Quantitative Economics with Python using JAX

Thomas J. Sargent & John Stachurski

Oct 12, 2026

CONTENTS

I Introduction **3**

1 About **5**
1.1 What is JAX? . . . 5
1.2 How to run these lectures . . . 6
1.3 Credits . . . 6
1.4 Prerequisites . . . 7

2 An Introduction to JAX **9**
2.1 JAX as a NumPy Replacement . . . 10
2.2 Random Numbers . . . 13
2.3 JIT compilation . . . 15
2.4 Functional Programming . . . 17
2.5 Gradients . . . 18
2.6 Writing vectorized code . . . 19
2.7 Exercises . . . 22

3 Adventures with Autodiff **25**
3.1 Overview . . . 25
3.2 What is automatic differentiation? . . . 26
3.3 Some experiments . . . 28
3.4 Gradient Descent . . . 33
3.5 Exercises . . . 37

4 Newton's Method via JAX **39**
4.1 Overview . . . 39
4.2 Newton in one dimension . . . 40
4.3 An Equilibrium Problem . . . 42
4.4 Computation . . . 42
4.5 Exercises . . . 45

II Simulation **51**

5 Inventory Dynamics **53**
5.1 Overview . . . 53
5.2 Sample paths . . . 54
5.3 Cross-sectional distributions . . . 55
5.4 Distribution dynamics . . . 58
5.5 Restock frequency . . . 60

6 Kesten Processes and Firm Dynamics **65**

6.1 Overview . . . 65
6.2 Kesten processes . . . 66
6.3 Exercises . . . 71

7 Wealth Distribution Dynamics **73**
7.1 Wealth dynamics . . . 74
7.2 Implementation . . . 75
7.3 Exercises . . . 84

III Asset Pricing **91**

8 Asset Pricing: The Lucas Asset Pricing Model **93**
8.1 Overview . . . 93
8.2 The Lucas model . . . 94
8.3 Computation . . . 98
8.4 Exercises . . . 104

9 An Asset Pricing Problem **107**
9.1 Overview . . . 107
9.2 Pricing a single payoff . . . 108
9.3 Pricing a cash flow . . . 109
9.4 Choosing the stochastic discount factor . . . 110
9.5 Solving for the price-dividend ratio . . . 110
9.6 Code . . . 111
9.7 An Extended Example . . . 115
9.8 Numpy Version . . . 116
9.9 JAX Version . . . 121
9.10 A memory-efficient JAX version . . . 123

IV Dynamic Programming **125**

10 Job Search **127**
10.1 Model . . . 128
10.2 Code . . . 128
10.3 Computing the solution . . . 130
10.4 Exercise . . . 132

11 Optimal Savings I: Value Function Iteration **137**
11.1 Overview . . . 138
11.2 Starting with NumPy . . . 139
11.3 Switching to JAX . . . 141
11.4 Switching to vmap . . . 145

12 Optimal Savings II: Alternative Algorithms **149**
12.1 Model primitives . . . 151
12.2 Operators . . . 152
12.3 Iteration . . . 154
12.4 Solvers . . . 155
12.5 Tests . . . 156

13 Shortest Paths **163**
13.1 Overview . . . 163
13.2 Solving for Minimum Cost-to-Go . . . 164

13.3 Exercises . . . 166

14 Optimal Investment **171**

15 Inventory Management Model **185**
15.1 A model with constant discounting . . . 185
15.2 Time varing discount rates . . . 186

16 Endogenous Grid Method **193**
16.1 Overview . . . 193
16.2 Setup . . . 194
16.3 Solution method . . . 195
16.4 Solutions . . . 200

V Macroeconomic Models **203**

17 Default Risk and Income Fluctuations **205**
17.1 Overview . . . 205
17.2 Structure . . . 207
17.3 Equilibrium . . . 209
17.4 Computation . . . 210
17.5 Results . . . 216
17.6 Exercises . . . 218

18 The Aiyagari Model **225**
18.1 Overview . . . 225
18.2 Firms . . . 227
18.3 Households . . . 228
18.4 Equilibrium . . . 235
18.5 Exercises . . . 237

19 The Hopenhayn Entry-Exit Model **241**
19.1 Outline . . . 241
19.2 The Model . . . 242
19.3 Code . . . 245
19.4 Solving the model . . . 250
19.5 Pareto tails . . . 253
19.6 Exercise . . . 255

20 Bianchi Overborrowing Model **259**
20.1 Markov dynamics . . . 259
20.2 Description of the model . . . 262
20.3 Overborrowing model in Python / JAX . . . 264
20.4 Planner problem . . . 269
20.5 Numerical solution . . . 271
20.6 Exercise . . . 273

VI Data and Empirics **277**

21 Maximum Likelihood Estimation **279**
21.1 Overview . . . 279
21.2 MLE with numerical methods (JAX) . . . 280
21.3 MLE with `statsmodels` . . . 285

22 Simple Neural Network Regression with Keras and JAX **289**
22.1 Data . . . 290
22.2 Models . . . 291
22.3 Training . . . 293

23 Neural Network Regression with JAX **297**
23.1 Outline . . . 297
23.2 Set up . . . 298
23.3 Training with Keras . . . 300
23.4 Training with JAX . . . 302
23.5 JAX plus Optax . . . 307
23.6 Summary . . . 311
23.7 Exercises . . . 311

24 Policy Gradient-Based Optimal Savings **317**
24.1 Introduction . . . 317
24.2 Theory . . . 318
24.3 Network . . . 320
24.4 Cake Eating Case . . . 322
24.5 IFP Model . . . 329

VII Racial Segregation **337**

25 Schelling's Model of Racial Segregation **339**
25.1 Outline . . . 339
25.2 Background . . . 340
25.3 The model . . . 345
25.4 Main Loop . . . 346
25.5 Results . . . 348
25.6 Performance . . . 353

26 Schelling Model with NumPy **355**
26.1 Overview . . . 355
26.2 Data Representation . . . 355
26.3 Helper Functions . . . 356
26.4 Visualization . . . 357
26.5 The Simulation . . . 357
26.6 Results . . . 358
26.7 Performance . . . 359

27 Schelling Model with JAX **361**
27.1 Overview . . . 361
27.2 Setup . . . 361
27.3 JAX-Compiled Functions . . . 362
27.4 The Simulation . . . 363
27.5 Results . . . 364
27.6 Limitations of This Approach . . . 365

28 Parallelizing the Algorithm **367**
28.1 Overview . . . 367
28.2 Setup . . . 367
28.3 The Parallel Algorithm . . . 368
28.4 Results . . . 370
28.5 Discussion . . . 372

29 Segregation with Persistent Shocks **373**
29.1 Overview . . . 373
29.2 Parameters . . . 373
29.3 Setup . . . 374
29.4 Type Flipping . . . 374
29.5 Simulation with Shocks . . . 374
29.6 Results . . . 375
29.7 Discussion . . . 379

VIII Other **381**

30 Troubleshooting **383**
30.1 Fixing Your Local Environment . . . 383
30.2 Reporting an Issue . . . 384

31 References **385**

32 Execution Statistics **387**

Bibliography **391**

Proof Index **393**

Index **395**

This website presents a set of lectures on quantitative economic modeling using GPUs and Google JAX.

- Introduction
  - *About*
  - *An Introduction to JAX*
  - *Adventures with Autodiff*
  - *Newton's Method via JAX*
- Simulation
  - *Inventory Dynamics*
  - *Kesten Processes and Firm Dynamics*
  - *Wealth Distribution Dynamics*
- Asset Pricing
  - *Asset Pricing: The Lucas Asset Pricing Model*
  - *An Asset Pricing Problem*
- Dynamic Programming
  - *Job Search*
  - *Optimal Savings I: Value Function Iteration*
  - *Optimal Savings II: Alternative Algorithms*
  - *Shortest Paths*
  - *Optimal Investment*
  - *Inventory Management Model*
  - *Endogenous Grid Method*
- Macroeconomic Models
  - *Default Risk and Income Fluctuations*
  - *The Aiyagari Model*
  - *The Hopenhayn Entry-Exit Model*
  - *Bianchi Overborrowing Model*
- Data and Empirics
  - *Maximum Likelihood Estimation*
  - *Simple Neural Network Regression with Keras and JAX*
  - *Neural Network Regression with JAX*
  - *Policy Gradient-Based Optimal Savings*
- Racial Segregation
  - *Schelling's Model of Racial Segregation*
  - *Schelling Model with NumPy*
  - *Schelling Model with JAX*
  - *Parallelizing the Algorithm*

  - *Segregation with Persistent Shocks*
- Other
  - *Troubleshooting*
  - *References*
  - *Execution Statistics*

Part I

Introduction

CHAPTER

ONE

ABOUT

Perhaps the single most notable feature of scientific computing in the past two decades is the rise and rise of parallel computation.

For example, the advanced artificial intelligence applications now shaking the worlds of business and academia require massive computer power to train, and the great majority of that computer power is supplied by GPUs.

For us economists, with our ever-growing need for more compute cycles, parallel computing provides both opportunities and new difficulties.

The main difficulty we face vis-a-vis parallel computation is accessibility.

Even for those with time to invest in careful parallelization of their programs, exploiting the full power of parallel hardware is challenging for non-experts.

Moreover, that hardware changes from year to year, so any human capital associated with mastering intricacies of a particular GPU has a very high depreciation rate.

For these reasons, we find Google JAX compelling.

In short, JAX makes high performance and parallel computing accessible (and fun!).

It provides a familiar array programming interface based on NumPy, and, as long as some simple conventions are adhered to, this code compiles to extremely efficient and well-parallelized machine code.

One of the most agreeable features of JAX is that the same code set and be run on either CPUs or GPUs, which allows users to test and develop locally, before deploying to a more powerful machine for heavier computations.

JAX is relatively easy to learn and highly portable, allowing us programmers to focus on the algorithms we want to implement, rather than particular features of our hardware.

This lecture series provides an introduction to using Google JAX for quantitative economics.

The rest of this page provides some background information on JAX, notes on how to run the lectures, and credits for our colleagues and RAs.

1.1 What is JAX?

JAX is an open source Python library developed by Google Research to support in-house artificial intelligence and machine learning.

JAX provides data types, functions and a compiler for fast linear algebra operations and automatic differentiation.

Loosely speaking, JAX is like NumPy with the addition of

- automatic differentiation
- automated GPU/TPU support

- a just-in-time compiler

In short, JAX delivers

1. high execution speeds on CPUs due to efficient parallelization and JIT compilation,
2. a powerful and convenient environment for GPU programming, and
3. the ability to efficiently differentiate smooth functions for optimization and estimation.

These features make JAX ideal for almost all quantitative economic modeling problems that require heavy-duty computing.

1.2 How to run these lectures

The easiest way to run these lectures is via Google Colab.

JAX is pre-installed with GPU support on Colab and Colab provides GPU access even on the free tier.

Each lecture has a "play" button on the top right that you can use to launch the lecture on Colab.

You might also like to try using JAX locally.

If you do not own a GPU, you can still install JAX for the CPU by following the relevant install instructions.

(We recommend that you install Anaconda Python first.)

If you do have a GPU, you can try installing JAX for the GPU by following the install instructions for GPUs.

(This is not always trivial but is starting to get easier.)

1.3 Credits

In building this lecture series, we had invaluable assistance from research assistants at QuantEcon and our QuantEcon colleagues.

In particular, we thank and credit

- Shu Hu
- Smit Lunagariya
- Matthew McKay
- Humphrey Yang
- Hengcheng Zhang
- Frank Wu

1.4 Prerequisites

We assume that readers have covered most of the QuantEcon lecture series on Python programming.

CHAPTER

TWO

AN INTRODUCTION TO JAX

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

This lecture provides a short introduction to Google JAX.

Let's see if we have an active GPU:

```
!nvidia-smi
```

```
Fri Oct  9 00:22:51 2026
+-----------------------------------------------------------------------------------------
↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
↪13.0     |
+-----------------------------------------+------------------------+----------------
↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
↪Compute M. |
|                                         |                        |              ␣
↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |              ␣
↪     0 |
| N/A   37C    P0             33W /   70W |       0MiB /  15360MiB |      0%      ␣
↪Default |
|                                         |                        |              ␣
↪   N/A |
+-----------------------------------------+------------------------+----------------
↪-------+

+-----------------------------------------------------------------------------------------
↪-------+
| Processes:                                                                            ␣
```

(continues on next page)

(continued from previous page)

```
↪        |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
↪Memory |
|        ID   ID                                                               ␣
↪Usage      |
|=============================================================================================|
|  No running processes found                                                           ␣
↪        |
+-----------------------------------------------------------------------------------------
↪-------+
```

2.1 JAX as a NumPy Replacement

One way to use JAX is as a plug-in NumPy replacement. Let's look at the similarities and differences.

2.1.1 Similarities

The following import is standard, replacing `import numpy as np`:

```
import jax
import jax.numpy as jnp
```

Now we can use `jnp` in place of `np` for the usual array operations:

```
a = jnp.asarray((1.0, 3.2, -1.5))
```

```
print(a)
```

```
[ 1.   3.2 -1.5]
```

```
print(jnp.sum(a))
```

```
2.6999998
```

```
print(jnp.mean(a))
```

```
0.9
```

```
print(jnp.dot(a, a))
```

```
13.490001
```

However, the array object `a` is not a NumPy array:

```
a
```

```
Array([ 1. ,  3.2, -1.5], dtype=float32)
```

```
type(a)
```

```
jaxlib._jax.ArrayImpl
```

Even scalar-valued maps on arrays return JAX arrays.

```
jnp.sum(a)
```

```
Array(2.6999998, dtype=float32)
```

JAX arrays are also called "device arrays," where term "device" refers to a hardware accelerator (GPU or TPU).

(In the terminology of GPUs, the "host" is the machine that launches GPU operations, while the "device" is the GPU itself.)

Operations on higher dimensional arrays are also similar to NumPy:

```
A = jnp.ones((2, 2))
B = jnp.identity(2)
A @ B
```

```
Array([[1., 1.],
       [1., 1.]], dtype=float32)
```

```
from jax.numpy import linalg
```

```
linalg.inv(B)   # Inverse of identity is identity
```

```
Array([[1., 0.],
       [0., 1.]], dtype=float32)
```

```
out = linalg.eigh(B)  # Computes eigenvalues and eigenvectors
out.eigenvalues
```

```
Array([0.99999994, 0.99999994], dtype=float32)
```

```
out.eigenvectors
```

```
Array([[1., 0.],
       [0., 1.]], dtype=float32)
```

2.1.2 Differences

One difference between NumPy and JAX is that JAX currently uses 32 bit floats by default.

This is standard for GPU computing and can lead to significant speed gains with small loss of precision.

However, for some calculations precision matters. In these cases 64 bit floats can be enforced via the command

```
jax.config.update("jax_enable_x64", True)
```

Let's check this works:

```
jnp.ones(3)
```

```
Array([1., 1., 1.], dtype=float64)
```

As a NumPy replacement, a more significant difference is that arrays are treated as **immutable**.

For example, with NumPy we can write

```
import numpy as np
a = np.linspace(0, 1, 3)
a
```

```
array([0. , 0.5, 1. ])
```

and then mutate the data in memory:

```
a[0] = 1
a
```

```
array([1. , 0.5, 1. ])
```

In JAX this fails:

```
a = jnp.linspace(0, 1, 3)
a
```

```
Array([0. , 0.5, 1. ], dtype=float64)
```

```
a[0] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 1
----> 1 a[0] = 1

File ~/miniconda3/envs/quantecon/lib/python3.13/site-packages/jax/_src/numpy/array_
↪methods.py:975, in _unimplemented_setitem(self, i, x)
    971 def _unimplemented_setitem(self, i, x):
    972   msg = ("JAX arrays are immutable and do not support in-place item␣
↪assignment."
    973          " Instead of x[idx] = y, use x = x.at[idx].set(y) or another .
↪at[] method:"
    974          " https://docs.jax.dev/en/latest/_autosummary/jax.numpy.ndarray.
↪at.html")
--> 975   raise TypeError(msg)

TypeError: JAX arrays are immutable and do not support in-place item assignment.␣
↪Instead of x[idx] = y, use x = x.at[idx].set(y) or another .at[] method: https://
↪docs.jax.dev/en/latest/_autosummary/jax.numpy.ndarray.at.html
```

In line with immutability, JAX does not support inplace operations:

```
a = np.array((2, 1))
a.sort()
a
```

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

```
a = jnp.array((2, 1))
a_new = a.sort()
a, a_new
```

```
(Array([2, 1], dtype=int64), Array([1, 2], dtype=int64))
```

The designers of JAX chose to make arrays immutable because JAX uses a functional programming style. More on this below.

However, JAX provides a functionally pure equivalent of in-place array modification using the `at` method.

```
a = jnp.linspace(0, 1, 3)
id(a)
```

```
134689016677968
```

```
a
```

```
Array([0. , 0.5, 1. ], dtype=float64)
```

Applying `at[0].set(1)` returns a new copy of `a` with the first element set to 1

```
a = a.at[0].set(1)
a
```

```
Array([1. , 0.5, 1. ], dtype=float64)
```

Inspecting the identifier of `a` shows that it has been reassigned

```
id(a)
```

```
134687254436432
```

2.2 Random Numbers

Random numbers are also a bit different in JAX, relative to NumPy. Typically, in JAX, the state of the random number generator needs to be controlled explicitly.

```
import jax.random as random
```

First we produce a key, which seeds the random number generator.

```
key = random.key(1)
```

```
type(key)
```

```
jax._src.random.prng.PRNGKeyArray
```

```
print(key)
```

```
Array((), dtype=key<fry>) overlaying:
[0 1]
```

Now we can use the key to generate some random numbers:

```
x = random.normal(key, (3, 3))
x
```

```
Array([[-1.18428442, -0.11617041,  0.17269028],
       [ 0.95730718, -0.83295415,  0.69080517],
       [ 0.07545021, -0.7645271 , -0.05064539]], dtype=float64)
```

If we use the same key again, we initialize at the same seed, so the random numbers are the same:

```
random.normal(key, (3, 3))
```

```
Array([[-1.18428442, -0.11617041,  0.17269028],
       [ 0.95730718, -0.83295415,  0.69080517],
       [ 0.07545021, -0.7645271 , -0.05064539]], dtype=float64)
```

To produce a (quasi-) independent draw, we can `split` the existing key.

```
key, subkey = random.split(key)
```

```
random.normal(key, (3, 3))
```

```
Array([[ 1.09221959,  0.33192176, -0.90184197],
       [-1.37815779,  0.43052577,  1.6068202 ],
       [ 0.04053753, -0.78732842,  1.75917181]], dtype=float64)
```

```
random.normal(subkey, (3, 3))
```

```
Array([[ 0.7158846 ,  0.03955972,  0.71127682],
       [-0.40080158, -0.91609481,  0.23713062],
       [ 0.85253995, -0.80972695,  1.79431941]], dtype=float64)
```

As we will see, the `split` operation is particularly useful for parallel computing, where independent sequences or simulations can be given their own key.

Another option is `fold_in`, which produces new "independent" keys from a base key.

The function below produces `k` (quasi-) independent random `n x n` matrices using this procedure.

```
base_key = random.key(42)
def gen_random_matrices(key, n, k):
    matrices = []
    for i in range(k):
        key = random.fold_in(base_key, i)  # generate a fresh key
        matrices.append(random.uniform(key, (n, n)))
    return matrices
```

```
matrices = gen_random_matrices(key, 2, 2)
for A in matrices:
    print(A)
```

```
[[0.23566993 0.39719189]
 [0.95367373 0.42397776]]
[[0.74211901 0.54715578]
 [0.05988742 0.32206803]]
```

To get a one-dimensional array of normal random draws, we can either use `(len, )` for the shape, as in

```
random.normal(key, (5, ))
```

```
Array([ 1.09221959,  0.33192176, -0.90184197, -1.37815779,  0.43052577],
 ↪dtype=float64)
```

or simply use `5` as the shape argument:

```
random.normal(key, 5)
```

```
Array([ 1.09221959,  0.33192176, -0.90184197, -1.37815779,  0.43052577],
 ↪dtype=float64)
```

2.3 JIT compilation

The JAX just-in-time (JIT) compiler accelerates logic within functions by fusing linear algebra operations into a single optimized kernel that the host can launch on the GPU / TPU (or CPU if no accelerator is detected).

2.3.1 A first example

To see the JIT compiler in action, consider the following function.

```
def f(x):
    a = 3*x + jnp.sin(x) + jnp.cos(x**2) - jnp.cos(2*x) - x**2 * 0.4 * x**1.5
    return jnp.sum(a)
```

Let's build an array to call the function on.

```
n = 50_000_000
x = jnp.ones(n)
```

How long does the function take to execute?

```
%time f(x).block_until_ready()
```

```
CPU times: user 511 ms, sys: 7.67 ms, total: 519 ms
Wall time: 700 ms
```

```
Array(2.19896006e+08, dtype=float64)
```

Note

Here, in order to measure actual speed, we use the `block_until_ready()` method to hold the interpreter until the results of the computation are returned from the device. This is necessary because JAX uses asynchronous dispatch, which allows the Python interpreter to run ahead of GPU computations.

The code doesn't run as fast as we might hope, given that it's running on a GPU.

But if we run it a second time it becomes much faster:

```
%time f(x).block_until_ready()
```

```
CPU times: user 3.47 ms, sys: 0 ns, total: 3.47 ms
Wall time: 170 ms
```

```
Array(2.19896006e+08, dtype=float64)
```

This is because the built in functions like `jnp.cos` are JIT compiled and the first run includes compile time.

Why would JAX want to JIT-compile built in functions like `jnp.cos` instead of just providing pre-compiled versions, like NumPy?

The reason is that the JIT compiler can specialize on the *size* of the array being used, which is helpful for parallelization.

For example, in running the code above, the JIT compiler produced a version of `jnp.cos` that is specialized to floating point arrays of size `n = 50_000_000`.

We can check this by calling `f` with a new array of different size.

```
m = 50_000_001
y = jnp.ones(m)
```

```
%time f(y).block_until_ready()
```

```
CPU times: user 356 ms, sys: 13.9 ms, total: 370 ms
Wall time: 545 ms
```

```
Array(2.19896011e+08, dtype=float64)
```

Notice that the execution time increases, because now new versions of the built-ins like `jnp.cos` are being compiled, specialized to the new array size.

If we run again, the code is dispatched to the correct compiled version and we get faster execution.

```
%time f(y).block_until_ready()
```

```
CPU times: user 2.24 ms, sys: 1.23 ms, total: 3.46 ms
Wall time: 108 ms
```

```
Array(2.19896011e+08, dtype=float64)
```

The compiled versions for the previous array size are still available in memory too, and the following call is dispatched to the correct compiled code.

```
%time f(x).block_until_ref()
```

```
CPU times: user 2.47 ms, sys: 999 µs, total: 3.47 ms
Wall time: 123 ms
```

```
Array(2.19896006e+08, dtype=float64)
```

2.3.2 Compiling the outer function

We can do even better if we manually JIT-compile the outer function.

```
f_jit = jax.jit(f)   # target for JIT compilation
```

Let's run once to compile it:

```
f_jit(x)
```

```
Array(2.19896006e+08, dtype=float64)
```

And now let's time it.

```
%time f_jit(x).block_until_ready()
```

```
CPU times: user 533 µs, sys: 92 µs, total: 625 µs
Wall time: 69.2 ms
```

```
Array(2.19896006e+08, dtype=float64)
```

Note the speed gain.

This is because the array operations are fused and no intermediate arrays are created.

Incidentally, a more common syntax when targetting a function for the JIT compiler is

```
@jax.jit
def f(x):
    a = 3*x + jnp.sin(x) + jnp.cos(x**2) - jnp.cos(2*x) - x**2 * 0.4 * x**1.5
    return jnp.sum(a)
```

2.4 Functional Programming

From JAX's documentation:

When walking about the countryside of Italy, the people will not hesitate to tell you that JAX has "una anima di pura programmazione funzionale".

In other words, JAX assumes a functional programming style.

The major implication is that JAX functions should be pure.

A pure function will always return the same result if invoked with the same inputs.

In particular, a pure function has

- no dependence on global variables and
- no side effects

JAX will not usually throw errors when compiling impure functions but execution becomes unpredictable.

Here's an illustration of this fact, using global variables:

```
a = 1  # global

@jax.jit
def f(x):
    return a + x
```

```
x = jnp.ones(2)
```

```
f(x)
```

```
Array([2., 2.], dtype=float64)
```

In the code above, the global value `a=1` is fused into the jitted function.

Even if we change `a`, the output of `f` will not be affected — as long as the same compiled version is called.

```
a = 42
```

```
f(x)
```

```
Array([2., 2.], dtype=float64)
```

Changing the dimension of the input triggers a fresh compilation of the function, at which time the change in the value of `a` takes effect:

```
x = jnp.ones(3)
```

```
f(x)
```

```
Array([43., 43., 43.], dtype=float64)
```

Moral of the story: write pure functions when using JAX!

2.5 Gradients

JAX can use automatic differentiation to compute gradients.

This can be extremely useful for optimization and solving nonlinear systems.

We will see significant applications later in this lecture series.

For now, here's a very simple illustration involving the function

```
def f(x):
    return (x**2) / 2
```

Let's take the derivative:

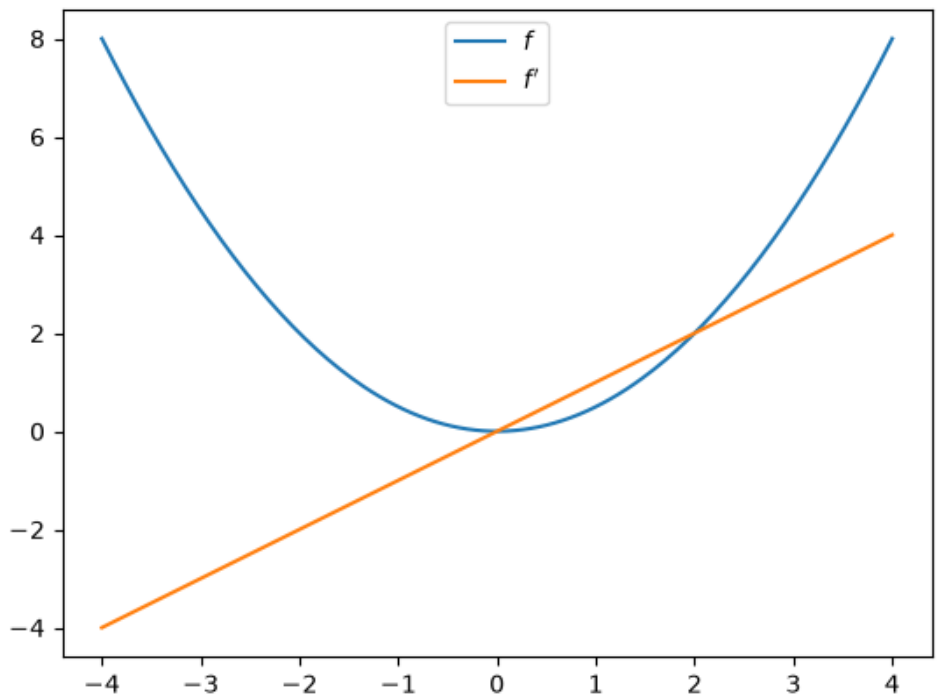
```
f_prime = jax.grad(f)
```

```
f_prime(10.0)
```

```
Array(10., dtype=float64, weak_type=True)
```

Let's plot the function and derivative, noting that $f'(x) = x$.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x_grid = jnp.linspace(-4, 4, 200)
ax.plot(x_grid, f(x_grid), label="$f$")
ax.plot(x_grid, [f_prime(x) for x in x_grid], label="$f'$")
ax.legend(loc='upper center')
plt.show()
```



We defer further exploration of automatic differentiation with JAX until *Adventures with Autodiff*.

2.6 Writing vectorized code

Writing fast JAX code requires shifting repetitive tasks from loops to array processing operations, so that the JAX compiler can easily understand the whole operation and generate more efficient machine code.

This procedure is called **vectorization** or **array programming**, and will be familiar to anyone who has used NumPy or MATLAB.

In most ways, vectorization is the same in JAX as it is in NumPy.

But there are also some differences, which we highlight here.

As a running example, consider the function

$$f(x, y) = \frac{\cos(x^2 + y^2)}{1 + x^2 + y^2}$$

Suppose that we want to evaluate this function on a square grid of x and y points and then plot it.

To clarify, here is the slow `for` loop version.

```
@jax.jit
def f(x, y):
    return jnp.cos(x**2 + y**2) / (1 + x**2 + y**2)

n = 80
x = jnp.linspace(-2, 2, n)
y = x

z_loops = np.empty((n, n))
```

```
%%time
for i in range(n):
    for j in range(n):
        z_loops[i, j] = f(x[i], y[j])
```

```
CPU times: user 6.98 s, sys: 2.09 s, total: 9.07 s
Wall time: 6.08 s
```

Even for this very small grid, the run time is extremely slow.

(Notice that we used a NumPy array for `z_loops` because we wanted to write to it.)

OK, so how can we do the same operation in vectorized form?

If you are new to vectorization, you might guess that we can simply write

```
z_bad = f(x, y)
```

But this gives us the wrong result because JAX doesn't understand the nested for loop.

```
z_bad.shape
```

```
(80,)
```

Here is what we actually wanted:

```
z_loops.shape
```

```
(80, 80)
```

To get the right shape and the correct nested for loop calculation, we can use a `meshgrid` operation designed for this purpose:

```
x_mesh, y_mesh = jnp.meshgrid(x, y)
```

Now we get what we want and the execution time is very fast.

```
%%time
z_mesh = f(x_mesh, y_mesh).block_until_ready()
```

```
CPU times: user 43.7 ms, sys: 1.69 ms, total: 45.3 ms
Wall time: 80.2 ms
```

Let's run again to eliminate compile time.

```
%%time
z_mesh = f(x_mesh, y_mesh).block_until_ready()
```

```
CPU times: user 469 µs, sys: 107 µs, total: 576 µs
Wall time: 303 µs
```

Let's confirm that we got the right answer.

```
jnp.allclose(z_mesh, z_loops)
```

```
Array(True, dtype=bool)
```

Now we can set up a serious grid and run the same calculation (on the larger grid) in a short amount of time.

```
n = 6000
x = jnp.linspace(-2, 2, n)
y = x
x_mesh, y_mesh = jnp.meshgrid(x, y)
```

```
%%time
z_mesh = f(x_mesh, y_mesh).block_until_ready()
```

```
CPU times: user 55.2 ms, sys: 1.67 ms, total: 56.9 ms
Wall time: 133 ms
```

Let's run again to get rid of compile time.

```
%%time
z_mesh = f(x_mesh, y_mesh).block_until_ready()
```

```
CPU times: user 697 µs, sys: 0 ns, total: 697 µs
Wall time: 29.1 ms
```

But there is one problem here: the mesh grids use a lot of memory.

```
x_mesh.nbytes + y_mesh.nbytes
```

```
576000000
```

By comparison, the flat array `x` is just

```
x.nbytes  # and y is just a pointer to x
```

```
48000
```

This extra memory usage can be a big problem in actual research calculations.

So let's try a different approach using jax.vmap

First we vectorize `f` in `y`.

```
f_vec_y = jax.vmap(f, in_axes=(None, 0))
```

In the line above, `(None, 0)` indicates that we are vectorizing in the second argument, which is `y`.

Next, we vectorize in the first argument, which is `x`.

```
f_vec = jax.vmap(f_vec_y, in_axes=(0, None))
```

With this construction, we can now call the function f on flat (low memory) arrays.

```
%%time
z_vmap = f_vec(x, y).block_until_ready()
```

```
CPU times: user 76.2 ms, sys: 1.62 ms, total: 77.8 ms
Wall time: 132 ms
```

We run it again to eliminate compile time.

```
%%time
z_vmap = f_vec(x, y).block_until_ready()
```

```
CPU times: user 1.58 ms, sys: 115 µs, total: 1.69 ms
Wall time: 26.4 ms
```

The execution time is essentially the same as the mesh operation but we are using much less memory.

And we produce the correct answer:

```
jnp.allclose(z_vmap, z_mesh)
```

```
Array(True, dtype=bool)
```

2.7 Exercises

Exercise 2.7.1

In the Exercise section of a lecture on Numba, we used Monte Carlo to price a European call option.

The code was accelerated by Numba-based multithreading.

Try writing a version of this operation for JAX, using all the same parameters.

If you are running your code on a GPU, you should be able to achieve significantly faster execution.

Solution

Here is one solution:

```
M = 10_000_000

n, β, K = 20, 0.99, 100
μ, ρ, ν, S0, h0 = 0.0001, 0.1, 0.001, 10, 0
```

```
@jax.jit
def compute_call_price_jax(β=β,
                           μ=μ,
                           S0=S0,
                           h0=h0,
                           K=K,
                           n=n,
                           ρ=ρ,
                           ν=ν,
                           M=M,
                           key=jax.random.key(1)):

    s = jnp.full(M, np.log(S0))
    h = jnp.full(M, h0)

    def update(i, loop_state):
        s, h, key = loop_state
        key, subkey = jax.random.split(key)
        Z = jax.random.normal(subkey, (2, M))
        s = s + μ + jnp.exp(h) * Z[0, :]
        h = ρ * h + ν * Z[1, :]
        new_loop_state = s, h, key
        return new_loop_state

    initial_loop_state = s, h, key
    final_loop_state = jax.lax.fori_loop(0, n, update, initial_loop_state)
    s, h, key = final_loop_state

    expectation = jnp.mean(jnp.maximum(jnp.exp(s) - K, 0))

    return β**n * expectation
```

Note

We use `jax.lax.fori_loop` instead of a Python `for` loop. This allows JAX to compile the loop efficiently without unrolling it, which significantly reduces compilation time for large arrays.

Let's run it once to compile it:

```
%%time
compute_call_price_jax().block_until_ready()
```

```
CPU times: user 1.78 s, sys: 27.9 ms, total: 1.81 s
Wall time: 1.58 s
```

```
Array(699495.97040563, dtype=float64)
```

And now let's time it:

```
%%time
compute_call_price_jax().block_until_ref()
```

```
CPU times: user 1.16 ms, sys: 35 µs, total: 1.2 ms
Wall time: 476 ms
```

```
Array(699495.97040563, dtype=float64)
```

CHAPTER

THREE

ADVENTURES WITH AUTODIFF

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

3.1 Overview

This lecture gives an introduction to automatic differentiation using Google JAX.

Automatic differentiation is one of the key elements of modern machine learning and artificial intelligence.

As such it has attracted a great deal of investment and there are several powerful implementations available.

One of the best of these is the automatic differentiation routines contained in JAX.

While other software packages also offer this feature, the JAX version is particularly powerful because it integrates so well with other core components of JAX (e.g., JIT compilation and parallelization).

Automatic differentiation can be used not only for AI but also for many problems faced in mathematical modeling, such as multi-dimensional nonlinear optimization and root-finding problems.

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

```
!pip install jax
```

We need the following imports

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
from sympy import symbols
```

3.2 What is automatic differentiation?

Autodiff is a technique for calculating derivatives on a computer.

3.2.1 Autodiff is not finite differences

The derivative of $f(x) = \exp(2x)$ is

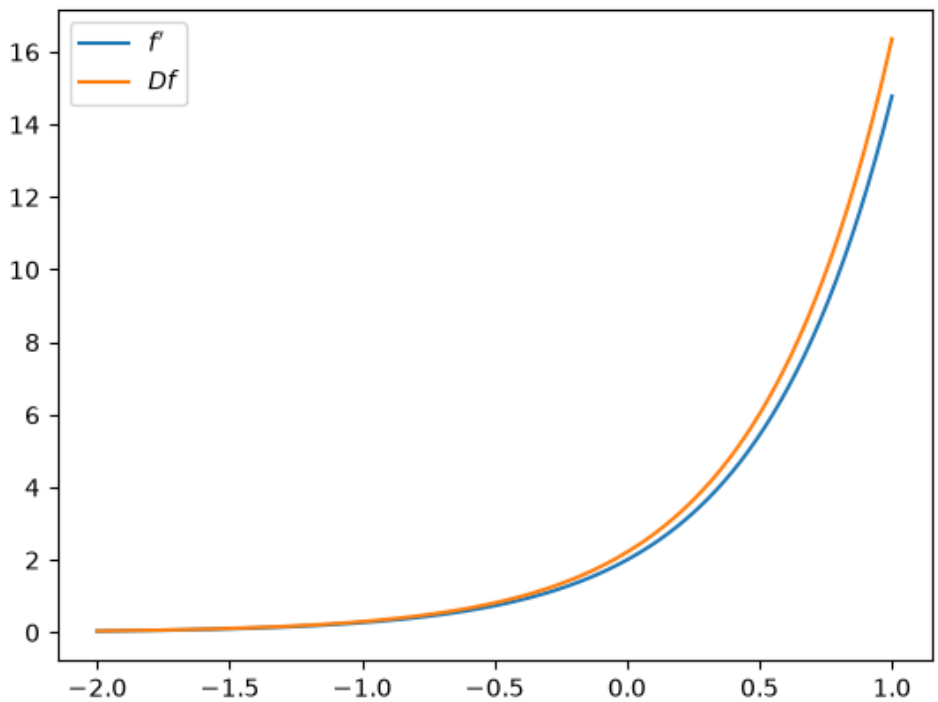
$$f'(x) = 2\exp(2x)$$

A computer that doesn't know how to take derivatives might approximate this with the finite difference ratio

$$(Df)(x) := \frac{f(x+h) - f(x)}{h}$$

where h is a small positive number.

```
def f(x):
    "Original function."
    return np.exp(2 * x)

def f_prime(x):
    "True derivative."
    return 2 * np.exp(2 * x)

def Df(x, h=0.1):
    "Approximate derivative (finite difference)."
    return (f(x + h) - f(x))/h

x_grid = np.linspace(-2, 1, 200)
fig, ax = plt.subplots()
ax.plot(x_grid, f_prime(x_grid), label="$f'$")
ax.plot(x_grid, Df(x_grid), label="$Df$")
ax.legend()
plt.show()
```



This kind of numerical derivative is often inaccurate and unstable.

One reason is that

$$\frac{f(x+h) - f(x)}{h} \approx \frac{0}{0}$$

Small numbers in the numerator and denominator cause rounding errors.

The situation is exponentially worse in high dimensions / with higher order derivatives.

3.2.2 Autodiff is not symbolic calculus

Symbolic calculus tries to use rules for differentiation to produce a single closed-form expression representing a derivative.

```
m, a, b, x = symbols('m a b x')
f_x = (a*x + b)**m
f_x.diff((x, 6))  # 6-th order derivative
```

$$\frac{a^6 m \left(ax+b\right)^m \left(m^5 - 15m^4 + 85m^3 - 225m^2 + 274m - 120\right)}{\left(ax+b\right)^6}$$

Symbolic calculus is not well suited to high performance computing.

One disadvantage is that symbolic calculus cannot differentiate through control flow.

Also, using symbolic calculus might involve redundant calculations.

For example, consider

$$(fgh)' = (f'g + g'f)h + (fg)h'$$

If we evaluate at x, then we evaluate $f(x)$ and $g(x)$ twice each.

Also, computing $f'(x)$ and $f(x)$ might involve similar terms (e.g., $f(x) = \exp(2x) \implies f'(x) = 2f(x)$) but this is not exploited in symbolic algebra.

3.2.3 Autodiff

Autodiff produces functions that evaluate derivatives at numerical values passed in by the calling code, rather than producing a single symbolic expression representing the entire derivative.

Derivatives are constructed by breaking calculations into component parts via the chain rule.

The chain rule is applied until the point where the terms reduce to primitive functions that the program knows how to differentiate exactly (addition, subtraction, exponentiation, sine and cosine, etc.)

3.3 Some experiments

Let's start with some real-valued functions on $\mathbb{R}$.

3.3.1 A differentiable function

Let's test JAX's autodiff with a relatively simple function.

```
def f(x):
    return jnp.sin(x) - 2 * jnp.cos(3 * x) * jnp.exp(- x**2)
```

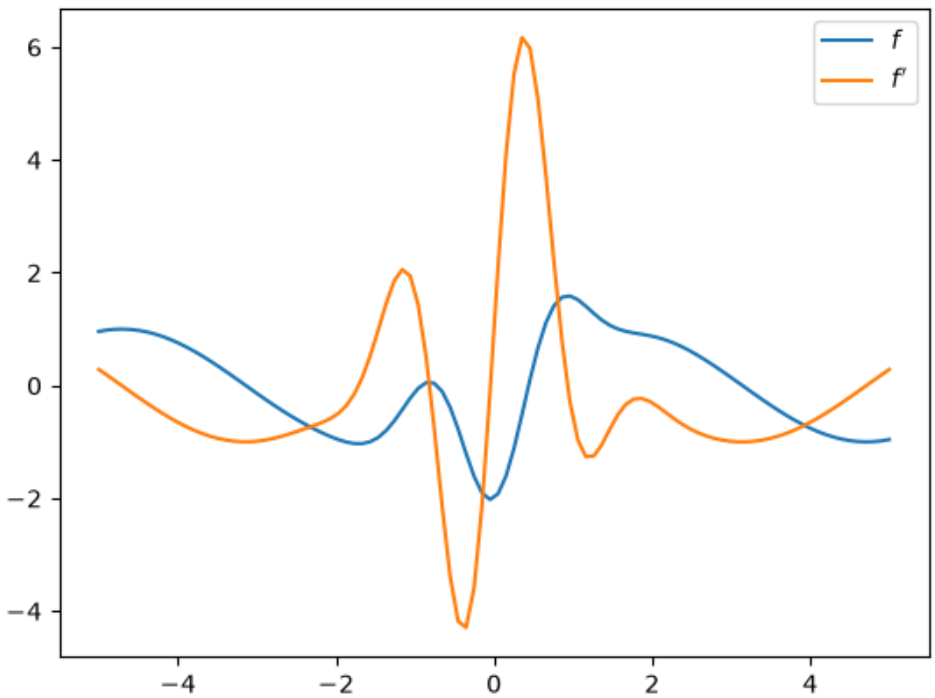
We use `grad` to compute the gradient of a real-valued function:

```
f_prime = jax.grad(f)
```

Let's plot the result:

```
x_grid = jnp.linspace(-5, 5, 100)
```

```
fig, ax = plt.subplots()
ax.plot(x_grid, [f(x) for x in x_grid], label="$f$")
ax.plot(x_grid, [f_prime(x) for x in x_grid], label="$f'$")
ax.legend()
plt.show()
```



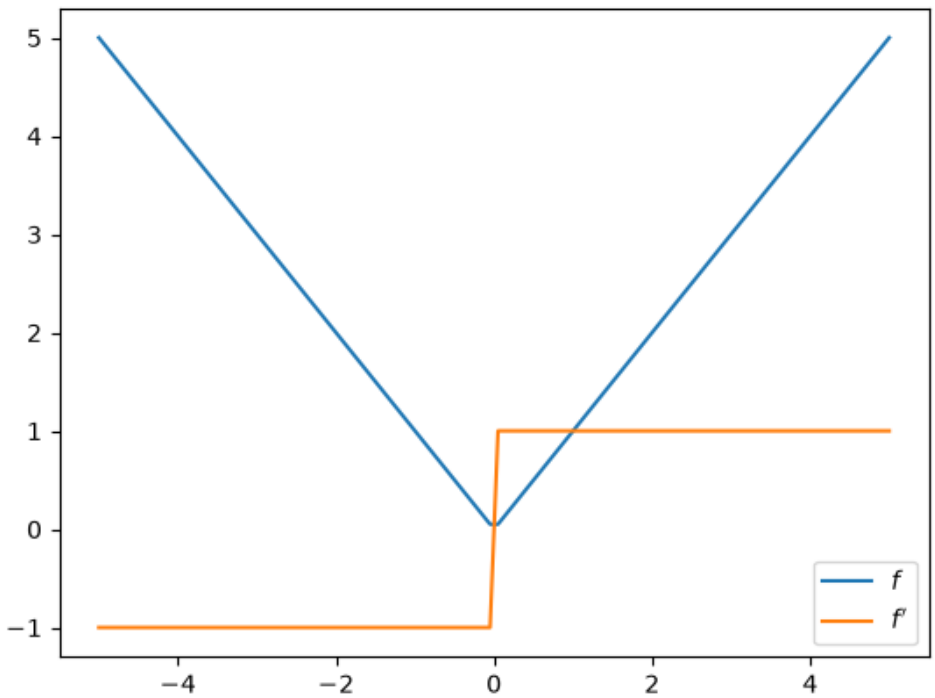
3.3.2 Absolute value function

What happens if the function is not differentiable?

```
def f(x):
    return jnp.abs(x)
```

```
f_prime = jax.grad(f)
```

```
fig, ax = plt.subplots()
ax.plot(x_grid, [f(x) for x in x_grid], label="$f$")
ax.plot(x_grid, [f_prime(x) for x in x_grid], label="$f'$")
ax.legend()
plt.show()
```



At the nondifferentiable point 0, `jax.grad` returns the right derivative:

```
f_prime(0.0)
```

```
Array(1., dtype=float32, weak_type=True)
```

3.3.3 Differentiating through control flow

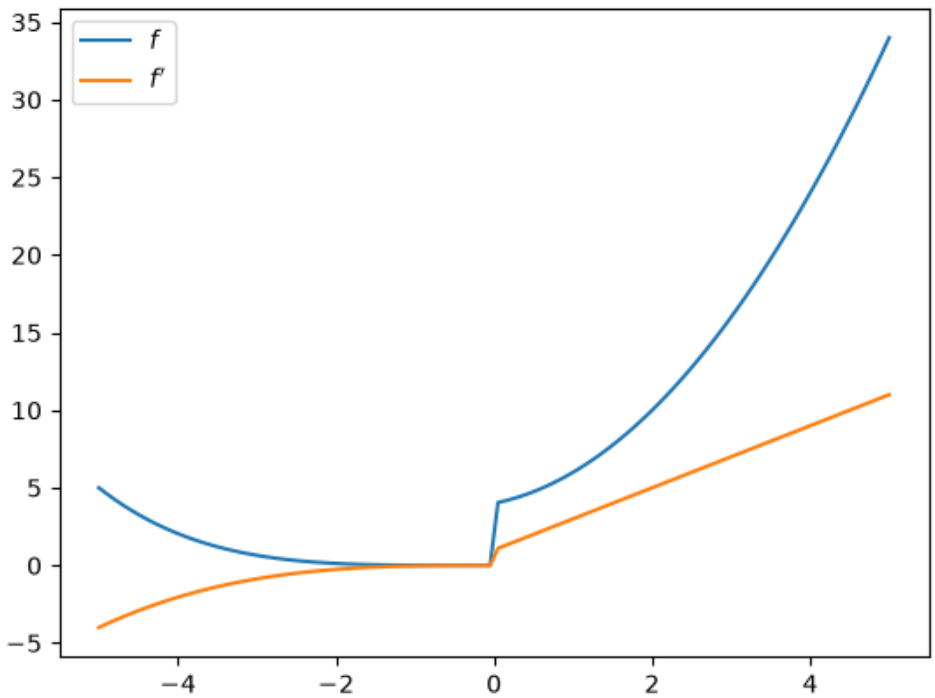
Let's try differentiating through some loops and conditions.

```
def f(x):
    def f1(x):
        for i in range(2):
            x *= 0.2 * x
        return x
    def f2(x):
        x = sum((x**i + i) for i in range(3))
        return x
    y = f1(x) if x < 0 else f2(x)
    return y
```

```
f_prime = jax.grad(f)
```

```
x_grid = jnp.linspace(-5, 5, 100)
```

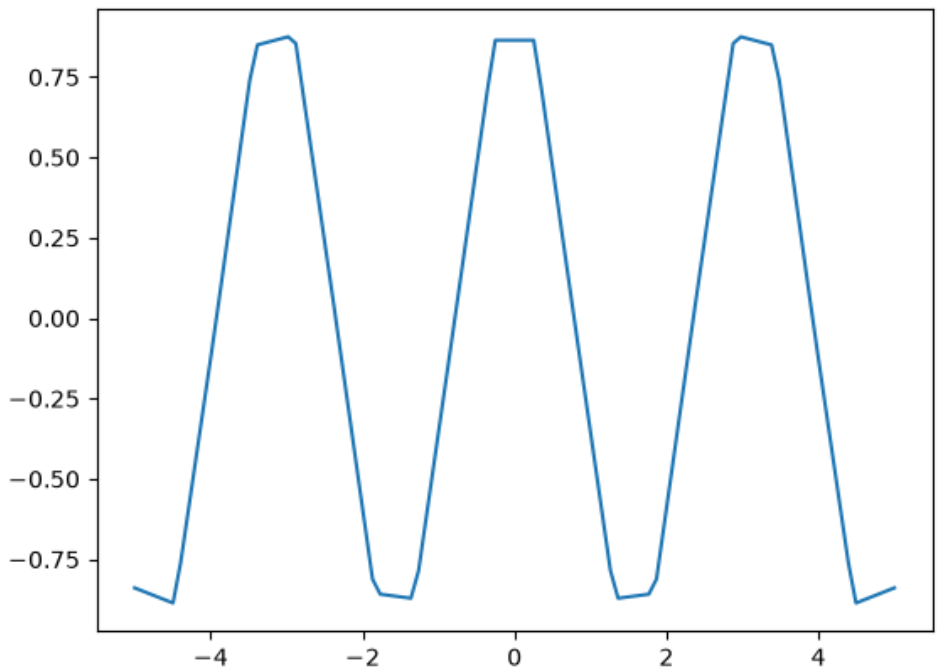
```
fig, ax = plt.subplots()
ax.plot(x_grid, [f(x) for x in x_grid], label="$f$")
ax.plot(x_grid, [f_prime(x) for x in x_grid], label="$f'$")
ax.legend()
plt.show()
```



3.3.4 Differentiating through a linear interpolation

We can differentiate through linear interpolation, even though the function is not smooth:

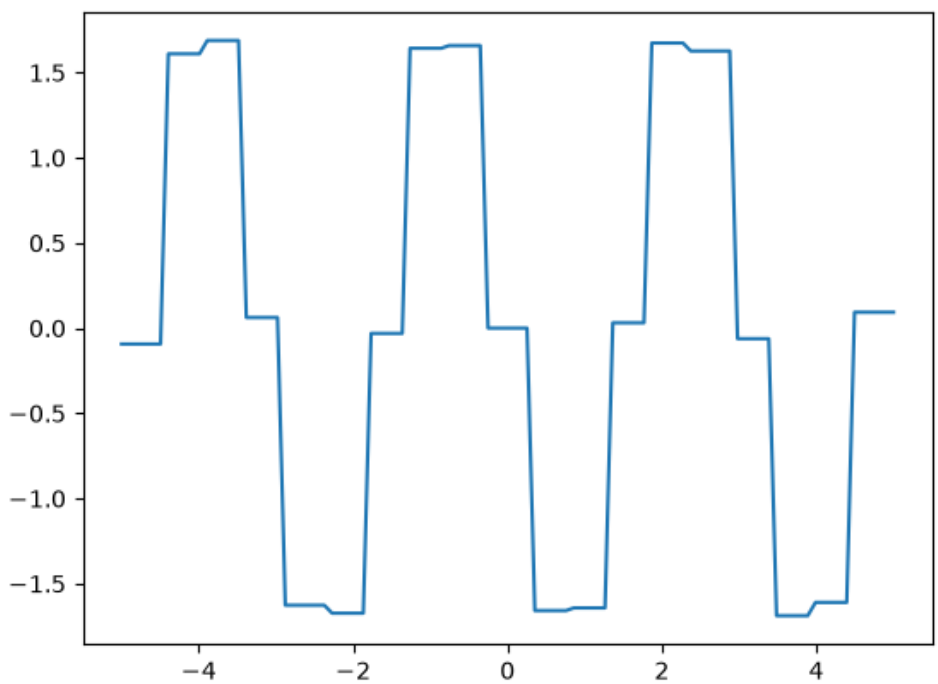
```
n = 20
xp = jnp.linspace(-5, 5, n)
yp = jnp.cos(2 * xp)

fig, ax = plt.subplots()
ax.plot(x_grid, jnp.interp(x_grid, xp, yp))
plt.show()
```




```
f_prime = jax.grad(jnp.interp)
```

```
f_prime_vec = jax.vmap(f_prime, in_axes=(0, None, None))
```

```
fig, ax = plt.subplots()
ax.plot(x_grid, f_prime_vec(x_grid, xp, yp))
plt.show()
```



3.4 Gradient Descent

Let's try implementing gradient descent.

As a simple application, we'll use gradient descent to solve for the OLS parameter estimates in simple linear regression.

3.4.1 A function for gradient descent

Here's an implementation of gradient descent.

```
def grad_descent(f,        # Function to be minimized
                 args,     # Extra arguments to the function
                 x0,       # Initial condition
                 λ=0.1,    # Initial learning rate
                 tol=1e-5,
                 max_iter=1_000):
    """
    Minimize the function f via gradient descent, starting from guess x0.

    The learning rate is computed according to the Barzilai-Borwein method.

    """

    f_grad = jax.grad(f)
    x = jnp.array(x0)
    df = f_grad(x, args)
    ϵ = tol + 1
    i = 0
    while ϵ > tol and i < max_iter:
        new_x = x - λ * df
        new_df = f_grad(new_x, args)
        Δx = new_x - x
        Δdf = new_df - df
        λ = jnp.abs(Δx @ Δdf) / (Δdf @ Δdf)
        ϵ = jnp.max(jnp.abs(Δx))
        x, df = new_x, new_df
        i += 1

    return x
```

3.4.2 Simulated data

We're going to test our gradient descent function by minimizing a sum of least squares in a regression problem.

Let's generate some simulated data:

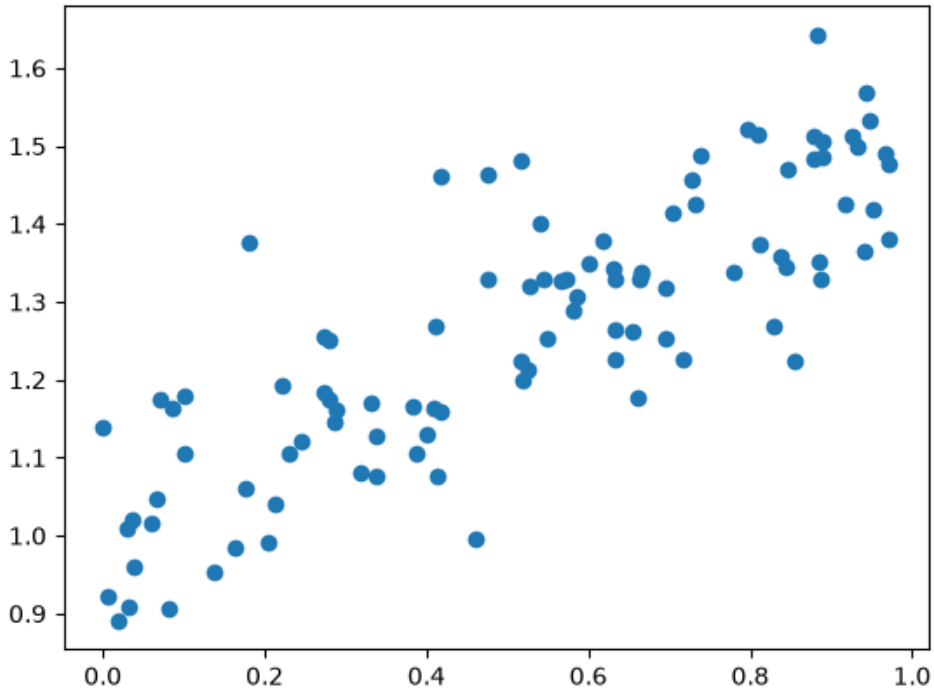
```
n = 100
key = jax.random.key(1234)
key, x_key, ϵ_key = jax.random.split(key, 3)
x = jax.random.uniform(x_key, (n,))

α, β, σ = 0.5, 1.0, 0.1  # Set the true intercept and slope.
ϵ = jax.random.normal(ϵ_key, (n,))
```

(continues on next page)

(continued from previous page)

```
y = α * x + β + σ * ϵ
```

```
fig, ax = plt.subplots()
ax.scatter(x, y)
plt.show()
```



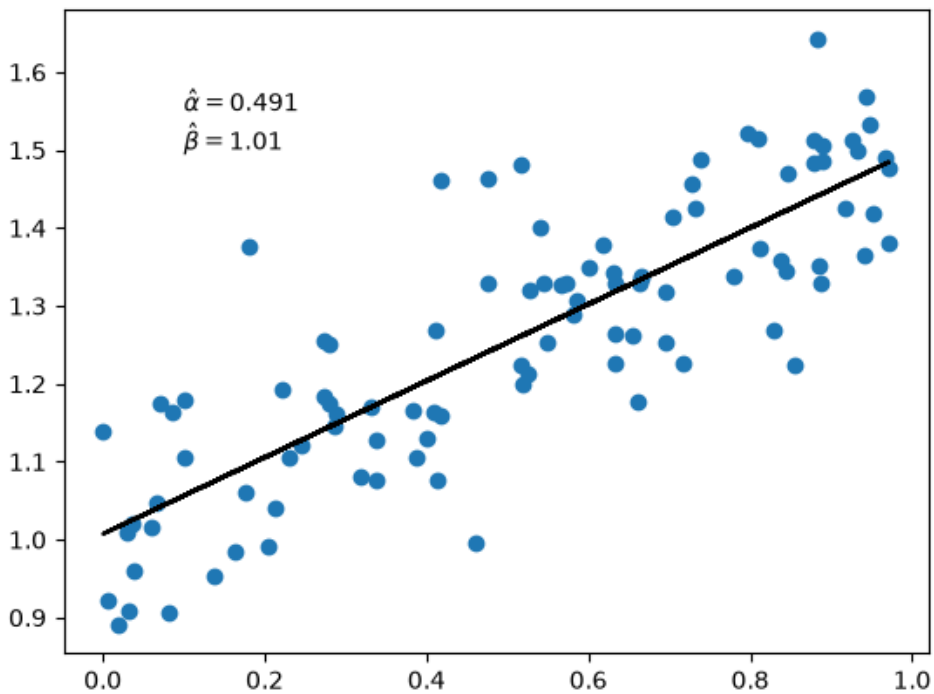
Let's start by calculating the estimated slope and intercept using closed form solutions.

```
mx = x.mean()
my = y.mean()
α_hat = jnp.sum((x - mx) * (y - my)) / jnp.sum((x - mx)**2)
β_hat = my - α_hat * mx
```

```
α_hat, β_hat
```

```
(Array(0.4914626, dtype=float32), Array(1.007149, dtype=float32))
```

```
fig, ax = plt.subplots()
ax.scatter(x, y)
ax.plot(x, α_hat * x + β_hat, 'k-')
ax.text(0.1, 1.55, rf'$\hat \alpha = {α_hat:.3}$')
ax.text(0.1, 1.50, rf'$\hat \beta = {β_hat:.3}$')
plt.show()
```



3.4.3 Minimizing squared loss by gradient descent

Let's see if we can get the same values with our gradient descent function.

First we set up the least squares loss function.

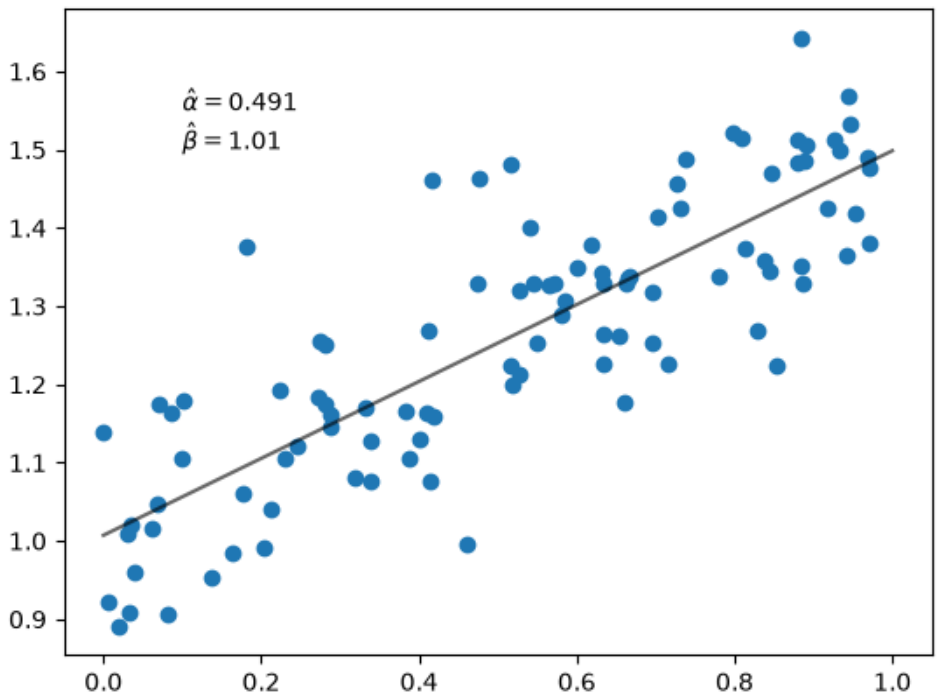
```
@jax.jit
def loss(params, data):
    a, b = params
    x, y = data
    return jnp.sum((y - a * x - b)**2)
```

Now we minimize it:

```
p0 = jnp.zeros(2)  # Initial guess for α, β
data = x, y
α_hat, β_hat = grad_descent(loss, data, p0)
```

Let's plot the results.

```
fig, ax = plt.subplots()
x_grid = jnp.linspace(0, 1, 100)
ax.scatter(x, y)
ax.plot(x_grid, α_hat * x_grid + β_hat, 'k-', alpha=0.6)
ax.text(0.1, 1.55, rf'$\hat \alpha = {α_hat:.3}$')
ax.text(0.1, 1.50, rf'$\hat \beta = {β_hat:.3}$')
plt.show()
```



Notice that we get the same estimates as we did from the closed form solutions.

3.4.4 Adding a squared term

Now let's try fitting a second order polynomial.

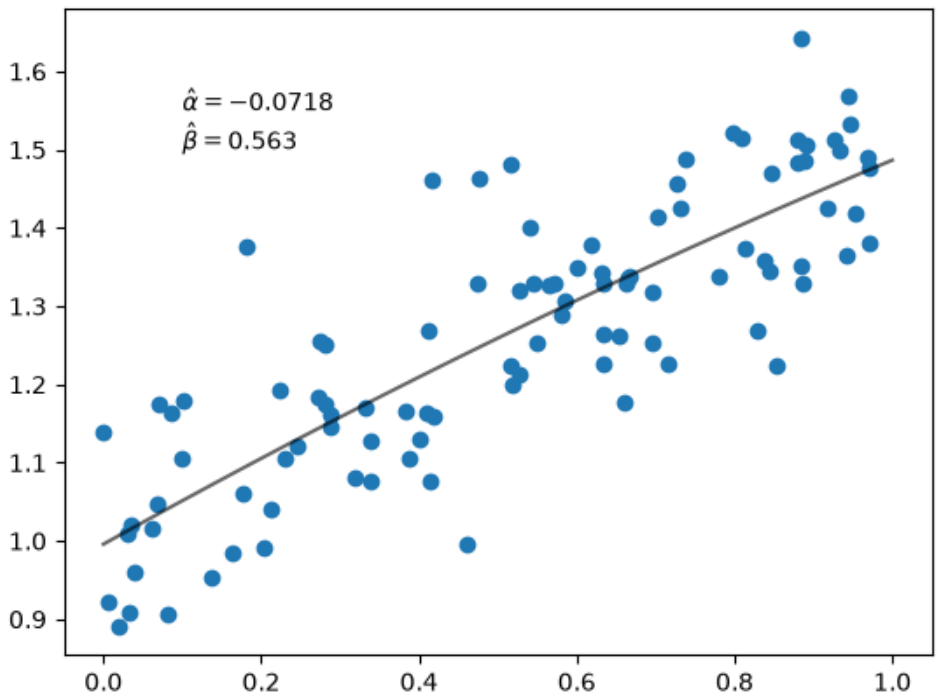
Here's our new loss function.

```
@jax.jit
def loss(params, data):
    a, b, c = params
    x, y = data
    return jnp.sum((y - a * x**2 - b * x - c)**2)
```

Now we're minimizing in three dimensions.

Let's try it.

```
p0 = jnp.zeros(3)
α_hat, β_hat, γ_hat = grad_descent(loss, data, p0)

fig, ax = plt.subplots()
ax.scatter(x, y)
ax.plot(x_grid, α_hat * x_grid**2 + β_hat * x_grid + γ_hat, 'k-', alpha=0.6)
ax.text(0.1, 1.55, rf'$\hat \alpha = {α_hat:.3}$')
ax.text(0.1, 1.50, rf'$\hat \beta = {β_hat:.3}$')
plt.show()
```



3.5 Exercises

Exercise 3.5.1

The function `jnp.polyval` evaluates polynomials.

For example, if `len(p)` is 3, then `jnp.polyval(p, x)` returns

$$f(p, x) := p_0 x^2 + p_1 x + p_2$$

Use this function for polynomial regression.

The (empirical) loss becomes

$$\ell(p, x, y) = \sum_{i=1}^{n} (y_i - f(p, x_i))^2$$

Set $k = 4$ and set the initial guess of `params` to `jnp.zeros(k)`.

Use gradient descent to find the array `params` that minimizes the loss function and plot the result (following the examples above).

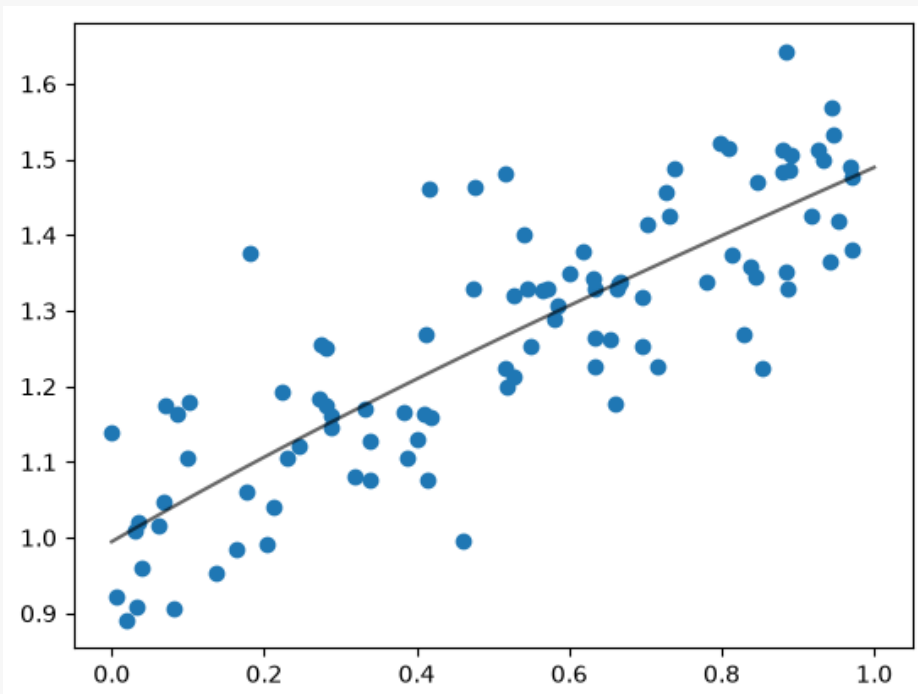
Solution

Here's one solution.

```
def loss(params, data):
    x, y = data
    return jnp.sum((y - jnp.polyval(params, x))**2)
```

```
k = 4
p0 = jnp.zeros(k)
p_hat = grad_descent(loss, data, p0)
print('Estimated parameter vector:')
print(p_hat)
print('\n\n')

fig, ax = plt.subplots()
ax.scatter(x, y)
ax.plot(x_grid, jnp.polyval(p_hat, x_grid), 'k-', alpha=0.6)
plt.show()
```

```
Estimated parameter vector:
[ 0.04280353 -0.13148506  0.58347857  0.9946171 ]
```



CHAPTER

FOUR

NEWTON'S METHOD VIA JAX

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

4.1 Overview

One of the key features of JAX is automatic differentiation.

We introduced this feature in *Adventures with Autodiff*.

In this lecture we apply automatic differentiation to the problem of computing economic equilibria via Newton's method.

Newton's method is a relatively simple root and fixed point solution algorithm, which we discussed in a more elementary QuantEcon lecture.

JAX is ideally suited to implementing Newton's method efficiently, even in high dimensions.

We use the following imports in this lecture

```
import jax
import jax.numpy as jnp
from scipy.optimize import root
import matplotlib.pyplot as plt
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:25:25 2026
+-----------------------------------------------------------------------------------
↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
↪13.0     |
+-----------------------------------------+------------------------+---------------
↪-------+
```

(continues on next page)

(continued from previous page)

```
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
↪Compute M. |
|                                         |                        |              ␣
↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |              ␣
↪     0 |
| N/A   36C    P0             27W /   70W |       0MiB /  15360MiB |      0%      ␣
↪Default |
|                                         |                        |              ␣
↪   N/A |
+-----------------------------------------+------------------------+---------------
↪-------+

+-----------------------------------------------------------------------------------
↪-------+
| Processes:                                                                        ␣
↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
↪Memory |
|        ID   ID                                                               ␣
↪Usage      |
|===================================================================================|
|  No running processes found                                                       ␣
↪       |
+-----------------------------------------------------------------------------------
↪-------+
```

4.2 Newton in one dimension

As a warm up, let's implement Newton's method in JAX for a simple one-dimensional root-finding problem.

Let f be a function from $\mathbb{R}$ to itself.

A **root** of f is an $x \in \mathbb{R}$ such that $f(x) = 0$.

Recall that Newton's method for solving for the root of f involves iterating with the map q defined by

$$q(x) = x - \frac{f(x)}{f'(x)}$$

Here is a function called `newton` that takes a function f plus a scalar value x_0, iterates with q starting from x_0, and returns an approximate fixed point.

```python
def newton(f, x_0, tol=1e-5):
    f_prime = jax.grad(f)
    def q(x):
        return x - f(x) / f_prime(x)

    error = tol + 1
    x = x_0
    while error > tol:
        y = q(x)
```

(continues on next page)

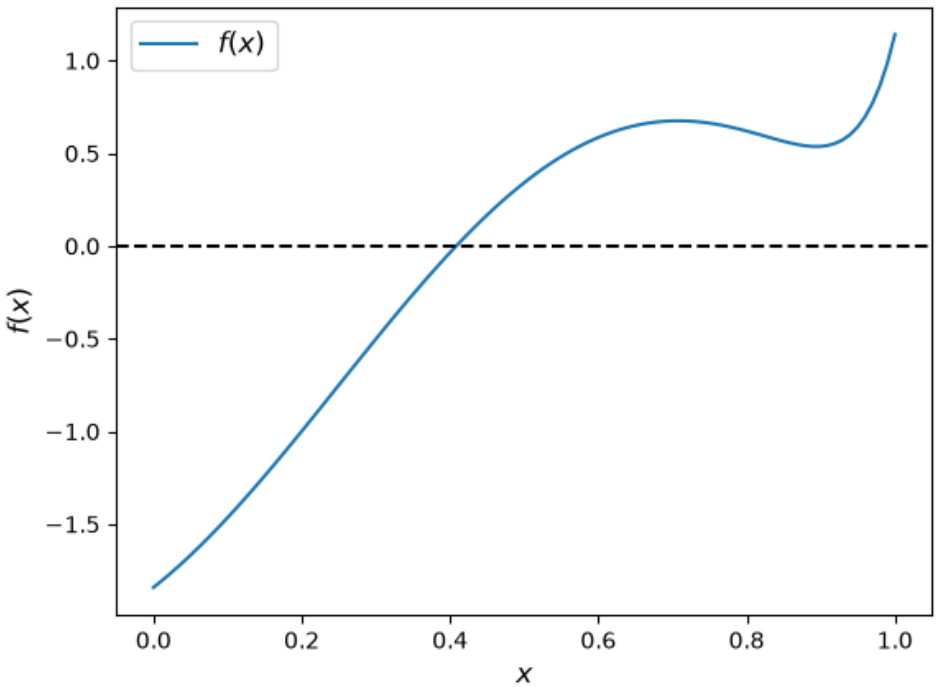
(continued from previous page)

```
        error = abs(x - y)
        x = y

    return x
```

The code above uses automatic differentiation to calculate f' via the call to `jax.grad`.

Let's test our `newton` routine on the function shown below.

```
f = lambda x: jnp.sin(4 * (x - 1/4)) + x + x**20 - 1
x = jnp.linspace(0, 1, 100)

fig, ax = plt.subplots()
ax.plot(x, f(x), label='$f(x)$')
ax.axhline(ls='--', c='k')
ax.set_xlabel('$x$', fontsize=12)
ax.set_ylabel('$f(x)$', fontsize=12)
ax.legend(fontsize=12)
plt.show()
```



Here we go

```
newton(f, 0.2)
```

```
Array(0.4082935, dtype=float32, weak_type=True)
```

This number looks to be close to the root, given the figure.

4.3 An Equilibrium Problem

Now let's move up to higher dimensions.

First we describe a market equilibrium problem we will solve with JAX via root-finding.

The market is for n goods.

(We are extending a two-good version of the market from an earlier lecture.)

The supply function for the i-th good is

$$q_i^s(p) = b_i \sqrt{p_i}$$

which we write in vector form as

$$q^s(p) = b\sqrt{p}$$

(Here $\sqrt{p}$ is the square root of each p_i and $b\sqrt{p}$ is the vector formed by taking the pointwise product $b_i\sqrt{p_i}$ at each i.)

The demand function is

$$q^d(p) = \exp(-Ap) + c$$

(Here A is an $n \times n$ matrix containing parameters, c is an $n \times 1$ vector and the exp function acts pointwise (element-by-element) on the vector $-Ap$.)

The excess demand function is

$$e(p) = \exp(-Ap) + c - b\sqrt{p}$$

An **equilibrium price** vector is an n-vector p such that $e(p) = 0$.

The function below calculates the excess demand for given parameters

```
def e(p, A, b, c):
    return jnp.exp(- A @ p) + c - b * jnp.sqrt(p)
```

4.4 Computation

In this section we describe and then implement the solution method.

4.4.1 Newton's Method

We use a multivariate version of Newton's method to compute the equilibrium price.

The rule for updating a guess p_n of the equilibrium price vector is

$$p_{n+1} = p_n - J_e(p_n)^{-1} e(p_n) \tag{4.1}$$

Here $J_e(p_n)$ is the Jacobian of e evaluated at p_n.

Iteration starts from initial guess p_0.

Instead of coding the Jacobian by hand, we use automatic differentiation via `jax.jacobian()`.

```
def newton(f, x_0, tol=1e-5, max_iter=15):
    """
    A multivariate Newton root-finding routine.

    """
    x = x_0
    f_jac = jax.jacobian(f)
    @jax.jit
    def q(x):
        " Updates the current guess. "
        return x - jnp.linalg.solve(f_jac(x), f(x))
    error = tol + 1
    n = 0
    while error > tol:
        n += 1
        if(n > max_iter):
            raise Exception('Max iteration reached without convergence')
        y = q(x)
        error = jnp.linalg.norm(x - y)
        x = y
        print(f'iteration {n}, error = {error}')
    return x
```

4.4.2 Application

Let's now apply the method just described to investigate a large market with 5,000 goods.

We randomly generate the matrix A and set the parameter vectors b, c to 1.

```
dim = 5_000
seed = 32

# Create a random matrix A and normalize the rows to sum to one
key = jax.random.key(seed)
A = jax.random.uniform(key, [dim, dim])
s = jnp.sum(A, axis=0)
A = A / s

# Set up b and c
b = jnp.ones(dim)
c = jnp.ones(dim)
```

Here's our initial condition p_0

```
init_p = jnp.ones(dim)
```

By combining the power of Newton's method, JAX accelerated linear algebra, automatic differentiation, and a GPU, we obtain a relatively small error for this high-dimensional problem in just a few seconds:

```
%%time
p = newton(lambda p: e(p, A, b, c), init_p).block_until_ready()
```

```
iteration 1, error = 29.977436065673828
iteration 2, error = 5.0928263664245605
iteration 3, error = 0.10971631854772568
```

(continues on next page)

(continued from previous page)

```
iteration 4, error = 5.183630491956137e-05
iteration 5, error = 1.2464071005524602e-05
iteration 6, error = 5.521076218428789e-06
CPU times: user 7.58 s, sys: 1.65 s, total: 9.23 s
Wall time: 9.66 s
```

We run it again to eliminate compile time.

```
%%time
p = newton(lambda p: e(p, A, b, c), init_p).block_until_ready()
```

```
iteration 1, error = 29.977436065673828
iteration 2, error = 5.0928263664245605
iteration 3, error = 0.10971631854772568
iteration 4, error = 5.183630491956137e-05
iteration 5, error = 1.2464071005524602e-05
iteration 6, error = 5.521076218428789e-06
CPU times: user 6.97 s, sys: 1.52 s, total: 8.49 s
Wall time: 9.09 s
```

Here's the size of the error:

```
jnp.max(jnp.abs(e(p, A, b, c)))
```

```
Array(1.1920929e-07, dtype=float32)
```

With the same tolerance, SciPy's `root` function takes much longer to run, even with the Jacobian supplied.

```
%%time
solution = root(lambda p: e(p, A, b, c),
                init_p,
                jac=lambda p: jax.jacobian(e)(p, A, b, c),
                method='hybr',
                tol=1e-5)
```

```
CPU times: user 2min 39s, sys: 538 ms, total: 2min 40s
Wall time: 2min 39s
```

The result is also slightly less accurate:

```
p = solution.x
jnp.max(jnp.abs(e(p, A, b, c)))
```

```
Array(9.536743e-07, dtype=float32)
```

4.5 Exercises

Exercise 4.5.1

Consider a three-dimensional extension of the Solow fixed point problem with

$$A = \begin{pmatrix} 2 & 3 & 3 \\ 2 & 4 & 2 \\ 1 & 5 & 1 \end{pmatrix}, \quad s = 0.2, \quad \alpha = 0.5, \quad \delta = 0.8$$

As before the law of motion is

$$k_{t+1} = g(k_t) \quad \text{where} \quad g(k) := sAk^\alpha + (1-\delta)k$$

However k_t is now a 3×1 vector.

Solve for the fixed point using Newton's method with the following initial values:

$$\begin{aligned} k1_0 &= (1, 1, 1) \\ k2_0 &= (3, 5, 5) \\ k3_0 &= (50, 50, 50) \end{aligned}$$

Hint

- The computation of the fixed point is equivalent to computing k^* such that $f(k^*) - k^* = 0$.
- If you are unsure about your solution, you can start with the solved example:

$$A = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

with $s = 0.3$, $\alpha = 0.3$, and $\delta = 0.4$ and starting value:

$$k_0 = (1, 1, 1)$$

The result should converge to the analytical solution.

Solution

Let's first define the parameters for this problem

```
A = jnp.array([[2.0, 3.0, 3.0],
               [2.0, 4.0, 2.0],
               [1.0, 5.0, 1.0]])
s = 0.2
α = 0.5
δ = 0.8
initLs = [jnp.ones(3),
          jnp.array([3.0, 5.0, 5.0]),
          jnp.repeat(50.0, 3)]
```

Then we define the multivariate version of the formula for the law of motion of capital

```
def multivariate_solow(k, A=A, s=s, α=α, δ=δ):
    return s * jnp.dot(A, k**α) + (1 - δ) * k
```

Let's run through each starting value and see the output

```
attempt = 1
for init in initLs:
    print(f'Attempt {attempt}: Starting value is {init} \n')
    %time k = newton(lambda k: multivariate_solow(k) - k, \
                     init).block_until_ready()
    print('-'*64)
    attempt += 1
```

```
Attempt 1: Starting value is [1. 1. 1.]
iteration 1, error = 50.496315002441406
iteration 2, error = 41.1093864440918
iteration 3, error = 4.294127464294434
iteration 4, error = 0.3854290544986725
iteration 5, error = 0.0054382034577429295
iteration 6, error = 8.92080606718082e-07
CPU times: user 236 ms, sys: 15.9 ms, total: 252 ms
Wall time: 245 ms
----------------------------------------------------------------
Attempt 2: Starting value is [3. 5. 5.]

iteration 1, error = 2.0701100826263428
iteration 2, error = 0.12642373144626617
iteration 3, error = 0.0006017307168804109
iteration 4, error = 3.3717478231665154e-07
CPU times: user 159 ms, sys: 3.01 ms, total: 162 ms
Wall time: 116 ms
----------------------------------------------------------------
Attempt 3: Starting value is [50. 50. 50.]
iteration 1, error = 73.00942993164062
iteration 2, error = 6.493789196014404
iteration 3, error = 0.6806989312171936
iteration 4, error = 0.016202213242650032
iteration 5, error = 1.0600916539260652e-05
iteration 6, error = 9.830249609876773e-07
CPU times: user 372 ms, sys: 12.2 ms, total: 385 ms
Wall time: 293 ms
----------------------------------------------------------------
```

We find that the results are invariant to the starting values.

But the number of iterations it takes to converge is dependent on the starting values.

Let substitute the output back into the formulate to check our last result

```
multivariate_solow(k) - k
```

```
Array([ 4.7683716e-07,  0.0000000e+00, -2.3841858e-07], dtype=float32)
```

Note the error is very small.

We can also test our results on the known solution

```
A = jnp.array([[2.0, 0.0, 0.0],
               [0.0, 2.0, 0.0],
               [0.0, 0.0, 2.0]])
s = 0.3
α = 0.3
δ = 0.4
init = jnp.repeat(1.0, 3)
%time k = newton(lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k, \
                 init).block_until_ready()
```

```
iteration 1, error = 1.5745922327041626
iteration 2, error = 0.21344946324825287
iteration 3, error = 0.002045975998044014
iteration 4, error = 8.259061701210157e-07
CPU times: user 306 ms, sys: 14.1 ms, total: 320 ms
Wall time: 230 ms
```

```
# Now we time it without compile
%time k = newton(lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k, \
                 init).block_until_ready()
```

```
iteration 1, error = 1.5745922327041626
iteration 2, error = 0.21344946324825287
iteration 3, error = 0.002045975998044014
iteration 4, error = 8.259061701210157e-07
CPU times: user 328 ms, sys: 11.2 ms, total: 339 ms
Wall time: 240 ms
```

The result is very close to the true solution but still slightly different.

We can increase the precision of the floating point numbers and restrict the tolerance to obtain a more accurate approximation (see detailed discussion in the lecture on JAX)

```
# We will use 64 bit floats with JAX in order to increase the precision.
jax.config.update("jax_enable_x64", True)
init = init.astype('float64')

%time k = newton(lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k,\
                 init, tol=1e-7).block_until_ready()
```

```
iteration 1, error = 1.5745916432444333
iteration 2, error = 0.21344933091258958
iteration 3, error = 0.0020465547718452695
iteration 4, error = 2.0309190076799282e-07
iteration 5, error = 1.538370149106851e-15
CPU times: user 258 ms, sys: 11.1 ms, total: 269 ms
Wall time: 302 ms
```

```
# Now we time it without compile
%time k = newton(lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k,\
                 init, tol=1e-7).block_until_ready()
```

```
iteration 1, error = 1.5745916432444333
iteration 2, error = 0.21344933091258958
iteration 3, error = 0.0020465547718452695
iteration 4, error = 2.0309190076799282e-07
iteration 5, error = 1.538370149106851e-15
CPU times: user 146 ms, sys: 6.09 ms, total: 152 ms
Wall time: 114 ms
```

We can see it steps towards a more accurate solution.

Exercise 4.5.2

In this exercise, let's try different initial values and check how Newton's method responds to different starting points.

Let's define a three-good problem with the following default values:

$$A = \begin{pmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.2 & 0.5 \\ 0.1 & 0.8 & 0.1 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \qquad \text{and} \qquad c = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

For this exercise, use the following extreme price vectors as initial values:

$$\begin{aligned} p1_0 &= (5, 5, 5) \\ p2_0 &= (1, 1, 1) \\ p3_0 &= (4.5, 0.1, 4) \end{aligned}$$

Set the tolerance to 10^{-15} for more accurate output.

Hint

Similar to exercise 1, enabling `float64` for JAX can improve the precision of our results.

Solution

Define parameters and initial values

```
A = jnp.array([
    [0.2, 0.1, 0.7],
    [0.3, 0.2, 0.5],
    [0.1, 0.8, 0.1]
])
b = jnp.array([1.0, 1.0, 1.0])
c = jnp.array([1.0, 1.0, 1.0])
initLs = [jnp.repeat(5.0, 3),
          jnp.array([4.5, 0.1, 4.0])]
```

Let's run through each initial guess and check the output

```
attempt = 1
for init in initLs:
    print(f'Attempt {attempt}: Starting value is {init} \n')
    init = init.astype('float64')
    %time p = newton(lambda p: e(p, A, b, c), \
                     init, \
                     tol=1e-15, max_iter=15).block_until_ready()
    print('-'*64)
    attempt +=1
```

```
Attempt 1: Starting value is [5. 5. 5.]
iteration 1, error = 9.243805733085065
iteration 2, error = nan
CPU times: user 132 ms, sys: 6.92 ms, total: 139 ms
Wall time: 112 ms
----------------------------------------------------------------
Attempt 2: Starting value is [4.5 0.1 4. ]
iteration 1, error = 4.892018895185869
iteration 2, error = 1.2120550201694784
iteration 3, error = 0.6942087122866175
iteration 4, error = 0.168951089180319
iteration 5, error = 0.005209730313222213
iteration 6, error = 4.3632751705775364e-06
iteration 7, error = 3.0460818773540415e-12
iteration 8, error = 0.0
CPU times: user 122 ms, sys: 9.96 ms, total: 132 ms
Wall time: 103 ms
----------------------------------------------------------------
```

We can find that Newton's method may fail for some starting values.

Sometimes it may take a few initial guesses to achieve convergence.

Substitute the result back to the formula to check our result

```
e(p, A, b, c)
```

```
Array([0., 0., 0.], dtype=float64)
```

We can see the result is very accurate.

Part II

Simulation

CHAPTER

FIVE

INVENTORY DYNAMICS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

5.1 Overview

This lecture explores the inventory dynamics of a firm using so-called s-S inventory control.

Loosely speaking, this means that the firm

- waits until inventory falls below some value s
- and then restocks up to level S (or, in some models, restocks with a bulk order of S units).

We will be interested in the distribution of the associated Markov process, which can be thought of as cross-sectional distributions of inventory levels across a large number of firms, all of which

1. evolve independently and
2. have the same dynamics.

Note that we also studied this model in a separate lecture, using Numba.

Here we study the same problem using JAX.

We will use the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from jax import random, lax
from typing import NamedTuple
from time import time
```

Here's a description of our GPU:

```
!nvidia-smi
```

```
Fri Oct  9 00:22:33 2026
+-----------------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+----------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |               ␣
 ↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                  ␣
 ↪     0 |
| N/A   36C    P0             27W /   70W |       0MiB /  15360MiB |      0%      ␣
 ↪Default |
|                                         |                        |                  ␣
 ↪   N/A |
+-----------------------------------------+------------------------+----------------
 ↪-------+

+-----------------------------------------------------------------------------------------
 ↪-------+
| Processes:                                                                            ␣
 ↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
 ↪Memory |
|        ID   ID                                                               ␣
 ↪Usage      |
|=========================================================================================|
|  No running processes found                                                           ␣
 ↪       |
+-----------------------------------------------------------------------------------------
 ↪-------+
```

5.2 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 across time and firms.

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 `namedtuple` that stores parameters.

```
class ModelParameters(NamedTuple):
    s: int = 10
    S: int = 100
    μ: float = 1.0
    σ: float = 0.5
```

5.3 Cross-sectional distributions

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

The probability distribution ψ_T is the time T distribution of firm inventory levels implied by the model.

We will approximate this distribution by

1. fixing n to be some large number, indicating the number of firms in the simulation,
2. fixing T, the time period we are interested in,
3. generating n independent draws from some fixed distribution ψ_0 that gives the initial cross-section of inventories for the n firms, and
4. shifting this distribution forward in time T periods, updating each firm T times via the dynamics described above (independent of other firms).

We will then visualize ψ_T by histogramming the cross-section.

We will use the following code to update the cross-section of firms by one period.

```
@jax.jit
def update_cross_section(params: ModelParameters,
                         X_vec: jnp.ndarray,
                         D: jnp.ndarray) -> jnp.ndarray:
    """
    Update by one period a cross-section of firms with inventory levels given by
    X_vec, given the vector of demand shocks in D. Here D[i] is the demand shock
    for firm i with current inventory X_vec[i].

    """
    # Unpack
    s, S = params.s, params.S
    # Restock if the inventory is below the threshold
    X_new = jnp.where(X_vec <= s,
                      jnp.maximum(S - D, 0),
                      jnp.maximum(X_vec - D, 0))
    return X_new
```

5.3.1 For loop version

Now we provide code to compute the cross-sectional distribution ψ_T given some initial distribution ψ_0 and a positive integer T.

In this code we use an ordinary Python `for` loop to step forward through time

(Below we will squeeze out more speed by compiling the outer loop as well as the update rule.)

In the code below, the initial distribution ψ_0 takes all firms to have initial inventory `x_init`.

```
def project_cross_section(params: ModelParameters,
                          x_init: jnp.ndarray,
                          T: int,
                          key: jnp.ndarray,
                          num_firms: int = 50_000) -> jnp.ndarray:
    # Set up initial distribution
    X_vec = jnp.full((num_firms, ), x_init)
    # Loop
    for i in range(T):
        key, subkey = random.split(key)
        Z = random.normal(subkey, shape=(num_firms, ))
        D = jnp.exp(params.μ + params.σ * Z)

        X_vec = update_cross_section(params, X_vec, D)

    return X_vec
```

We'll use the following specification

```
params = ModelParameters()
x_init = 50
T = 500
# Initialize random number generator
key = random.key(10)
```

Let's look at the timing.

```
start_time = time()
X_vec = project_cross_section(
    params, x_init, T, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 1227.519512 ms
```

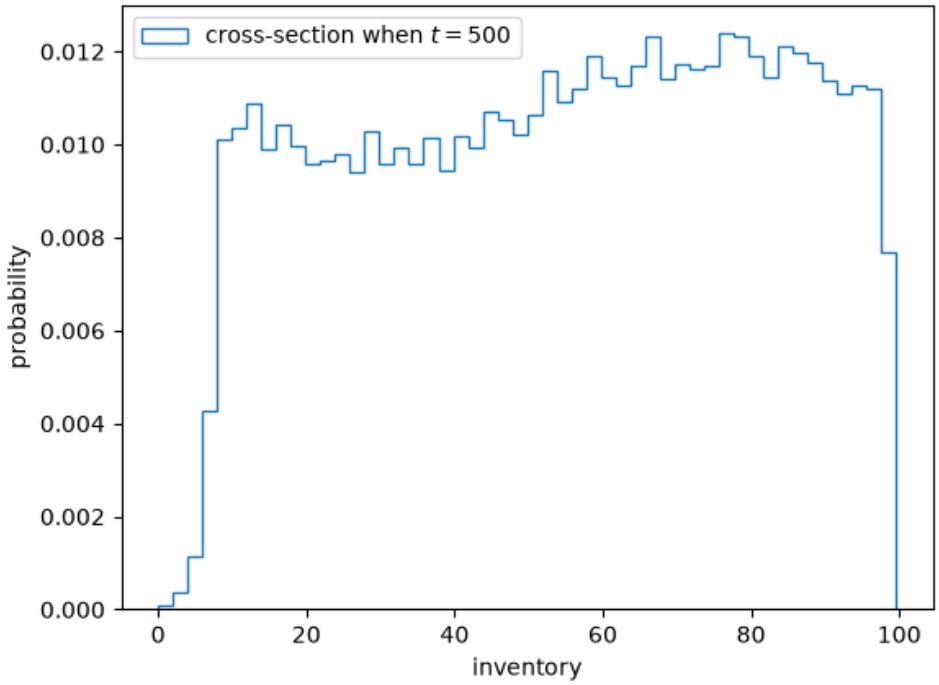
Let's run again to eliminate compile time.

```
start_time = time()
X_vec = project_cross_section(
    params, x_init, T, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 466.963291 ms
```

Here's a histogram of inventory levels at time T.

```
fig, ax = plt.subplots()
ax.hist(X_vec, bins=50,
        density=True,
        histtype='step',
        label=f'cross-section when $t = {T}$')
ax.set_xlabel('inventory')
ax.set_ylabel('probability')
ax.legend()
plt.show()
```



5.3.2 Compiling the outer loop

Now let's see if we can gain some speed by compiling the outer loop, which steps through the time dimension.

We will do this using `jax.jit` and a `fori_loop`, which is a compiler-ready version of a `for` loop provided by JAX.

```
def project_cross_section_fori(
        params: ModelParameters,
        x_init: jnp.ndarray,
        T: int,
        key: jnp.ndarray,
        num_firms: int = 50_000
    ) -> jnp.ndarray:

    s, S, μ, σ = params.s, params.S, params.μ, params.σ
    X = jnp.full((num_firms, ), x_init)

    # Define the function for each update
    def fori_update(t, loop_state):
        # Unpack
        X, key = loop_state
```

(continues on next page)

(continued from previous page)

```
        # Draw shocks using key
        key, subkey = random.split(key)
        Z = random.normal(subkey, shape=(num_firms,))
        D = jnp.exp(μ + σ * Z)
        # Update X
        X = jnp.where(X <= s,
                  jnp.maximum(S - D, 0),
                  jnp.maximum(X - D, 0))
        return X, key

    # Loop t from 0 to T, applying fori_update each time.
    initial_loop_state = X, key
    X, key = lax.fori_loop(0, T, fori_update, initial_loop_state)
    return X

# Compile taking T and num_firms as static (changes trigger recompile)
project_cross_section_fori = jax.jit(
    project_cross_section_fori, static_argnums=(2, 4))
```

Let's see how fast this runs with compile time.

```
start_time = time()
X_vec = project_cross_section_fori(
        params, x_init, T, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 485.568523 ms
```

And let's see how fast it runs without compile time.

```
start_time = time()
X_vec = project_cross_section_fori(
        params, x_init, T, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 10.133982 ms
```

Compared to the original version with a pure Python outer loop, we have produced a nontrivial speed gain.

This is due to the fact that we have compiled the entire sequence of operations.

5.4 Distribution dynamics

Next let's take a look at how the distribution sequence evolves over time.

We will go back to using ordinary Python `for` loops.

Here is code that repeatedly shifts the cross-section forward while recording the cross-section at the dates in `sample_dates`.

```
def shift_forward_and_sample(x_init, params, sample_dates,
                             key, num_firms=50_000, sim_length=750):
```

(continues on next page)

(continued from previous page)

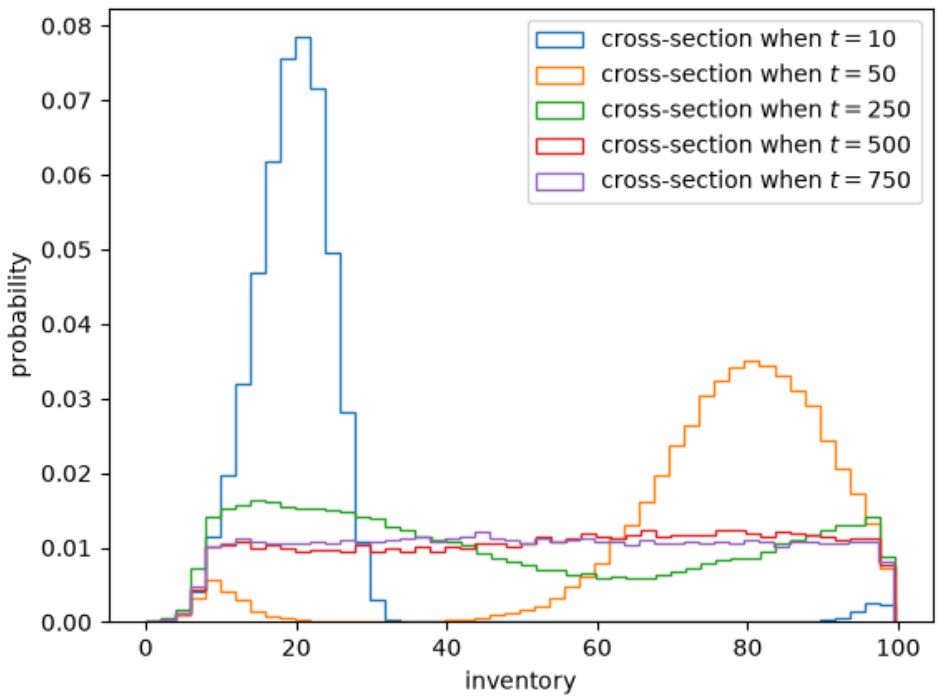
```
    X = res = jnp.full((num_firms, ), x_init)

    # Use for loop to update X and collect samples
    for i in range(sim_length):
        key, subkey = random.split(key)
        Z = random.normal(subkey, shape=(num_firms, ))
        D = jnp.exp(params.μ + params.σ * Z)

        X = update_cross_section(params, X, D)

        # draw a sample at the sample dates
        if (i+1 in sample_dates):
          res = jnp.vstack((res, X))

    return res[1:]
```

Let's test it

```
x_init = 50
num_firms = 10_000
sample_dates = 10, 50, 250, 500, 750
key = random.key(10)

X = shift_forward_and_sample(
    x_init, params, sample_dates, key).block_until_ready()
```

Let's plot the output.

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

for i, date in enumerate(sample_dates):
    ax.hist(X[i, :], bins=50,
            density=True,
            histtype='step',
            label=f'cross-section when $t = {date}$')

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



This model for inventory dynamics is asymptotically stationary, with a unique stationary distribution.

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

Intuitively, demand shocks repeatedly push inventories downward, while the threshold rule returns low-inventory firms to a common upper inventory level, so the process gradually forgets its initial condition.

Although we will not prove this here, we can see it in the simulation above.

By $t = 500$ or $t = 750$ the distributions are barely changing.

If you test a few different initial conditions, you will see that they do not affect long-run outcomes.

5.5 Restock frequency

As an exercise, let's study the probability that firms need to restock over a given time period.

In the exercise, we will

- set the starting stock level to $X_0 = 70$ and
- calculate the proportion of firms that need to order twice or more in the first 50 periods.

This proportion approximates the probability of the event when the sample size is large.

5.5.1 For loop version

We start with an easier `for` loop implementation

```
# Define a jitted function for each update
@jax.jit
def update_stock(n_restock, X, params, D):
    n_restock = jnp.where(X <= params.s,
                          n_restock + 1,
                          n_restock)
    X = jnp.where(X <= params.s,
                  jnp.maximum(params.S - D, 0),
                  jnp.maximum(X - D, 0))
    return n_restock, X

def compute_freq(params, key,
                 x_init=70,
                 sim_length=50,
                 num_firms=1_000_000):

    # Prepare initial arrays
    X = jnp.full((num_firms, ), x_init)

    # Stack the restock counter on top of the inventory
    n_restock = jnp.zeros((num_firms, ))

    # Use a for loop to perform the calculations on all states
    for i in range(sim_length):
        key, subkey = random.split(key)
        Z = random.normal(subkey, shape=(num_firms, ))
        D = jnp.exp(params.μ + params.σ * Z)
        n_restock, X = update_stock(
            n_restock, X, params, D)

    return jnp.mean(n_restock > 1, axis=0)
```

```
key = random.key(27)

start_time = time()
freq = compute_freq(params, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 842.149734 ms
```

We run the code again to get rid of compile time.

```
start_time = time()
freq = compute_freq(params, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 45.720577 ms
```

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

```
Frequency of at least two stock outs = 0.44729599356651306
```

Exercise 5.5.1

Write a `fori_loop` version of the last function. See if you can increase the speed while generating a similar answer.

Solution

Here is a `lax.fori_loop` version that JIT compiles the whole function

```
def compute_freq(params, key,
                 x_init=70,
                 sim_length=50,
                 num_firms=1_000_000):

    s, S, μ, σ = params.s, params.S, params.μ, params.σ
    # Prepare initial arrays
    X = jnp.full((num_firms, ), x_init)
    Z = random.normal(key, shape=(sim_length, num_firms))
    D = jnp.exp(μ + σ * Z)

    # Stack the restock counter on top of the inventory
    restock_count = jnp.zeros((num_firms, ))
    Xs = (X, restock_count)

    # Define the function for each update
    def update_cross_section(i, Xs):
        # Separate the inventory and restock counter
        x, restock_count = Xs[0], Xs[1]
        restock_count = jnp.where(x <= s,
                                  restock_count + 1,
                                  restock_count)
        x = jnp.where(x <= s,
                      jnp.maximum(S - D[i], 0),
                      jnp.maximum(x - D[i], 0))

        Xs = (x, restock_count)
        return Xs

    # Use lax.fori_loop to perform the calculations on all states
    X_final = lax.fori_loop(0, sim_length, update_cross_section, Xs)

    return jnp.mean(X_final[1] > 1)

# Compile taking sim_length and num_firms as static (changes trigger recompile)
compute_freq = jax.jit(compute_freq, static_argnums=(3, 4))
```

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

```
start_time = time()
freq = compute_freq(params, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 459.605694 ms
```

We run the code again to eliminate the compile time.

```
start_time = time()
freq = compute_freq(params, key).block_until_ready()
end_time = time()
print(f"Elapsed time: {(end_time - start_time) * 1000:.6f} ms")
```

```
Elapsed time: 8.201838 ms
```

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

```
Frequency of at least two stock outs = 0.4476909935474396
```

CHAPTER

SIX

KESTEN PROCESSES AND FIRM DYNAMICS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to JAX and Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

6.1 Overview

This lecture describes Kesten processes, which are an important class of stochastic processes, and an application of firm dynamics.

The lecture draws on an earlier QuantEcon lecture, which uses Numba to accelerate the computations.

In that earlier lecture you can find a more detailed discussion of the concepts involved.

This lecture focuses on implementing the same computations in JAX.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import quantecon as qe
import jax
import jax.numpy as jnp
from jax import random
from jax import lax
from typing import NamedTuple
from functools import partial
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:24:32 2026
+-----------------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+----------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |              ␣
 ↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |              ␣
 ↪     0 |
| N/A   35C    P8             14W /   70W |       0MiB /  15360MiB |      0%      ␣
 ↪Default |
|                                         |                        |              ␣
 ↪   N/A |
+-----------------------------------------+------------------------+----------------
 ↪-------+

+-----------------------------------------------------------------------------------------
 ↪-------+
| Processes:                                                                              ␣
 ↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
 ↪Memory |
|        ID   ID                                                               ␣
 ↪Usage      |
|=========================================================================================|
|  No running processes found                                                             ␣
 ↪       |
+-----------------------------------------------------------------------------------------
 ↪-------+
```

6.2 Kesten processes

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

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{6.1}$$

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.

6.2.1 Application: firm dynamics

In this section we apply Kesten process theory to the study of firm dynamics.

Gibrat's law

It was postulated many years ago by Robert Gibrat 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

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{6.2}$$

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

Subsequent empirical research has shown that this specification is not accurate, particularly for small firms.

However, we can get close to the data by modifying (6.2) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{6.3}$$

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

We now study the implications of this specification.

Heavy tails

If the conditions of the Kesten–Goldie Theorem are satisfied, then (6.3) implies that the firm size distribution will have Pareto tails.

This matches empirical findings across many data sets.

But there is another unrealistic aspect of the firm dynamics specified in (6.3) that we need to address: 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.

In this setting, firm dynamics can be expressed as

$$s_{t+1} = e_{t+1} \mathbb{1}\{s_t < \bar{s}\} + (a_{t+1} s_t + b_{t+1}) \mathbb{1}\{s_t \geq \bar{s}\} \tag{6.4}$$

The motivation behind and interpretation of *(6.2.4)* can be found in our earlier Kesten process lecture.

What can we say about dynamics?

Although (6.4) 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?

We can 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, we assume that each of a_t, b_t and e_t is lognormal.

Here's a class to store parameters:

```
class Firm(NamedTuple):
    μ_a:   float = -0.5
    σ_a:   float = 0.1
    μ_b:   float = 0.0
    σ_b:   float = 0.5
    μ_e:   float = 0.0
    σ_e:   float = 0.5
    s_bar: float = 1.0
```

Here's code to update a cross-section of firms according to the dynamics in *(6.2.4)*.

```
@jax.jit
def update_cross_section(s, a, b, e, firm):
    μ_a, σ_a, μ_b, σ_b, μ_e, σ_e, s_bar = firm
    s = jnp.where(s < s_bar, e, a * s + b)
    return s
```

Now we write a for loop that repeatedly calls this function, to push a cross-section of firms forward in time.

For sufficiently large `T`, the cross-section it returns (the cross-section at time `T`) corresponds to firm size distribution in (approximate) equilibrium.

```
def generate_cross_section(
        firm, M=500_000, T=500, s_init=1.0, seed=123
    ):

    μ_a, σ_a, μ_b, σ_b, μ_e, σ_e, s_bar = firm
    key = random.key(seed)

    # Initialize the cross-section to a common value
    s = jnp.full((M, ), s_init)

    # Perform updates on s for time t
    for t in range(T):
        key, *subkeys = random.split(key, 4)
        a = μ_a + σ_a * random.normal(subkeys[0], (M,))
        b = μ_b + σ_b * random.normal(subkeys[1], (M,))
        e = μ_e + σ_e * random.normal(subkeys[2], (M,))
        # Exponentiate shocks
        a, b, e = jax.tree.map(jnp.exp, (a, b, e))
        # Update the cross-section of firms
        s = update_cross_section(s, a, b, e, firm)

    return s
```

Let's try running the code and generating a cross-section.

```
firm = Firm()
with qe.Timer():
    data = generate_cross_section(firm).block_until_ready()
```

```
2.3979 seconds elapsed
```

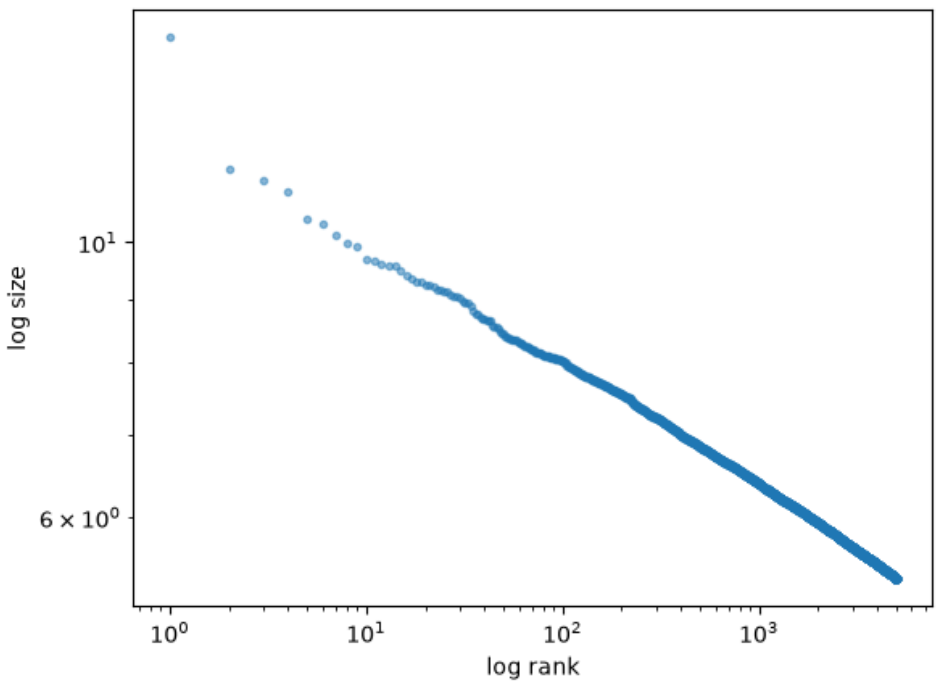
We run the function again so we can see the speed without compile time.

```
with qe.Timer():
    data = generate_cross_section(firm).block_until_ready()
```

```
0.9081 seconds elapsed
```

Let's produce the rank-size plot and check the distribution:

```
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.

Alternative implementation with `lax.fori_loop`

Although we JIT-compiled some of the code above, we did not JIT-compile the `for` loop.

Let's try squeezing out a bit more speed by

- replacing the `for` loop with `lax.fori_loop` and
- JIT-compiling the whole function.

Here is the `lax.fori_loop` version:

```
@partial(jax.jit, static_argnames=("T", "M"))
def generate_cross_section_lax(
        firm, T=500, M=500_000, s_init=1.0, seed=123
    ):

    μ_a, σ_a, μ_b, σ_b, μ_e, σ_e, s_bar = firm
    key = random.key(seed)

    # Initial cross section
    s = jnp.full((M, ), s_init)

    def update_cross_section(t, state):
        s, key = state
        key, *subkeys = jax.random.split(key, 4)
        # Generate current random draws
        a = μ_a + σ_a * random.normal(subkeys[0], (M,))
        b = μ_b + σ_b * random.normal(subkeys[1], (M,))
        e = μ_e + σ_e * random.normal(subkeys[2], (M,))
        # Exponentiate them
        a, b, e = jax.tree.map(jnp.exp, (a, b, e))
        # Update the cross-section of firms
        s = jnp.where(s < s_bar, e, a * s + b)
        new_state = s, key
        return new_state

    # Use fori_loop
    initial_state = s, key
    final_s, final_key = lax.fori_loop(
        0, T, update_cross_section, initial_state
    )
    return final_s
```

Let's see if we get any speed gain

```
with qe.Timer():
    data = generate_cross_section_lax(firm).block_until_ready()
```

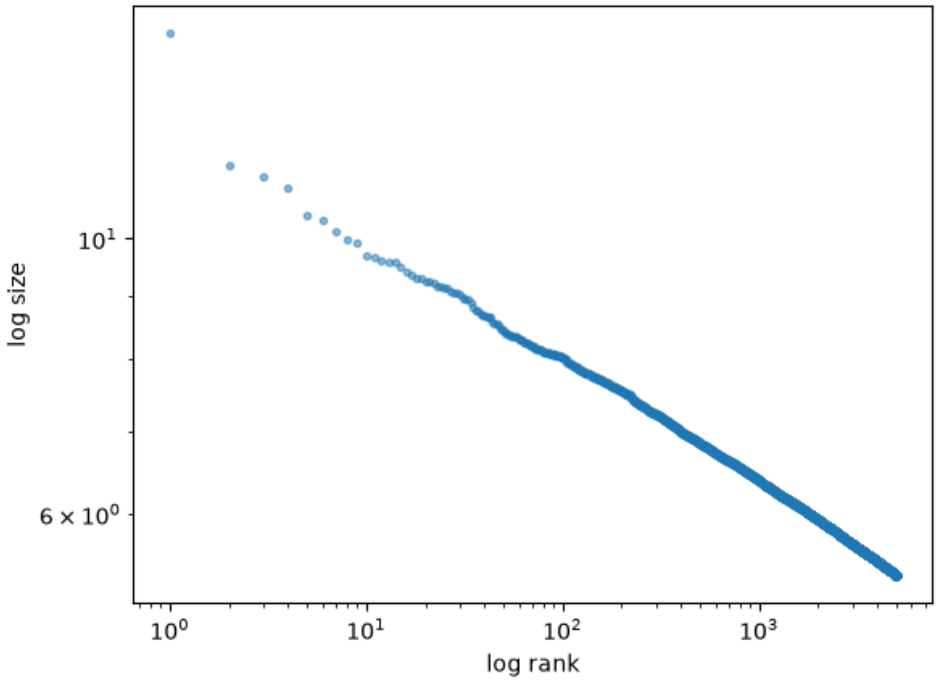
```
1.2641 seconds elapsed
```

```
with qe.Timer():
    data = generate_cross_section_lax(firm).block_until_ready()
```

```
0.0607 seconds elapsed
```

Here we produce the same 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()
```



6.3 Exercises

Exercise 6.3.1

Try writing an alternative version of `generate_cross_section_lax()` where the entire sequence of random draws is generated at once, so that all of `a`, `b`, and `e` are of shape `(T, M)`.

(The `update_cross_section()` function should not generate any random numbers.)

Does it improve the runtime?

What are the pros and cons of this approach?

Solution

```
@partial(jax.jit, static_argnames=("T", "M"))
def generate_cross_section_lax(
        firm, T=500, M=500_000, s_init=1.0, seed=123
    ):

    μ_a, σ_a, μ_b, σ_b, μ_e, σ_e, s_bar = firm
    key = random.key(seed)
    subkey_1, subkey_2, subkey_3 = random.split(key, 3)

    # Generate entire sequence of random draws
    a = μ_a + σ_a * random.normal(subkey_1, (T, M))
    b = μ_b + σ_b * random.normal(subkey_2, (T, M))
    e = μ_e + σ_e * random.normal(subkey_3, (T, M))
    # Exponentiate them
```

```
    a, b, e = jax.tree.map(jnp.exp, (a, b, e))
    # Initial cross section
    s = jnp.full((M, ), s_init)

    def update_cross_section(t, s):
        # Pull out the t-th cross-section of shocks
        a_t, b_t, e_t = a[t], b[t], e[t]
        s = jnp.where(s < s_bar, e_t, a_t * s + b_t)
        return s

    # Use lax.fori_loop to perform the calculations on all states
    s_final = lax.fori_loop(0, T, update_cross_section, s)
    return s_final
```

Here are the run times.

```
with qe.Timer():
    data = generate_cross_section_lax(firm).block_until_ready()
```

```
1.1489 seconds elapsed
```

```
with qe.Timer():
    data = generate_cross_section_lax(firm).block_until_ready()
```

```
0.0761 seconds elapsed
```

This method might or might not be faster.

In general, the relative speed will depend on the size of the cross-section and the length of the simulation paths.

However, this method is far more memory intensive.

It will fail when T and M become sufficiently large.

CHAPTER

SEVEN

WEALTH DISTRIBUTION DYNAMICS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In this lecture we examine wealth dynamics in large cross-section of agents who are subject to both

- idiosyncratic shocks, which affect labor income and returns, and
- an aggregate shock, which also impacts on labor income and returns

In most macroeconomic models savings and consumption are determined by optimization.

Here savings and consumption behavior is taken as given – you can plug in your favorite model to obtain savings behavior and then analyze distribution dynamics using the techniques described below.

One of our interests will be how different aspects of wealth dynamics – such as labor income and the rate of return on investments – feed into measures of inequality, such as the Gini coefficient.

Because the analysis below requires simulating large cross sections of households, it provides a natural setting for JAX's array-based programming model and accelerator support.

In addition to JAX and Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

We will use the following imports:

```
import numba
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import quantecon as qe
import jax
import jax.numpy as jnp
from time import time
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:31:44 2026
+-----------------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+----------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |              ␣
 ↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |              ␣
 ↪     0 |
| N/A   37C    P8             14W /   70W |       0MiB /  15360MiB |      0%      ␣
 ↪Default |
|                                         |                        |              ␣
 ↪   N/A |
+-----------------------------------------+------------------------+----------------
 ↪-------+

+-----------------------------------------------------------------------------------------
 ↪-------+
| Processes:                                                                           ␣
 ↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
 ↪Memory |
|        ID   ID                                                               ␣
 ↪Usage      |
|=========================================================================================|
|  No running processes found                                                          ␣
 ↪       |
+-----------------------------------------------------------------------------------------
 ↪-------+
```

7.1 Wealth dynamics

Wealth evolves as follows:

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

Here

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

There is an aggregate state process

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

that affects the interest rate and labor income.

In particular, the gross interest rates obey

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

while

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

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

(Each household receives their own idiosyncratic shocks.)

Regarding the savings function s, our default model will be

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{7.1}$$

where s_0 is a positive constant.

Thus,

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

7.2 Implementation

7.2.1 Numba implementation

Here's a function that collects parameters and useful constants

```
def create_wealth_model(w_hat=1.0,    # Savings parameter
                        s_0=0.75,     # Savings parameter
                        c_y=1.0,      # Labor income parameter
                        μ_y=1.0,      # Labor income parameter
                        σ_y=0.2,      # Labor income parameter
                        c_r=0.05,     # Rate of return parameter
                        μ_r=0.1,      # Rate of return parameter
                        σ_r=0.5,      # Rate of return parameter
                        a=0.5,        # Aggregate shock parameter
                        b=0.0,        # Aggregate shock parameter
                        σ_z=0.1):     # Aggregate shock parameter
    """
    Create a wealth model with given parameters.

    Return a tuple model = (household_params, aggregate_params), where
    household_params collects household information and aggregate_params
    collects information relevant to the aggregate shock process.

    """
    # Mean and variance of z process
    z_mean = b / (1 - a)
    z_var = σ_z**2 / (1 - a**2)
    exp_z_mean = np.exp(z_mean + z_var / 2)
    # Mean of R and y processes
    R_mean = c_r * exp_z_mean + np.exp(μ_r + σ_r**2 / 2)
```

(continues on next page)

(continued from previous page)

```
    y_mean = c_y * exp_z_mean + np.exp(μ_y + σ_y**2 / 2)
    # Test stability condition ensuring wealth does not diverge
    # to infinity.
    α = R_mean * s_0
    if α >= 1:
        raise ValueError("Stability condition failed.")
    # Pack values into tuples and return them
    household_params = (w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, y_mean)
    aggregate_params = (a, b, σ_z, z_mean, z_var)
    model = household_params, aggregate_params
    return model
```

Here's a function that generates the aggregate state process

```
@numba.jit
def generate_aggregate_state_sequence(aggregate_params, length=100):
    a, b, σ_z, z_mean, z_var = aggregate_params
    z = np.empty(length+1)
    z[0] = z_mean   # Initialize at z_mean
    for t in range(length):
        z[t+1] = a * z[t] + b + σ_z * np.random.randn()
    return z
```

Here's a function that updates household wealth by one period, taking the current value of the aggregate shock

```
@numba.jit
def update_wealth(household_params, w, z):
    """
    Generate w_{t+1} given w_t and z_{t+1}.
    """
    # Unpack
    w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, y_mean = household_params
    # Update wealth
    y = c_y * np.exp(z) + np.exp(μ_y + σ_y * np.random.randn())
    wp = y
    if w >= w_hat:
        R = c_r * np.exp(z) + np.exp(μ_r + σ_r * np.random.randn())
        wp += R * s_0 * w
    return wp
```

Here's a function to simulate the time series of wealth for an individual household

```
@numba.jit
def wealth_time_series(model, w_0, sim_length):
    """
    Generate a single time series of length sim_length for wealth given initial
    value w_0.  The function generates its own aggregate shock sequence.

    """
    # Unpack
    household_params, aggregate_params = model
    a, b, σ_z, z_mean, z_var = aggregate_params
    # Initialize and update
    z = generate_aggregate_state_sequence(aggregate_params,
                                          length=sim_length-1)
    w = np.empty(sim_length)
    w[0] = w_0
```

(continues on next page)

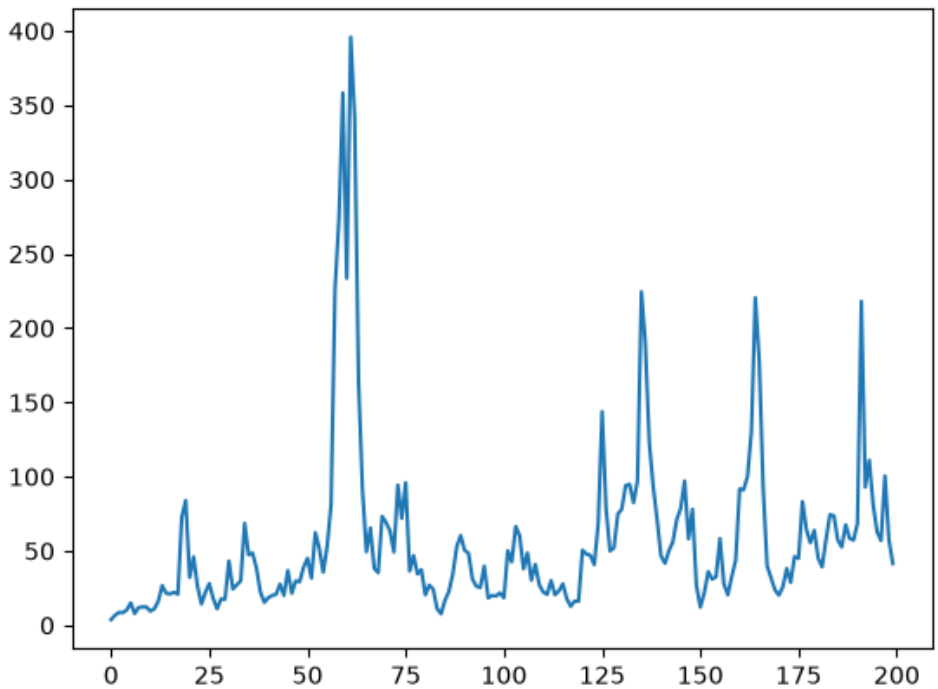
(continued from previous page)

```
    for t in range(sim_length-1):
        w[t+1] = update_wealth(household_params, w[t], z[t+1])
    return w
```

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

```
model = create_wealth_model()
household_params, aggregate_params = model
w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, y_mean = household_params
a, b, σ_z, z_mean, z_var = aggregate_params
ts_length = 200
w = wealth_time_series(model, y_mean, ts_length)
```

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



Notice the large spikes in wealth over time.

Such spikes are related to heavy tails in the wealth distribution, which we discuss below.

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

Note the use of parallelization to speed up computation.

```
@numba.jit(parallel=True)
def update_cross_section(model, w_distribution, z_sequence):
    """
    Shifts a cross-section of households forward in time

    Takes
```

(continues on next page)

(continued from previous page)

```
        * a current distribution of wealth values as w_distribution and
        * an aggregate state sequence z_sequence

    The first entry of z_sequence is the initial aggregate state.

    It then updates each w_t in w_distribution to w_{t+j}, where
    j = len(z_sequence) - 1.

    Returns the new distribution.

    """
    # Unpack
    household_params, aggregate_params = model

    num_households = len(w_distribution)
    new_distribution = np.empty_like(w_distribution)
    z = z_sequence

    # Update each household
    for i in numba.prange(num_households):
        w = w_distribution[i]
        for t in range(1, len(z)):
            w = update_wealth(household_params, w, z[t])
        new_distribution[i] = w
    return new_distribution
```

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

Let's see how long it takes to shift a large cross-section of households forward 200 periods

```
sim_length = 200
num_households = 10_000_000
ψ_0 = np.full(num_households, y_mean)  # Initial distribution
z_sequence = generate_aggregate_state_sequence(aggregate_params,
                                               length=sim_length)
print("Generating cross-section using Numba")
start = time()
ψ_star = update_cross_section(model, ψ_0, z_sequence)
numba_with_compile = time() - start
print(f"Generated cross-section in {numba_with_compile} seconds.\n")
```

```
Generating cross-section using Numba
Generated cross-section in 40.004812240600586 seconds.
```

We run it again to eliminate compile time.

```
start = time()
ψ_star = update_cross_section(model, ψ_0, z_sequence)
numba_without_compile = time() - start
print(f"Generated cross-section in {numba_without_compile} seconds.\n")
```

```
Generated cross-section in 40.23064661026001 seconds.
```

7.2.2 JAX implementation

Let's redo some of the preceding calculations using JAX and see how execution speed compares

```
def update_cross_section_jax(model, w_distribution, z_sequence, key):
    """
    Shifts a cross-section of households forward in time

    Takes

        * a current distribution of wealth values as w_distribution and
        * an aggregate state sequence z_sequence

    The first entry of z_sequence is the initial aggregate state.

    It then updates each w_t in w_distribution to w_{t+j}, where
    j = len(z_sequence) - 1.

    Returns the new distribution.

    """
    # Unpack, simplify names
    household_params, aggregate_params = model
    w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, y_mean = household_params
    w = w_distribution
    n = len(w)

    # Update wealth
    for z in z_sequence[1:]:
        key, subkey = jax.random.split(key)
        U = jax.random.normal(subkey, (2, n))
        y = c_y * jnp.exp(z) + jnp.exp(μ_y + σ_y * U[0, :])
        R = c_r * jnp.exp(z) + jnp.exp(μ_r + σ_r * U[1, :])
        w = y + jnp.where(w < w_hat, 0.0, R * s_0 * w)

    return w
```

Let's see how long it takes to shift the cross-section of households forward using JAX

```
sim_length = 200
num_households = 10_000_000
ψ_0 = jnp.full(num_households, y_mean)  # Initial distribution
z_sequence = generate_aggregate_state_sequence(aggregate_params,
                                               length=sim_length)
z_sequence = jnp.array(z_sequence)
```

```
print("Generating cross-section using JAX")
key = jax.random.key(1234)
start = time()
ψ_star = update_cross_section_jax(model, ψ_0, z_sequence, key).block_until_ready()
jax_with_compile = time() - start
print(f"Generated cross-section in {jax_with_compile} seconds.\n")
```

```
Generating cross-section using JAX
Generated cross-section in 2.459756374359131 seconds.
```

```
print("Repeating without compile time.")
key = jax.random.key(1234)
start = time()
ψ_star = update_cross_section_jax(model, ψ_0, z_sequence, key).block_until_ready()
jax_without_compile = time() - start
print(f"Generated cross-section in {jax_without_compile} seconds")
```

```
Repeating without compile time.
Generated cross-section in 1.3223481178283691 seconds
```

And let's see how long it takes if we compile the loop.

```
def update_cross_section_jax_compiled(model,
                                      w_distribution,
                                      w_size,
                                      z_sequence,
                                      key):
    """
    Shifts a cross-section of households forward in time

    Takes

        * a current distribution of wealth values as w_distribution and
        * an aggregate state sequence z_sequence

    The first entry of z_sequence is the initial aggregate state.

    It then updates each w_t in w_distribution to w_{t+j}, where
    j = len(z_sequence) - 1.

    Returns the new distribution.

    """
    # Unpack, simplify names
    household_params, aggregate_params = model
    w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, y_mean = household_params
    w = w_distribution
    n = len(w)
    z = z_sequence[1:]
    sim_length = len(z)

    def body_function(t, state):
        key, w = state
        key, subkey = jax.random.split(key)
        U = jax.random.normal(subkey, (2, n))
        y = c_y * jnp.exp(z[t]) + jnp.exp(μ_y + σ_y * U[0, :])
        R = c_r * jnp.exp(z[t]) + jnp.exp(μ_r + σ_r * U[1, :])
        w = y + jnp.where(w < w_hat, 0.0, R * s_0 * w)
        return key, w

    key, w = jax.lax.fori_loop(0, sim_length, body_function, (key, w))
    return w
```

```
update_cross_section_jax_compiled = jax.jit(
        update_cross_section_jax_compiled, static_argnums=(2,)
)
```

```
print("Generating cross-section using JAX with compiled loop")
key = jax.random.key(1234)
start = time()
ψ_star = update_cross_section_jax_compiled(
        model, ψ_0, num_households, z_sequence, key
).block_until_ready()
jax_fori_with_compile = time() - start
print(f"Generated cross-section in {jax_fori_with_compile} seconds.\n")
```

```
Generating cross-section using JAX with compiled loop
Generated cross-section in 0.8133029937744141 seconds.
```

```
print("Repeating without compile time")
key = jax.random.key(1234)
start = time()
ψ_star = update_cross_section_jax_compiled(
        model, ψ_0, num_households, z_sequence, key
).block_until_ready()
jax_fori_without_compile = time() - start
print(f"Generated cross-section in {jax_fori_without_compile} seconds")
```

```
Repeating without compile time
Generated cross-section in 0.1720719337463379 seconds
```

```
print(f"JAX is {numba_without_compile/jax_fori_without_compile:.4f} times faster.\n")
```

```
JAX is 233.8013 times faster.
```

7.2.3 Pareto tails

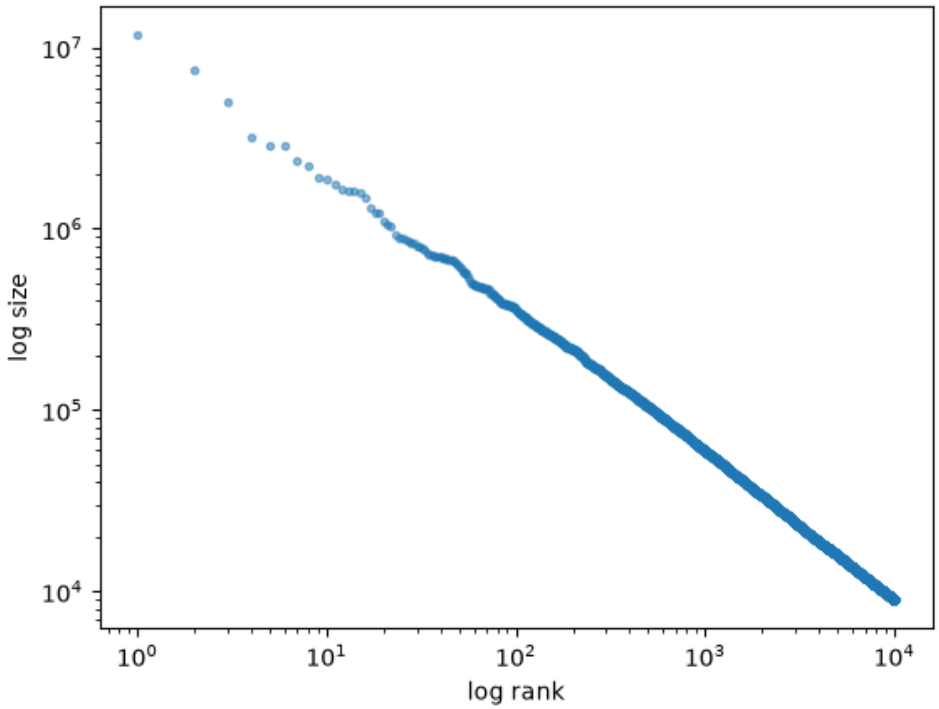
In most countries, the cross-sectional distribution of wealth exhibits a Pareto tail (power law).

Let's see if our model can replicate this stylized fact by running a simulation that generates a cross-section of wealth and generating a suitable rank-size plot.

We will use the function `rank_size` from `quantecon` library.

In the limit, data that obeys a power law generates a straight line.

```
model = create_wealth_model()
key = jax.random.key(1234)
ψ_star = update_cross_section_jax_compiled(
        model, ψ_0, num_households, z_sequence, key
)
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()
```



7.2.4 Lorenz curves and Gini coefficients

To study the impact of parameters on inequality, we examine Lorenz curves and the Gini coefficients at different parameters.

QuantEcon provides functions to compute Lorenz curves and Gini coefficients that are accelerated using Numba.

Here we provide JAX-based functions that do the same job and are faster for large data sets on parallel hardware.

Lorenz curve

Recall that, for sorted data $w_1, \dots, w_n$, the Lorenz curve generates data points $(x_i, y_i)_{i=0}^n$ according to

$$x_0 = y_0 = 0 \qquad \text{and, for } i \geq 1, \quad x_i = \frac{i}{n}, \qquad y_i = \frac{\sum_{j \leq i} w_j}{\sum_{j \leq n} w_j}$$

```
def _lorenz_curve_jax(w, w_size):
    n = w.shape[0]
    w = jnp.sort(w)
    x = jnp.arange(n + 1) / n
    s = jnp.concatenate((jnp.zeros(1), jnp.cumsum(w)))
    y = s / s[n]
    return x, y

lorenz_curve_jax = jax.jit(_lorenz_curve_jax, static_argnums=(1,))
```

Let's test

```
sim_length = 200
num_households = 1_000_000
ψ_0 = jnp.full(num_households, y_mean)  # Initial distribution
z_sequence = generate_aggregate_state_sequence(aggregate_params,
                                               length=sim_length)
z_sequence = jnp.array(z_sequence)
```

```
key = jax.random.key(1234)
ψ_star = update_cross_section_jax_compiled(
        model, ψ_0, num_households, z_sequence, key
)
```

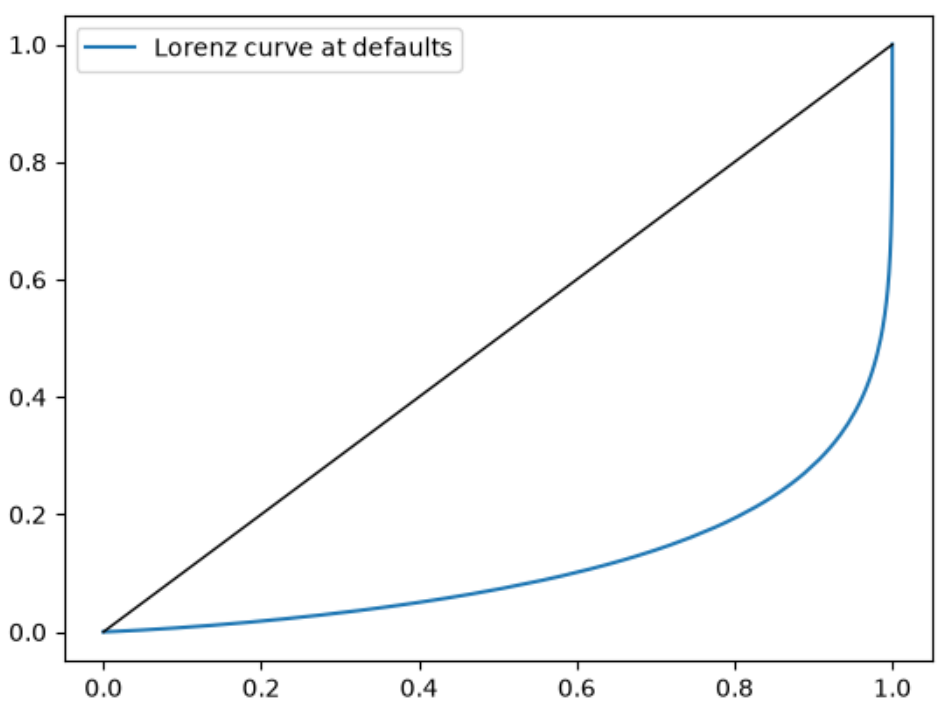
```
%time _ = lorenz_curve_jax(ψ_star, num_households)
```

```
CPU times: user 345 ms, sys: 16 ms, total: 361 ms
Wall time: 150 ms
```

```
# Now time it without compile time
%time x, y = lorenz_curve_jax(ψ_star, num_households)
```

```
CPU times: user 0 ns, sys: 583 µs, total: 583 µs
Wall time: 345 µs
```

```
fig, ax = plt.subplots()
ax.plot(x, y, label="Lorenz curve at defaults")
ax.plot(x, x, 'k-', lw=1)
ax.legend()
plt.show()
```



Gini Coefficient

Recall that, for sorted data $w_1, \dots, w_n$, the Gini coefficient takes the form

$$G := \frac{\sum_{i=1}^n \sum_{j=1}^n |w_j - w_i|}{2n \sum_{i=1}^n w_i}. \tag{7.2}$$

Here's a function that computes the Gini coefficient using vectorization.

```
def _gini_jax(w, w_size):
    w_1 = jnp.reshape(w, (w_size, 1))
    w_2 = jnp.reshape(w, (1, w_size))
    g_sum = jnp.sum(jnp.abs(w_1 - w_2))
    return g_sum / (2 * w_size * jnp.sum(w))

gini_jax = jax.jit(_gini_jax, static_argnums=(1,))
```

```
%time gini = gini_jax(ψ_star, num_households).block_until_ready()
```

```
CPU times: user 273 ms, sys: 6 ms, total: 279 ms
Wall time: 6.74 s
```

```
# Now time it without compilation
%time gini = gini_jax(ψ_star, num_households).block_until_ready()
```

```
CPU times: user 3.37 ms, sys: 1 ms, total: 4.37 ms
Wall time: 6.49 s
```

```
gini
```

```
Array(0.77136517, dtype=float32)
```

7.3 Exercises

Exercise 7.3.1

In this exercise, write an alternative version of `gini_jax` that uses `vmap` instead of reshaping and broadcasting.

Test with the same array to see if you can obtain the same output

Solution

Here's one solution:

```
@jax.jit
def gini_jax_vmap(w):

    def _inner_sum(x):
        return jnp.sum(jnp.abs(x - w))

    inner_sum = jax.vmap(_inner_sum)
```

```
    full_sum = jnp.sum(inner_sum(w))
    return full_sum / (2 * len(w) * jnp.sum(w))
```

```
%time gini = gini_jax_vmap(ψ_star).block_until_ready()
```

```
CPU times: user 183 ms, sys: 2 ms, total: 185 ms
Wall time: 6.6 s
```

```
# Now time it without compile time
%time gini = gini_jax_vmap(ψ_star).block_until_ready()
```

```
CPU times: user 2.34 ms, sys: 1.99 ms, total: 4.34 ms
Wall time: 6.54 s
```

```
gini
```

```
Array(0.77136517, dtype=float32)
```

Exercise 7.3.2

In this exercise we investigate how the parameters determining the rate of return on assets and labor income shape inequality.

In doing so we recall that

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

while

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

Investigate how the Lorenz curves and the Gini coefficient associated with the wealth distribution change as return to savings varies.

In particular, plot Lorenz curves for the following three different values of μ_r

```
μ_r_vals = (0.0, 0.025, 0.05)
```

Use the following as your initial cross-sectional distribution

```
num_households = 1_000_000
ψ_0 = jnp.full(num_households, y_mean)  # Initial distribution
```

Once you have done that, plot the Gini coefficients as well.

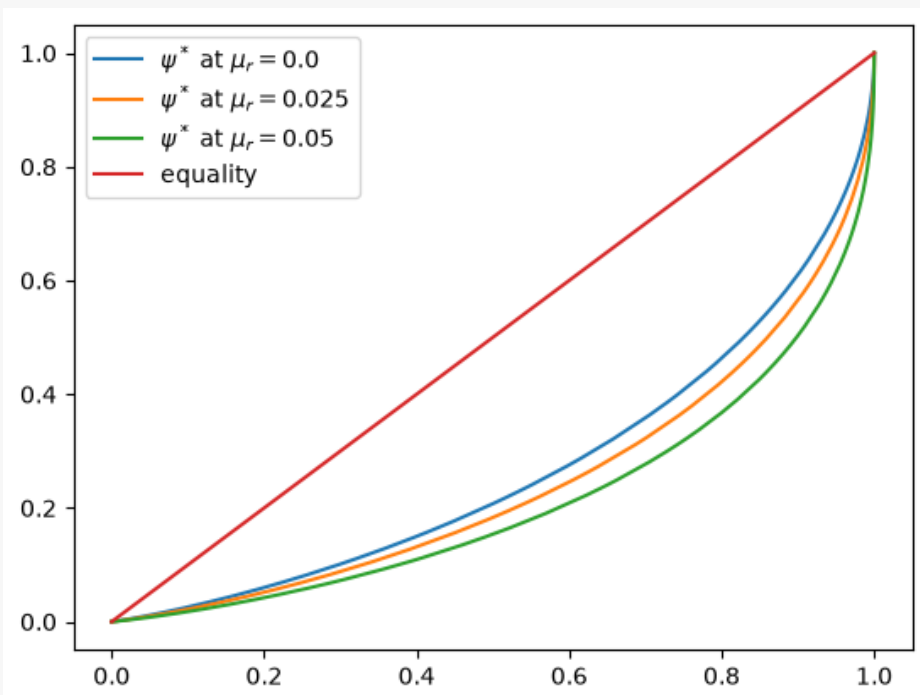
Do the outcomes match your intuition?

Solution

Here is one solution

In this and the following parameter comparisons, we reuse the same random seed across parameter values so that differences in the curves mainly reflect parameter changes rather than different random draws.

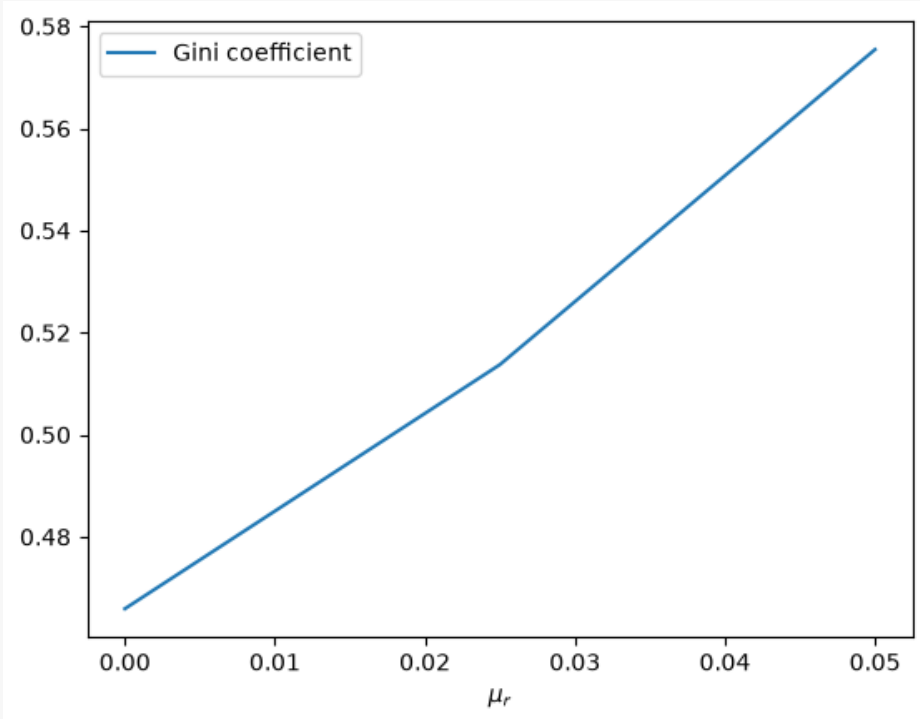
```
key = jax.random.key(1234)
fig, ax = plt.subplots()
gini_vals = []
for μ_r in μ_r_vals:
    model = create_wealth_model(μ_r=μ_r)
    ψ_star = update_cross_section_jax_compiled(
            model, ψ_0, num_households, z_sequence, key
    )
    x, y = lorenz_curve_jax(ψ_star, num_households)
    g = gini_jax(ψ_star, num_households)
    ax.plot(x, y, label=f'$\psi^*$ at $\mu_r = {μ_r:0.2}$')
    gini_vals.append(g)
ax.plot(x, x, label='equality')
ax.legend(loc="upper left")
plt.show()
```



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

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()
```



As expected, inequality increases as returns on financial income rise.

Exercise 7.3.3

Now investigate what happens when we change the volatility term σ_r in financial returns.

Use the same initial condition as before and the sequence

```
σ_r_vals = (0.35, 0.45, 0.52)
```

To isolate the role of volatility, set $\mu_r = -\sigma_r^2/2$ at each σ_r.

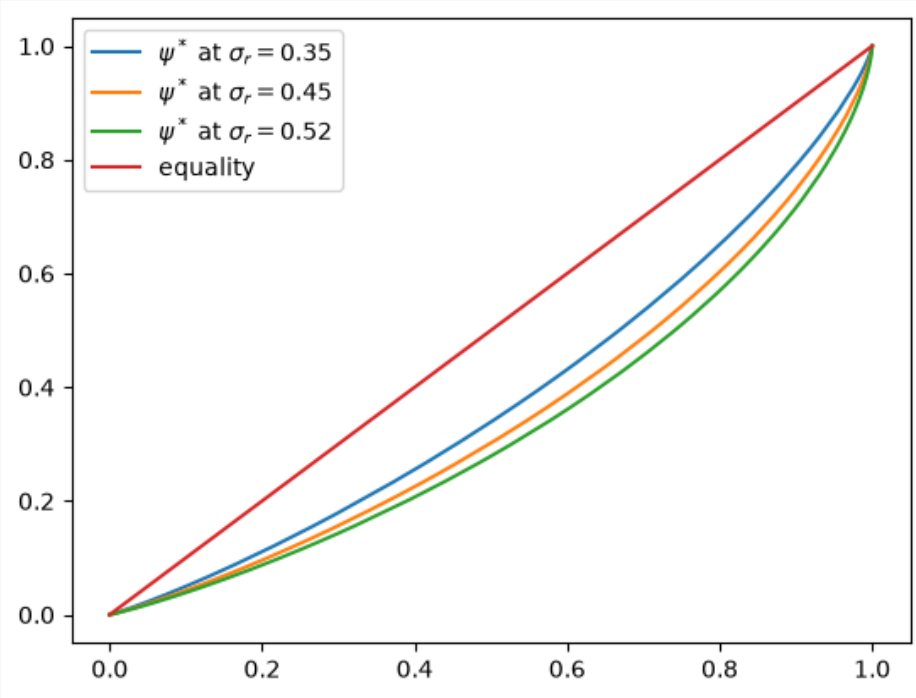
(This holds the mean of the idiosyncratic term $\exp(\mu_r + \sigma_r \xi)$ constant.)

Solution

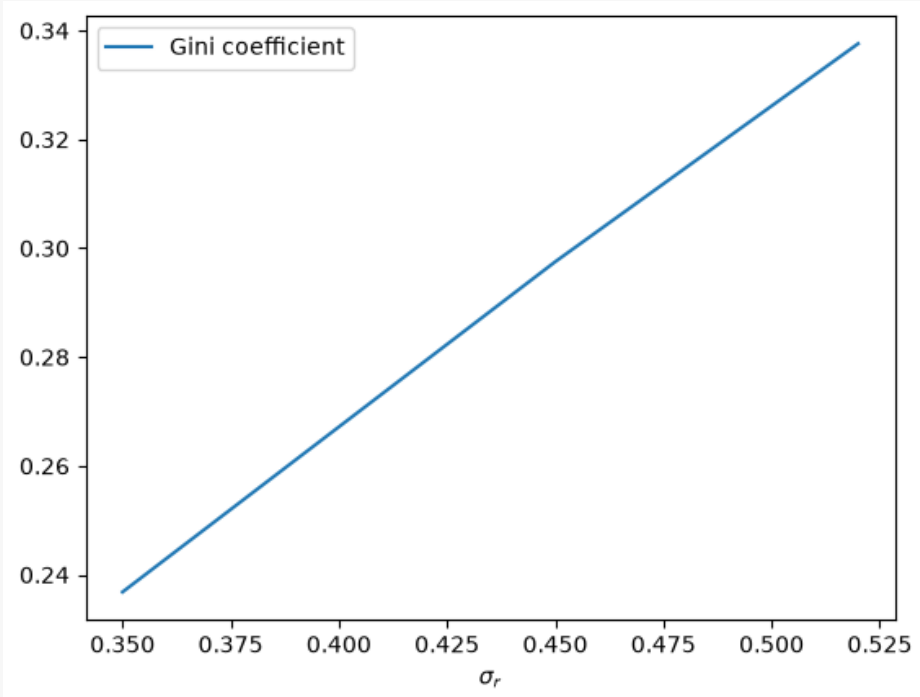
Here's one solution

```
key = jax.random.key(1234)
fig, ax = plt.subplots()

gini_vals = []
for σ_r in σ_r_vals:
    model = create_wealth_model(σ_r=σ_r, μ_r=(-σ_r**2/2))
    ψ_star = update_cross_section_jax_compiled(
            model, ψ_0, num_households, z_sequence, key
    )
    x, y = lorenz_curve_jax(ψ_star, num_households)
    g = gini_jax(ψ_star, num_households)
    ax.plot(x, y, label=f'$\psi^*$ at $\sigma_r = {σ_r:0.2}$')
```

```
    gini_vals.append(g)
ax.plot(x, x, label='equality')
ax.legend(loc="upper left")
plt.show()
```




```
fig, ax = plt.subplots()
ax.plot(σ_r_vals, gini_vals, label='Gini coefficient')
ax.set_xlabel("$\sigma_r$")
ax.legend()
plt.show()
```



Exercise 7.3.4

In this exercise, examine which has more impact on inequality:

- a 5% rise in volatility of the rate of return,
- or a 5% rise in volatility of labor income.

Test this by

1. Shifting σ_r up 5% from the baseline and plotting the Lorenz curve
2. Shifting σ_y up 5% from the baseline and plotting the Lorenz curve

Plot both on the same figure and examine the result.

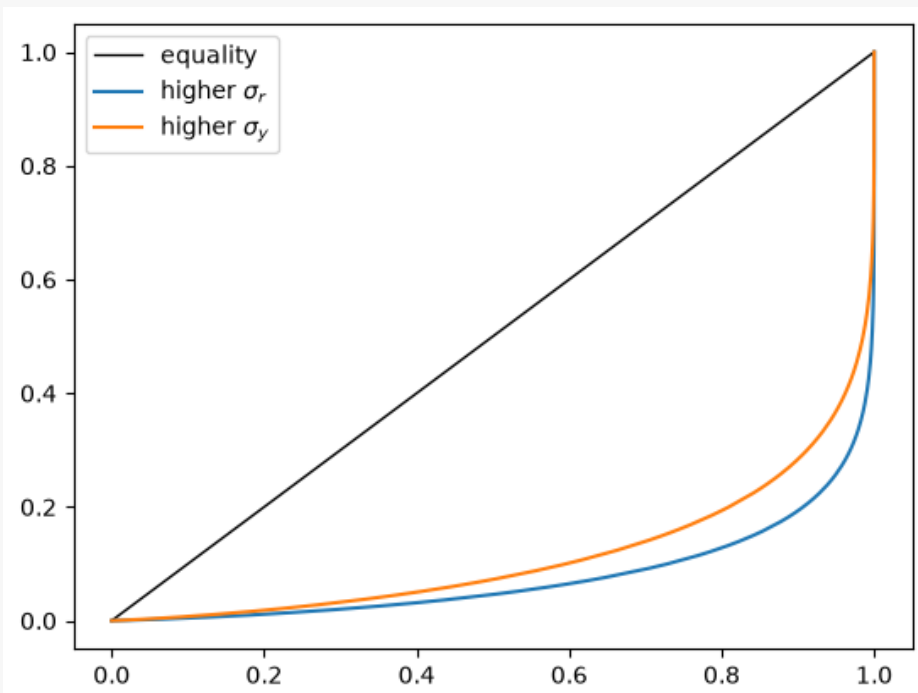
Solution

Here's one solution.

It shows that increasing volatility in financial income has a greater effect

```
model = create_wealth_model()
household_params, aggregate_params = model
w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, y_mean = household_params
σ_r_default = σ_r
σ_y_default = σ_y

ψ_star = update_cross_section_jax_compiled(
            model, ψ_0, num_households, z_sequence, key
)
x_default, y_default = lorenz_curve_jax(ψ_star, num_households)
```

```
model = create_wealth_model(σ_r=(1.05 * σ_r_default))
ψ_star = update_cross_section_jax_compiled(
            model, ψ_0, num_households, z_sequence, key
)
x_financial, y_financial = lorenz_curve_jax(ψ_star, num_households)

model = create_wealth_model(σ_y=(1.05 * σ_y_default))
ψ_star = update_cross_section_jax_compiled(
            model, ψ_0, num_households, z_sequence, key
)
x_labor, y_labor = lorenz_curve_jax(ψ_star, num_households)

fig, ax = plt.subplots()
ax.plot(x_default, x_default, 'k-', lw=1, label='equality')
ax.plot(x_financial, y_financial, label=r'higher $\sigma_r$')
ax.plot(x_labor, y_labor, label=r'higher $\sigma_y$')
ax.legend()
plt.show()
```



Part III

Asset Pricing

CHAPTER

EIGHT

ASSET PRICING: THE LUCAS ASSET PRICING MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

8.1 Overview

An asset is a claim on a stream of prospective payments.

What is the correct price to pay for such a claim?

The asset pricing model of Lucas [Lucas, 1978] attempts to answer this question in an equilibrium setting with risk-averse agents.

Lucas' model provides a beautiful illustration of model building in general and equilibrium pricing in competitive models in particular.

In this lecture we work through the Lucas model and show where the fundamental asset pricing equation comes from.

We'll write code in both Numba and JAX.

Since the model is relatively small, the speed gain from using JAX is not as large as it is in some other lectures.

Nonetheless, the gain is nontrivial.

Let's start with installing `quantecon` package and some imports:

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

```
Requirement already satisfied: quantecon in /home/runner/miniconda3/envs/quantecon/
↪lib/python3.13/site-packages (0.13.0)
Requirement already satisfied: numba>=0.56.0 in /home/runner/miniconda3/envs/
↪quantecon/lib/python3.13/site-packages (from quantecon) (0.65.1)
Requirement already satisfied: numpy>=1.17.0 in /home/runner/miniconda3/envs/
↪quantecon/lib/python3.13/site-packages (from quantecon) (2.4.6)
Requirement already satisfied: scipy>=1.5.0 in /home/runner/miniconda3/envs/
```

(continues on next page)

(continued from previous page)

```
↪quantecon/lib/python3.13/site-packages (from quantecon) (1.18.0)
Requirement already satisfied: llvmlite<0.48,>=0.47.0dev0 in /home/runner/
↪miniconda3/envs/quantecon/lib/python3.13/site-packages (from numba>=0.56.0->
↪quantecon) (0.47.0)
```

```
import jax.numpy as jnp
import jax
import numpy as np
import numba
import quantecon as qe
import matplotlib.pyplot as plt
```

8.2 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.

8.2.1 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{8.1}$$

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.

8.2.2 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 (8.1) 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

$$c + \pi' p(y) \leq \pi y + \pi p(y) \tag{8.2}$$

We can invoke the fact that utility is increasing to claim equality in (8.2) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\} \tag{8.3}$$

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 (8.3) 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 (8.3) 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

$$p(y) = \beta \int \frac{u'[G(y,z)]}{u'(y)} [G(y,z) + p(G(y,z))] \phi(dz) \tag{8.4}$$

In sequential rather than functional notation, we can also write this as

$$p_t = \mathbb{E}_t \left[\beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \tag{8.5}$$

This is the famous consumption-based asset pricing equation.

Before discussing it further we want to solve out for prices.

8.2.3 Solving the model

Equation (8.4) 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

$$f(y) := u'(y) p(y) \tag{8.6}$$

so that (8.4) becomes

$$f(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{8.7}$$

Here $h(y) := \beta \int u'[G(y,z)] G(y,z) \phi(dz)$ is a function that depends only on the primitives.

Equation (8.7) is a functional equation in f.

The plan is to solve out for f and convert back to p via (8.6).

To solve (8.7) 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

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{8.8}$$

In what follows, we refer to T as the Lucas operator.

The reason we do this is that a solution to (8.7) 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 $C_b(\mathbb{R}_+)$ 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 $C_b(\mathbb{R}_+)$.
2. For any $f \in C_b(\mathbb{R}_+)$, 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

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \qquad \forall\, f, g \in C_b(\mathbb{R}_+) \tag{8.9}$$

Here $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$.

To see that (8.9) is valid, pick any $f, g \in C_b(\mathbb{R}_+)$ 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 (8.9) with $\alpha := \beta$.

8.3 Computation

The preceding discussion tells us that we can compute f^* by picking any arbitrary $f \in C_b(\mathbb{R}_+)$ 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'll use Monte Carlo to compute the integral

$$\int f[G(y,z)]\phi(dz)$$

Monte Carlo is not always the fastest method for computing low-dimensional integrals, but it is extremely flexible (for example, it's straightforward to change the underlying state process).

8.3.1 Numba code

Let's start with code using NumPy / Numba (and then compare it to code using JAX).

We create a function that returns tuples containing parameters and arrays needed for computation.

```
def create_lucas_tree_model(
    γ=2,      # CRRA utility parameter
    β=0.95,   # Discount factor
    α=0.90,   # Correlation coefficient
    σ=0.1,    # Volatility coefficient
    grid_size=500,
    draw_size=1_000,
    seed=11,
):
    # Set the grid interval to contain most of the mass of the
    # stationary distribution of the consumption endowment
    ssd = σ / np.sqrt(1 - α**2)
    grid_min, grid_max = np.exp(-4 * ssd), np.exp(4 * ssd)
    grid = np.linspace(grid_min, grid_max, grid_size)
    # Set up distribution for shocks
    np.random.seed(seed)
    draws = np.random.lognormal(mean=0, sigma=σ, size=500)
    # And the vector h
    h = np.empty(grid_size)
    for i, y in enumerate(grid):
        h[i] = β * np.mean((y**α * draws) ** (1 - γ))
    # Pack and return
    params = γ, β, α, σ
    arrays = grid, draws, h
    return params, arrays
```

Here's a Numba-jitted version of the Lucas operator

```
@numba.jit
def T(params, arrays, f):
    """
    The Lucas pricing operator.
    """
    # Unpack
    γ, β, α, σ = params
    grid, draws, h = arrays
    # Turn f into a function
    Af = lambda x: np.interp(x, grid, f)
    # Compute Tf and return
    Tf = np.empty_like(f)
    # Apply the T operator to f using Monte Carlo integration
    for i in range(len(grid)):
        y = grid[i]
        Tf[i] = h[i] + β * np.mean(Af(y**α * draws))
    return Tf
```

To solve the model, we write a function that iterates using the Lucas operator to find the fixed point.

```
def solve_model(params, arrays, tol=1e-6, max_iter=500):
    """
    Compute the equilibrium price function.
    """
```

(continues on next page)

(continued from previous page)

```
    # Unpack
    γ, β, α, σ = params
    grid, draws, h = arrays
    # Set up and loop
    i = 0
    f = np.ones_like(grid)  # Initial guess of f
    error = tol + 1
    while error > tol and i < max_iter:
        Tf = T(params, arrays, f)
        error = np.max(np.abs(Tf - f))
        f = Tf
        i += 1
    price = f * grid**γ  # Back out price vector
    return price
```

Let's solve the model and plot the resulting price function

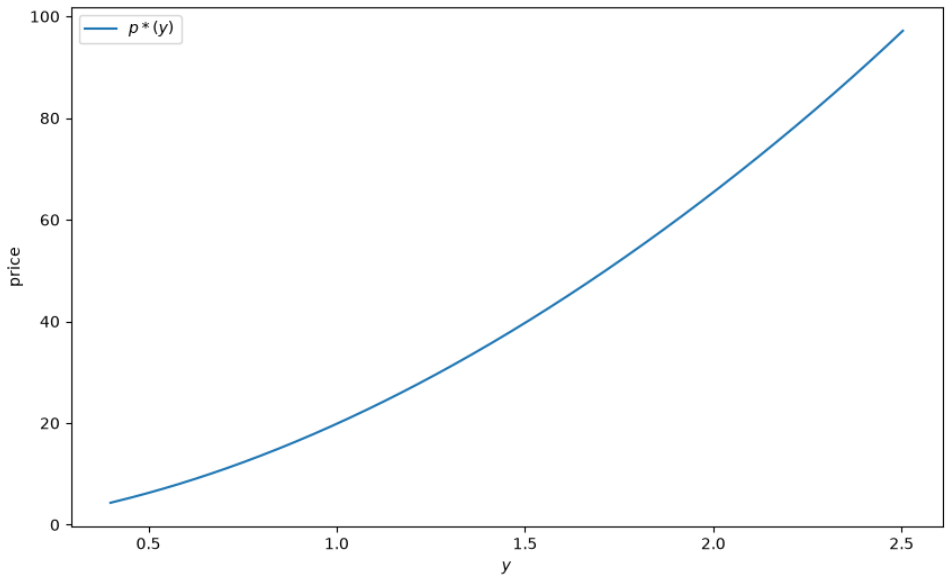
```
params, arrays = create_lucas_tree_model()
γ, β, α, σ = params
grid, draws, h = arrays

# Solve once to compile
with qe.Timer() as numba_with_compile_time:
    price_vals = solve_model(params, arrays)
```

```
5.6516 seconds elapsed
```

```
# Now time execution without compile time
with qe.Timer() as numba_without_compile_time:
    price_vals = solve_model(params, arrays)
```

```
4.1556 seconds elapsed
```

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(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).

8.3.2 JAX code

Here's a JAX version of the same problem.

```
def create_lucas_tree_model(
    γ=2,      # CRRA utility parameter
    β=0.95,   # Discount factor
    α=0.90,   # Correlation coefficient
    σ=0.1,    # Volatility coefficient
    grid_size=500,
    draw_size=1_000,
    seed=11,
):
    # Set the grid interval to contain most of the mass of the
    # stationary distribution of the consumption endowment
    ssd = σ / jnp.sqrt(1 - α**2)
    grid_min, grid_max = jnp.exp(-4 * ssd), jnp.exp(4 * ssd)
    grid = jnp.linspace(grid_min, grid_max, grid_size)

    # Set up distribution for shocks
    key = jax.random.key(seed)
    draws = jax.random.lognormal(key, σ, shape=(draw_size,))
    grid_reshaped = grid.reshape((grid_size, 1))
    draws_reshaped = draws.reshape((-1, draw_size))
    h = β * jnp.mean((grid_reshaped**α * draws_reshaped) ** (1 - γ), axis=1)
    params = γ, β, α, σ
    arrays = grid, draws, h
    return params, arrays
```

We'll use the following function to simultaneously compute the expectation

$$\int f[G(y,z)]\phi(dz)$$

over all y in the grid, under the current specifications.

```
@jax.jit
def compute_expectation(y, α, draws, grid, f):
    return jnp.mean(jnp.interp(y**α * draws, grid, f))


# Vectorize over y
compute_expectation = jax.vmap(
    compute_expectation, in_axes=(0, None, None, None, None)
)
```

Here's the Lucas operator

```
@jax.jit
def T(params, arrays, f):
    """
    The Lucas operator

    """
    grid, draws, h = arrays
    γ, β, α, σ = params
    mci = compute_expectation(grid, α, draws, grid, f)
    return h + β * mci
```

We'll use successive approximation to compute the fixed point.

```
def successive_approx_jax(
    T,          # Operator (callable)
    x_0,        # Initial condition
    tol=1e-6,   # Error tolerance
    max_iter=10_000,
):  # Max iteration bound
    def body_fun(k_x_err):
        k, x, error = k_x_err
        x_new = T(x)
        error = jnp.max(jnp.abs(x_new - x))
        return k + 1, x_new, error

    def cond_fun(k_x_err):
        k, x, error = k_x_err
        return jnp.logical_and(error > tol, k < max_iter)

    k, x, error = jax.lax.while_loop(cond_fun, body_fun, (1, x_0, tol + 1))
    return x


successive_approx_jax = jax.jit(successive_approx_jax, static_argnums=(0,))
```

Here's a function that solves the model

```
def solve_model(params, arrays, tol=1e-6, max_iter=500):
    """
```

(continues on next page)

(continued from previous page)

```
    Compute the equilibrium price function.
    """
    # Simplify notation
    grid, draws, h = arrays
    γ, β, α, σ = params
    _T = lambda f: T(params, arrays, f)
    f = jnp.ones_like(grid)  # Initial guess of f

    f = successive_approx_jax(_T, f, tol=tol, max_iter=max_iter)

    price = f * grid**γ  # Back out price vector

    return price
```

Now let's solve the model again and compare timing

```
params, arrays = create_lucas_tree_model()
grid, draws, h = arrays
γ, β, α, σ = params

# Solve once to compile
with qe.Timer() as jax_with_compile_time:
    price_vals = solve_model(params, arrays).block_until_ready()
```

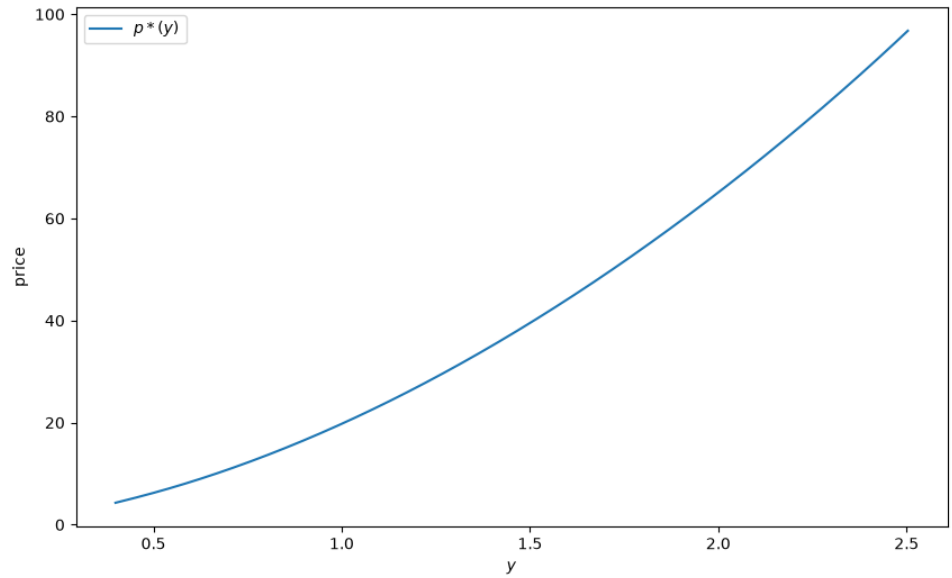
```
0.7160 seconds elapsed
```

```
# Now time execution without compile time
with qe.Timer() as jax_without_compile_time:
    price_vals = solve_model(params, arrays).block_until_ready()
print(
    "Speedup factor = ",
    numba_without_compile_time.elapsed / jax_without_compile_time.elapsed,
)
```

```
0.4480 seconds elapsed
Speedup factor =  9.27610267468928
```

Let's check the solutions are similar

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(grid, price_vals, label="$p*(y)$")
ax.set_xlabel("$y$")
ax.set_ylabel("price")
ax.legend()
plt.show()
```



8.4 Exercises

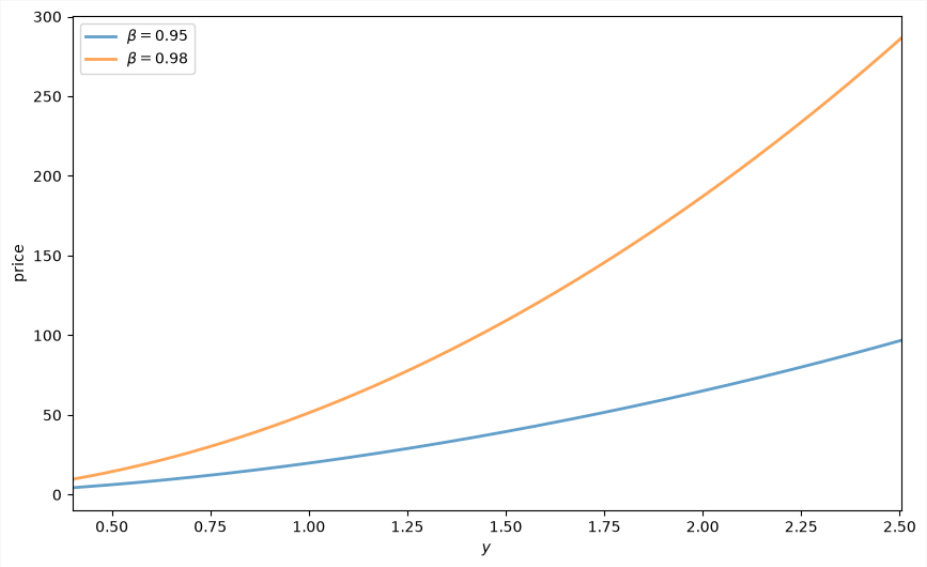
Exercise 8.4.1

When consumers are more patient the asset becomes more valuable, and the price of the Lucas tree shifts up.

Show this by plotting the price function for the Lucas tree when $\beta = 0.95$ and 0.98.

Solution

```
fig, ax = plt.subplots(figsize=(10, 6))

for β in (0.95, 0.98):
    params, arrays = create_lucas_tree_model(β=β)
    grid, draws, h = arrays
    γ, beta, α, σ = params
    price_vals = solve_model(params, arrays)
    label = rf"$\beta = {beta}$"
    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()
```


$\beta = 0.95$
$\beta = 0.98$
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

CHAPTER

NINE

AN ASSET PRICING PROBLEM

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

9.1 Overview

In this lecture we consider some asset pricing problems and use them to illustrate some foundations of JAX programming.

The main difference from the lecture *Asset Pricing: The Lucas Asset Pricing Model*, which also considers asset prices, is that the the state spaces will be discrete and multi-dimensional.

Most of the heavy lifting is done through routines from linear algebra.

Along the way, we will show how to solve some memory-intensive problems with large state spaces.

We do this using elegant techniques made available by JAX, involving the use of linear operators to avoid instantiating large matrices.

If you wish to skip all motivation and move straight to the first equation we plan to solve, you can jump to *(9.5.5)*.

The code outputs below are generated by machine connected to the following GPU

```
!nvidia-smi
```

```
Fri Oct  9 00:25:00 2026
+-----------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+---------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
```

(continues on next page)

(continued from previous page)

```
|                                         |                         |              ␣
↪MIG M. |
|=========================================+=========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off  |              ␣
↪     0 |
| N/A   36C    P0             33W /   70W |       0MiB /  15360MiB  |      0%      ␣
↪Default |
|                                         |                         |              ␣
↪   N/A |
+-----------------------------------------+-------------------------+---------------
↪-------+

+-----------------------------------------------------------------------------------
↪-------+
| Processes:                                                                        ␣
↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
↪Memory |
|        ID   ID                                                               ␣
↪Usage      |
|===================================================================================|
|  No running processes found                                                       ␣
↪       |
+-----------------------------------------------------------------------------------
↪-------+
```

In addition to JAX and Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

Below we use the following imports

```
import scipy
import quantecon as qe
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from collections import namedtuple
from time import time
```

We will use 64 bit floats with JAX in order to increase precision.

```
jax.config.update("jax_enable_x64", True)
```

9.2 Pricing a single payoff

Suppose, at time t, we have an asset that pays a random amount D_{t+1} at time $t+1$ and nothing after that.

The simplest way to price this asset is to use "risk-neutral" asset pricing, which asserts that the price of the asset at time t should be

$$P_t = \beta \, \mathbb{E}_t D_{t+1} \tag{9.1}$$

Here β is a constant discount factor and $\mathbb{E}_t D_{t+1}$ is the expectation of D_{t+1} at time t.

Roughly speaking, *(9.2.1)* says that the cost (i.e., price) equals expected benefit.

The discount factor is introduced because most people prefer payments now to payments in the future.

One problem with this very simple model is that it does not take into account attitudes to risk.

For example, investors often demand higher rates of return for holding risky assets.

This feature of asset prices cannot be captured by risk neutral pricing.

Hence we modify *(9.2.1)* to

$$P_t = \mathbb{E}_t M_{t+1} D_{t+1} \tag{9.2}$$

In this expression, M_{t+1} replaces β and is called the **stochastic discount factor**.

In essence, allowing discounting to become a random variable gives us the flexibility to combine temporal discounting and attitudes to risk.

We leave further discussion to other lectures because our aim is to move to the computational problem.

9.3 Pricing a cash flow

Now let's try to price an asset like a share, which delivers a cash flow $D_t, D_{t+1}, \ldots$.

We will call these payoffs "dividends".

If we buy the share, hold it for one period and sell it again, we receive one dividend and our payoff is $D_{t+1} + P_{t+1}$.

Therefore, by *(9.2.2)*, the price should be

$$P_t = \mathbb{E}_t M_{t+1} [D_{t+1} + P_{t+1}] \tag{9.3}$$

Because prices generally grow over time, which complicates analysis, it will be easier for us to solve for 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 (9.3) by D_t to get

$$V_t = \mathbb{E}_t \left[M_{t+1} \frac{D_{t+1}}{D_t} (1 + V_{t+1}) \right] \tag{9.4}$$

We can also write this as

$$V_t = \mathbb{E}_t \left[M_{t+1} \exp(G^d_{t+1}) (1 + V_{t+1}) \right] \tag{9.5}$$

where

$$G^d_{t+1} = \ln \frac{D_{t+1}}{D_t}$$

is the growth rate of dividends.

Our aim is to solve *(9.3.3)* but before that we need to specify

1. the stochastic discount factor M_{t+1} and
2. the growth rate of dividends G^d_{t+1}

9.4 Choosing the stochastic discount factor

We will adopt the stochastic discount factor described in *Asset Pricing: The Lucas Asset Pricing Model*, which has the form

$$M_{t+1} = \beta \frac{u'(C_{t+1})}{u'(C_t)} \tag{9.6}$$

where u is a utility function and C_t is time t consumption of a representative consumer.

For utility, we'll assume the **constant relative risk aversion** (CRRA) specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \tag{9.7}$$

Inserting the CRRA specification into (9.6) and letting

$$G^c_{t+1} = \ln\left(\frac{C_{t+1}}{C_t}\right)$$

the growth rate rate of consumption, we obtain

$$M_{t+1} = \beta \left(\frac{C_{t+1}}{C_t}\right)^{-\gamma} = \beta \exp(G^c_{t+1})^{-\gamma} = \beta \exp(-\gamma G^c_{t+1}) \tag{9.8}$$

9.5 Solving for the price-dividend ratio

Substituting *(9.4.3)* into (9.5) gives the price-dividend ratio formula

$$V_t = \beta \mathbb{E}_t \left[\exp(G^d_{t+1} - \gamma G^c_{t+1})(1 + V_{t+1})\right] \tag{9.9}$$

We assume there is a Markov chain $\{X_t\}$, which we call the **state process**, such that

$$\begin{aligned} G^c_{t+1} &= \mu_c + X_t + \sigma_c \epsilon_{c,t+1} \\ G^d_{t+1} &= \mu_d + X_t + \sigma_d \epsilon_{d,t+1} \end{aligned}$$

Here $\{\epsilon_{c,t}\}$ and $\{\epsilon_{d,t}\}$ are IID and standard normal, and independent of each other.

We can think of $\{X_t\}$ as an aggregate shock that affects both consumption growth and firm profits (and hence dividends).

We let P be the stochastic matrix that governs $\{X_t\}$ and assume $\{X_t\}$ takes values in some finite set S.

We guess that V_t is a fixed function of this state process (and this guess turns out to be correct).

This means that $V_t = v(X_t)$ for some unknown function v.

By *(9.5.1)*, the unknown function v satisfies the equation

$$v(X_t) = \beta \mathbb{E}_t \left\{\exp[a + (1-\gamma)X_t + \sigma_d \epsilon_{d,t+1} - \gamma \sigma_c \epsilon_{c,t+1}](1 + v(X_{t+1}))\right\} \tag{9.10}$$

where $a := \mu_d - \gamma \mu_c$

Since the shocks $\epsilon_{c,t+1}$ and $\epsilon_{d,t+1}$ are independent of $\{X_t\}$, we can integrate them out.

We use the following property of lognormal distributions: if $Y = \exp(c\epsilon)$ for constant c and $\epsilon \sim N(0,1)$, then $\mathbb{E}Y = \exp(c^2/2)$.

This yields

$$v(X_t) = \beta \mathbb{E}_t \left\{ \exp\left[a + (1-\gamma) X_t + \frac{\sigma_d^2 + \gamma^2 \sigma_c^2}{2} \right] (1 + v(X_{t+1})) \right\} \tag{9.11}$$

Conditioning on $X_t = x$, we can write this as

$$v(x) = \beta \sum_{y \in S} \left\{ \exp\left[a + (1-\gamma) x + \frac{\sigma_d^2 + \gamma^2 \sigma_c^2}{2} \right] (1 + v(y)) \right\} P(x, y) \tag{9.12}$$

for all $x \in S$.

Suppose $S = \{x_1, \dots, x_N\}$.

Then we can think of v as an N-vector and, using square brackets for indices on arrays, write

$$v[i] = \beta \sum_{j=1}^{N} \left\{ \exp\left[a + (1-\gamma) x[i] + \frac{\sigma_d^2 + \gamma^2 \sigma_c^2}{2} \right] (1 + v[j]) \right\} P[i, j] \tag{9.13}$$

for $i = 1, \dots, N$.

Equivalently, we can write

$$v[i] = \sum_{j=1}^{N} K[i, j](1 + v[j]) \tag{9.14}$$

where K is the matrix defined by

$$K[i, j] = \beta \left\{ \exp\left[a + (1-\gamma) x[i] + \frac{\sigma_d^2 + \gamma^2 \sigma_c^2}{2} \right] \right\} P[i, j] \tag{9.15}$$

Rewriting *(9.5.6)* in vector form yields

$$v = K(\mathbb{1} + v) \tag{9.16}$$

Notice that *(9.5.8)* can be written as $(I - K)v = K\mathbb{1}$.

The Neumann series lemma tells us that $I - K$ is invertible and the solution is

$$v = (I - K)^{-1} K \mathbb{1} \tag{9.17}$$

whenever $r(K)$, the spectral radius of K, is strictly less than one.

Once we specify P and all the parameters, we can

1. obtain K
2. check the spectral radius condition $r(K) < 1$ and, assuming it holds,
3. compute the solution via *(9.5.9)*.

9.6 Code

We will use the power iteration algorithm to check the spectral radius condition.

The function below computes the spectral radius of `A`.

```
def power_iteration_sr(A, num_iterations=15, seed=1234):
    " Estimates the spectral radius of A via power iteration. "

    # Initialize
    key = jax.random.key(seed)
    b_k = jax.random.normal(key, (A.shape[1],))
    sr = 0

    for _ in range(num_iterations):
        # calculate the matrix-by-vector product Ab
        b_k1 = jnp.dot(A, b_k)

        # calculate the norm
        b_k1_norm = jnp.linalg.norm(b_k1)

        # Record the current estimate of the spectral radius
        sr = jnp.sum(b_k1 * b_k)/jnp.sum(b_k * b_k)

        # re-normalize the vector and continue
        b_k = b_k1 / b_k1_norm

    return sr

power_iteration_sr = jax.jit(power_iteration_sr)
```

The next function verifies that the spectral radius of a given matrix is < 1.

```
def test_stability(Q):
    """
    Assert that the spectral radius of matrix Q is < 1.
    """
    sr = power_iteration_sr(Q)
    assert sr < 1, f"Spectral radius condition failed with radius = {sr}"
```

In what follows we assume that $\{X_t\}$, the state process, is a discretization of the AR(1) process

$$X_{t+1} = \rho X_t + \sigma \eta_{t+1}$$

where ρ, σ are parameters and $\{\eta_t\}$ is IID and standard normal.

To discretize this process we use QuantEcon.py's `tauchen` function.

Below we write a function called `create_model()` that returns a namedtuple storing the relevant parameters and arrays.

```
Model = namedtuple('Model',
                   ('P', 'S', 'β', 'γ', 'μ_c', 'μ_d', 'σ_c', 'σ_d'))

def create_model(N=100,          # size of state space for Markov chain
                 ρ=0.9,          # persistence parameter for Markov chain
                 σ=0.01,         # persistence parameter for Markov chain
                 β=0.98,         # discount factor
                 γ=2.5,          # coefficient of risk aversion
                 μ_c=0.01,       # mean growth of consumption
                 μ_d=0.01,       # mean growth of dividends
                 σ_c=0.02,       # consumption volatility
                 σ_d=0.04):      # dividend volatility
    # Create the state process
```

(continues on next page)

(continued from previous page)

```
    mc = qe.tauchen(N, ρ, σ)
    S = mc.state_values
    P = mc.P
    # Shift arrays to the device
    S, P = map(jax.device_put, (S, P))
    # Return the namedtuple
    return Model(P=P, S=S, β=β, γ=γ, μ_c=μ_c, μ_d=μ_d, σ_c=σ_c, σ_d=σ_d)
```

Our first step is to construct the matrix K defined in *(9.5.7)*.

Here's a function that does this using loops.

```
def compute_K_loop(model):
    # unpack
    P, S, β, γ, μ_c, μ_d, σ_c, σ_d = model
    N = len(S)
    K = np.empty((N, N))
    a = μ_d - γ * μ_c
    for i, x in enumerate(S):
        for j, y in enumerate(S):
            e = np.exp(a + (1 - γ) * x + (σ_d**2 + γ**2 * σ_c**2) / 2)
            K[i, j] = β * e * P[i, j]
    return K
```

To exploit the parallelization capabilities of JAX, let's also write a vectorized (i.e., loop-free) implementation.

```
def compute_K(model):
    # unpack
    P, S, β, γ, μ_c, μ_d, σ_c, σ_d = model
    N = len(S)
    # Reshape and multiply pointwise using broadcasting
    x = np.reshape(S, (N, 1))
    a = μ_d - γ * μ_c
    e = np.exp(a + (1 - γ) * x + (σ_d**2 + γ**2 * σ_c**2) / 2)
    K = β * e * P
    return K
```

These two functions produce the same output:

```
model = create_model(N=10)
K1 = compute_K(model)
K2 = compute_K_loop(model)
np.allclose(K1, K2)
```

```
True
```

Now we can compute the price-dividend ratio:

```
def price_dividend_ratio(model, test_stable=True):
    """
    Computes the price-dividend ratio of the asset.

    Parameters
    ----------
    model: an instance of Model
        contains primitives
```

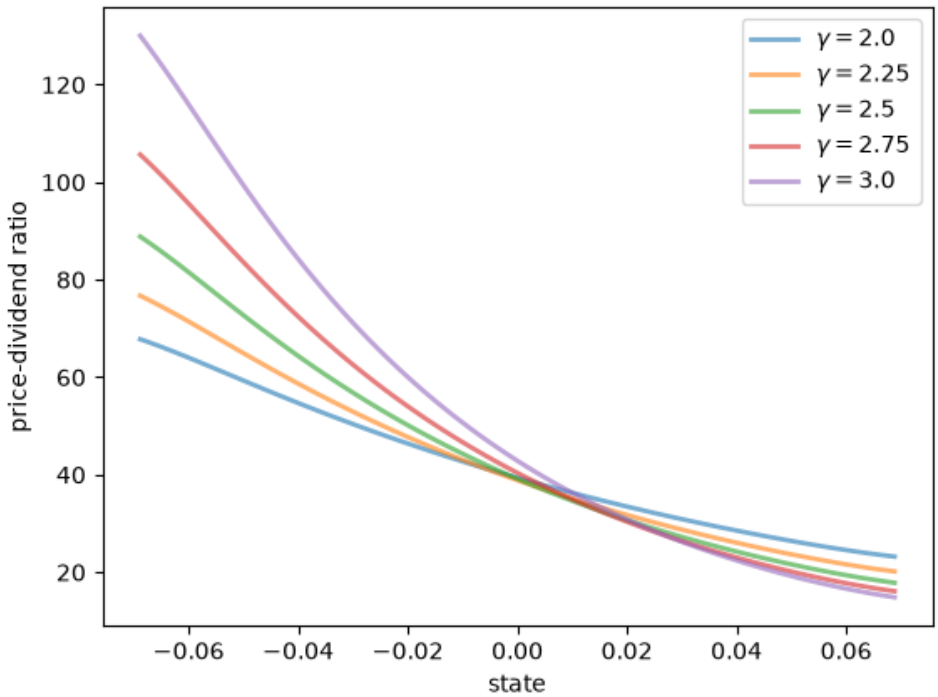
(continues on next page)

(continued from previous page)

```
    Returns
    -------
    v : array_like
        price-dividend ratio

    """
    K = compute_K(model)
    N = len(model.S)

    if test_stable:
        test_stability(K)

    # Compute v
    I = np.identity(N)
    ones_vec = np.ones(N)
    v = np.linalg.solve(I - K, K @ ones_vec)

    return v
```

Here's a plot of v as a function of the state for several values of γ.

```
model = create_model()
S = model.S
ys = np.linspace(2.0, 3.0, 5)

fig, ax = plt.subplots()

for y in ys:
    model = create_model(y=y)
    v = price_dividend_ratio(model)
    ax.plot(S, v, lw=2, alpha=0.6, label=rf"$\gamma = {y}$")

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 (9.8), higher growth decreases the discount factor, lowering the weight placed on future dividends.

9.7 An Extended Example

One problem with the last set is that volatility is constant through time (i.e., σ_c and σ_d are constants).

In reality, financial markets and growth rates of macroeconomic variables exhibit bursts of volatility.

To accommodate this, we now develop a *stochastic volatility* model.

To begin, suppose that consumption and dividends grow as follows.

$$G^i_{t+1} = \mu_i + Z_t + \bar{\sigma} \exp(H^i_t) \epsilon_{i,t+1}, \qquad i \in \{c, d\}$$

where $\{Z_t\}$ is a finite Markov chain and $\{H^c_t\}$ and $\{H^d_t\}$ are volatility processes.

We assume that $\{H^c_t\}$ and $\{H^d_t\}$ are AR(1) processes of the form

$$H^i_{t+1} = \rho_i H^i_t + \sigma_i \eta_{i,t+1}, \qquad i \in \{c, d\}$$

Here $\{\eta^c_t\}$ and $\{\eta^d_t\}$ are IID and standard normal.

Let $X_t = (H^c_t, H^d_t, Z_t)$.

We call $\{X_t\}$ the state process and guess that V_t is a function of this state process, so that $V_t = v(X_t)$ for some unknown function v.

Modifying *(9.5.2)* to accommodate the new growth specifications, we find that v satisfies

$$\begin{aligned} v(X_t) =& \beta \times \\ & \mathbb{E}_t \left\{ \exp[a + (1-\gamma) Z_t + \bar{\sigma} \exp(H_t^d) \epsilon_{d,t+1} - \gamma \bar{\sigma} \exp(H_t^c) \epsilon_{c,t+1}] (1 + v(X_{t+1})) \right\} \end{aligned} \tag{9.18}$$

where, as before, $a := \mu_d - \gamma \mu_c$

Conditioning on state $x = (h_c, h_d, z)$, this becomes

$$v(x) = \beta \mathbb{E}_t \exp[a + (1-\gamma) z + \bar{\sigma} \exp(h_d) \epsilon_{d,t+1} - \gamma \bar{\sigma} \exp(h_c) \epsilon_{c,t+1}] (1 + v(X_{t+1})) \tag{9.19}$$

As before, we integrate out the independent shocks and use the rules for expectations of lognormals to obtain

$$v(x) = \beta \mathbb{E}_t \exp \left[a + (1-\gamma) z + \bar{\sigma}^2 \frac{\exp(2h_d) + \gamma^2 \exp(2h_c)}{2} \right] (1 + v(X_{t+1})) \tag{9.20}$$

Let

$$\begin{aligned} & A(h_c, h_d, z, h_c', h_d', z') := \\ & \qquad \beta \ \exp \left[a + (1-\gamma) z + \bar{\sigma}^2 \frac{\exp(2h_d) + \gamma^2 \exp(2h_c)}{2} \right] P(h_c, h_c') Q(h_d, h_d') R(z, z') \end{aligned}$$

where P, Q, R are the stochastic matrices for, respectively, discretized $\{H_t^c\}$, discretized $\{H_t^d\}$ and $\{Z_t\}$,

With this notation, we can write *(9.7.3)* more explicitly as

$$v(h_c, h_d, z) = \sum_{h_c', h_d', z'} (1 + v(h_c', h_d', z')) A(h_c, h_d, z, h_c', h_d', z') \tag{9.21}$$

Let's now write the state using indices, with (i, j, k) being the indices for (h_c, h_d, z).

Then (9.21) becomes

$$v[i, j, k] = \sum_{i', j', k'} A[i, j, k, i', j', k'] (1 + v[i', j', k']) \tag{9.22}$$

One way to understand this is to reshape v into an N-vector, where $N = I \times J \times K$, and A into an $N \times N$ matrix.

Then we can write *(9.7.5)* as

$$v = A(\mathbb{1} + v)$$

Provided that the spectral radius condition $r(A) < 1$ holds, the solution is given by

$$v = (I - A)^{-1} A \mathbb{1}$$

9.8 Numpy Version

Our first implementation will be in NumPy.

Once we have a NumPy version working, we will convert it to JAX and check the difference in the run times.

The code block below provides a function called `create_sv_model()` that returns a namedtuple containing arrays and other data that form the primitives of the problem.

It assumes that $\{Z_t\}$ is a discretization of

$$Z_{t+1} = \rho_z Z_t + \sigma_z \xi_{t+1}$$

```
SVModel = namedtuple('SVModel',
                        ('P', 'hc_grid',
                         'Q', 'hd_grid',
                         'R', 'z_grid',
                         'β', 'γ', 'bar_σ', 'μ_c', 'μ_d'))

def create_sv_model(β=0.98,         # discount factor
                    γ=2.5,          # coefficient of risk aversion
                    I=14,           # size of state space for h_c
                    ρ_c=0.9,        # persistence parameter for h_c
                    σ_c=0.01,       # volatility parameter for h_c
                    J=14,           # size of state space for h_d
                    ρ_d=0.9,        # persistence parameter for h_d
                    σ_d=0.01,       # volatility parameter for h_d
                    K=14,           # size of state space for z
                    bar_σ=0.01,     # volatility scaling parameter
                    ρ_z=0.9,        # persistence parameter for z
                    σ_z=0.01,       # persistence parameter for z
                    μ_c=0.001,      # mean growth of consumption
                    μ_d=0.005):     # mean growth of dividends

    mc = qe.tauchen(I, ρ_c, σ_c)
    hc_grid = mc.state_values
    P = mc.P
    mc = qe.tauchen(J, ρ_d, σ_d)
    hd_grid = mc.state_values
    Q = mc.P
    mc = qe.tauchen(K, ρ_z, σ_z)
    z_grid = mc.state_values
    R = mc.P

    return SVModel(P=P, hc_grid=hc_grid,
                   Q=Q, hd_grid=hd_grid,
                   R=R, z_grid=z_grid,
                   β=β, γ=γ, bar_σ=bar_σ, μ_c=μ_c, μ_d=μ_d)
```

Now we provide a function to compute the matrix A.

```
def compute_A(sv_model):
    # Set up
    P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
    I, J, K = len(hc_grid), len(hd_grid), len(z_grid)
    N = I * J * K
    # Reshape and broadcast over (i, j, k, i', j', k')
    hc = np.reshape(hc_grid,      (I, 1, 1, 1,  1,  1))
    hd = np.reshape(hd_grid,      (1, J, 1, 1,  1,  1))
    z = np.reshape(z_grid,        (1, 1, K, 1,  1,  1))
    P = np.reshape(P,             (I, 1, 1, I,  1,  1))
    Q = np.reshape(Q,             (1, J, 1, 1,  J,  1))
    R = np.reshape(R,             (1, 1, K, 1,  1,  K))
    # Compute A and then reshape to create a matrix
    a = μ_d - γ * μ_c
    b = bar_σ**2 * (np.exp(2 * hd) + γ**2 * np.exp(2 * hc)) / 2
    κ = np.exp(a + (1 - γ) * z + b)
    A = β * κ * P * Q * R
    A = np.reshape(A, (N, N))
    return A
```

Here's our function to compute the price-dividend ratio for the stochastic volatility model.

```
def sv_pd_ratio(sv_model, test_stable=True):
    """
    Computes the price-dividend ratio of the asset for the stochastic volatility
    model.

    Parameters
    ----------
    sv_model: an instance of Model
        contains primitives

    Returns
    -------
    v : array_like
        price-dividend ratio

    """
    # unpack
    P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
    I, J, K = len(hc_grid), len(hd_grid), len(z_grid)
    N = I * J * K

    A = compute_A(sv_model)
    # Make sure that a unique solution exists
    if test_stable:
        test_stability(A)

    # Compute v
    ones_array = np.ones(N)
    Id = np.identity(N)
    v = scipy.linalg.solve(Id - A, A @ ones_array)
    # Reshape into an array of the form v[i, j, k]
    v = np.reshape(v, (I, J, K))
    return v
```

Let's create an instance of the model and solve it.

```
sv_model = create_sv_model()
P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
```

Let's run it to compile.

```
start = time()
v = sv_pd_ratio(sv_model)
numpy_with_compile = time() - start
print("Numpy compile plus execution time = ", numpy_with_compile)
```

```
Numpy compile plus execution time =  0.9052658081054688
```

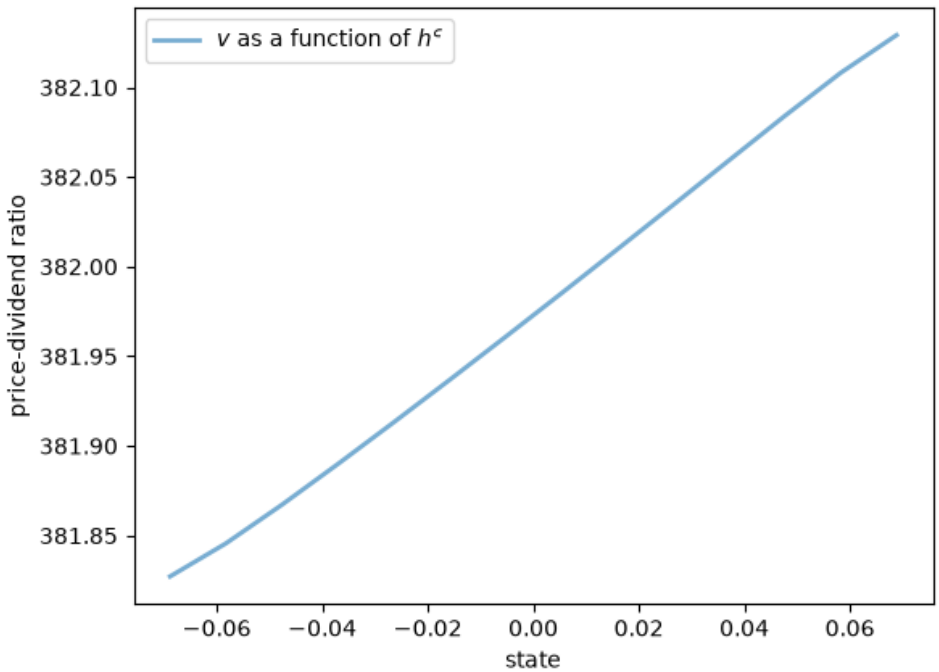
Let's run it again to remove the compile.

```
start = time()
v = sv_pd_ratio(sv_model)
numpy_without_compile = time() - start
print("Numpy execution time = ", numpy_without_compile)
```

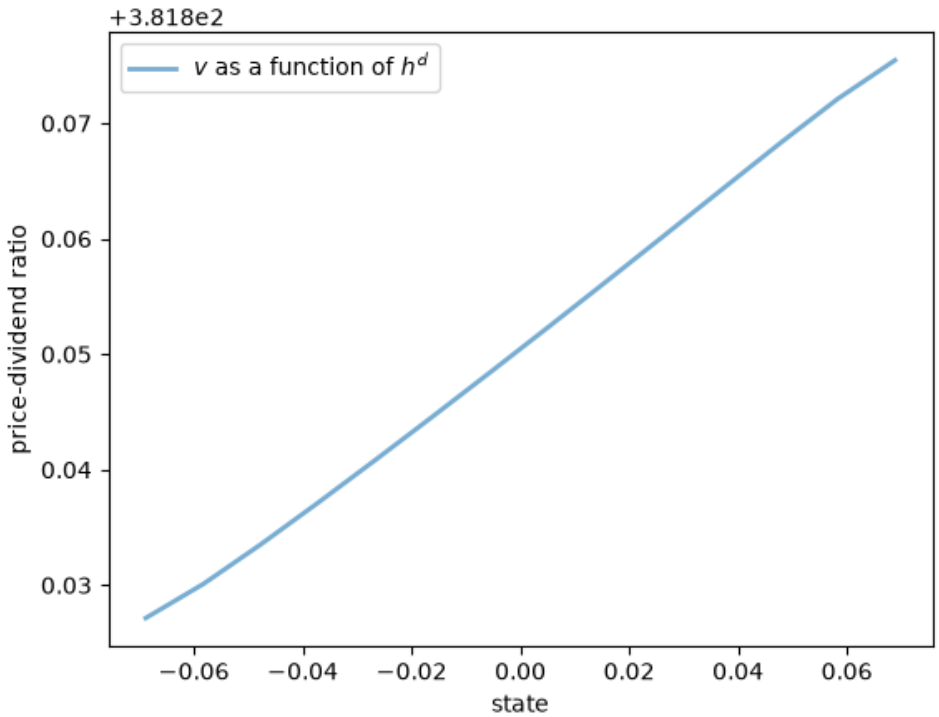
```
Numpy execution time =  0.32196855545043945
```

Here are some plots of the solution v along the three dimensions.

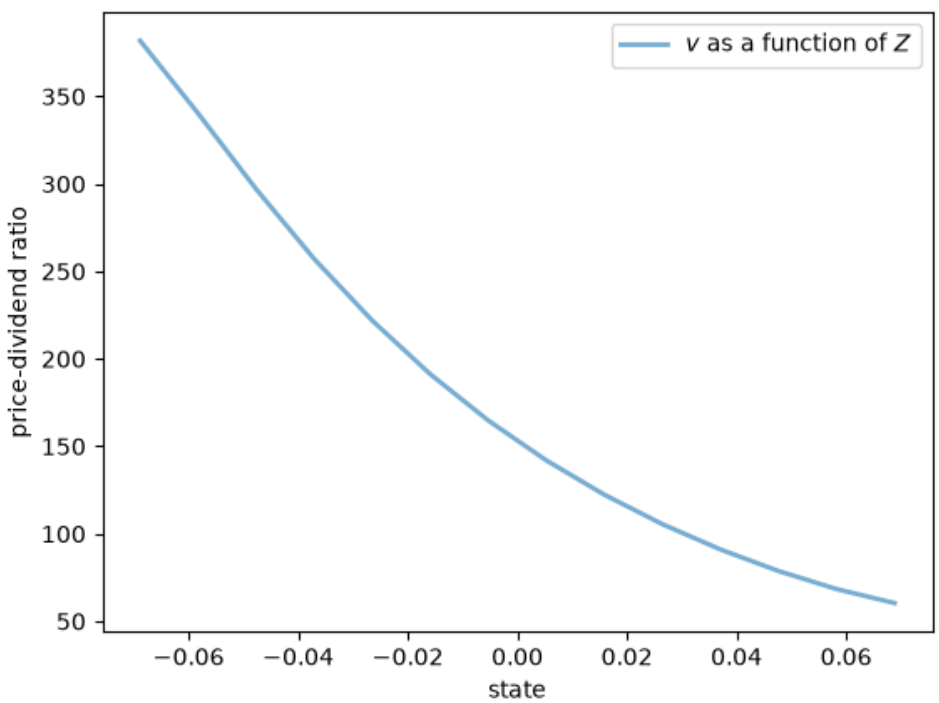
```
fig, ax = plt.subplots()
ax.plot(hc_grid, v[:, 0, 0], lw=2, alpha=0.6, label="$v$ as a function of $h^c$")
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend()
plt.show()
```




```
fig, ax = plt.subplots()
ax.plot(hd_grid, v[0, :, 0], lw=2, alpha=0.6, label="$v$ as a function of $h^d$")
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend()
plt.show()
```




```
fig, ax = plt.subplots()
ax.plot(z_grid, v[0, 0, :], lw=2, alpha=0.6, label="$v$ as a function of $Z$")
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend()
plt.show()
```



9.9 JAX Version

Now let's write a JAX version that is a simple transformation of the NumPy version.

(Below we will write a more efficient version using JAX's ability to work with linear operators.)

```
def create_sv_model_jax(sv_model):    # mean growth of dividends

    # Take the contents of a NumPy sv_model instance
    P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model

    # Shift the arrays to the device (GPU if available)
    hc_grid, hd_grid, z_grid = map(jax.device_put, (hc_grid, hd_grid, z_grid))
    P, Q, R = map(jax.device_put, (P, Q, R))

    # Create a new instance and return it
    return SVModel(P=P, hc_grid=hc_grid,
                   Q=Q, hd_grid=hd_grid,
                   R=R, z_grid=z_grid,
                   β=β, γ=γ, bar_σ=bar_σ, μ_c=μ_c, μ_d=μ_d)
```

Here's a function to compute A.

We include the extra argument `shapes` to help the compiler understand the size of the arrays.

This is important when we JIT-compile the function below.

```
def compute_A_jax(sv_model, shapes):
    # Set up
    P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
    I, J, K = shapes
    N = I * J * K
    # Reshape and broadcast over (i, j, k, i', j', k')
    hc = jnp.reshape(hc_grid,      (I, 1, 1, 1,  1,  1))
    hd = jnp.reshape(hd_grid,      (1, J, 1, 1,  1,  1))
    z = jnp.reshape(z_grid,        (1, 1, K, 1,  1,  1))
    P = jnp.reshape(P,             (I, 1, 1, I,  1,  1))
    Q = jnp.reshape(Q,             (1, J, 1, 1,  J,  1))
    R = jnp.reshape(R,             (1, 1, K, 1,  1,  K))
    # Compute A and then reshape to create a matrix
    a = μ_d - γ * μ_c
    b = bar_σ**2 * (jnp.exp(2 * hd) + γ**2 * jnp.exp(2 * hc)) / 2
    κ = jnp.exp(a + (1 - γ) * z + b)
    A = β * κ * P * Q * R
    A = jnp.reshape(A, (N, N))
    return A
```

Here's the function that computes the solution.

```
def sv_pd_ratio_jax(sv_model, shapes):
    """
    Computes the price-dividend ratio of the asset for the stochastic volatility
    model.

    Parameters
    ----------
    sv_model: an instance of Model
        contains primitives
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    v : array_like
        price-dividend ratio

    """
    # unpack
    P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
    I, J, K = len(hc_grid), len(hd_grid), len(z_grid)
    shapes = I, J, K
    N = I * J * K

    A = compute_A_jax(sv_model, shapes)

    # Compute v, reshape and return
    ones_array = jnp.ones(N)
    Id = jnp.identity(N)
    v = jax.scipy.linalg.solve(Id - A, A @ ones_array)
    return jnp.reshape(v, (I, J, K))
```

Now let's target these functions for JIT-compilation, while using `static_argnums` to indicate that the function will need to be recompiled when `shapes` changes.

```
compute_A_jax = jax.jit(compute_A_jax, static_argnums=(1,))
sv_pd_ratio_jax = jax.jit(sv_pd_ratio_jax, static_argnums=(1,))
```

```
sv_model = create_sv_model()
sv_model_jax = create_sv_model_jax(sv_model)
P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model_jax
shapes = len(hc_grid), len(hd_grid), len(z_grid)
```

Let's see how long it takes to run with compile time included.

```
start = time()
v_jax = sv_pd_ratio_jax(sv_model_jax, shapes).block_until_ready()
jnp_with_compile = time() - start
print("JAX compile plus execution time = ", jnp_with_compile)
```

```
JAX compile plus execution time =  0.6196620464324951
```

And now let's see without compile time.

```
start = time()
v_jax = sv_pd_ratio_jax(sv_model_jax, shapes).block_until_ready()
jnp_without_compile = time() - start
print("JAX execution time = ", jnp_without_compile)
```

```
JAX execution time =  0.1388852596282959
```

Here's the ratio of times:

```
jnp_without_compile / numpy_without_compile
```

```
0.43136280632744733
```

Let's check that the NumPy and JAX versions realize the same solution.

```
v = jax.device_put(v)

print(jnp.allclose(v, v_jax))
```

```
True
```

9.10 A memory-efficient JAX version

One problem with the code above is that we instantiate a matrix of size $N = I \times J \times K$.

This quickly becomes impossible as I, J, K increase.

Fortunately, JAX makes it possible to solve for the price-dividend ratio without instantiating this large matrix.

The first step is to think of A not as a matrix, but rather as the linear operator that transforms g into Ag.

```
def A(g, sv_model, shapes):
    # Set up
    P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
    I, J, K = shapes
    # Reshape and broadcast over (i, j, k, i', j', k')
    hc = jnp.reshape(hc_grid,      (I, 1, 1, 1,  1,  1))
    hd = jnp.reshape(hd_grid,      (1, J, 1, 1,  1,  1))
    z = jnp.reshape(z_grid,        (1, 1, K, 1,  1,  1))
    P = jnp.reshape(P,             (I, 1, 1, I,  1,  1))
    Q = jnp.reshape(Q,             (1, J, 1, 1,  J,  1))
    R = jnp.reshape(R,             (1, 1, K, 1,  1,  K))
    g = jnp.reshape(g,             (1, 1, 1, I,  J,  K))
    a = μ_d - γ * μ_c
    b = bar_σ**2 * (jnp.exp(2 * hd) + γ**2 * jnp.exp(2 * hc)) / 2
    κ = jnp.exp(a + (1 - γ) * z + b)
    A = β * κ * P * Q * R
    Ag = jnp.sum(A * g, axis=(3, 4, 5))
    return Ag
```

Now we write a version of the solution function for the price-dividend ratio that acts directly on the linear operator `A`.

```
def sv_pd_ratio_linop(sv_model, shapes):
    P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
    I, J, K = shapes

    ones_array = jnp.ones((I, J, K))
    # Set up the operator g -> (I - A) g
    J = lambda g: g - A(g, sv_model, shapes)
    # Solve v = (I - A)^{-1} A 1
    A1 = A(ones_array, sv_model, shapes)
    # Apply an iterative solver that works for linear operators
    v = jax.scipy.sparse.linalg.bicgstab(J, A1)[0]
    return v
```

Let's target these functions for JIT compilation.

```
A = jax.jit(A, static_argnums=(2,))
sv_pd_ratio_linop = jax.jit(sv_pd_ratio_linop, static_argnums=(1,))
```

Let's time the solution with compile time included.

```
start = time()
v_jax_linop = sv_pd_ratio_linop(sv_model, shapes).block_until_ready()
jnp_linop_with_compile = time() - start
print("JAX compile plus execution time = ", jnp_linop_with_compile)
```

```
JAX compile plus execution time =  0.3022291660308838
```

And now let's see without compile time.

```
start = time()
v_jax_linop = sv_pd_ratio_linop(sv_model, shapes).block_until_ready()
jnp_linop_without_compile = time() - start
print("JAX execution time = ", jnp_linop_without_compile)
```

```
JAX execution time =  0.006482124328613281
```

Let's verify the solution again:

```
print(jnp.allclose(v, v_jax_linop))
```

```
True
```

Here's the ratio of times between memory-efficient and direct version:

```
jnp_linop_without_compile / jnp_without_compile
```

```
0.04667251475038925
```

The speed is somewhat faster and, moreover, we can now work with much larger grids.

Here's a moderately large example, where the state space has 15,625 elements.

```
sv_model = create_sv_model(I=25, J=25, K=25)
sv_model_jax = create_sv_model_jax(sv_model)
P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model_jax
shapes = len(hc_grid), len(hd_grid), len(z_grid)

%time _ = sv_pd_ratio_linop(sv_model_jax, shapes).block_until_ready()
%time _ = sv_pd_ratio_linop(sv_model_jax, shapes).block_until_ready()
```

```
CPU times: user 761 ms, sys: 23.6 ms, total: 784 ms
Wall time: 509 ms
CPU times: user 168 ms, sys: 24 µs, total: 169 ms
Wall time: 168 ms
```

The solution is computed relatively quickly and without memory issues.

Readers will find that they can push these numbers further, although we refrain from doing so here.

Part IV

Dynamic Programming

CHAPTER

TEN

JOB SEARCH

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In this lecture we study a basic infinite-horizon job search problem with Markov wage draws

Note

For background on infinite horizon job search see, e.g., DP1.

The exercise at the end asks you to add risk-sensitive preferences and see how the main results change.

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

```
!pip install quantecon
```

We use the following imports.

```
import matplotlib.pyplot as plt
import quantecon as qe
import jax
import jax.numpy as jnp
from collections import namedtuple

jax.config.update("jax_enable_x64", True)
```

10.1 Model

We study an elementary model where

- jobs are permanent
- unemployed workers receive current compensation c
- the horizon is infinite
- an unemployment agent discounts the future via discount factor $\beta \in (0, 1)$

10.1.1 Set up

At the start of each period, an unemployed worker receives wage offer W_t.

To build a wage offer process we consider the dynamics

$$W_{t+1} = \rho W_t + \nu Z_{t+1}$$

where $(Z_t)_{t \geq 0}$ is IID and standard normal.

We then discretize this wage process using Tauchen's method to produce a stochastic matrix P.

Successive wage offers are drawn from P.

10.1.2 Rewards

Since jobs are permanent, the return to accepting wage offer w today is

$$w + \beta w + \beta^2 w + \cdots = \frac{w}{1 - \beta}$$

The Bellman equation is

$$v(w) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \sum_{w'} v(w') P(w, w') \right\}$$

We solve this model using value function iteration.

10.2 Code

Let's set up a `namedtuple` to store information needed to solve the model.

```
Model = namedtuple('Model', ('n', 'w_vals', 'P', 'β', 'c'))
```

The function below holds default values and populates the `namedtuple`.

```
def create_js_model(
        n=500,        # wage grid size
        ρ=0.9,        # wage persistence
        ν=0.2,        # wage volatility
        β=0.99,       # discount factor
        c=1.0,        # unemployment compensation
```

(continues on next page)

(continued from previous page)

```
    ):
    "Creates an instance of the job search model with Markov wages."
    mc = qe.tauchen(n, ρ, ν)
    w_vals, P = jnp.exp(mc.state_values), jnp.array(mc.P)
    return Model(n, w_vals, P, β, c)
```

Let's test it:

```
model = create_js_model(β=0.98)
```

```
model.c
```

```
1.0
```

```
model.β
```

```
0.98
```

```
model.w_vals.mean()
```

```
Array(1.34861482, dtype=float64)
```

Here's the Bellman operator.

```
@jax.jit
def T(v, model):
    """
    The Bellman operator Tv = max{e, c + β E v} with

        e(w) = w / (1-β) and (Ev)(w) = E_w[ v(W')]

    """
    n, w_vals, P, β, c = model
    h = c + β * P @ v
    e = w_vals / (1 - β)

    return jnp.maximum(e, h)
```

The next function computes the optimal policy under the assumption that v is the value function.

The policy takes the form

$$\sigma(w) = \mathbf{1}\left\{\frac{w}{1-\beta} \geq c + \beta \sum_{w'} v(w')P(w,w')\right\}$$

Here $\mathbf{1}$ is an indicator function.

- $\sigma(w) = 1$ means stop
- $\sigma(w) = 0$ means continue.

```
@jax.jit
def get_greedy(v, model):
    "Get a v-greedy policy."
    n, w_vals, P, β, c = model
```

(continues on next page)

(continued from previous page)

```
    e = w_vals / (1 - β)
    h = c + β * P @ v
    σ = jnp.where(e >= h, 1, 0)
    return σ
```

Here's a routine for value function iteration.

```
def vfi(model, max_iter=10_000, tol=1e-4):
    "Solve the infinite-horizon Markov job search model by VFI."
    print("Starting VFI iteration.")
    v = jnp.zeros_like(model.w_vals)    # Initial guess
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        new_v = T(v, model)
        error = jnp.max(jnp.abs(new_v - v))
        i += 1
        v = new_v

    v_star = v
    σ_star = get_greedy(v_star, model)
    return v_star, σ_star
```

10.3 Computing the solution

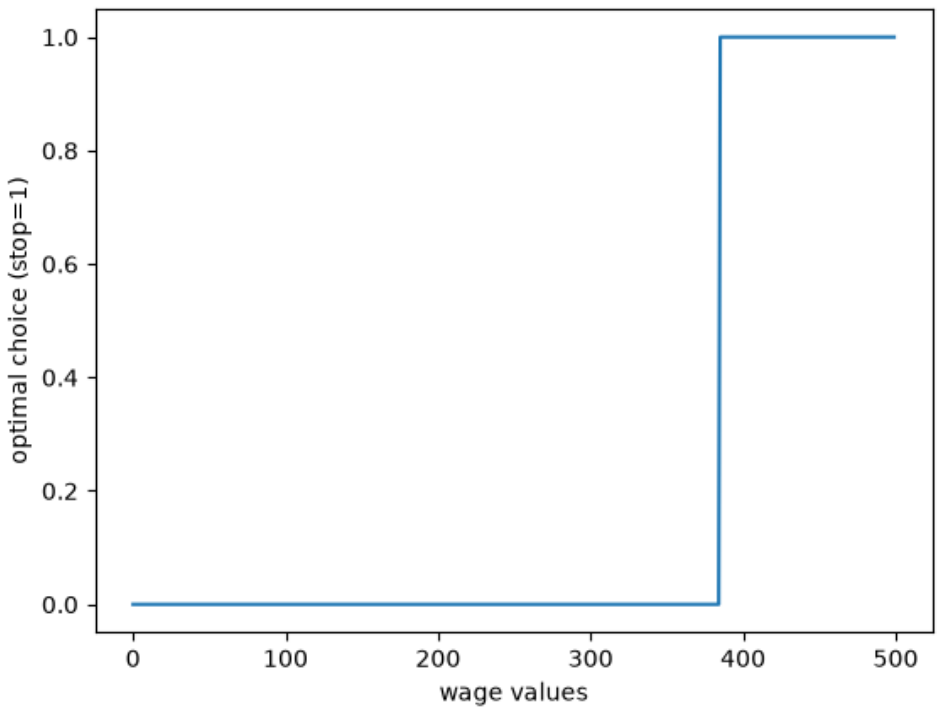
Let's set up and solve the model.

```
model = create_js_model()
n, w_vals, P, β, c = model

v_star, σ_star = vfi(model)
```

```
Starting VFI iteration.
```

Here's the optimal policy:

```
fig, ax = plt.subplots()
ax.plot(σ_star)
ax.set_xlabel("wage values")
ax.set_ylabel("optimal choice (stop=1)")
plt.show()
```



We compute the reservation wage as the first w such that $\sigma(w) = 1$.

```
stop_indices = jnp.where(σ_star == 1)
stop_indices
```

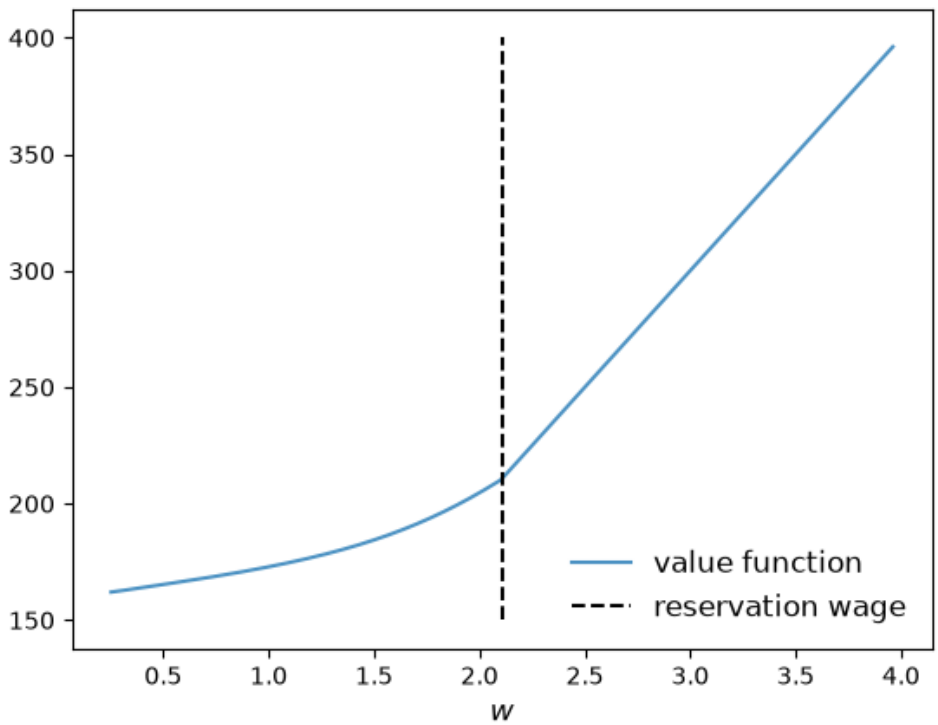
```
(Array([385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397,
        398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410,
        411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423,
        424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436,
        437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449,
        450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462,
        463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475,
        476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488,
        489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499],      dtype=int64),)
```

```
res_wage_index = min(stop_indices[0])
```

```
res_wage = w_vals[res_wage_index]
```

Here's a joint plot of the value function and the reservation wage.

```
fig, ax = plt.subplots()
ax.plot(w_vals, v_star, alpha=0.8, label="value function")
ax.vlines((res_wage,), 150, 400, 'k', ls='--', label="reservation wage")
ax.legend(frameon=False, fontsize=12, loc="lower right")
ax.set_xlabel("$w$", fontsize=12)
plt.show()
```



10.4 Exercise

Exercise 10.4.1

In the setting above, the agent is risk-neutral vis-a-vis future utility risk.

Now solve the same problem but this time assuming that the agent has risk-sensitive preferences, which are a type of nonlinear recursive preferences.

The Bellman equation becomes

$$v(w) = \max \left\{ \frac{w}{1-\beta}, c + \frac{\beta}{\theta} \ln \left[\sum_{w'} \exp(\theta v(w')) P(w, w') \right] \right\}$$

When $\theta < 0$ the agent is risk averse.

Solve the model when $\theta = -0.1$ and compare your result to the risk neutral case.

Try to interpret your result.

You can start with the following code:

```
RiskModel = namedtuple('Model', ('n', 'w_vals', 'P', 'β', 'c', 'θ'))

def create_risk_sensitive_js_model(
        n=500,        # wage grid size
        ρ=0.9,        # wage persistence
        ν=0.2,        # wage volatility
        β=0.99,       # discount factor
        c=1.0,        # unemployment compensation
        θ=-0.1        # risk parameter
    ):
    "Creates an instance of the job search model with Markov wages."
    mc = qe.tauchen(n, ρ, ν)
    w_vals, P = jnp.exp(mc.state_values), mc.P
    P = jnp.array(P)
    return RiskModel(n, w_vals, P, β, c, θ)
```

Now you need to modify `T` and `get_greedy` and then run value function iteration again.

Solution

```
RiskModel = namedtuple('Model', ('n', 'w_vals', 'P', 'β', 'c', 'θ'))

def create_risk_sensitive_js_model(
        n=500,        # wage grid size
        ρ=0.9,        # wage persistence
        ν=0.2,        # wage volatility
        β=0.99,       # discount factor
        c=1.0,        # unemployment compensation
        θ=-0.1        # risk parameter
    ):
    "Creates an instance of the job search model with Markov wages."
    mc = qe.tauchen(n, ρ, ν)
    w_vals, P = jnp.exp(mc.state_values), mc.P
    P = jnp.array(P)
    return RiskModel(n, w_vals, P, β, c, θ)


@jax.jit
def T_rs(v, model):
    """
    The Bellman operator Tv = max{e, c + β R v} with

        e(w) = w / (1-β) and

        (Rv)(w) = (1/θ) ln{E_w[ exp(θ v(W'))]}

    """
    n, w_vals, P, β, c, θ = model
    h = c + (β / θ) * jnp.log(P @ (jnp.exp(θ * v)))
    e = w_vals / (1 - β)

    return jnp.maximum(e, h)


@jax.jit
def get_greedy_rs(v, model):
    " Get a v-greedy policy."
    n, w_vals, P, β, c, θ = model
    e = w_vals / (1 - β)
    h = c + (β / θ) * jnp.log(P @ (jnp.exp(θ * v)))
    σ = jnp.where(e >= h, 1, 0)
    return σ



def vfi(model, max_iter=10_000, tol=1e-4):
    "Solve the infinite-horizon Markov job search model by VFI."
    print("Starting VFI iteration.")
    v = jnp.zeros_like(model.w_vals)     # Initial guess
    i = 0
    error = tol + 1
```

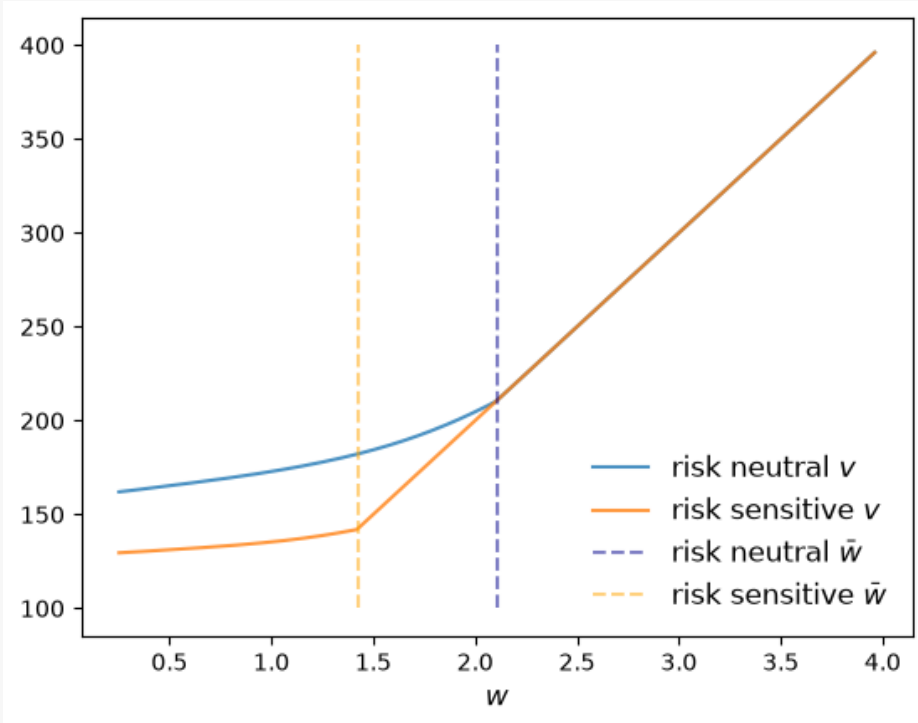
```
    while error > tol and i < max_iter:
        new_v = T_rs(v, model)
        error = jnp.max(jnp.abs(new_v - v))
        i += 1
        v = new_v

    v_star = v
    σ_star = get_greedy_rs(v_star, model)
    return v_star, σ_star



model_rs = create_risk_sensitive_js_model()

n, w_vals, P, β, c, θ = model_rs

v_star_rs, σ_star_rs = vfi(model_rs)
```

```
Starting VFI iteration.
```

Let's plot the results together with the original risk neutral case and see what we get.

```
stop_indices = jnp.where(σ_star_rs == 1)
res_wage_index = min(stop_indices[0])
res_wage_rs = w_vals[res_wage_index]
```

```
fig, ax = plt.subplots()
ax.plot(w_vals, v_star,  alpha=0.8, label="risk neutral $v$")
ax.plot(w_vals, v_star_rs, alpha=0.8, label="risk sensitive $v$")
ax.vlines((res_wage,), 100, 400,  ls='--', color='darkblue',
          alpha=0.5, label=r"risk neutral $\bar w$")
ax.vlines((res_wage_rs,), 100, 400, ls='--', color='orange',
          alpha=0.5, label=r"risk sensitive $\bar w$")
ax.legend(frameon=False, fontsize=12, loc="lower right")
ax.set_xlabel("$w$", fontsize=12)
plt.show()
```



The figure shows that the reservation wage under risk sensitive preferences (RS $\bar{w}$) shifts down.

This makes sense – the agent does not like risk and hence is more inclined to accept the current offer, even when it's lower.

CHAPTER

ELEVEN

OPTIMAL SAVINGS I: VALUE FUNCTION ITERATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to JAX and Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

We will use the following imports:

```
import quantecon as qe
import numpy as np
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from collections import namedtuple
from time import time
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:28:56 2026
+-----------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+---------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |              ␣
 ↪MIG M. |
```

(continues on next page)

(continued from previous page)

```
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                    ␣
↪     0 |
| N/A   38C    P8             14W /   70W |       0MiB /  15360MiB |      0%      ␣
↪Default |
|                                         |                        |                    ␣
↪   N/A |
+-----------------------------------------+------------------------+---------------
↪-------+

+-----------------------------------------------------------------------------------------
↪-------+
| Processes:                                                                              ␣
↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
↪Memory |
|        ID   ID                                                               ␣
↪Usage      |
|=========================================================================================|
|  No running processes found                                                             ␣
↪       |
+-----------------------------------------------------------------------------------------
↪-------+
```

We'll use 64 bit floats to gain extra precision.

```
jax.config.update("jax_enable_x64", True)
```

11.1 Overview

We consider an optimal savings problem with CRRA utility and budget constraint

$$W_{t+1} + C_t \leq RW_t + Y_t$$

where

- C_t is consumption and $C_t \geq 0$,
- W_t is wealth and $W_t \geq 0$,
- $R > 0$ is a gross rate of return, and
- (Y_t) is labor income.

We assume below that labor income is a discretized AR(1) process.

The Bellman equation is

$$v(w) = \max_{0 \leq w' \leq Rw+y} \left\{ u(Rw + y - w') + \beta \sum_{y'} v(w', y') Q(y, y') \right\}$$

where

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

In the code we use the function

$$B((w, y), w', v) = u(Rw + y - w') + \beta \sum_{y'} v(w', y')Q(y, y').$$

the encapsulate the right hand side of the Bellman equation.

11.2 Starting with NumPy

Let's start with a standard NumPy version running on the CPU.

Starting with this traditional approach will allow us to record the speed gain associated with switching to JAX.

(NumPy operations are similar to MATLAB operations, so this also serves as a rough comparison with MATLAB.)

11.2.1 Functions and operators

The following function contains default parameters and returns tuples that contain the key computational components of the model.

```
def create_consumption_model(R=1.01,                    # Gross interest rate
                             β=0.98,                    # Discount factor
                             γ=2,                       # CRRA parameter
                             w_min=0.01,                # Min wealth
                             w_max=5.0,                 # Max wealth
                             w_size=150,                # Grid side
                             ρ=0.9, ν=0.1, y_size=100): # Income parameters
    """
    A function that takes in parameters and returns parameters and grids
    for the optimal savings problem.
    """
    # Build grids and transition probabilities
    w_grid = np.linspace(w_min, w_max, w_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = np.exp(mc.state_values), mc.P
    # Pack and return
    params = β, R, γ
    sizes = w_size, y_size
    arrays = w_grid, y_grid, Q
    return params, sizes, arrays
```

(The function returns sizes of arrays because we use them later to help compile functions in JAX.)

To produce efficient NumPy code, we will use a vectorized approach.

The first step is to create the right hand side of the Bellman equation as a multi-dimensional array with dimensions over all states and controls.

```
def B(v, params, sizes, arrays):
    """
    A vectorized version of the right-hand side of the Bellman equation
    (before maximization), which is a 3D array representing

        B(w, y, w') = u(Rw + y - w') + β Σ_y' v(w', y') Q(y, y')

    for all (w, y, w').
```

(continues on next page)

(continued from previous page)

```
    """

    # Unpack
    β, R, γ = params
    w_size, y_size = sizes
    w_grid, y_grid, Q = arrays

    # Compute current rewards r(w, y, wp) as array r[i, j, ip]
    w  = np.reshape(w_grid, (w_size, 1, 1))     # w[i]   ->  w[i, j, ip]
    y  = np.reshape(y_grid, (1, y_size, 1))     # z[j]   ->  z[i, j, ip]
    wp = np.reshape(w_grid, (1, 1, w_size))     # wp[ip] -> wp[i, j, ip]
    c = R * w + y - wp

    # Calculate continuation rewards at all combinations of (w, y, wp)
    v = np.reshape(v, (1, 1, w_size, y_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Q = np.reshape(Q, (1, y_size, 1, y_size))  # Q[j, jp]  -> Q[i, j, ip, jp]
    EV = np.sum(v * Q, axis=3)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return np.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -np.inf)
```

Here are two functions we need for value function iteration.

The first is the Bellman operator.

The second computes a v-greedy policy given v (i.e., the policy that maximizes the right-hand side of the Bellman equation.)

```
def T(v, params, sizes, arrays):
    "The Bellman operator."
    return np.max(B(v, params, sizes, arrays), axis=2)

def get_greedy(v, params, sizes, arrays):
    "Computes a v-greedy policy, returned as a set of indices."
    return np.argmax(B(v, params, sizes, arrays), axis=2)
```

11.2.2 Value function iteration

Here's a routine that performs value function iteration.

```
def value_function_iteration(model, max_iter=10_000, tol=1e-5):
    params, sizes, arrays = model
    v = np.zeros(sizes)
    i, error = 0, tol + 1
    while error > tol and i < max_iter:
        v_new = T(v, params, sizes, arrays)
        error = np.max(np.abs(v_new - v))
        i += 1
        v = v_new
    return v, get_greedy(v, params, sizes, arrays)
```

Now we create an instance, unpack it, and test how long it takes to solve the model.

```
model = create_consumption_model()
# Unpack
```

(continues on next page)

(continued from previous page)

```
params, sizes, arrays = model
β, R, γ = params
w_size, y_size = sizes
w_grid, y_grid, Q = arrays
```

```
print("Starting VFI.")
start = time()
v_star, σ_star = value_function_iteration(model)
numpy_with_compile = time() - start
print(f"VFI completed in {numpy_with_compile} seconds.")
```

```
Starting VFI.
VFI completed in 17.270034313201904 seconds.
```

Let's run it again to eliminate compile time.

```
start = time()
v_star, σ_star = value_function_iteration(model)
numpy_without_compile = time() - start
print(f"VFI completed in {numpy_without_compile} seconds.")
```

```
VFI completed in 16.910991191864014 seconds.
```

Here's a plot of the policy function.

```
fig, ax = plt.subplots()
ax.plot(w_grid, w_grid, "k--", label="45")
ax.plot(w_grid, w_grid[σ_star[:, 1]], label="$\\sigma^*(\cdot, y_1)$")
ax.plot(w_grid, w_grid[σ_star[:, -1]], label="$\\sigma^*(\cdot, y_N)$")
ax.legend()
plt.show()
```

11.3 Switching to JAX

To switch over to JAX, we change `np` to `jnp` throughout and add some `jax.jit` requests.

11.3.1 Functions and operators

We redefine `create_consumption_model` to produce JAX arrays.

```
def create_consumption_model(R=1.01,                        # Gross interest rate
                             β=0.98,                        # Discount factor
                             γ=2,                           # CRRA parameter
                             w_min=0.01,                    # Min wealth
                             w_max=5.0,                     # Max wealth
                             w_size=150,                    # Grid side
                             ρ=0.9, ν=0.1, y_size=100):     # Income parameters
    """
    A function that takes in parameters and returns parameters and grids
    for the optimal savings problem.
```

(continues on next page)

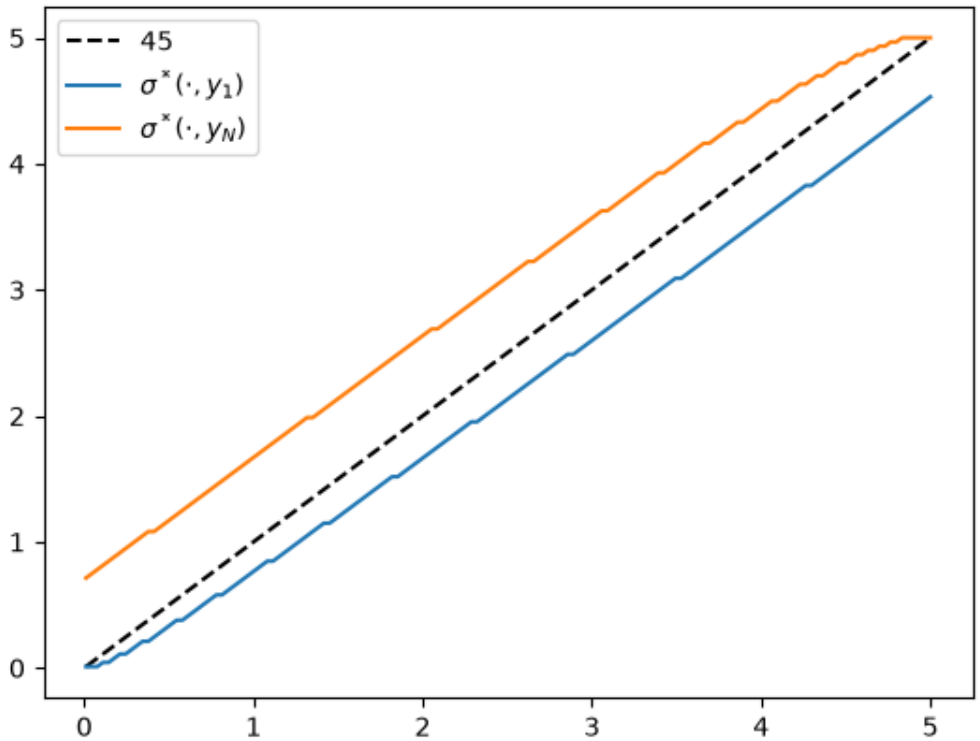


Fig. 11.1: Policy function

(continued from previous page)

```
    """
    w_grid = jnp.linspace(w_min, w_max, w_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), jax.device_put(mc.P)
    sizes = w_size, y_size
    return (β, R, γ), sizes, (w_grid, y_grid, Q)
```

The right hand side of the Bellman equation is the same as the NumPy version after switching `np` to `jnp`.

```
def B(v, params, sizes, arrays):
    """
    A vectorized version of the right-hand side of the Bellman equation
    (before maximization), which is a 3D array representing

        B(w, y, w') = u(Rw + y - w') + β Σ_y' v(w', y') Q(y, y')

    for all (w, y, w').
    """

    # Unpack
    β, R, γ = params
    w_size, y_size = sizes
    w_grid, y_grid, Q = arrays

    # Compute current rewards r(w, y, wp) as array r[i, j, ip]
    w  = jnp.reshape(w_grid, (w_size, 1, 1))     # w[i]   ->  w[i, j, ip]
```

(continues on next page)

(continued from previous page)

```
    y  = jnp.reshape(y_grid, (1, y_size, 1))     # z[j]   ->  z[i, j, ip]
    wp = jnp.reshape(w_grid, (1, 1, w_size))     # wp[ip] -> wp[i, j, ip]
    c = R * w + y - wp

    # Calculate continuation rewards at all combinations of (w, y, wp)
    v = jnp.reshape(v, (1, 1, w_size, y_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Q = jnp.reshape(Q, (1, y_size, 1, y_size))  # Q[j, jp]  -> Q[i, j, ip, jp]
    EV = jnp.sum(v * Q, axis=3)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Some readers might be concerned that we are creating high dimensional arrays, leading to inefficiency.

Could they be avoided by more careful vectorization?

In fact this is not necessary: this function will be JIT-compiled by JAX, and the JIT compiler will optimize compiled code to minimize memory use.

```
B = jax.jit(B, static_argnums=(2,))
```

In the call above, we indicate to the compiler that `sizes` is static, so the compiler can parallelize optimally while taking array sizes as fixed.

The Bellman operator T can be implemented by

```
def T(v, params, sizes, arrays):
    "The Bellman operator."
    return jnp.max(B(v, params, sizes, arrays), axis=2)

T = jax.jit(T, static_argnums=(2,))
```

The next function computes a v-greedy policy given v (i.e., the policy that maximizes the right-hand side of the Bellman equation.)

```
def get_greedy(v, params, sizes, arrays):
    "Computes a v-greedy policy, returned as a set of indices."
    return jnp.argmax(B(v, params, sizes, arrays), axis=2)

get_greedy = jax.jit(get_greedy, static_argnums=(2,))
```

11.3.2 Successive approximation

Now we define a solver that implements VFI.

We could use the one we built for NumPy above, after changing `np` to `jnp`.

Alternatively, we can push a bit harder and write a compiled version using `jax.lax.while_loop`.

This will give us just a bit more speed.

The first step is to write a compiled successive approximation routine that performs fixed point iteration on some given function `T`.

```
def successive_approx_jax(T,                        # Operator (callable)
                          x_0,                      # Initial condition
                          tolerance=1e-6,           # Error tolerance
```

(continues on next page)

(continued from previous page)

```
                            max_iter=10_000):       # Max iteration bound
    def body_fun(k_x_err):
        k, x, error = k_x_err
        x_new = T(x)
        error = jnp.max(jnp.abs(x_new - x))
        return k + 1, x_new, error

    def cond_fun(k_x_err):
        k, x, error = k_x_err
        return jnp.logical_and(error > tolerance, k < max_iter)

    k, x, error = jax.lax.while_loop(cond_fun, body_fun,
                                     (1, x_0, tolerance + 1))
    return x

successive_approx_jax = \
    jax.jit(successive_approx_jax, static_argnums=(0,))
```

Our value function iteration routine calls `successive_approx_jax` while passing in the Bellman operator.

```
def value_function_iteration(model, tol=1e-5):
    params, sizes, arrays = model
    vz = jnp.zeros(sizes)
    _T = lambda v: T(v, params, sizes, arrays)
    v_star = successive_approx_jax(_T, vz, tolerance=tol)
    return v_star, get_greedy(v_star, params, sizes, arrays)
```

11.3.3 Timing

Let's create an instance and unpack it.

```
model = create_consumption_model()
# Unpack
params, sizes, arrays = model
β, R, γ = params
w_size, y_size = sizes
w_grid, y_grid, Q = arrays
```

Let's see how long it takes to solve this model.

```
print("Starting VFI using vectorization.")
start = time()
v_star_jax, σ_star_jax = value_function_iteration(model)
jax_with_compile = time() - start
print(f"VFI completed in {jax_with_compile} seconds.")
```

```
Starting VFI using vectorization.
VFI completed in 0.8012347221374512 seconds.
```

Let's run it again to eliminate compile time.

```
start = time()
v_star_jax, σ_star_jax = value_function_iteration(model)
jax_without_compile = time() - start
print(f"VFI completed in {jax_without_compile} seconds.")
```

```
VFI completed in 0.3697173595428467 seconds.
```

The relative speed gain is

```
print(f"Relative speed gain = {numpy_without_compile / jax_without_compile}")
```

```
Relative speed gain = 45.740322317497764
```

This is an impressive speed up and in fact we can do better still by switching to alternative algorithms that are better suited to parallelization.

These algorithms are discussed in a *separate lecture*.

11.4 Switching to vmap

Before we discuss alternative algorithms, let's take another look at value function iteration.

For this simple optimal savings problem, direct vectorization is relatively easy.

In particular, it's straightforward to express the right hand side of the Bellman equation as an array that stores evaluations of the function at every state and control.

For more complex models direct vectorization can be much harder.

For this reason, it helps to have another approach to fast JAX implementations up our sleeves.

Here's a version that

1. writes the right hand side of the Bellman operator as a function of individual states and controls, and
2. applies `jax.vmap` on the outside to achieve a parallelized solution.

First let's rewrite `B`

```
def B(v, params, arrays, i, j, ip):
    """
    The right-hand side of the Bellman equation before maximization, which takes
    the form

        B(w, y, w') = u(Rw + y - w') + β Σ_y' v(w', y') Q(y, y')

    The indices are (i, j, ip) -> (w, y, w').
    """
    β, R, γ = params
    w_grid, y_grid, Q = arrays
    w, y, wp  = w_grid[i], y_grid[j], w_grid[ip]
    c = R * w + y - wp
    EV = jnp.sum(v[ip, :] * Q[j, :])
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Now we successively apply `vmap` to simulate nested loops.

```
B_1    = jax.vmap(B,   in_axes=(None, None, None, None, None, 0))
B_2    = jax.vmap(B_1, in_axes=(None, None, None, None, 0,    None))
B_vmap = jax.vmap(B_2, in_axes=(None, None, None, 0,    None, None))
```

Here's the Bellman operator and the `get_greedy` functions for the `vmap` case.

```
def T_vmap(v, params, sizes, arrays):
    "The Bellman operator."
    w_size, y_size = sizes
    w_indices, y_indices = jnp.arange(w_size), jnp.arange(y_size)
    B_values = B_vmap(v, params, arrays, w_indices, y_indices, w_indices)
    return jnp.max(B_values, axis=-1)

T_vmap = jax.jit(T_vmap, static_argnums=(2,))

def get_greedy_vmap(v, params, sizes, arrays):
    "Computes a v-greedy policy, returned as a set of indices."
    w_size, y_size = sizes
    w_indices, y_indices = jnp.arange(w_size), jnp.arange(y_size)
    B_values = B_vmap(v, params, arrays, w_indices, y_indices, w_indices)
    return jnp.argmax(B_values, axis=-1)

get_greedy_vmap = jax.jit(get_greedy_vmap, static_argnums=(2,))
```

Here's the iteration routine.

```
def value_iteration_vmap(model, tol=1e-5):
    params, sizes, arrays = model
    vz = jnp.zeros(sizes)
    _T = lambda v: T_vmap(v, params, sizes, arrays)
    v_star = successive_approx_jax(_T, vz, tolerance=tol)
    return v_star, get_greedy(v_star, params, sizes, arrays)
```

Let's see how long it takes to solve the model using the `vmap` method.

```
print("Starting VFI using vmap.")
start = time()
v_star_vmap, σ_star_vmap = value_iteration_vmap(model)
jax_vmap_with_compile = time() - start
print(f"VFI completed in {jax_vmap_with_compile} seconds.")
```

```
Starting VFI using vmap.
VFI completed in 0.48206210136413574 seconds.
```

Let's run it again to get rid of compile time.

```
start = time()
v_star_vmap, σ_star_vmap = value_iteration_vmap(model)
jax_vmap_without_compile = time() - start
print(f"VFI completed in {jax_vmap_without_compile} seconds.")
```

```
VFI completed in 0.3916950225830078 seconds.
```

We need to make sure that we got the same result.

```
print(jnp.allclose(v_star_vmap, v_star_jax))
print(jnp.allclose(σ_star_vmap, σ_star_jax))
```

```
True
True
```

Here's the speed gain associated with switching from the NumPy version to JAX with `vmap`:

```
print(f"Relative speed = {numpy_without_compile/jax_vmap_without_compile}")
```

```
Relative speed = 43.173873082036025
```

And here's the comparison with the first JAX implementation (which used direct vectorization).

```
print(f"Relative speed = {jax_without_compile / jax_vmap_without_compile}")
```

```
Relative speed = 0.9438908799626025
```

The execution times for the two JAX versions are relatively similar.

However, as emphasized above, having a second method up our sleeves (i.e, the `vmap` approach) will be helpful when confronting dynamic programs with more sophisticated Bellman equations.

CHAPTER

TWELVE

OPTIMAL SAVINGS II: ALTERNATIVE ALGORITHMS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In *Optimal Savings I: Value Function Iteration* we solved a simple version of the household optimal savings problem via value function iteration (VFI) using JAX.

In this lecture we tackle exactly the same problem while adding in two alternative algorithms:

- optimistic policy iteration (OPI) and
- Howard policy iteration (HPI).

We will see that both of these algorithms outperform traditional VFI.

One reason for this is that the algorithms have good convergence properties.

Another is that one of them, HPI, is particularly well suited to pairing with JAX.

The reason is that HPI uses a relatively small number of computationally expensive steps, whereas VFI uses a longer sequence of small steps.

In other words, VFI is inherently more sequential than HPI, and sequential routines are hard to parallelize.

By comparison, HPI is less sequential – the small number of computationally intensive steps can be effectively parallelized by JAX.

This is particularly valuable when the underlying hardware includes a GPU.

Details on VFI, HPI and OPI can be found in this book, for which a PDF is freely available.

Here we assume readers have some knowledge of the algorithms and focus on computation.

For the details of the savings model, readers can refer to *Optimal Savings I: Value Function Iteration*.

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

```
!pip install quantecon
```

We will use the following imports:

```
import quantecon as qe
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from time import time
```

Let's check the GPU we are running.

```
!nvidia-smi
```

```
Fri Oct  9 00:29:39 2026
+-----------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+---------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |              ␣
 ↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |              ␣
 ↪     0 |
| N/A   33C    P8              9W /   70W |       0MiB /  15360MiB |      0%      ␣
 ↪Default |
|                                         |                        |              ␣
 ↪   N/A |
+-----------------------------------------+------------------------+---------------
 ↪-------+

+-----------------------------------------------------------------------------------
 ↪-------+
| Processes:                                                                        ␣
 ↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
 ↪Memory |
|        ID   ID                                                               ␣
 ↪Usage      |
|===================================================================================|
|  No running processes found                                                       ␣
 ↪       |
+-----------------------------------------------------------------------------------
 ↪-------+
```

We'll use 64 bit floats to gain extra precision.

```
jax.config.update("jax_enable_x64", True)
```

12.1 Model primitives

First we define a model that stores parameters and grids.

The *following code* is repeated from *Optimal Savings I: Value Function Iteration*.

```
def create_consumption_model(R=1.01,                    # Gross interest rate
                             β=0.98,                    # Discount factor
                             γ=2,                       # CRRA parameter
                             w_min=0.01,                # Min wealth
                             w_max=5.0,                 # Max wealth
                             w_size=150,                # Grid side
                             ρ=0.9, ν=0.1, y_size=100): # Income parameters
    """
    A function that takes in parameters and returns parameters and grids
    for the optimal savings problem.
    """
    # Build grids and transition probabilities
    w_grid = jnp.linspace(w_min, w_max, w_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), mc.P
    # Pack and return
    params = β, R, γ
    sizes = w_size, y_size
    arrays = w_grid, y_grid, jnp.array(Q)
    return params, sizes, arrays
```

Here's the right hand side of the Bellman equation:

```
def _B(v, params, arrays, i, j, ip):
    """
    The right-hand side of the Bellman equation before maximization, which takes
    the form

        B(w, y, w') = u(Rw + y - w') + β Σ_y' v(w', y') Q(y, y')

    The indices are (i, j, ip) -> (w, y, w').
    """
    β, R, γ = params
    w_grid, y_grid, Q = arrays
    w, y, wp  = w_grid[i], y_grid[j], w_grid[ip]
    c = R * w + y - wp
    EV = jnp.sum(v[ip, :] * Q[j, :])
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Now we successively apply `vmap` to vectorize B by simulating nested loops.

```
B_1    = jax.vmap(_B,  in_axes=(None, None, None, None, None, 0))
B_2    = jax.vmap(B_1, in_axes=(None, None, None, None, 0,    None))
B_vmap = jax.vmap(B_2, in_axes=(None, None, None, 0,    None, None))
```

Here's a fully vectorized version of B.

```
def B(v, params, sizes, arrays):
    w_size, y_size = sizes
    w_indices, y_indices = jnp.arange(w_size), jnp.arange(y_size)
    return B_vmap(v, params, arrays, w_indices, y_indices, w_indices)
```

(continues on next page)

(continued from previous page)

```
B = jax.jit(B, static_argnums=(2,))
```

12.2 Operators

Here's the Bellman operator T

```
def T(v, params, sizes, arrays):
    "The Bellman operator."
    return jnp.max(B(v, params, sizes, arrays), axis=-1)

T = jax.jit(T, static_argnums=(2,))
```

The next function computes a v-greedy policy given v

```
def get_greedy(v, params, sizes, arrays):
    "Computes a v-greedy policy, returned as a set of indices."
    return jnp.argmax(B(v, params, sizes, arrays), axis=-1)

get_greedy = jax.jit(get_greedy, static_argnums=(2,))
```

We define a function to compute the current rewards r_σ given policy σ, which is defined as the vector

$$r_\sigma(w, y) := r(w, y, \sigma(w, y))$$

```
def _compute_r_σ(σ, params, arrays, i, j):
    """
    With indices (i, j) -> (w, y) and wp = σ[i, j], compute

        r_σ[i, j] = u(Rw + y - wp)

    which gives current rewards under policy σ.
    """

    # Unpack model
    β, R, γ = params
    w_grid, y_grid, Q = arrays
    # Compute r_σ[i, j]
    w, y, wp = w_grid[i], y_grid[j], w_grid[σ[i, j]]
    c = R * w + y - wp
    r_σ = c**(1-γ)/(1-γ)

    return r_σ
```

Now we successively apply `vmap` to simulate nested loops.

```
r_1 = jax.vmap(_compute_r_σ,  in_axes=(None, None, None, None, 0))
r_σ_vmap = jax.vmap(r_1,      in_axes=(None, None, None, 0,    None))
```

Here's a fully vectorized version of r_σ.

```
def compute_r_σ(σ, params, sizes, arrays):
    w_size, y_size = sizes
```

(continues on next page)

(continued from previous page)

```
    w_indices, y_indices = jnp.arange(w_size), jnp.arange(y_size)
    return r_σ_vmap(σ, params, arrays, w_indices, y_indices)

compute_r_σ = jax.jit(compute_r_σ, static_argnums=(2,))
```

Now we define the policy operator T_σ going through similar steps

```
def _T_σ(v, σ, params, arrays, i, j):
    "The σ-policy operator."

    # Unpack model
    β, R, γ = params
    w_grid, y_grid, Q = arrays

    r_σ  = _compute_r_σ(σ, params, arrays, i, j)
    # Calculate the expected sum Σ_jp v[σ[i, j], jp] * Q[i, j, jp]
    EV = jnp.sum(v[σ[i, j], :] * Q[j, :])

    return r_σ + β * EV


T_1 = jax.vmap(_T_σ,      in_axes=(None, None, None, None, None, 0))
T_σ_vmap = jax.vmap(T_1,  in_axes=(None, None, None, None, 0,    None))

def T_σ(v, σ, params, sizes, arrays):
    w_size, y_size = sizes
    w_indices, y_indices = jnp.arange(w_size), jnp.arange(y_size)
    return T_σ_vmap(v, σ, params, arrays, w_indices, y_indices)

T_σ = jax.jit(T_σ, static_argnums=(3,))
```

The function below computes the value v_σ of following policy σ.

This lifetime value is a function v_σ that satisfies

$$v_\sigma(w, y) = r_\sigma(w, y) + \beta \sum_{y'} v_\sigma(\sigma(w, y), y') Q(y, y')$$

We wish to solve this equation for v_σ.

Suppose we define the linear operator L_σ by

$$(L_\sigma v)(w, y) = v(w, y) - \beta \sum_{y'} v(\sigma(w, y), y') Q(y, y')$$

With this notation, the problem is to solve for v via

$$(L_\sigma v)(w, y) = r_\sigma(w, y)$$

In vector for this is $L_\sigma v = r_\sigma$, which tells us that the function we seek is

$$v_\sigma = L_\sigma^{-1} r_\sigma$$

JAX allows us to solve linear systems defined in terms of operators; the first step is to define the function L_σ.

```
def _L_σ(v, σ, params, arrays, i, j):
    """
```

(continues on next page)

(continued from previous page)

```
    Here we set up the linear map v -> L_σ v, where

        (L_σ v)(w, y) = v(w, y) - β Σ_y' v(σ(w, y), y') Q(y, y')

    """
    # Unpack
    β, R, γ = params
    w_grid, y_grid, Q = arrays
    # Compute and return v[i, j] - β Σ_jp v[σ[i, j], jp] * Q[j, jp]
    return v[i, j]  - β * jnp.sum(v[σ[i, j], :] * Q[j, :])

L_1 = jax.vmap(_L_σ,      in_axes=(None, None, None, None, None, 0))
L_σ_vmap = jax.vmap(L_1,  in_axes=(None, None, None, None, 0,    None))

def L_σ(v, σ, params, sizes, arrays):
    w_size, y_size = sizes
    w_indices, y_indices = jnp.arange(w_size), jnp.arange(y_size)
    return L_σ_vmap(v, σ, params, arrays, w_indices, y_indices)

L_σ = jax.jit(L_σ, static_argnums=(3,))
```

Now we can define a function to compute v_σ

```
def get_value(σ, params, sizes, arrays):
    "Get the value v_σ of policy σ by inverting the linear map L_σ."

    # Unpack
    β, R, γ = params
    w_size, y_size = sizes
    w_grid, y_grid, Q = arrays

    r_σ = compute_r_σ(σ, params, sizes, arrays)

    # Reduce L_σ to a function in v
    partial_L_σ = lambda v: L_σ(v, σ, params, sizes, arrays)

    return jax.scipy.sparse.linalg.bicgstab(partial_L_σ, r_σ)[0]

get_value = jax.jit(get_value, static_argnums=(2,))
```

12.3 Iteration

We use successive approximation for VFI.

```
def successive_approx_jax(T,                        # Operator (callable)
                          x_0,                      # Initial condition
                          tol=1e-6,                 # Error tolerance
                          max_iter=10_000):         # Max iteration bound
    def update(inputs):
        k, x, error = inputs
        x_new = T(x)
        error = jnp.max(jnp.abs(x_new - x))
        return k + 1, x_new, error
```

(continues on next page)

(continued from previous page)

```
    def condition_function(inputs):
        k, x, error = inputs
        return jnp.logical_and(error > tol, k < max_iter)

    k, x, error = jax.lax.while_loop(condition_function,
                                     update,
                                     (1, x_0, tol + 1))
    return x

successive_approx_jax = jax.jit(successive_approx_jax, static_argnums=(0,))
```

For OPI we'll add a compiled routine that computes $T_\sigma^m v$.

```
def iterate_policy_operator(σ, v, m, params, sizes, arrays):

    def update(i, v):
        v = T_σ(v, σ, params, sizes, arrays)
        return v

    v = jax.lax.fori_loop(0, m, update, v)
    return v

iterate_policy_operator = jax.jit(iterate_policy_operator,
                                  static_argnums=(4,))
```

12.4 Solvers

Now we define the solvers, which implement VFI, HPI and OPI.

Here's VFI.

```
def value_function_iteration(model, tol=1e-5):
    """
    Implements value function iteration.
    """
    params, sizes, arrays = model
    vz = jnp.zeros(sizes)
    _T = lambda v: T(v, params, sizes, arrays)
    v_star = successive_approx_jax(_T, vz, tol=tol)
    return get_greedy(v_star, params, sizes, arrays)
```

For OPI we will use a compiled JAX `lax.while_loop` operation to speed execution.

```
def opi_loop(params, sizes, arrays, m, tol, max_iter):
    """
    Implements optimistic policy iteration (see dp.quantecon.org) with
    step size m.

    """
    v_init = jnp.zeros(sizes)

    def condition_function(inputs):
        i, v, error = inputs
        return jnp.logical_and(error > tol, i < max_iter)
```

(continues on next page)

(continued from previous page)

```
    def update(inputs):
        i, v, error = inputs
        last_v = v
        σ = get_greedy(v, params, sizes, arrays)
        v = iterate_policy_operator(σ, v, m, params, sizes, arrays)
        error = jnp.max(jnp.abs(v - last_v))
        i += 1
        return i, v, error

    num_iter, v, error = jax.lax.while_loop(condition_function,
                                            update,
                                            (0, v_init, tol + 1))

    return get_greedy(v, params, sizes, arrays)

opi_loop = jax.jit(opi_loop, static_argnums=(1,))
```

Here's a friendly interface to OPI

```
def optimistic_policy_iteration(model, m=10, tol=1e-5, max_iter=10_000):
    params, sizes, arrays = model
    σ_star = opi_loop(params, sizes, arrays, m, tol, max_iter)
    return σ_star
```

Here's HPI.

```
def howard_policy_iteration(model, maxiter=250):
    """
    Implements Howard policy iteration (see dp.quantecon.org)
    """
    params, sizes, arrays = model
    σ = jnp.zeros(sizes, dtype=int)
    i, error = 0, 1.0
    while error > 0 and i < maxiter:
        v_σ = get_value(σ, params, sizes, arrays)
        σ_new = get_greedy(v_σ, params, sizes, arrays)
        error = jnp.max(jnp.abs(σ_new - σ))
        σ = σ_new
        i = i + 1
        print(f"Concluded loop {i} with error {error}.")
    return σ
```

12.5 Tests

Let's create a model for consumption, and plot the resulting optimal policy function using all the three algorithms and also check the time taken by each solver.

```
model = create_consumption_model()
# Unpack
params, sizes, arrays = model
β, R, γ = params
w_size, y_size = sizes
w_grid, y_grid, Q = arrays
```

```
print("Starting HPI.")
start = time()
σ_star_hpi = howard_policy_iteration(model).block_until_ready()
hpi_with_compile = time() - start
print(f"HPI completed in {hpi_with_compile} seconds.")
```

```
Starting HPI.
Concluded loop 1 with error 77.
Concluded loop 2 with error 53.
Concluded loop 3 with error 28.
Concluded loop 4 with error 17.
Concluded loop 5 with error 8.
Concluded loop 6 with error 4.
Concluded loop 7 with error 1.
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
Concluded loop 10 with error 0.
HPI completed in 0.9452497959136963 seconds.
```

We run it again to get rid of compile time.

```
start = time()
σ_star_hpi = howard_policy_iteration(model).block_until_ready()
hpi_without_compile = time() - start
print(f"HPI completed in {hpi_without_compile} seconds.")
```

```
Concluded loop 1 with error 77.
Concluded loop 2 with error 53.
Concluded loop 3 with error 28.
Concluded loop 4 with error 17.
Concluded loop 5 with error 8.
Concluded loop 6 with error 4.
Concluded loop 7 with error 1.
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
Concluded loop 10 with error 0.
HPI completed in 0.13255095481872559 seconds.
```

```
fig, ax = plt.subplots()
ax.plot(w_grid, w_grid, "k--", label="45")
ax.plot(w_grid, w_grid[σ_star_hpi[:, 1]], label="$\\sigma^{*}_{HPI}(\cdot, y_1)$")
ax.plot(w_grid, w_grid[σ_star_hpi[:, -1]], label="$\\sigma^{*}_{HPI}(\cdot, y_N)$")
ax.legend()
plt.show()
```

```
print("Starting VFI.")
start = time()
σ_star_vfi = value_function_iteration(model).block_until_ready()
vfi_with_compile = time() - start
print(f"VFI completed in {vfi_with_compile} seconds.")
```

```
Starting VFI.
VFI completed in 0.50162672996521 seconds.
```

We run it again to eliminate compile time.

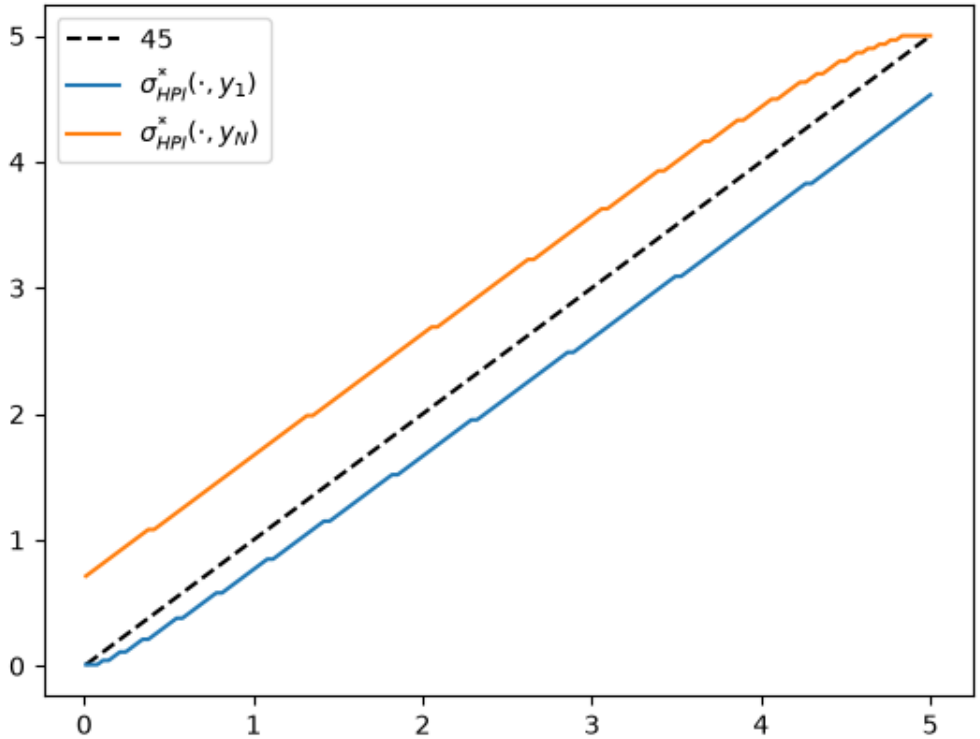


Fig. 12.1: Optimal policy function (HPI)

```
start = time()
σ_star_vfi = value_function_iteration(model).block_until_ready()
vfi_without_compile = time() - start
print(f"VFI completed in {vfi_without_compile} seconds.")
```

```
VFI completed in 0.3915982246398926 seconds.
```

```
fig, ax = plt.subplots()
ax.plot(w_grid, w_grid, "k--", label="45")
ax.plot(w_grid, w_grid[σ_star_vfi[:, 1]], label="$\\sigma^{*}_{VFI}(\cdot, y_1)$")
ax.plot(w_grid, w_grid[σ_star_vfi[:, -1]], label="$\\sigma^{*}_{VFI}(\cdot, y_N)$")
ax.legend()
plt.show()
```

```
print("Starting OPI.")
start = time()
σ_star_opi = optimistic_policy_iteration(model, m=100).block_until_ready()
opi_with_compile = time() - start
print(f"OPI completed in {opi_with_compile} seconds.")
```

```
Starting OPI.
OPI completed in 0.4397850036621094 seconds.
```

Let's run it again to get rid of compile time.

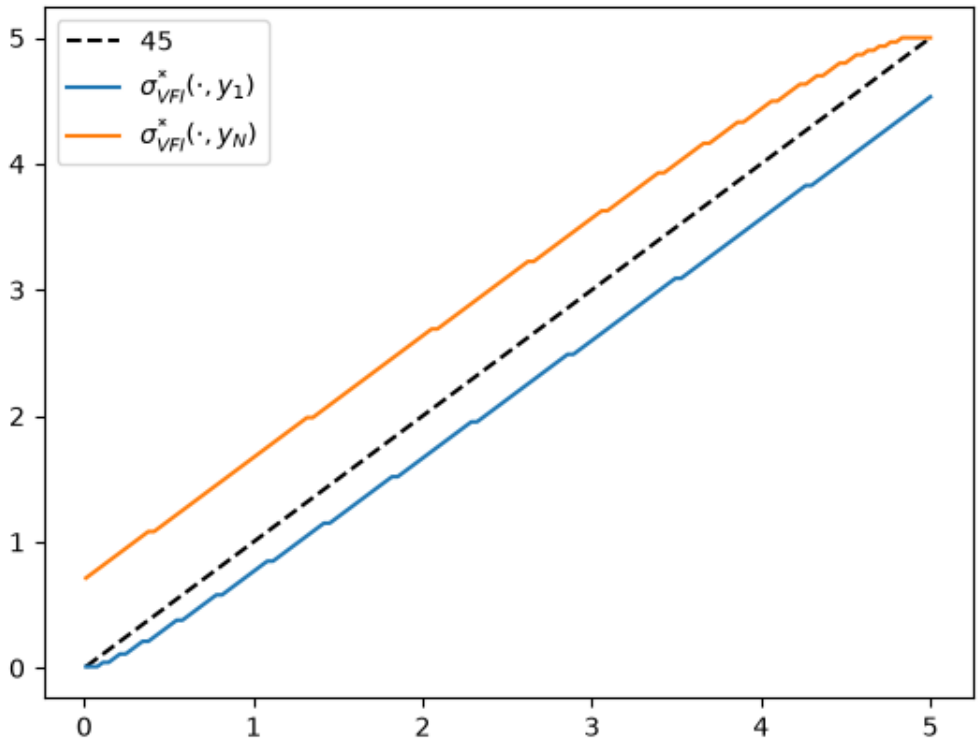


Fig. 12.2: Optimal policy function (VFI)

```
start = time()
σ_star_opi = optimistic_policy_iteration(model, m=100).block_until_ready()
opi_without_compile = time() - start
print(f"OPI completed in {opi_without_compile} seconds.")
```

```
OPI completed in 0.126204252243042 seconds.
```

```
fig, ax = plt.subplots()
ax.plot(w_grid, w_grid, "k--", label="45")
ax.plot(w_grid, w_grid[σ_star_opi[:, 1]], label="$\\sigma^{*}_{OPI}(\cdot, y_1)$")
ax.plot(w_grid, w_grid[σ_star_opi[:, -1]], label="$\\sigma^{*}_{OPI}(\cdot, y_N)$")
ax.legend()
plt.show()
```

We observe that all the solvers produce the same output from the above three plots.

Now, let's create a plot to visualize the time differences among these algorithms.

```
def run_algorithm(algorithm, model, **kwargs):
    result = algorithm(model, **kwargs)

    # Now time it without compile time
    start = time()
    result = algorithm(model, **kwargs).block_until_ready()
    algorithm_without_compile = time() - start
```

(continues on next page)

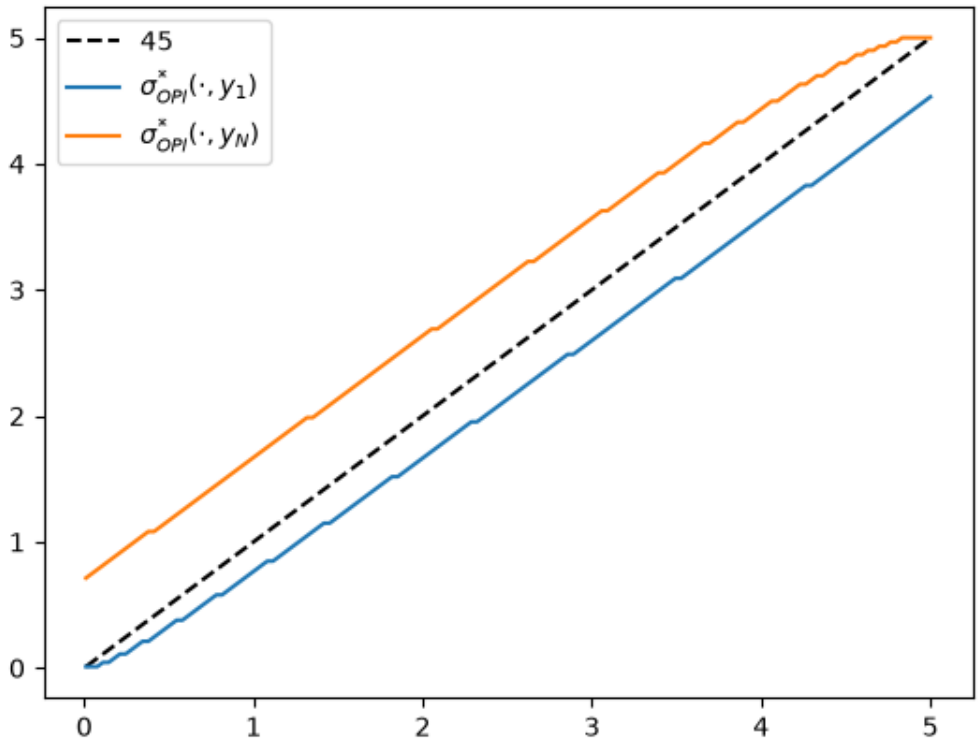


Fig. 12.3: Optimal policy function (OPI)

(continued from previous page)

```
    print(f"{algorithm.__name__} completed in {algorithm_without_compile:.2f} seconds.
↪")
    return result, algorithm_without_compile
```

```
σ_pi, pi_time = run_algorithm(howard_policy_iteration, model)
σ_vfi, vfi_time = run_algorithm(value_function_iteration, model, tol=1e-5)
```

```
Concluded loop 1 with error 77.
Concluded loop 2 with error 53.
Concluded loop 3 with error 28.
Concluded loop 4 with error 17.
Concluded loop 5 with error 8.
Concluded loop 6 with error 4.
Concluded loop 7 with error 1.
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
Concluded loop 10 with error 0.
Concluded loop 1 with error 77.
Concluded loop 2 with error 53.
Concluded loop 3 with error 28.
Concluded loop 4 with error 17.
Concluded loop 5 with error 8.
Concluded loop 6 with error 4.
Concluded loop 7 with error 1.
Concluded loop 8 with error 1.
```

(continues on next page)

(continued from previous page)

```
Concluded loop 9 with error 1.
Concluded loop 10 with error 0.
howard_policy_iteration completed in 0.08 seconds.
value_function_iteration completed in 0.39 seconds.
```

```
m_vals = range(5, 600, 40)
opi_times = []
for m in m_vals:
    σ_opi, opi_time = run_algorithm(optimistic_policy_iteration,
                                    model, m=m, tol=1e-5)
    opi_times.append(opi_time)
```

```
optimistic_policy_iteration completed in 0.38 seconds.
optimistic_policy_iteration completed in 0.12 seconds.
optimistic_policy_iteration completed in 0.12 seconds.
optimistic_policy_iteration completed in 0.14 seconds.
optimistic_policy_iteration completed in 0.17 seconds.
optimistic_policy_iteration completed in 0.21 seconds.
optimistic_policy_iteration completed in 0.24 seconds.
optimistic_policy_iteration completed in 0.28 seconds.
optimistic_policy_iteration completed in 0.31 seconds.
optimistic_policy_iteration completed in 0.35 seconds.
optimistic_policy_iteration completed in 0.38 seconds.
optimistic_policy_iteration completed in 0.42 seconds.
optimistic_policy_iteration completed in 0.45 seconds.
optimistic_policy_iteration completed in 0.49 seconds.
optimistic_policy_iteration completed in 0.53 seconds.
```

```
fig, ax = plt.subplots()
ax.plot(m_vals,
        jnp.full(len(m_vals), hpi_without_compile),
        lw=2, label="Howard policy iteration")
ax.plot(m_vals,
        jnp.full(len(m_vals), vfi_without_compile),
        lw=2, label="value function iteration")
ax.plot(m_vals, opi_times,
        lw=2, label="optimistic policy iteration")
ax.legend(frameon=False, loc='lower right')
ax.set_xlabel("$m$")
ax.set_ylabel("time")
plt.show()
```

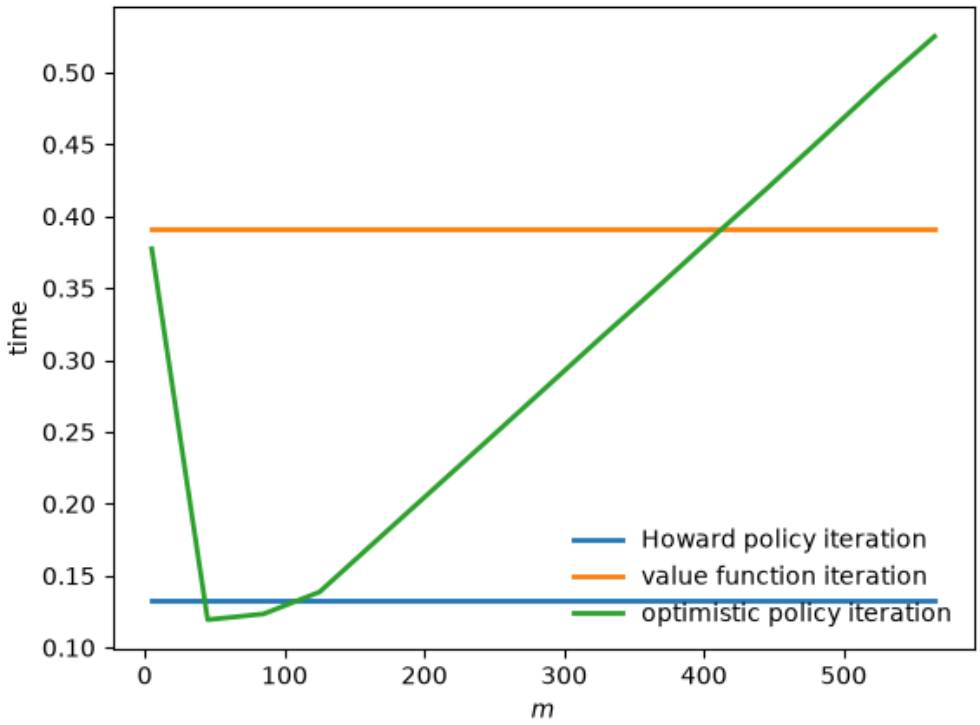


Fig. 12.4: Solver times

CHAPTER

THIRTEEN

SHORTEST PATHS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

13.1 Overview

This lecture is the extended version of the shortest path lecture using JAX. Please see that lecture for all background and notation.

Let's start by importing the libraries.

```
import numpy as np
import jax.numpy as jnp
import jax
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:31:30 2026
+-----------------------------------------------------------------------------------------
↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
↪13.0     |
+-----------------------------------------+------------------------+----------------
↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
↪Compute M. |
|                                         |                        |               ␣
↪MIG M. |
|=========================================+========================+======================|
```

(continues on next page)

(continued from previous page)

```
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                  ␣
↪     0 |
| N/A   41C    P0             28W /   70W |       0MiB /  15360MiB |      0%      ␣
↪Default |
|                                         |                        |                  ␣
↪   N/A |
+-----------------------------------------+------------------------+---------------
↪-------+

+-----------------------------------------------------------------------------------
↪-------+
| Processes:                                                                        ␣
↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
↪Memory |
|        ID   ID                                                               ␣
↪Usage      |
|=====================================================================================|
|  No running processes found                                                       ␣
↪       |
+-----------------------------------------------------------------------------------
↪-------+
```

13.2 Solving for Minimum Cost-to-Go

Let $J(v)$ denote the minimum cost-to-go from node v, understood as the total cost from v if we take the best route.

Let's look at an algorithm for computing J and then think about how to implement it.

13.2.1 The Algorithm

The standard algorithm for finding J is to start an initial guess and then iterate.

This is a standard approach to solving nonlinear equations, often called the method of **successive approximations**.

Our initial guess will be

$$J_0(v) = 0 \text{ for all } v \tag{13.1}$$

Now

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v} \{c(v, w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n, go to 2

This sequence converges to J.

Let's start by defining the **distance matrix** Q.

```
inf = jnp.inf
Q = jnp.array([[inf, 1,   5,   3,   inf, inf, inf],
               [inf, inf, inf, 9,   6,   inf, inf],
               [inf, inf, inf, inf, inf, 2,   inf],
```

(continues on next page)

(continued from previous page)

```
               [inf, inf, inf, inf, inf, 4,   8],
               [inf, inf, inf, inf, inf, inf, 4],
               [inf, inf, inf, inf, inf, inf, 1],
               [inf, inf, inf, inf, inf, inf, 0]])
```

Notice that the cost of staying still (on the principle diagonal) is set to

- `jnp.inf` for non-destination nodes — moving on is required.
- `0` for the destination node — here is where we stop.

Let's try with this example using python `while` loop and some `jax` vectorized code:

```
%%time

num_nodes = Q.shape[0]
J = jnp.zeros(num_nodes)

max_iter = 500
i = 0

while i < max_iter:
    next_J = jnp.min(Q + J, axis=1)
    if jnp.allclose(next_J, J):
        break
    else:
        J = next_J.copy()
        i += 1

print("The cost-to-go function is", J)
```

```
The cost-to-go function is [ 8. 10.  3.  5.  4.  1.  0.]
CPU times: user 149 ms, sys: 11.3 ms, total: 160 ms
Wall time: 245 ms
```

We can further optimize the above code by using jax.lax.while_loop. The extra acceleration is due to the fact that the entire operation can be optimized by the JAX compiler and launched as a single kernel on the GPU.

```
max_iter = 500
num_nodes = Q.shape[0]
J = jnp.zeros(num_nodes)
```

```
def body_fun(values):
    # Define the body function of while loop
    i, J, break_cond = values

    # Update J and break condition
    next_J = jnp.min(Q + J, axis=1)
    break_condition = jnp.allclose(next_J, J)

    # Return next iteration values
    return i + 1, next_J, break_condition
```

```
def cond_fun(values):
    i, J, break_condition = values
    return ~break_condition & (i < max_iter)
```

Let's see the timing for JIT compilation of the functions and runtime results.

```
%%time
jax.lax.while_loop(cond_fun, body_fun, init_val=(0, J, False))[1].block_until_ready()
```

```
CPU times: user 109 ms, sys: 7.43 ms, total: 117 ms
Wall time: 114 ms
```

```
Array([ 8., 10.,  3.,  5.,  4.,  1.,  0.], dtype=float32)
```

Now, this runs faster once we have the JIT compiled JAX version of the functions.

```
%%time
jax.lax.while_loop(cond_fun, body_fun, init_val=(0, J, False))[1].block_until_ready()
```

```
CPU times: user 1.48 ms, sys: 199 µs, total: 1.68 ms
Wall time: 1.02 ms
```

```
Array([ 8., 10.,  3.,  5.,  4.,  1.,  0.], dtype=float32)
```

Note

Large speed gains while using `jax.lax.while_loop` won't be realized unless the shortest path problem is relatively large.

13.3 Exercises

Exercise 13.3.1

The text below describes a weighted directed graph.

The line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04
- node8 at cost 11.11
- node14 at cost 72.21

No other nodes can be reached directly from node0.

Other lines have a similar interpretation.

Your task is to use the algorithm given above to find the optimal path and its cost.

```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
```

```
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
```

```
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Writing graph.txt
```

Solution

First let's write a function that reads in the graph data above and builds a distance matrix.

```
num_nodes = 100
destination_node = 99
def map_graph_to_distance_matrix(in_file):

    # First let's set of the distance matrix Q with inf everywhere
    Q = np.full((num_nodes, num_nodes), np.inf)

    # Now we read in the data and modify Q
    with open(in_file) as infile:
        for line in infile:
            elements = line.split(',')
            node = elements.pop(0)
            node = int(node[4:])    # convert node description to integer
            if node != destination_node:
```

```
                for element in elements:
                    destination, cost = element.split()
                    destination = int(destination[4:])
                    Q[node, destination] = float(cost)
            Q[destination_node, destination_node] = 0
    return jnp.array(Q)
```

Let's write a function `compute_cost_to_go` that returns J given any valid Q.

```
@jax.jit
def compute_cost_to_go(Q):
    num_nodes = Q.shape[0]
    J = jnp.zeros(num_nodes)       # Initial guess
    max_iter = 500
    i = 0

    def body_fun(values):
        # Define the body function of while loop
        i, J, break_cond = values

        # Update J and break condition
        next_J = jnp.min(Q + J, axis=1)
        break_condition = jnp.allclose(next_J, J)

        # Return next iteration values
        return i + 1, next_J, break_condition

    def cond_fun(values):
        i, J, break_condition = values
        return ~break_condition & (i < max_iter)

    return jax.lax.while_loop(cond_fun, body_fun,
                              init_val=(0, J, False))[1]
```

Finally, here's a function that uses the `cost-to-go` function to obtain the optimal path (and its cost).

```
def print_best_path(J, Q):
    sum_costs = 0
    current_node = 0
    while current_node != destination_node:
        print(current_node)
        # Move to the next node and increment costs
        next_node = jnp.argmin(Q[current_node, :] + J)
        sum_costs += Q[current_node, next_node]
        current_node = next_node
    print(destination_node)
    print('Cost: ', sum_costs)
```

Okay, now we have the necessary functions, let's call them to do the job we were assigned.

```
Q = map_graph_to_distance_matrix('graph.txt')
```

Let's see the timings for jitting the function and runtime results.

```
%%time
J = compute_cost_to_go(Q).block_until_ready()
```

```
CPU times: user 136 ms, sys: 8.56 ms, total: 145 ms
Wall time: 136 ms
```

Let's run again to eliminate compile time.

```
%%time
J = compute_cost_to_go(Q).block_until_ready()
```

```
CPU times: user 1.34 ms, sys: 89 µs, total: 1.43 ms
Wall time: 1.12 ms
```

```
print_best_path(J, Q)
```

```
0
8
11
18
23
33
41
53
56
57
60
67
70
73
76
85
87
88
93
94
96
97
98
99
Cost:  160.55
```

The total cost of the path should agree with $J[0]$ so let's check this.

```
J[0].item()
```

```
160.5500030517578
```

CHAPTER

FOURTEEN

OPTIMAL INVESTMENT

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to JAX and Anaconda, this lecture will need the following libraries:

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

We study a monopolist who faces inverse demand curve

$$P_t = a_0 - a_1 Y_t + Z_t,$$

where

- P_t is price,
- Y_t is output and
- Z_t is a demand shock.

We assume that Z_t is a discretized AR(1) process, specified below.

Current profits are

$$P_t Y_t - cY_t - \gamma(Y_{t+1} - Y_t)^2$$

Combining with the demand curve and writing y, y' for Y_t, Y_{t+1}, this becomes

$$r(y, z, y') := (a_0 - a_1 y + z - c)y - \gamma(y' - y)^2$$

The firm maximizes present value of expected discounted profits. The Bellman equation is

$$v(y, z) = \max_{y'} \left\{ r(y, z, y') + \beta \sum_{z'} v(y', z') Q(z, z') \right\}.$$

We discretize y to a finite grid `y_grid`.

In essence, the firm tries to choose output close to the monopolist profit maximizer, given Z_t, but is constrained by adjustment costs.

Let's begin with the following imports

```
import quantecon as qe
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from time import time
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:28:34 2026
+-----------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+---------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |              ␣
 ↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |              ␣
 ↪     0 |
| N/A   35C    P8              9W /   70W |       0MiB /  15360MiB |      0%      ␣
 ↪Default |
|                                         |                        |              ␣
 ↪   N/A |
+-----------------------------------------+------------------------+---------------
 ↪-------+

+-----------------------------------------------------------------------------------
 ↪-------+
| Processes:                                                                       ␣
 ↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
 ↪Memory |
|        ID   ID                                                               ␣
 ↪Usage      |
|=================================================================================================|
|  No running processes found                                                      ␣
 ↪       |
+-----------------------------------------------------------------------------------
 ↪-------+
```

We will use 64 bit floats with JAX in order to increase the precision.

```
jax.config.update("jax_enable_x64", True)
```

Let's define a function to create an investment model using the given parameters.

```
def create_investment_model(
        r=0.01,                              # Interest rate
        a_0=10.0, a_1=1.0,                   # Demand parameters
        γ=25.0, c=1.0,                       # Adjustment and unit cost
        y_min=0.0, y_max=20.0, y_size=100,   # Grid for output
        ρ=0.9, ν=1.0,                        # AR(1) parameters
        z_size=150):                         # Grid size for shock
    """
    A function that takes in parameters and returns an instance of Model that
    contains data for the investment problem.
    """
    β = 1 / (1 + r)
    y_grid = jnp.linspace(y_min, y_max, y_size)
    mc = qe.tauchen(z_size, ρ, ν)
    z_grid, Q = mc.state_values, mc.P

    # Break up parameters into static and nonstatic components
    constants = β, a_0, a_1, γ, c
    sizes = y_size, z_size
    arrays = y_grid, z_grid, Q

    # Shift arrays to the device (e.g., GPU)
    arrays = tuple(map(jax.device_put, arrays))
    return constants, sizes, arrays
```

Let's re-write the vectorized version of the right-hand side of the Bellman equation (before maximization), which is a 3D array representing

$$B(y, z, y') = r(y, z, y') + \beta \sum_{z'} v(y', z') Q(z, z')$$

for all (y, z, y').

```
def B(v, constants, sizes, arrays):
    """
    A vectorized version of the right-hand side of the Bellman equation
    (before maximization)
    """

    # Unpack
    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
    y_grid, z_grid, Q = arrays

    # Compute current rewards r(y, z, yp) as array r[i, j, ip]
    y  = jnp.reshape(y_grid, (y_size, 1, 1))     # y[i]   ->  y[i, j, ip]
    z  = jnp.reshape(z_grid, (1, z_size, 1))     # z[j]   ->  z[i, j, ip]
    yp = jnp.reshape(y_grid, (1, 1, y_size))     # yp[ip] -> yp[i, j, ip]
    r = (a_0 - a_1 * y + z - c) * y - γ * (yp - y)**2

    # Calculate continuation rewards at all combinations of (y, z, yp)
    v = jnp.reshape(v, (1, 1, y_size, z_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Q = jnp.reshape(Q, (1, z_size, 1, z_size))  # Q[j, jp]  -> Q[i, j, ip, jp]
    EV = jnp.sum(v * Q, axis=3)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return r + β * EV
```

(continues on next page)

(continued from previous page)

```
# Create a jitted function
B = jax.jit(B, static_argnums=(2,))
```

We define a function to compute the current rewards r_σ given policy σ, which is defined as the vector

$$r_\sigma(y, z) := r(y, z, \sigma(y, z))$$

```
def compute_r_σ(σ, constants, sizes, arrays):
    """
    Compute the array r_σ[i, j] = r[i, j, σ[i, j]], which gives current
    rewards given policy σ.
    """

    # Unpack model
    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
    y_grid, z_grid, Q = arrays

    # Compute r_σ[i, j]
    y = jnp.reshape(y_grid, (y_size, 1))
    z = jnp.reshape(z_grid, (1, z_size))
    yp = y_grid[σ]
    r_σ = (a_0 - a_1 * y + z - c) * y - γ * (yp - y)**2

    return r_σ


# Create the jitted function
compute_r_σ = jax.jit(compute_r_σ, static_argnums=(2,))
```

Define the Bellman operator.

```
def T(v, constants, sizes, arrays):
    """The Bellman operator."""
    return jnp.max(B(v, constants, sizes, arrays), axis=2)

T = jax.jit(T, static_argnums=(2,))
```

The following function computes a v-greedy policy.

```
def get_greedy(v, constants, sizes, arrays):
    "Computes a v-greedy policy, returned as a set of indices."
    return jnp.argmax(B(v, constants, sizes, arrays), axis=2)

get_greedy = jax.jit(get_greedy, static_argnums=(2,))
```

Define the σ-policy operator.

```
def T_σ(v, σ, constants, sizes, arrays):
    """The σ-policy operator."""

    # Unpack model
    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
    y_grid, z_grid, Q = arrays
```

(continues on next page)

(continued from previous page)

```
    r_σ = compute_r_σ(σ, constants, sizes, arrays)

    # Compute the array v[σ[i, j], jp]
    zp_idx = jnp.arange(z_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, z_size))
    σ = jnp.reshape(σ, (y_size, z_size, 1))
    V = v[σ, zp_idx]

    # Convert Q[j, jp] to Q[i, j, jp]
    Q = jnp.reshape(Q, (1, z_size, z_size))

    # Calculate the expected sum Σ_jp v[σ[i, j], jp] * Q[i, j, jp]
    Ev = jnp.sum(V * Q, axis=2)

    return r_σ + β * Ev

T_σ = jax.jit(T_σ, static_argnums=(3,))
```

Next, we want to computes the lifetime value of following policy σ.

This lifetime value is a function v_σ that satisfies

$$v_\sigma(y, z) = r_\sigma(y, z) + \beta \sum_{z'} v_\sigma(\sigma(y, z), z') Q(z, z')$$

We wish to solve this equation for v_σ.

Suppose we define the linear operator L_σ by

$$(L_\sigma v)(y, z) = v(y, z) - \beta \sum_{z'} v(\sigma(y, z), z') Q(z, z')$$

With this notation, the problem is to solve for v via

$$(L_\sigma v)(y, z) = r_\sigma(y, z)$$

In vector for this is $L_\sigma v = r_\sigma$, which tells us that the function we seek is

$$v_\sigma = L_\sigma^{-1} r_\sigma$$

JAX allows us to solve linear systems defined in terms of operators; the first step is to define the function L_σ.

```
def L_σ(v, σ, constants, sizes, arrays):

    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
    y_grid, z_grid, Q = arrays

    # Set up the array v[σ[i, j], jp]
    zp_idx = jnp.arange(z_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, z_size))
    σ = jnp.reshape(σ, (y_size, z_size, 1))
    V = v[σ, zp_idx]

    # Expand Q[j, jp] to Q[i, j, jp]
    Q = jnp.reshape(Q, (1, z_size, z_size))

    # Compute and return v[i, j] - β Σ_jp v[σ[i, j], jp] * Q[j, jp]
```

(continues on next page)

(continued from previous page)

```
    return v - β * jnp.sum(V * Q, axis=2)

L_σ = jax.jit(L_σ, static_argnums=(3,))
```

Now we can define a function to compute v_σ

```
def get_value(σ, constants, sizes, arrays):

    # Unpack
    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
    y_grid, z_grid, Q = arrays

    r_σ = compute_r_σ(σ, constants, sizes, arrays)

    # Reduce L_σ to a function in v
    partial_L_σ = lambda v: L_σ(v, σ, constants, sizes, arrays)

    return jax.scipy.sparse.linalg.bicgstab(partial_L_σ, r_σ)[0]

get_value = jax.jit(get_value, static_argnums=(2,))
```

We use successive approximation for VFI.

```
def successive_approx_jax(T,                      # Operator (callable)
                          x_0,                    # Initial condition
                          tol=1e-6,               # Error tolerance
                          max_iter=10_000):       # Max iteration bound
    def body_fun(k_x_err):
        k, x, error = k_x_err
        x_new = T(x)
        error = jnp.max(jnp.abs(x_new - x))
        return k + 1, x_new, error

    def cond_fun(k_x_err):
        k, x, error = k_x_err
        return jnp.logical_and(error > tol, k < max_iter)

    k, x, error = jax.lax.while_loop(cond_fun, body_fun, (1, x_0, tol + 1))
    return x

successive_approx_jax = jax.jit(successive_approx_jax, static_argnums=(0,))
```

For OPI we'll add a compiled routine that computes $T_\sigma^m v$.

```
def iterate_policy_operator(σ, v, m, params, sizes, arrays):

    def update(i, v):
        v = T_σ(v, σ, params, sizes, arrays)
        return v

    v = jax.lax.fori_loop(0, m, update, v)
    return v

iterate_policy_operator = jax.jit(iterate_policy_operator,
                                  static_argnums=(4,))
```

Finally, we introduce the solvers that implement VFI, HPI and OPI.

```
def value_function_iteration(model, tol=1e-5):
    """
    Implements value function iteration.
    """
    params, sizes, arrays = model
    vz = jnp.zeros(sizes)
    _T = lambda v: T(v, params, sizes, arrays)
    v_star = successive_approx_jax(_T, vz, tol=tol)
    return get_greedy(v_star, params, sizes, arrays)
```

For OPI we will use a compiled JAX `lax.while_loop` operation to speed execution.

```
def opi_loop(params, sizes, arrays, m, tol, max_iter):
    """
    Implements optimistic policy iteration (see dp.quantecon.org) with
    step size m.

    """
    v_init = jnp.zeros(sizes)

    def condition_function(inputs):
        i, v, error = inputs
        return jnp.logical_and(error > tol, i < max_iter)

    def update(inputs):
        i, v, error = inputs
        last_v = v
        σ = get_greedy(v, params, sizes, arrays)
        v = iterate_policy_operator(σ, v, m, params, sizes, arrays)
        error = jnp.max(jnp.abs(v - last_v))
        i += 1
        return i, v, error

    num_iter, v, error = jax.lax.while_loop(condition_function,
                                            update,
                                            (0, v_init, tol + 1))

    return get_greedy(v, params, sizes, arrays)

opi_loop = jax.jit(opi_loop, static_argnums=(1,))
```

Here's a friendly interface to OPI

```
def optimistic_policy_iteration(model, m=10, tol=1e-5, max_iter=10_000):
    params, sizes, arrays = model
    σ_star = opi_loop(params, sizes, arrays, m, tol, max_iter)
    return σ_star
```

Here's HPI

```
def howard_policy_iteration(model, maxiter=250):
    """
    Implements Howard policy iteration (see dp.quantecon.org)
    """
    params, sizes, arrays = model
    σ = jnp.zeros(sizes, dtype=int)
```

(continues on next page)

(continued from previous page)

```
    i, error = 0, 1.0
    while error > 0 and i < maxiter:
        v_σ = get_value(σ, params, sizes, arrays)
        σ_new = get_greedy(v_σ, params, sizes, arrays)
        error = jnp.max(jnp.abs(σ_new - σ))
        σ = σ_new
        i = i + 1
        print(f"Concluded loop {i} with error {error}.")
    return σ
```

```
model = create_investment_model()
constants, sizes, arrays = model
β, a_0, a_1, γ, c = constants
y_size, z_size = sizes
y_grid, z_grid, Q = arrays
```

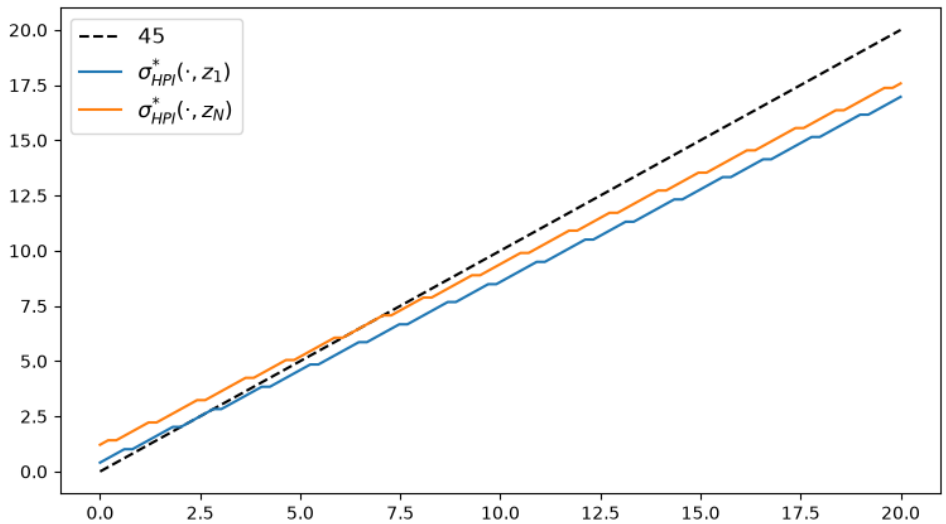
```
print("Starting HPI.")
%time σ_star_hpi = howard_policy_iteration(model).block_until_ready()
```

```
# Now time it without compile time
start = time()
σ_star_hpi = howard_policy_iteration(model).block_until_ready()
hpi_without_compile = time() - start
print(σ_star_hpi)
print(f"HPI completed in {hpi_without_compile} seconds.")
```

```
Concluded loop 1 with error 50.
Concluded loop 2 with error 26.
Concluded loop 3 with error 17.
Concluded loop 4 with error 10.
Concluded loop 5 with error 7.
Concluded loop 6 with error 4.
Concluded loop 7 with error 3.
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
Concluded loop 10 with error 1.
Concluded loop 11 with error 1.
Concluded loop 12 with error 0.
[[ 2  2  2 ...  6  6  6]
 [ 3  3  3 ...  7  7  7]
 [ 4  4  4 ...  7  7  7]
 ...
 [82 82 82 ... 86 86 86]
 [83 83 83 ... 86 86 86]
 [84 84 84 ... 87 87 87]]
HPI completed in 0.14902281761169434 seconds.
```

Here's the plot of the Howard policy, as a function of y at the highest and lowest values of z.

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(y_grid, y_grid, "k--", label="45")
ax.plot(y_grid, y_grid[σ_star_hpi[:, 1]], label="$\\sigma^{*}_{HPI}(\cdot, z_1)$")
ax.plot(y_grid, y_grid[σ_star_hpi[:, -1]], label="$\\sigma^{*}_{HPI}(\cdot, z_N)$")
ax.legend(fontsize=12)
plt.show()
```



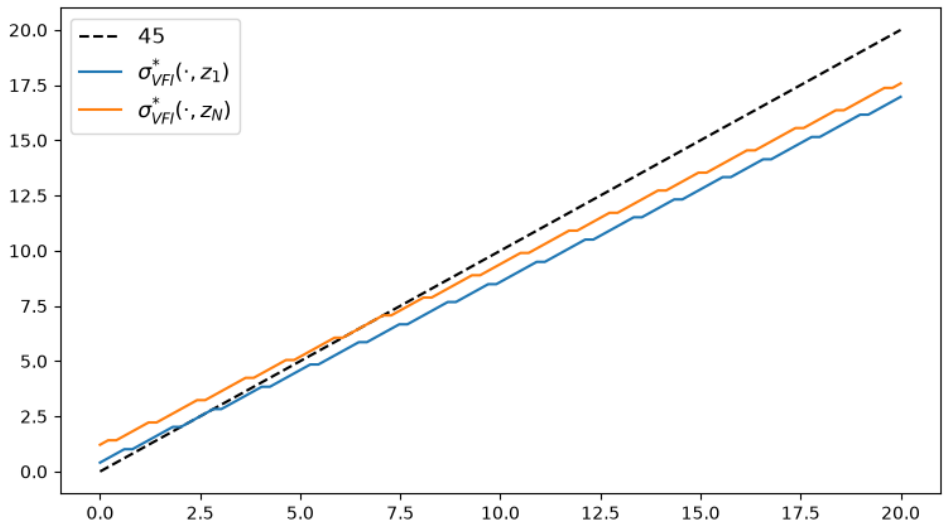

```
print("Starting VFI.")
%time σ_star_vfi = value_function_iteration(model).block_until_ready()
```

```
# Now time it without compile time
start = time()
σ_star_vfi = value_function_iteration(model).block_until_ready()
vfi_without_compile = time() - start
print(σ_star_vfi)
print(f"VFI completed in {vfi_without_compile} seconds.")
```

```
[[ 2  2  2 ...  6  6  6]
 [ 3  3  3 ...  7  7  7]
 [ 4  4  4 ...  7  7  7]
 ...
 [82 82 82 ... 86 86 86]
 [83 83 83 ... 86 86 86]
 [84 84 84 ... 87 87 87]]
VFI completed in 0.5243029594421387 seconds.
```

Here's the plot of the VFI, as a function of y at the highest and lowest values of z.

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(y_grid, y_grid, "k--", label="45")
ax.plot(y_grid, y_grid[σ_star_vfi[:, 1]], label="$\\sigma^{*}_{VFI}(\cdot, z_1)$")
ax.plot(y_grid, y_grid[σ_star_vfi[:, -1]], label="$\\sigma^{*}_{VFI}(\cdot, z_N)$")
ax.legend(fontsize=12)
plt.show()
```



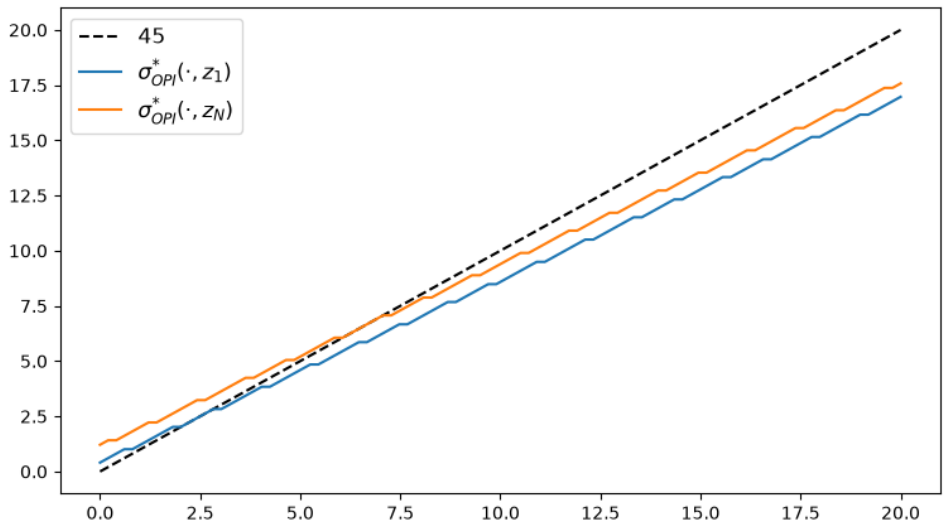

```
print("Starting OPI.")
%time σ_star_opi = optimistic_policy_iteration(model, m=100).block_until_ready()
```

```
# Now time it without compile time
start = time()
σ_star_opi = optimistic_policy_iteration(model, m=100).block_until_ready()
opi_without_compile = time() - start
print(σ_star_opi)
print(f"OPI completed in {opi_without_compile} seconds.")
```

```
[[ 2  2  2 ...  6  6  6]
 [ 3  3  3 ...  7  7  7]
 [ 4  4  4 ...  7  7  7]
 ...
 [82 82 82 ... 86 86 86]
 [83 83 83 ... 86 86 86]
 [84 84 84 ... 87 87 87]]
OPI completed in 0.2092909812927246 seconds.
```

Here's the plot of the optimal policy, as a function of y at the highest and lowest values of z.

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(y_grid, y_grid, "k--", label="45")
ax.plot(y_grid, y_grid[σ_star_opi[:, 1]], label="$\\sigma^{*}_{OPI}(\cdot, z_1)$")
ax.plot(y_grid, y_grid[σ_star_opi[:, -1]], label="$\\sigma^{*}_{OPI}(\cdot, z_N)$")
ax.legend(fontsize=12)
plt.show()
```



We observe that all the solvers produce the same output from the above three plots.

Let's plot the time taken by each of the solvers and compare them.

```
m_vals = range(5, 600, 40)
```

```
print("Running Howard policy iteration.")
%time σ_hpi = howard_policy_iteration(model).block_until_ready()
```

```
Running Howard policy iteration.
Concluded loop 1 with error 50.
Concluded loop 2 with error 26.
Concluded loop 3 with error 17.
Concluded loop 4 with error 10.
Concluded loop 5 with error 7.
Concluded loop 6 with error 4.
Concluded loop 7 with error 3.
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
Concluded loop 10 with error 1.
Concluded loop 11 with error 1.
Concluded loop 12 with error 0.
CPU times: user 113 ms, sys: 6.75 ms, total: 120 ms
Wall time: 103 ms
```

```
# Now time it without compile time
start = time()
σ_hpi = howard_policy_iteration(model).block_until_ready()
hpi_without_compile = time() - start
print(f"HPI completed in {hpi_without_compile} seconds.")
```

```
Concluded loop 1 with error 50.
Concluded loop 2 with error 26.
Concluded loop 3 with error 17.
Concluded loop 4 with error 10.
Concluded loop 5 with error 7.
Concluded loop 6 with error 4.
Concluded loop 7 with error 3.
```

(continues on next page)

(continued from previous page)

```
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
Concluded loop 10 with error 1.
Concluded loop 11 with error 1.
Concluded loop 12 with error 0.
HPI completed in 0.10373187065124512 seconds.
```

```
print("Running value function iteration.")
%time σ_vfi = value_function_iteration(model, tol=1e-5).block_until_ready()
```

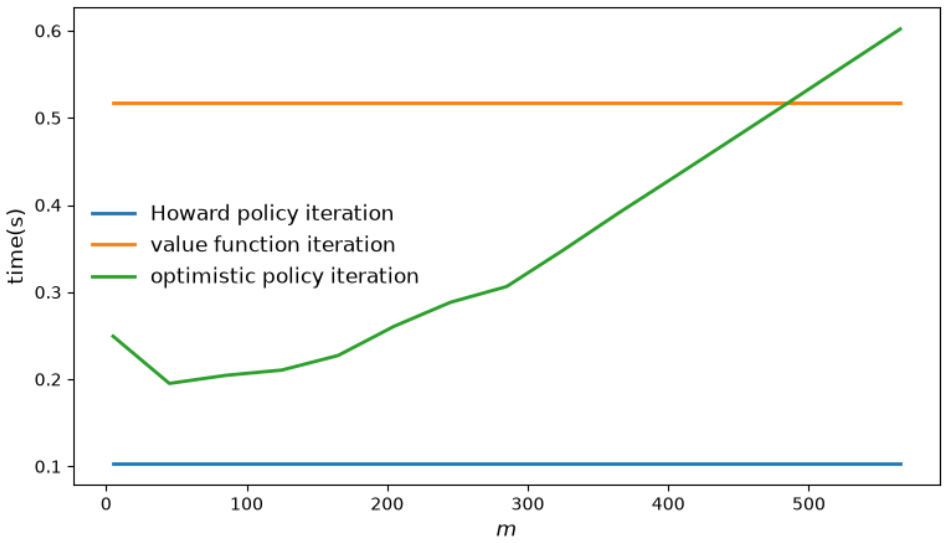
```
Running value function iteration.
CPU times: user 584 ms, sys: 7.11 ms, total: 591 ms
Wall time: 526 ms
```

```
# Now time it without compile time
start = time()
σ_vfi = value_function_iteration(model, tol=1e-5).block_until_ready()
vfi_without_compile = time() - start
print(f"VFI completed in {vfi_without_compile} seconds.")
```

```
VFI completed in 0.5180642604827881 seconds.
```

```
opi_times = []
for m in m_vals:
    print(f"Running optimistic policy iteration with m={m}.")
    σ_opi = optimistic_policy_iteration(model, m=m, tol=1e-5).block_until_ready()

    # Now time it without compile time
    start = time()
    σ_opi = optimistic_policy_iteration(model, m=m, tol=1e-5).block_until_ready()
    opi_without_compile = time() - start
    print(f"OPI with m={m} completed in {opi_without_compile} seconds.")
    opi_times.append(opi_without_compile)
```

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(m_vals, jnp.full(len(m_vals), hpi_without_compile),
        lw=2, label="Howard policy iteration")
ax.plot(m_vals, jnp.full(len(m_vals), vfi_without_compile),
        lw=2, label="value function iteration")
ax.plot(m_vals, opi_times, lw=2, label="optimistic policy iteration")
ax.legend(fontsize=12, frameon=False)
ax.set_xlabel("$m$", fontsize=12)
ax.set_ylabel("time(s)", fontsize=12)
plt.show()
```


0.6
0.5
0.4
0.3
0.2
0.1
time(s)
Howard policy iteration
value function iteration
optimistic policy iteration
0
100
200
300
400
500
m

CHAPTER

FIFTEEN

INVENTORY MANAGEMENT MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

This lecture provides a JAX implementation of a model in Dynamic Programming.

In addition to JAX and Anaconda, this lecture will need the following libraries:

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

15.1 A model with constant discounting

We study a firm where a manager tries to maximize shareholder value.

To simplify the problem, we assume that the firm only sells one product.

Letting π_t be profits at time t and $r > 0$ be the interest rate, the value of the firm is

$$V_0 = \sum_{t \geq 0} \beta^t \pi_t \qquad \text{where} \qquad \beta := \frac{1}{1+r}.$$

Suppose the firm faces exogenous demand process $(D_t)_{t \geq 0}$.

We assume $(D_t)_{t \geq 0}$ is IID with common distribution $\phi \in (Z_+)$.

Inventory $(X_t)_{t \geq 0}$ of the product obeys

$$X_{t+1} = f(X_t, D_{t+1}, A_t) \qquad \text{where} \qquad f(x, a, d) := (x - d) \vee 0 + a.$$

The term A_t is units of stock ordered this period, which take one period to arrive.

We assume that the firm can store at most K items at one time.

Profits are given by

$$\pi_t := X_t \wedge D_{t+1} - cA_t - \kappa 1\{A_t > 0\}.$$

We take the minimum of current stock and demand because orders in excess of inventory are assumed to be lost rather than back-filled.

Here c is unit product cost and κ is a fixed cost of ordering inventory.

We can map our inventory problem into a dynamic program with state space $X := \{0, \dots, K\}$ and action space $A := X$.

The feasible correspondence Γ is

$$\Gamma(x) := \{0, \dots, K - x\},$$

which represents the set of feasible orders when the current inventory state is x.

The reward function is expected current profits, or

$$r(x, a) := \sum_{d \geq 0} (x \wedge d)\phi(d) - ca - \kappa 1\{a > 0\}.$$

The stochastic kernel (i.e., state-transition probabilities) from the set of feasible state-action pairs is

$$P(x, a, x') := P\{f(x, a, D) = x'\} \qquad \text{when} \quad D \sim \phi.$$

When discounting is constant, the Bellman equation takes the form

$$v(x) = \max_{a \in \Gamma(x)} \left\{ r(x, a) + \beta \sum_{d \geq 0} v(f(x, a, d))\phi(d) \right\} \tag{15.1}$$

15.2 Time varing discount rates

We wish to consider a more sophisticated model with time-varying discounting.

This time variation accommodates non-constant interest rates.

To this end, we replace the constant β in (15.1) with a stochastic process (β_t) where

- $\beta_t = 1/(1 + r_t)$ and
- r_t is the interest rate at time t

We suppose that the dynamics can be expressed as $\beta_t = \beta(Z_t)$, where the exogenous process $(Z_t)_{t \geq 0}$ is a Markov chain on Z with Markov matrix Q.

After relabeling inventory X_t as Y_t and x as y, the Bellman equation becomes

$$v(y, z) = \max_{a \in \Gamma(x)} B((y, z), a, v)$$

where

$$B((y, z), a, v) = r(y, a) + \beta(z) \sum_{d,\, z'} v(f(y, a, d), z')\phi(d)Q(z, z'). \tag{15.2}$$

We set $\beta(z) := z$ and

$$R(y, a, y') := P\{f(y, a, d) = y'\} \quad \text{when} \quad D \sim \phi,$$

Now $R(y, a, y')$ is the probability of realizing next period inventory level y' when the current level is y and the action is a.

Hence we can rewrite (15.2) as

$$B((y, z), a, v) = r(y, a) + \beta(z) \sum_{y', z'} v(y', z')Q(z, z')R(y, a, y').$$

Let's begin with the following imports

```
import quantecon as qe
import jax
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
from time import time
from functools import partial
from typing import NamedTuple
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:22:44 2026
+-----------------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+----------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |               ␣
 ↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                ␣
 ↪     0 |
| N/A   35C    P8             14W /   70W |       0MiB /  15360MiB |      0%      ␣
 ↪Default |
|                                         |                        |               ␣
 ↪   N/A |
+-----------------------------------------+------------------------+----------------
 ↪-------+

+-----------------------------------------------------------------------------------------
 ↪-------+
| Processes:                                                                              ␣
 ↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
 ↪Memory |
|        ID   ID                                                               ␣
 ↪Usage      |
|=========================================================================================|
|  No running processes found                                                             ␣
 ↪       |
+-----------------------------------------------------------------------------------------
 ↪-------+
```

We will use 64 bit floats with JAX in order to increase the precision.

```
jax.config.update("jax_enable_x64", True)
```

Let's define a model to represent the inventory management.

```
# NamedTuple Model
class Model(NamedTuple):
    z_values: jnp.ndarray        # Exogenous shock values
    Q: jnp.ndarray               # Exogenous shock probabilities
    x_values: jnp.ndarray        # Inventory values
    d_values: jnp.ndarray        # Demand values for summation
    ϕ_values: jnp.ndarray        # Demand probabilities
    p: float                     # Demand parameter
    c: float = 0.2               # Unit cost
    κ: float = 0.8               # Fixed cost
```

```
def create_sdd_inventory_model(
        ρ: float = 0.98,     # Exogenous state autocorrelation parameter
        ν: float = 0.002,    # Exogenous state volatility parameter
        n_z: int = 10,       # Exogenous state discretization size
        b: float = 0.97,     # Exogenous state offset
        K: int = 100,        # Max inventory
        D_MAX: int = 101,    # Demand upper bound for summation
        p: float = 0.6
    ) -> Model:

    # Demand
    def demand_pdf(p, d):
        return (1 - p)**d * p

    d_values = jnp.arange(D_MAX)
    ϕ_values = demand_pdf(p, d_values)

    # Exogenous state process
    mc = qe.tauchen(n_z, ρ, ν)
    z_values, Q = map(jnp.array, (mc.state_values + b, mc.P))

    # Endogenous state
    x_values = jnp.arange(K + 1)   # 0, 1, ..., K

    return Model(
        z_values=z_values, Q=Q,
        x_values=x_values, d_values=d_values, ϕ_values=ϕ_values,
        p=p
    )
```

Here's the function `B` on the right-hand side of the Bellman equation.

```
@jax.jit
def B(x, z_idx, v, model):
    """
    Take z_idx and convert it to z.  Then compute

        B(x, z, a, v) = r(x, a) + β(z) Σ_x' v(x') P(x, a, x')

    for all possible choices of a.
    """

    z_values, Q, x_values, d_values, ϕ_values, p, c, κ = model
    z = z_values[z_idx]

    def _B(a):
```

(continues on next page)

(continued from previous page)

```
        """
        Returns r(x, a) + β(z) Σ_x′ v(x′) P(x, a, x′) for each a.
        """
        revenue = jnp.sum(jnp.minimum(x, d_values) * ϕ_values)
        profit = revenue - c * a - κ * (a > 0)
        v_R = jnp.sum(v[jnp.maximum(x - d_values, 0) + a].T * ϕ_values, axis=1)
        cv = jnp.sum(v_R * Q[z_idx])
        return profit + z * cv

    a_values = x_values   # Set of possible order sizes
    B_values = jax.vmap(_B)(a_values)
    max_x = len(x_values) - 1

    return jnp.where(a_values <= max_x - x, B_values, -jnp.inf)
```

We need to vectorize this function so that we can use it efficiently in JAX.

We apply a sequence of `vmap` operations to vectorize appropriately in each argument.

```
B = jax.vmap(B, in_axes=(None, 0, None, None))
B = jax.vmap(B, in_axes=(0, None, None, None))
```

Next we define the Bellman operator.

```
@jax.jit
def T(v, model):
    """The Bellman operator."""
    z_values, Q, x_values, d_values, ϕ_values, p, c, κ = model
    z_indices = jnp.arange(len(z_values))
    res = B(x_values, z_indices, v, model)
    return jnp.max(res, axis=2)
```

The following function computes a v-greedy policy.

```
@jax.jit
def get_greedy(v, model):
    """Get a v-greedy policy.  Returns a zero-based array."""
    z_values, Q, x_values, d_values, ϕ_values, p, c, κ = model
    z_indices = jnp.arange(len(z_values))
    res = B(x_values, z_indices, v, model)
    return jnp.argmax(res, axis=2)
```

Here's code to solve the model using value function iteration.

```
@jax.jit
def solve_inventory_model(v_init, model, max_iter=10_000, tol=1e-6):
    """Use successive_approx to get v_star and then compute greedy."""

    def update(state):
        error, i, v = state
        new_v = T(v, model)
        new_error = jnp.max(jnp.abs(new_v - v))
        new_i = i + 1
        return new_error, new_i, new_v

    def test(state):
        error, i, v = state
```

(continues on next page)

(continued from previous page)

```
        return (i < max_iter) & (error > tol)

    i, error = 0, tol + 1
    initial_state = error, i, v_init
    final_state = jax.lax.while_loop(test, update, initial_state)
    error, i, v_star = final_state
    σ_star = get_greedy(v_star, model)
    return v_star, σ_star
```

Now let's create an instance and solve it.

```
model = create_sdd_inventory_model()
z_values, Q, x_values, d_values, ϕ_values, p, c, κ = model
n_z = len(z_values)
n_x = len(x_values)
v_init = jnp.zeros((n_x, n_z), dtype=float)
```

```
start = time()
v_star, σ_star = solve_inventory_model(v_init, model)

# Pause until execution finishes
jax.tree_util.tree_map(lambda x: x.block_until_ready(), (v_star, σ_star))

jax_time_with_compile = time() - start
print(f"compile plus execution time = {jax_time_with_compile * 1000:.6f} ms")
```

```
compile plus execution time = 1269.429445 ms
```

Let's run again to get rid of the compile time.

```
start = time()
v_star, σ_star = solve_inventory_model(v_init, model)

# Pause until execution finishes
jax.tree_util.tree_map(lambda x: x.block_until_ready(), (v_star, σ_star))

jax_time_without_compile = time() - start
print(f"execution time = {jax_time_without_compile * 1000:.6f} ms")
```

```
execution time = 815.770388 ms
```

Now let's do a simulation.

We'll begin by converting back to NumPy arrays for convenience

```
Q = np.array(Q)
z_values = np.array(z_values)
z_mc = qe.MarkovChain(Q, z_values)
```

Here's code to simulate inventories

```
def sim_inventories(ts_length, X_init=0):
    """Simulate given the optimal policy."""
    global p, z_mc

    z_idx = z_mc.simulate_indices(ts_length, init=1)
```

(continues on next page)

(continued from previous page)

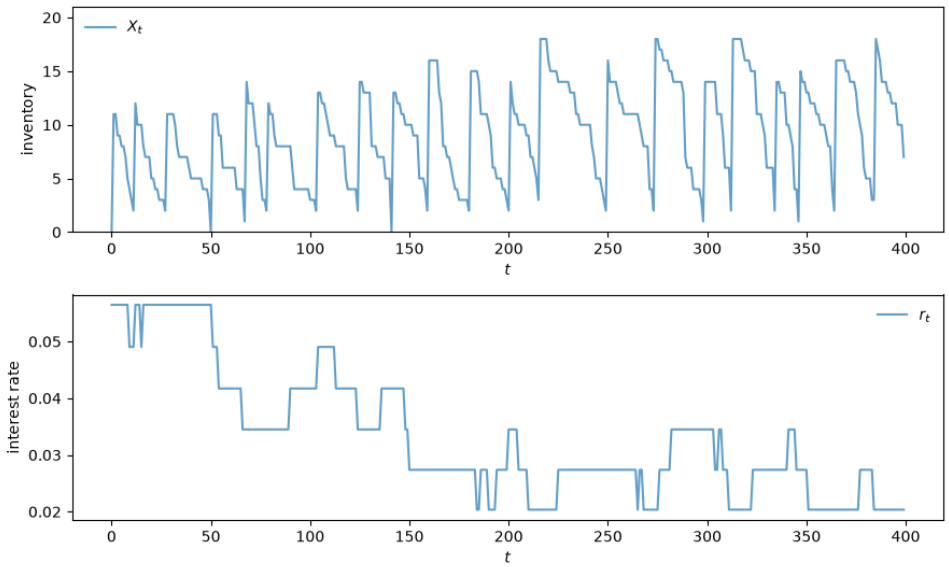
```
    X = np.zeros(ts_length, dtype=np.int32)
    X[0] = X_init
    rand = np.random.default_rng().geometric(p=p, size=ts_length-1) - 1

    for t in range(ts_length-1):
        X[t+1] = np.maximum(X[t] - rand[t], 0) + σ_star[X[t], z_idx[t]]

    return X, z_values[z_idx]
```

Here's code to generate a plot.

```
def plot_ts(ts_length=400, fontsize=10):
    X, Z = sim_inventories(ts_length)
    fig, axes = plt.subplots(2, 1, figsize=(9, 5.5))

    ax = axes[0]
    ax.plot(X, label=r"$X_t$", alpha=0.7)
    ax.set_xlabel(r"$t$", fontsize=fontsize)
    ax.set_ylabel("inventory", fontsize=fontsize)
    ax.legend(fontsize=fontsize, frameon=False)
    ax.set_ylim(0, np.max(X)+3)

    # calculate interest rate from discount factors
    r = (1 / Z) - 1

    ax = axes[1]
    ax.plot(r, label=r"$r_t$", alpha=0.7)
    ax.set_xlabel(r"$t$", fontsize=fontsize)
    ax.set_ylabel("interest rate", fontsize=fontsize)
    ax.legend(fontsize=fontsize, frameon=False)

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

Let's take a look.

```
plot_ts()
```


X_t
inventory
t
r_t
interest rate
t

CHAPTER

SIXTEEN

ENDOGENOUS GRID METHOD

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

16.1 Overview

In this lecture we use the endogenous grid method (EGM) to solve a basic income fluctuation (optimal savings) problem.

Background on the endogenous grid method can be found in an earlier QuantEcon lecture.

Here we focus on providing an efficient JAX implementation.

In addition to JAX and Anaconda, this lecture will need the following libraries:

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

```
import quantecon as qe
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
import numba
from time import time
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:20:07 2026
+-----------------------------------------------------------------------------------
↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
↪13.0     |
+-----------------------------------------+------------------------+---------------
```

(continues on next page)

(continued from previous page)

```
↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
↪Compute M. |
|                                         |                        |              ␣
↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |              ␣
↪     0 |
| N/A   37C    P8             14W /   70W |       0MiB /  15360MiB |      0%      ␣
↪Default |
|                                         |                        |              ␣
↪   N/A |
+-----------------------------------------+------------------------+---------------
↪-------+

+-----------------------------------------------------------------------------------
↪-------+
| Processes:                                                                        ␣
↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
↪Memory |
|        ID   ID                                                               ␣
↪Usage      |
|===================================================================================|
|  No running processes found                                                       ␣
↪       |
+-----------------------------------------------------------------------------------
↪-------+
```

We use 64 bit floating point numbers for extra precision.

```
jax.config.update("jax_enable_x64", True)
```

16.2 Setup

We 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

$$a_{t+1} \leq R(a_t - c_t) + Y_{t+1}, \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots$$

Here $R = 1 + r$ where r is the interest rate.

The income process $\{Y_t\}$ is a Markov chain generated by stochastic matrix P.

The matrix P and the grid of values taken by Y_t are obtained by discretizing the AR(1) process

$$Y_{t+1} = \rho Y_t + \nu \epsilon_{t+1}$$

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

Utility has the CRRA specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

The following function stores default parameter values for the income fluctuation problem and creates suitable arrays.

```
def ifp(R=1.01,                # gross interest rate
        β=0.99,                # discount factor
        γ=1.5,                 # CRRA preference parameter
        s_max=16,              # savings grid max
        s_size=200,            # savings grid size
        ρ=0.99,                # income persistence
        ν=0.02,                # income volatility
        y_size=25):            # income grid size

    # require R β < 1 for convergence
    assert R * β < 1, "Stability condition failed."
    # Create income Markov chain
    mc = qe.tauchen(y_size, ρ, ν)
    y_grid, P = jnp.exp(mc.state_values), mc.P
    # Shift to JAX arrays
    P, y_grid = jax.device_put((P, y_grid))
    s_grid = jnp.linspace(0, s_max, s_size)
    # Pack and return
    constants = β, R, γ
    sizes = s_size, y_size
    arrays = s_grid, y_grid, P
    return constants, sizes, arrays
```

16.3 Solution method

Let $S = \mathbb{R}_+ \times \mathsf{Y}$ be the set of possible values for the state (a_t, Y_t).

We aim to compute an optimal consumption policy $\sigma^* \colon S \to \mathbb{R}$, under which dynamics are given by

$$c_t = \sigma^*(a_t, Y_t) \quad \text{and} \quad a_{t+1} = R(a_t - c_t) + Y_{t+1}$$

In this section we discuss how we intend to solve for this policy.

16.3.1 Euler equation

The Euler equation for the optimization problem is

$$u'(c_t) = \max\left\{\beta R\, \mathbb{E}_t u'(c_{t+1})\,,\ u'(a_t)\right\}$$

An explanation for this expression can be found here.

We rewrite the Euler equation in functional form

$$(u' \circ \sigma)(a, y) = \max\left\{\beta R\, \mathbb{E}_y (u' \circ \sigma)[R(a - \sigma(a, y)) + \hat{Y},\, \hat{Y}]\,,\ u'(a)\right\}$$

where $(u' \circ \sigma)(a, y) := u'(\sigma(a, y))$ and σ is a consumption policy.

For given consumption policy σ, we define $(K\sigma)(a, y)$ as the unique $c \in [0, a]$ that solves

$$u'(c) = \max\left\{\beta R\, \mathbb{E}_y (u' \circ \sigma)\left[R(a - c) + \hat{Y},\, \hat{Y}\right]\,,\ u'(a)\right\} \tag{16.1}$$

It can be shown that

1. iterating with K computes an optimal policy and
2. if σ is increasing in its first argument, then so is $K\sigma$

Hence below we always assume that σ is increasing in its first argument.

The EGM is a technique for computing the update $K\sigma$ given σ along a grid of asset values.

Notice that, since $u'(a) \to \infty$ as $a \downarrow 0$, the second term in the max in *(16.3.1)* dominates for sufficiently small a.

Also, again using *(16.3.1)*, we have $c = a$ for all such a.

Hence, for sufficiently small a,

$$u'(a) \geq \beta R \, \mathbb{E}_y (u' \circ \sigma) \, [\hat{Y}, \, \hat{Y}]$$

Equality holds at $\bar{a}(y)$ given by

$$\bar{a}(y) = (u')^{-1} \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) \, [\hat{Y}, \, \hat{Y}] \right\}$$

We can now write

$$u'(c) = \begin{cases} \beta R \, \mathbb{E}_y (u' \circ \sigma) \, [R(a-c) + \hat{Y}, \, \hat{Y}] & \text{if } a > \bar{a}(y) \\ u'(a) & \text{if } a \leq \bar{a}(y) \end{cases}$$

Equivalently, we can state that the c satisfying $c = (K\sigma)(a, y)$ obeys

$$c = \begin{cases} (u')^{-1} \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) \, [R(a-c) + \hat{Y}, \, \hat{Y}] \right\} & \text{if } a > \bar{a}(y) \\ a & \text{if } a \leq \bar{a}(y) \end{cases} \tag{16.2}$$

We begin with an *exogenous* grid of saving values $0 = s_0 < \ldots < s_{N-1}$

Using the exogenous savings grid, and a fixed value of y, we create an *endogenous* asset grid $a_0, \ldots, a_{N-1}$ and a consumption grid $c_0, \ldots, c_{N-1}$ as follows.

First we set $a_0 = c_0 = 0$, since zero consumption is an optimal (in fact the only) choice when $a = 0$.

Then, for $i > 0$, we compute

$$c_i = (u')^{-1} \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) \, [R s_i + \hat{Y}, \, \hat{Y}] \right\} \quad \text{for all } i \tag{16.3}$$

and we set

$$a_i = s_i + c_i$$

We claim that each pair a_i, c_i obeys *(16.3.2)*.

Indeed, since $s_i > 0$, choosing c_i according to *(16.3.3)* gives

$$c_i = (u')^{-1} \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) \, [R s_i + \hat{Y}, \, \hat{Y}] \right\} \geq \bar{a}(y)$$

where the inequality uses the fact that σ is increasing in its first argument.

If we now take $a_i = s_i + c_i$ we get $a_i > \bar{a}(y)$, so the pair (a_i, c_i) satisfies

$$c_i = (u')^{-1} \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) \, [R(a_i - c_i) + \hat{Y}, \, \hat{Y}] \right\} \quad \text{and} \quad a_i > \bar{a}(y)$$

Hence *(16.3.2)* holds.

We are now ready to iterate with K.

16.3.2 JAX version

First we define a vectorized operator K based on the EGM.

Notice in the code below that

- we avoid all loops and any mutation of arrays
- the function is pure (no globals, no mutation of inputs)

```
def K_egm(a_in, σ_in, constants, sizes, arrays):
    """
    The vectorized operator K using EGM.

    """

    # Unpack
    β, R, γ = constants
    s_size, y_size = sizes
    s_grid, y_grid, P = arrays

    def u_prime(c):
        return c**(-γ)

    def u_prime_inv(u):
            return u**(-1/γ)

    # Linearly interpolate σ(a, y)
    def σ(a, y):
        return jnp.interp(a, a_in[:, y], σ_in[:, y])
    σ_vec = jnp.vectorize(σ)

    # Broadcast and vectorize
    y_hat = jnp.reshape(y_grid, (1, 1, y_size))
    y_hat_idx = jnp.reshape(jnp.arange(y_size), (1, 1, y_size))
    s = jnp.reshape(s_grid, (s_size, 1, 1))
    P = jnp.reshape(P, (1, y_size, y_size))

    # Evaluate consumption choice
    a_next = R * s + y_hat
    σ_next = σ_vec(a_next, y_hat_idx)
    up = u_prime(σ_next)
    E = jnp.sum(up * P, axis=-1)
    c = u_prime_inv(β * R * E)

    # Set up a column vector with zero in the first row and ones elsewhere
    e_0 = jnp.ones(s_size) - jnp.identity(s_size)[:, 0]
    e_0 = jnp.reshape(e_0, (s_size, 1))

    # The policy is computed consumption with the first row set to zero
    σ_out = c * e_0

    # Compute a_out by a = s + c
    a_out = np.reshape(s_grid, (s_size, 1)) + σ_out

    return a_out, σ_out
```

Then we use `jax.jit` to compile K.

We use `static_argnums` to allow a recompile whenever `sizes` changes, since the compiler likes to specialize on

shapes.

```
K_egm_jax = jax.jit(K_egm, static_argnums=(3,))
```

Next we define a successive approximator that repeatedly applies K.

```
def successive_approx_jax(model,
            tol=1e-5,
            max_iter=100_000,
            verbose=True,
            print_skip=25):

    # Unpack
    constants, sizes, arrays = model
    β, R, γ = constants
    s_size, y_size = sizes
    s_grid, y_grid, P = arrays

    # Initial condition is to consume all in every state
    σ_init = jnp.repeat(s_grid, y_size)
    σ_init = jnp.reshape(σ_init, (s_size, y_size))
    a_init = jnp.copy(σ_init)
    a_vec, σ_vec = a_init, σ_init

    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        a_new, σ_new = K_egm_jax(a_vec, σ_vec, constants, sizes, arrays)
        error = jnp.max(jnp.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 = jnp.copy(a_new), jnp.copy(σ_new)

    if error > tol:
        print("Failed to converge!")
    else:
        print(f"\nConverged in {i} iterations.")

    return a_new, σ_new
```

16.3.3 Numba version

Below we provide a second set of code, which solves the same model with Numba.

The purpose of this code is to cross-check our results from the JAX version, as well as to do a runtime comparison.

Most readers will want to skip ahead to the next section, where we solve the model and run the cross-check.

```
@numba.jit
def K_egm_nb(a_in, σ_in, constants, sizes, arrays):
    """
    The operator K using Numba.

    """
```

(continues on next page)

(continued from previous page)

```
    # Simplify names
    β, R, γ = constants
    s_size, y_size = sizes
    s_grid, y_grid, P = arrays

    def u_prime(c):
        return c**(-γ)

    def u_prime_inv(u):
        return u**(-1/γ)

    # Linear interpolation of policy using endogenous grid
    def σ(a, z):
        return np.interp(a, a_in[:, z], σ_in[:, z])

    # Allocate memory for new consumption array
    σ_out = np.zeros_like(σ_in)
    a_out = np.zeros_like(σ_out)

    for i, s in enumerate(s_grid[1:]):
        i += 1
        for z in range(y_size):
            expect = 0.0
            for z_hat in range(y_size):
                expect += u_prime(σ(R * s + y_grid[z_hat], z_hat)) * \
                            P[z, z_hat]
            c = u_prime_inv(β * R * expect)
            σ_out[i, z] = c
            a_out[i, z] = s + c

    return a_out, σ_out
```

```
def successive_approx_numba(model,        # Class with model information
                            tol=1e-5,
                            max_iter=100_000,
                            verbose=True,
                            print_skip=25):

    # Unpack
    constants, sizes, arrays = model
    s_size, y_size = sizes
    # make NumPy versions of arrays
    arrays = tuple(map(np.array, arrays))
    s_grid, y_grid, P = arrays

    σ_init = np.repeat(s_grid, y_size)
    σ_init = np.reshape(σ_init, (s_size, y_size))
    a_init = np.copy(σ_init)
    a_vec, σ_vec = a_init, σ_init

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        a_new, σ_new = K_egm_nb(a_vec, σ_vec, constants, sizes, arrays)
```

(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!")
    else:
        print(f"\nConverged in {i} iterations.")

    return a_new, σ_new
```

16.4 Solutions

Here we solve the IFP with JAX and Numba.

We will compare both the outputs and the execution time.

16.4.1 Outputs

```
model = ifp()
```

Here's a first run of the JAX code.

```
%%time
a_star_egm_jax, σ_star_egm_jax = successive_approx_jax(model,
                                        print_skip=1000)
```

```
Error at iteration 1000 is 6.472028596182788e-05.
Error at iteration 2000 is 1.2994575430580468e-05.

Converged in 2192 iterations.
CPU times: user 3.79 s, sys: 1.14 s, total: 4.92 s
Wall time: 3.28 s
```

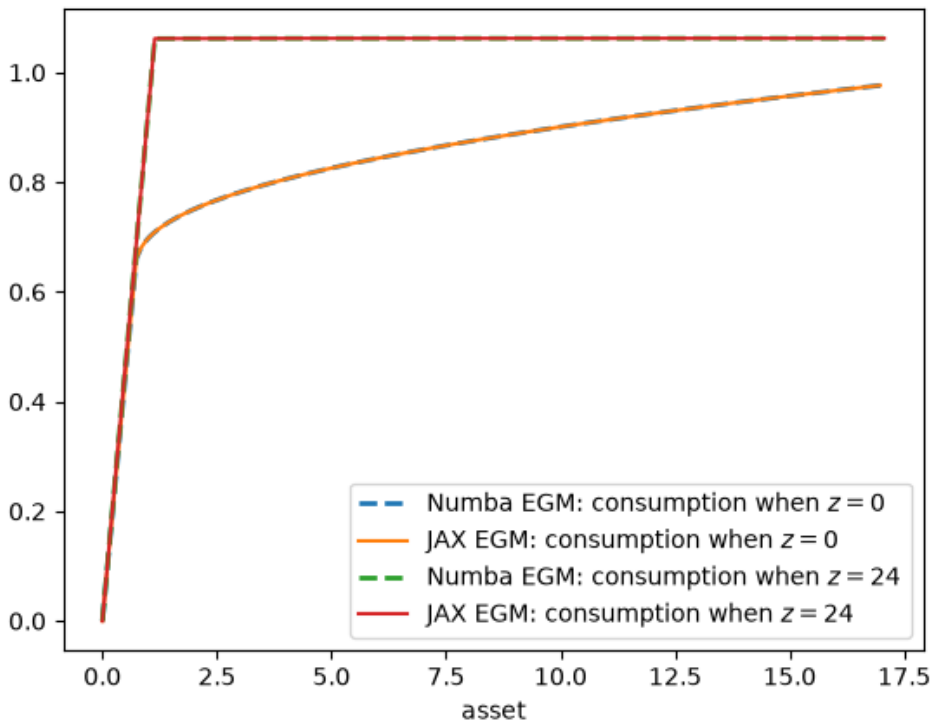
Next let's solve the same IFP with Numba.

```
%%time
a_star_egm_nb, σ_star_egm_nb = successive_approx_numba(model,
                                      print_skip=1000)
```

```
Error at iteration 1000 is 6.472028596182788e-05.
Error at iteration 2000 is 1.2994575430802513e-05.

Converged in 2192 iterations.
CPU times: user 1min 7s, sys: 0 ns, total: 1min 7s
Wall time: 1min 7s
```

Now let's check the outputs in a plot to make sure they are the same.

```
constants, sizes, arrays = model
β, R, γ = constants
s_size, y_size = sizes
s_grid, y_grid, P = arrays


fig, ax = plt.subplots()

for z in (0, y_size-1):
    ax.plot(a_star_egm_nb[:, z],
            σ_star_egm_nb[:, z],
            '--', lw=2,
            label=f"Numba EGM: consumption when $z={z}$")
    ax.plot(a_star_egm_jax[:, z],
            σ_star_egm_jax[:, z],
            label=f"JAX EGM: consumption when $z={z}$")

ax.set_xlabel('asset')
plt.legend()
plt.show()
```



16.4.2 Timing

Now let's compare execution time of the two methods.

```
start = time()
a_star_egm_jax, σ_star_egm_jax = successive_approx_jax(model,
                                        print_skip=1000)
jax_time_without_compile = time() - start
print("Jax execution time = ", jax_time_without_compile)
```

```
Error at iteration 1000 is 6.472028596182788e-05.
Error at iteration 2000 is 1.2994575430580468e-05.

Converged in 2192 iterations.
Jax execution time =  2.636720657348633
```

```
start = time()
a_star_egm_nb, σ_star_egm_nb = successive_approx_numba(model,
                                        print_skip=1000)
numba_time_without_compile = time() - start
print("Numba execution time = ", numba_time_without_compile)
```

```
Error at iteration 1000 is 6.472028596182788e-05.
Error at iteration 2000 is 1.2994575430802513e-05.

Converged in 2192 iterations.
Numba execution time =  66.93267107009888
```

```
jax_time_without_compile / numba_time_without_compile
```

```
0.03939362668773795
```

The JAX code is significantly faster, as expected.

This difference will increase when more features (and state variables) are added to the model.

Part V

Macroeconomic Models

CHAPTER

SEVENTEEN

DEFAULT RISK AND INCOME FLUCTUATIONS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to JAX and Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

17.1 Overview

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

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

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

The foreign lenders must be compensated for default risk.

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

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

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

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

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

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

This can lead to

- spikes in interest rates

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

Such dynamics are consistent with experiences of many countries.

Let's start with some imports:

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

import jax
import jax.numpy as jnp
from typing import NamedTuple
```

Let's check the GPU we are using

```
!nvidia-smi
```

```
Fri Oct  9 00:18:37 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version: 13.0     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                    0 |
| N/A   36C    P8              9W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                                   Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

We will use 64 bit floats with JAX in order to increase the precision.

```
jax.config.update("jax_enable_x64", True)
```

17.2 Structure

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

17.2.1 Output, Consumption and Debt

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

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

The output good can be traded or consumed.

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

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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{17.1}$$

Here

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

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

The government is benevolent in the sense that its aim is to maximize (17.1).

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

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

17.2.2 Asset Markets

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

The bond market has the following features

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

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

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

$$c = y + B - q(B', y)B' \tag{17.2}$$

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

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

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

17.2.3 Financial Markets

Foreign creditors

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

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

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

$$q = \frac{1 - \delta}{1 + r} \tag{17.3}$$

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

17.2.4 Government's Decisions

At each point in time t, the government chooses between

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

Defaulting means declining to repay all of its current obligations.

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

But a sovereign default has two consequences:

1. Output immediately falls from y to $h(y)$, where $0 \leq h(y) \leq y$.

- It returns to y only after the country regains access to international credit markets.

1. The country loses access to foreign credit markets.

17.2.5 Reentering International Credit Market

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

17.3 Equilibrium

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

1. Consumption and assets satisfy the national budget constraint.
2. The government maximizes household utility taking into account

- the resource constraint
- the effect of its choices on the price of bonds
- consequences of defaulting now for future net output and future borrowing and lending opportunities

1. The interest rate on the government's debt includes a risk-premium sufficient to make foreign creditors expect on average to earn the constant risk-free international interest rate.

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

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

In a recursive formulation,

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

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

Expressed recursively, the value of defaulting is

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

The value of paying is

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

The three value functions are linked by

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

The government chooses to default when

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

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

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \tag{17.4}$$

Given zero profits for foreign creditors in equilibrium, we can combine (17.3) and (17.4) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{17.5}$$

17.3.1 Definition of Equilibrium

An *equilibrium* is

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

such that

- The three Bellman equations for $(v_c(B, y), v_d(y), v(B, y))$ are satisfied
- Given the price function $q(B', y)$, the default decision rule and the asset accumulation decision rule attain the optimal value function $v(B, y)$, and
- The price function $q(B', y)$ satisfies equation (17.5)

17.4 Computation

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

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

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

After that we'll perform some additional simulations.

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

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

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

Here is a more detailed description of our algorithm:

1. Guess a pair of non-default and default value functions v_c and v_d.
2. Using these functions, calculate the value function v, the corresponding default probabilities and the price function q.
3. At each pair (B, y),
4. update the value of defaulting $v_d(y)$.

5. update the value of remaining $v_c(B, y)$.
6. Check for convergence. If converged, stop – if not, go to step 2.

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

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

As we have in other places, we accelerate our code using JAX and JIT compilation.

We define a namedtuple to store parameters, grids and transition probabilities.

```
class ArellanoEconomy(NamedTuple):
    β: float            # Time discount parameter
    γ: float            # Utility parameter
    r: float            # Lending rate
    ρ: float            # Persistence in the income process
    η: float            # Standard deviation of the income process
    θ: float            # Prob of re-entering financial markets
    B_size: int         # Grid size for bonds
    y_size: int         # Grid size for income
    P: jnp.ndarray      # Markov matrix governing the income process
    B_grid: jnp.ndarray # Bond grid
    y_grid: jnp.ndarray # State values of the income process
    def_y: jnp.ndarray  # Default income process
```

```
def create_arellano(
    B_size=251,    # Grid size for bonds
    B_min=-0.45,   # Smallest B value
    B_max=0.45,    # Largest B value
    y_size=51,     # Grid size for income
    β=0.953,       # Time discount parameter
    γ=2.0,         # Utility parameter
    r=0.017,       # Lending rate
    ρ=0.945,       # Persistence in the income process
    η=0.025,       # Standard deviation of the income process
    θ=0.282,       # Prob of re-entering financial markets
    def_y_param=0.969,
):  # Parameter governing income in default
    # Set up grids
    B_grid = jnp.linspace(B_min, B_max, B_size)
    mc = qe.markov.tauchen(y_size, ρ, η)
    y_grid, P = jnp.exp(mc.state_values), mc.P

    # Put grids on the device
    P = jax.device_put(P)

    # Output received while in default, with same shape as y_grid
    def_y = jnp.minimum(def_y_param * jnp.mean(y_grid), y_grid)

    return ArellanoEconomy(
        β=β,
        γ=γ,
        r=r,
        ρ=ρ,
        η=η,
        θ=θ,
        B_size=B_size,
        y_size=y_size,
```

(continues on next page)

(continued from previous page)

```
        P=P,
        B_grid=B_grid,
        y_grid=y_grid,
        def_y=def_y,
    )
```

Here is the utility function.

```
@jax.jit
def u(c, γ):
    return c ** (1 - γ) / (1 - γ)
```

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

```
def compute_q(v_c, v_d, params, sizes, arrays):
    """
    Compute the bond price function q(B, y) at each (B, y) pair.  The first
    step is to calculate the default probabilities

        δ(B, y) := Σ_{y'} 1{v_c(B, y') < v_d(y')} P(y, y') dy'
    """

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

    # Set up arrays with indices [i_B, i_y, i_yp]
    v_d = jnp.reshape(v_d, (1, 1, y_size))
    v_c = jnp.reshape(v_c, (B_size, 1, y_size))
    P = jnp.reshape(P, (1, y_size, y_size))

    # Compute δ[i_B, i_y]
    default_states = v_c < v_d
    delta = jnp.sum(default_states * P, axis=(2,))

    q = (1 - delta) / (1 + r)
    return q
```

Next we introduce Bellman operators that updated v_d and v_c.

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

    B0_idx = jnp.searchsorted(B_grid, 1e-10)  # Index at which B is near zero

    current_utility = u(def_y, γ)
    v = jnp.maximum(v_c[B0_idx, :], v_d)
    w = θ * v + (1 - θ) * v_d
    A = jnp.reshape(w, (1, y_size))
```

(continues on next page)

(continued from previous page)

```
    cont_value = jnp.sum(A * P, axis=(1,))

    return current_utility + β * cont_value
```

```
def bellman(v_c, v_d, q, params, sizes, arrays):
    """
    The RHS of the Bellman equation when the country is not in a
    defaulted state on their debt.  That is,

        bellman(B, y) =
            u(y - q(B', y) B' + B) + β Σ_{y'} v(B', y') P(y, y')

    If consumption is not positive then returns -np.inf
    """
    # Unpack
    β, γ, r, ρ, η, θ = params
    B_size, y_size = sizes
    P, B_grid, y_grid, def_y = arrays

    # Set up c[i_B, i_y, i_Bp]
    y_idx = jnp.reshape(jnp.arange(y_size), (1, y_size, 1))
    B_idx = jnp.reshape(jnp.arange(B_size), (B_size, 1, 1))
    Bp_idx = jnp.reshape(jnp.arange(B_size), (1, 1, B_size))
    c = y_grid[y_idx] - q[Bp_idx, y_idx] * B_grid[Bp_idx] + B_grid[B_idx]

    # Set up v[i_B, i_y, i_Bp, i_yp] and P[i_B, i_y, i_Bp, i_yp]
    v_d = jnp.reshape(v_d, (1, 1, 1, y_size))
    v_c = jnp.reshape(v_c, (1, 1, B_size, y_size))
    v = jnp.maximum(v_c, v_d)
    P = jnp.reshape(P, (1, y_size, 1, y_size))
    # Sum over i_yp
    continuation_value = jnp.sum(v * P, axis=(3,))

    # Return new_v_c[i_B, i_y, i_Bp]
    val = jnp.where(c > 0, u(c, γ) + β * continuation_value, -jnp.inf)
    return val
```

```
def T_c(v_c, v_d, q, params, sizes, arrays):
    vals = bellman(v_c, v_d, q, params, sizes, arrays)
    return jnp.max(vals, axis=2)
```

```
def get_greedy(v_c, v_d, q, params, sizes, arrays):
    vals = bellman(v_c, v_d, q, params, sizes, arrays)
    return jnp.argmax(vals, axis=2)
```

Let's make JIT-compiled versions of these functions, with the sizes of the arrays declared as static (compile-time constants) in order to help the compiler.

```
compute_q = jax.jit(compute_q, static_argnums=(3,))
T_d = jax.jit(T_d, static_argnums=(3,))
bellman = jax.jit(bellman, static_argnums=(4,))
T_c = jax.jit(T_c, static_argnums=(4,))
get_greedy = jax.jit(get_greedy, static_argnums=(4,))
```

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

```
def update_values_and_prices(v_c, v_d, params, sizes, arrays):

    q = compute_q(v_c, v_d, params, sizes, arrays)
    new_v_d = T_d(v_c, v_d, params, sizes, arrays)
    new_v_c = T_c(v_c, v_d, q, params, sizes, arrays)

    return new_v_c, new_v_d
```

We can now write a function that will use an instance of `ArellanoEconomy` and the functions defined above to compute the solution to our model.

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

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

    β, γ, r, ρ, η, θ, B_size, y_size, P, B_grid, y_grid, def_y = model

    params = β, γ, r, ρ, η, θ
    sizes = B_size, y_size
    arrays = P, B_grid, y_grid, def_y

    # Initial conditions for v_c and v_d
    v_c = jnp.zeros((B_size, y_size))
    v_d = jnp.zeros((y_size,))

    current_iter = 0
    error = tol + 1
    while (current_iter < max_iter) and (error > tol):
        if current_iter % 100 == 0:
            print(f"Entering iteration {current_iter} with error {error}.")
        new_v_c, new_v_d = update_values_and_prices(
            v_c, v_d, params, sizes, arrays
        )
        error = jnp.max(jnp.abs(new_v_c - v_c)) + jnp.max(
            jnp.abs(new_v_d - v_d)
        )
        v_c, v_d = new_v_c, new_v_d
        current_iter += 1

    print(f"Terminating at iteration {current_iter}.")

    q = compute_q(v_c, v_d, params, sizes, arrays)
    B_star = get_greedy(v_c, v_d, q, params, sizes, arrays)
    return v_c, v_d, q, B_star
```

Let's try solving the model.

```
ae = create_arellano()
```

```
with qe.Timer():
    v_c, v_d, q, B_star = solve(ae)
```

```
Entering iteration 0 with error 1.00000001.
Entering iteration 100 with error 0.017499341639204857.
Entering iteration 200 with error 0.00014189363558969603.
Entering iteration 300 with error 1.151467966309383e-06.
Terminating at iteration 399.
3.1051 seconds elapsed
```

We run it again to get rid of compile time.

```
with qe.Timer():
    v_c, v_d, q, B_star = solve(ae)
```

```
Entering iteration 0 with error 1.00000001.
Entering iteration 100 with error 0.017499341639204857.
Entering iteration 200 with error 0.00014189363558969603.
Entering iteration 300 with error 1.151467966309383e-06.
Terminating at iteration 399.
1.6061 seconds elapsed
```

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

```
def simulate(model, T, v_c, v_d, q, B_star, key):
    """
    Simulates the Arellano 2008 model of sovereign debt

    Here `model` is an instance of `ArellanoEconomy` and `T` is the length of
    the simulation.  Endogenous objects `v_c`, `v_d`, `q` and `B_star` are
    assumed to come from a solution to `model`.
    """
    # Unpack elements of the model
    B_size, y_size = model.B_size, model.y_size
    B_grid, y_grid, P = model.B_grid, model.y_grid, model.P

    B0_idx = jnp.searchsorted(B_grid, 1e-10)  # Index at which B is near zero

    # Set initial conditions
    y_idx = y_size // 2
    B_idx = B0_idx
    in_default = False

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

    y_sim, y_a_sim = [], []
    B_sim, q_sim, d_sim = [], [], []

    # Perform simulation
    t = 0
    while t < T:
        y_sim.append(y_grid[y_idx])
        B_sim.append(B_grid[B_idx])
        if v_c[B_idx, y_idx] < v_d[y_idx] or in_default:
            y_a_sim.append(model.def_y[y_idx])
            d_sim.append(1)
            Bp_idx = B0_idx
            # Re-enter financial markets next period with prob θ
```

(continues on next page)

(continued from previous page)

```
            in_default = False if jax.random.uniform(key) < model.θ else True
            key, _ = jax.random.split(key)  # Update the random key
        else:
            y_a_sim.append(y_sim[t])
            d_sim.append(0)
            Bp_idx = B_star[B_idx, y_idx]

        q_sim.append(q[Bp_idx, y_idx])
        # Update time and indices
        t += 1
        y_idx = y_sim_indices[t]
        B_idx = Bp_idx

    return (
        jnp.array(y_sim),
        jnp.array(y_a_sim),
        jnp.array(B_sim),
        jnp.array(q_sim),
        jnp.array(d_sim),
    )
```

17.5 Results

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

In what follows, all results are computed using parameter values of `ArellanoEconomy` created by `create_arellano`.

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

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

The first figure shows the bond price schedule and replicates Figure 3 of [Arellano, 2008], where y_L and Y_H are particular below average and above average values of output y.

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

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

The figure shows that

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

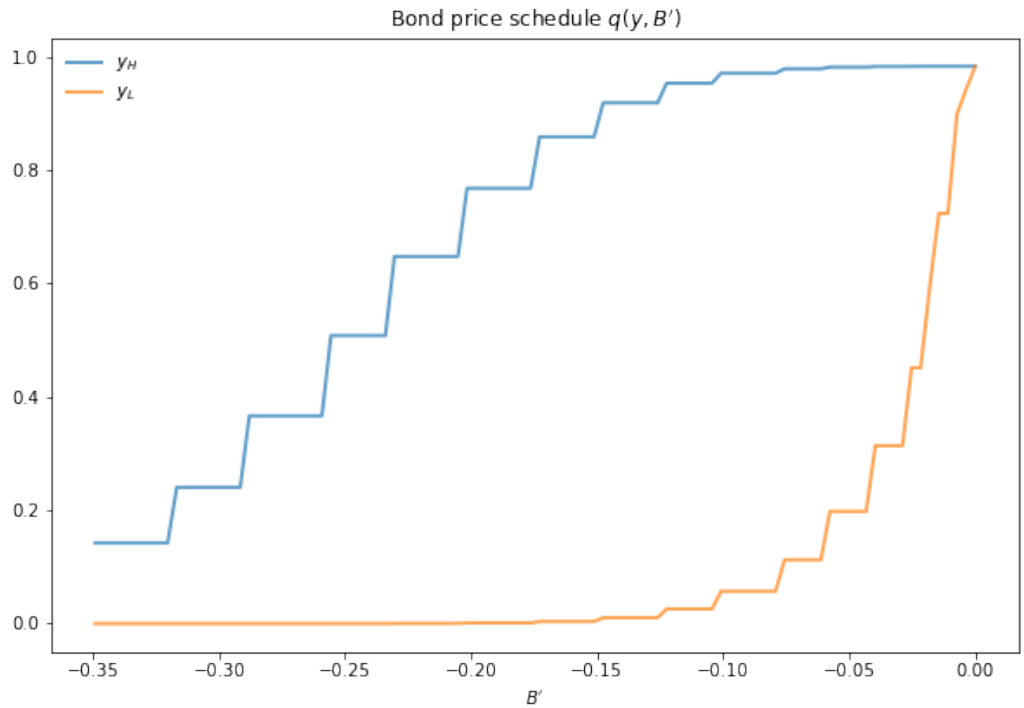
The next figure plots value functions and replicates the right hand panel of Figure 4 of [Arellano, 2008].

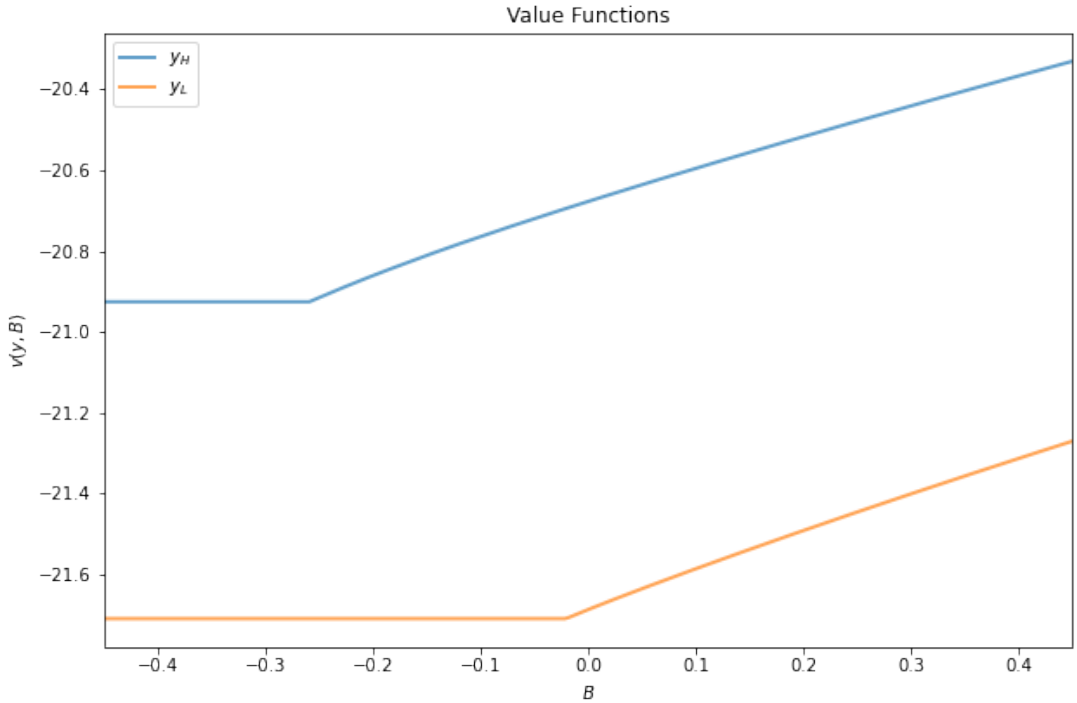
We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (17.4).

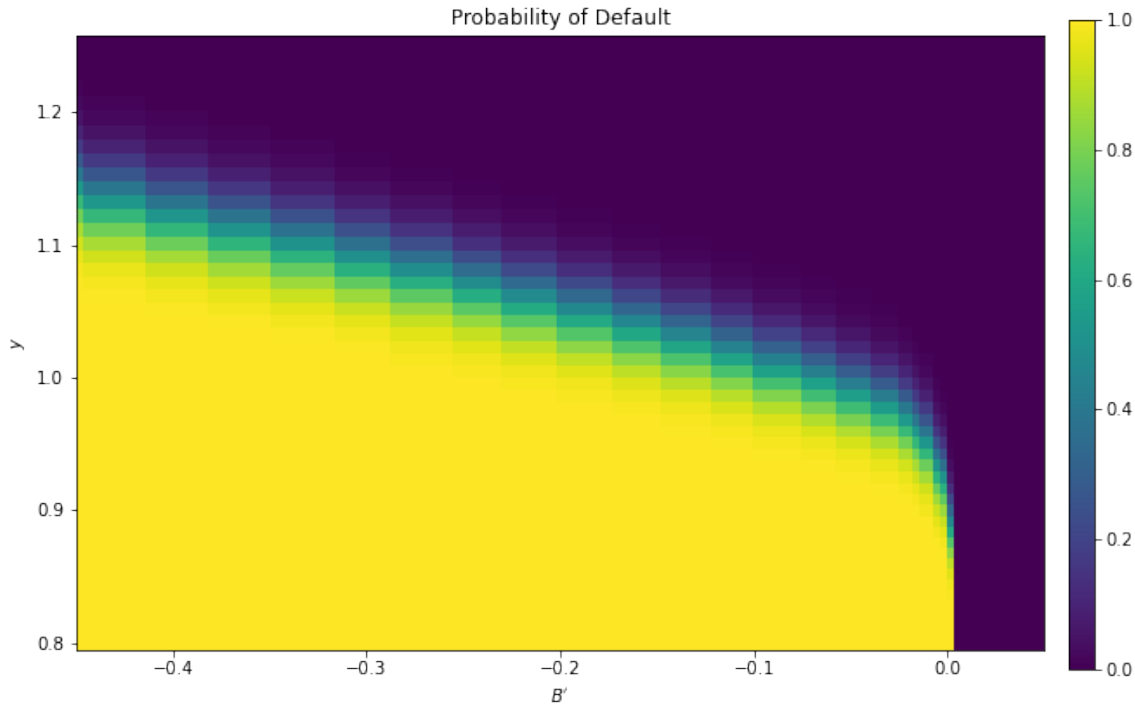
The next plot shows these default probabilities over (B', y) as a heat map.

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

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


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


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



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

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

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

17.6 Exercises

Exercise 17.6.1

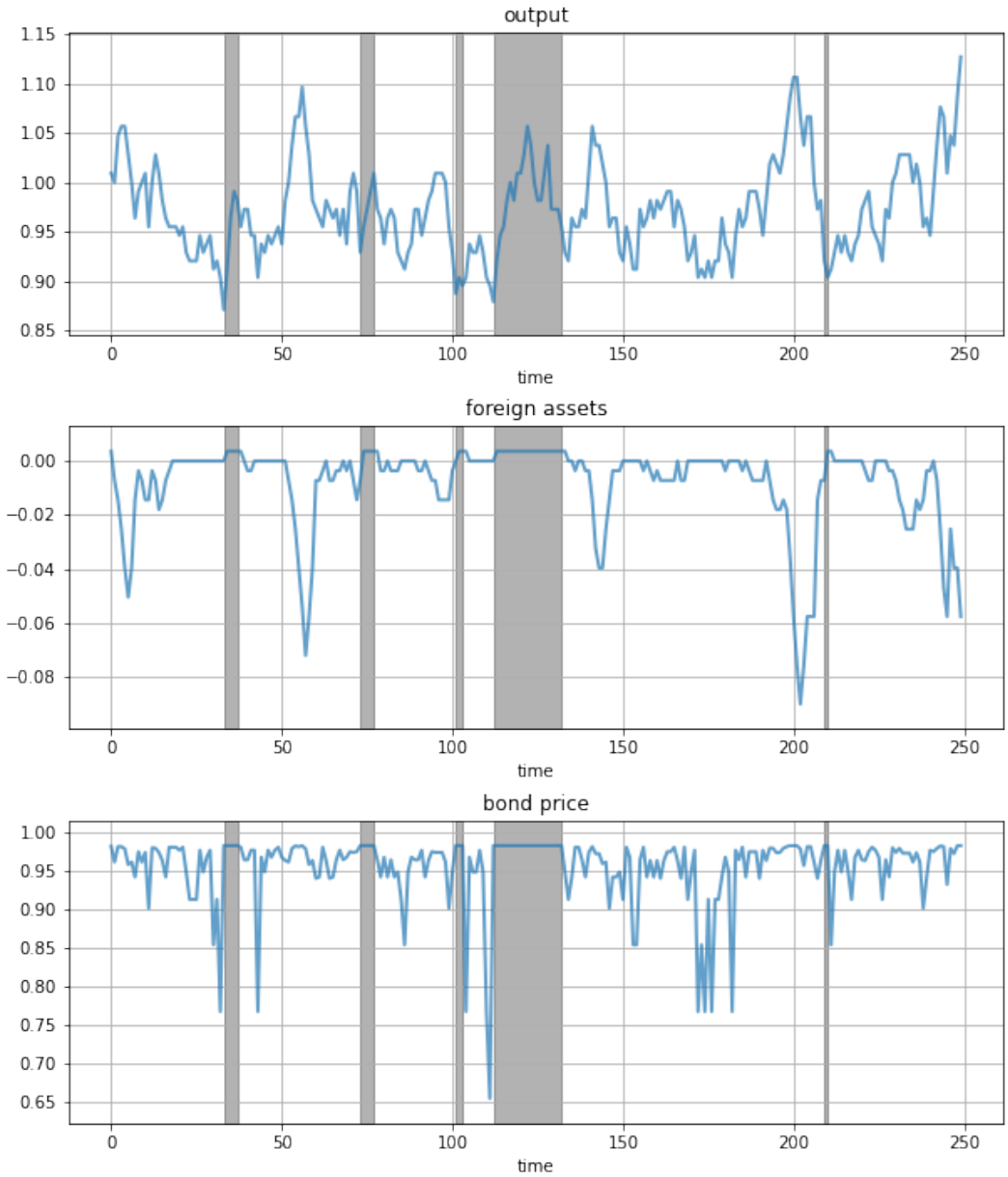
To the extent that you can, replicate the figures shown above

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

Solution

Compute the value function, policy and equilibrium prices

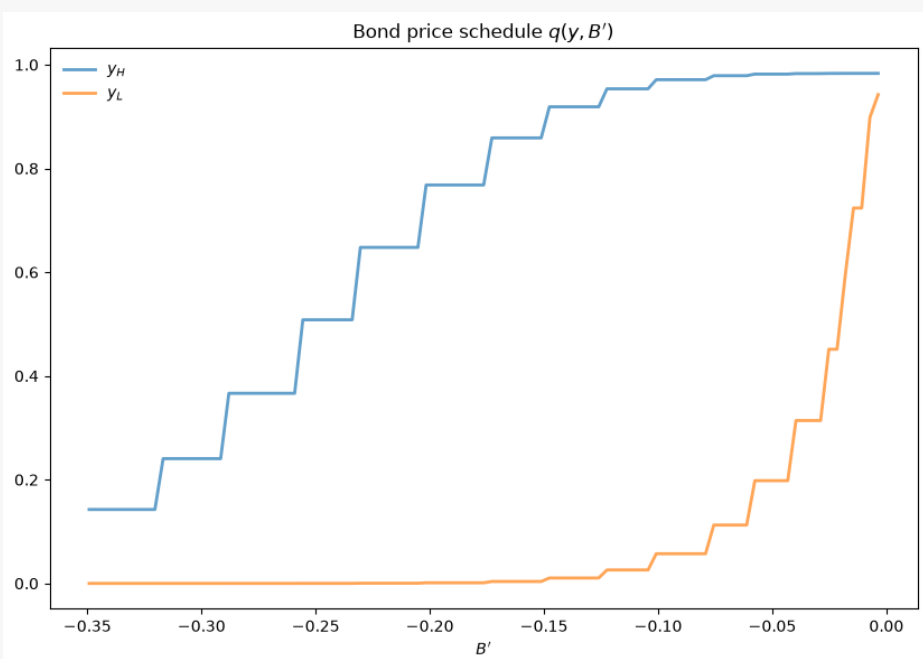
```
ae = create_arellano()
v_c, v_d, q, B_star = solve(ae)
```


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

```
Entering iteration 0 with error 1.00000001.
Entering iteration 100 with error 0.017499341639204857.
Entering iteration 200 with error 0.00014189363558969603.
Entering iteration 300 with error 1.151467966309383e-06.
Terminating at iteration 399.
```

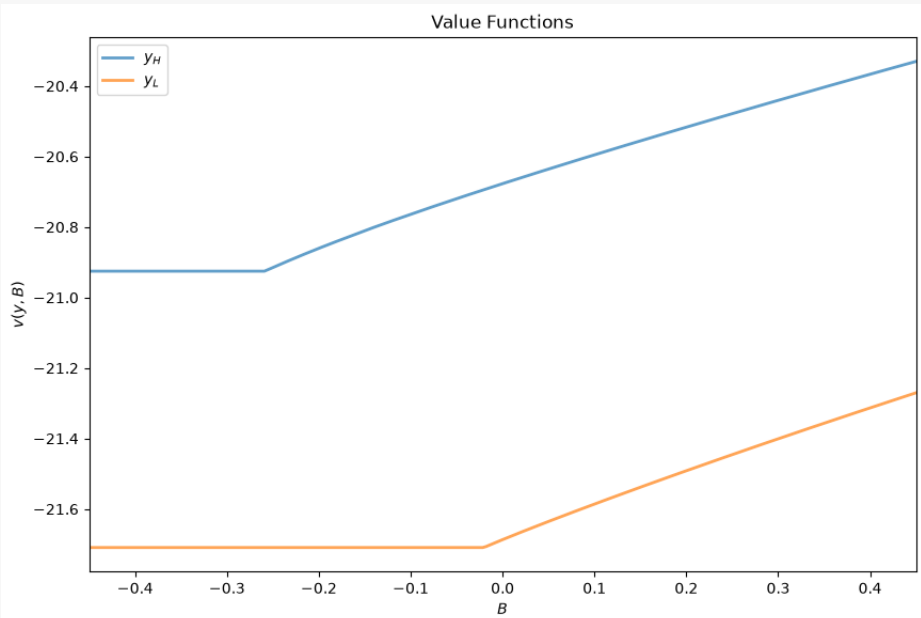
Compute the bond price schedule as seen in figure 3 of [Arellano, 2008]

```
# Unpack some useful names
B_grid, y_grid, P = ae.B_grid, ae.y_grid, ae.P
B_size, y_size = ae.B_size, ae.y_size
r = ae.r

# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = jnp.mean(y_grid) * 1.05, jnp.mean(y_grid) * 0.95
iy_high, iy_low = (jnp.searchsorted(y_grid, x) for x in (high, low))

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

# Extract a suitable plot grid
x = []
q_low = []
q_high = []
for i, B in enumerate(B_grid):
    if -0.35 <= B <= 0:  # To match fig 3 of Arellano (2008)
        x.append(B)
        q_low.append(q[i, iy_low])
        q_high.append(q[i, iy_high])
ax.plot(x, q_high, label="$y_H$", lw=2, alpha=0.7)
ax.plot(x, q_low, label="$y_L$", lw=2, alpha=0.7)
ax.set_xlabel("$B'$")
ax.legend(loc="upper left", frameon=False)
plt.show()
```



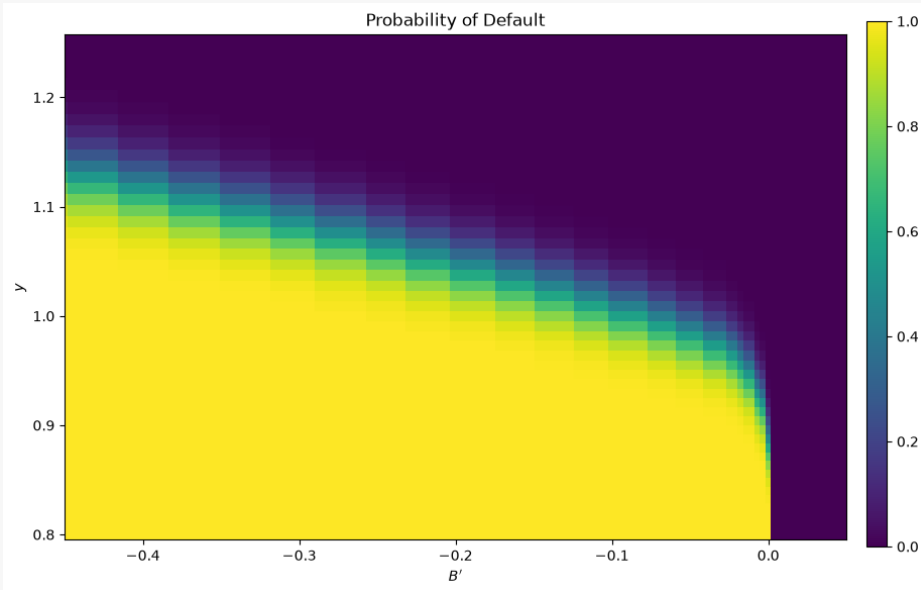
Draw a plot of the value functions

```
v = jnp.maximum(v_c, jnp.reshape(v_d, (1, y_size)))

fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Value Functions")
ax.plot(B_grid, v[:, iy_high], label="$y_H$", lw=2, alpha=0.7)
ax.plot(B_grid, v[:, iy_low], label="$y_L$", lw=2, alpha=0.7)
ax.legend(loc="upper left")
ax.set(xlabel="$B$", ylabel="$v(y, B)$")
ax.set_xlim(min(B_grid), max(B_grid))
plt.show()
```



Draw a heat map for default probability

```
# Set up arrays with indices [i_B, i_y, i_yp]
shaped_v_d = jnp.reshape(v_d, (1, 1, y_size))
shaped_v_c = jnp.reshape(v_c, (B_size, 1, y_size))
shaped_P = jnp.reshape(P, (1, y_size, y_size))

# Compute delta[i_B, i_y]
default_states = 1.0 * (shaped_v_c < shaped_v_d)
delta = jnp.sum(default_states * shaped_P, axis=(2,))

# Create figure
fig, ax = plt.subplots(figsize=(10, 6.5))
hm = ax.pcolormesh(B_grid, y_grid, delta.T)
cax = fig.add_axes([0.92, 0.1, 0.02, 0.8])
fig.colorbar(hm, cax=cax)
ax.axis([B_grid.min(), 0.05, y_grid.min(), y_grid.max()])
ax.set(xlabel="$B'$", ylabel="$y$", title="Probability of Default")
plt.show()
```

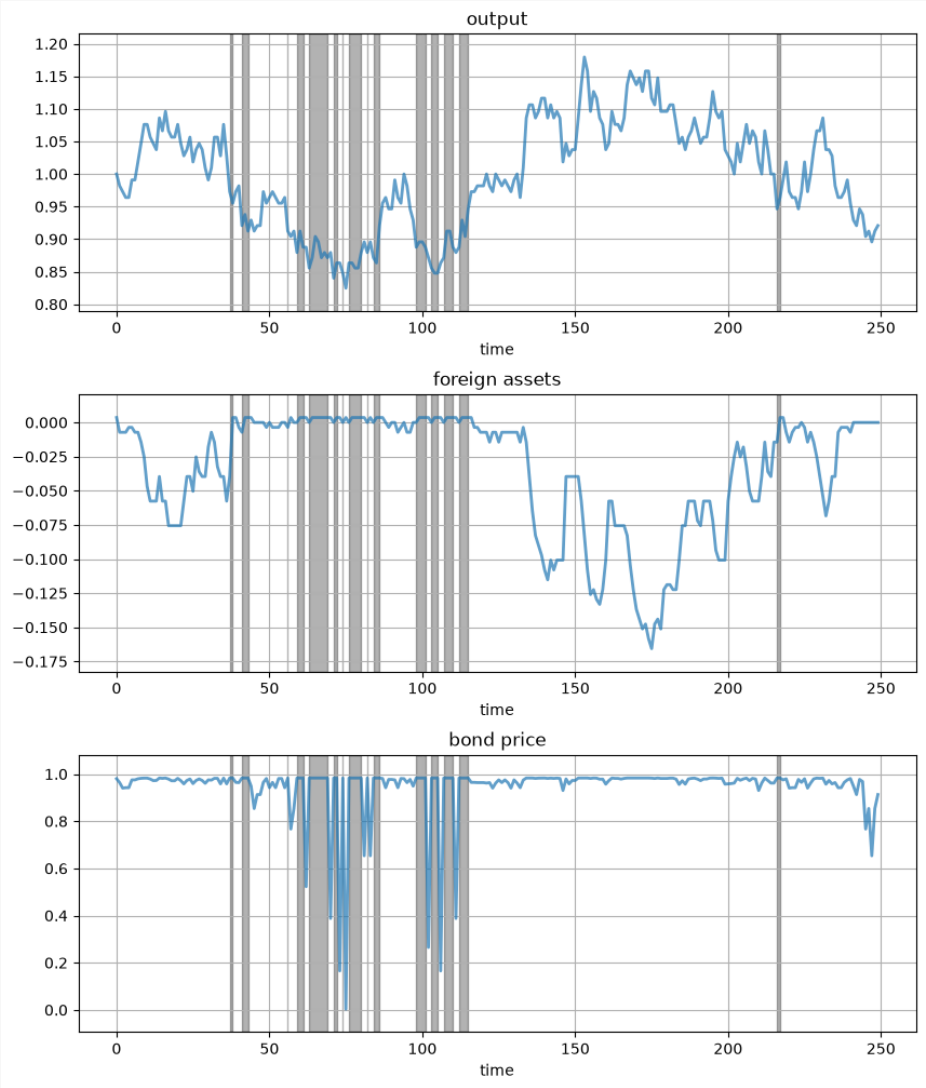


Plot a time series of major variables simulated from the model

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

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

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

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

for ax, series, title in zip(axes, plot_series, titles):
    # Determine suitable y limits
    s_max, s_min = max(series), min(series)
    s_range = s_max - s_min
    y_max = s_max + s_range * 0.1
    y_min = s_min - s_range * 0.1
    ax.set_ylim(y_min, y_max)
    for pair in start_end_pairs:
        ax.fill_between(
            pair, (y_min, y_min), (y_max, y_max), color="k", alpha=0.3
        )
```

```
    ax.grid()
    ax.plot(range(T), series, lw=2, alpha=0.7)
    ax.set(title=title, xlabel="time")

plt.show()
```



CHAPTER

EIGHTEEN

THE AIYAGARI MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

18.1 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [Bew77].

We begin by discussing an example of a Bewley model due to Rao Aiyagari [Aiy94].

The model features

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

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

- precautionary savings and the effect of liquidity constraints [Aiy94]
- risk sharing and asset pricing [HL96]
- the shape of the wealth distribution [BBZ15]

18.1.1 References

The primary reference for this lecture is [Aiy94].

A textbook treatment is available in chapter 18 of [LS18].

A less sophisticated version of this lecture (without JAX) can be found here.

18.1.2 Preliminaries

We use the following imports

```
import time
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from collections import namedtuple
```

```
Matplotlib is building the font cache; this may take a moment.
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:17:26 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                    0 |
| N/A   31C    P8             13W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                                 Usage      |
```

(continues on next page)

(continued from previous page)

```
|==============================================================================================|
|  No running processes found                                                                  ␣
↪        |
+----------------------------------------------------------------------------------------------
↪-------+
```

We will use 64 bit floats with JAX in order to increase the precision.

```
jax.config.update("jax_enable_x64", True)
```

We will use the following function to compute stationary distributions of stochastic matrices. (For a reference to the algorithm, see p. 88 of Economic Dynamics.)

```
@jax.jit
def compute_stationary(P):
    n = P.shape[0]
    I = jnp.identity(n)
    O = jnp.ones((n, n))
    A = I - jnp.transpose(P) + O
    return jnp.linalg.solve(A, jnp.ones(n))
```

18.2 Firms

Firms produce output by hiring capital and labor.

Firms act competitively and face constant returns to scale.

Since returns to scale are constant the number of firms does not matter.

Hence we can consider a single (but nonetheless competitive) representative firm.

The firm's output is

$$Y = AK^\alpha N^{1-\alpha}$$

where

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

The firm's problem is

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

The parameter δ is the depreciation rate.

These parameters are stored in the following namedtuple.

```
Firm = namedtuple('Firm', ('A', 'N', 'α', 'δ'))

def create_firm(A=1.0,
                N=1.0,
                α=0.33,
                δ=0.05):
```

(continues on next page)

(continued from previous page)

```
    """
    Create a namedtuple that stores firm data.

    """
    return Firm(A=A, N=N, α=α, δ=δ)
```

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

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta \tag{18.1}$$

```
def r_given_k(K, firm):
    """
    Inverse demand curve for capital.  The interest rate associated with a
    given demand for capital K.
    """
    A, N, α, δ = firm
    return A * α * (N / K)**(1 - α) - δ
```

Using (18.1) and the firm's first-order condition for labor,

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{18.2}$$

```
def r_to_w(r, firm):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    A, N, α, δ = firm
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))
```

18.3 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of *ex-ante* identical households face a common borrowing constraint.

The savings problem faced by a typical household is

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

subject to

$$a_{t+1} + c_t \leq wz_t + (1+r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate
- r is a net interest rate

- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with given stochastic matrix P.

In this simple version of the model, households supply labor inelastically because they do not value leisure.

Below we provide code to solve the household problem, taking r and w as fixed.

18.3.1 Primitives and Operators

We will solve the household problem using Howard policy iteration (see Ch 5 of Dynamic Programming).

First we set up a namedtuple to store the parameters that define a household asset accumulation problem, as well as the grids used to solve it.

```
Household = namedtuple('Household',
                       ('β', 'a_grid', 'z_grid', 'Π'))
```

```
def create_household(β=0.96,                        # Discount factor
                     Π=[[0.9, 0.1], [0.1, 0.9]],   # Markov chain
                     z_grid=[0.1, 1.0],            # Exogenous states
                     a_min=1e-10, a_max=20,        # Asset grid
                     a_size=200):
    """
    Create a namedtuple that stores all data needed to solve the household
    problem, given prices.

    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    z_grid, Π = map(jnp.array, (z_grid, Π))
    return Household(β=β, a_grid=a_grid, z_grid=z_grid, Π=Π)
```

For now we assume that $u(c) = \log(c)$.

(CRRA utility is treated in the exercises.)

```
u = jnp.log
```

Here's a tuple that stores the wage rate and interest rate, as well as a function that creates a price namedtuple with default values.

```
Prices = namedtuple('Prices', ('r', 'w'))

def create_prices(r=0.01,    # Interest rate
                  w=1.0):    # Wages
    return Prices(r=r, w=w)
```

Now we set up a vectorized version of the right-hand side of the Bellman equation (before maximization), which is a 3D array representing

$$B(a, z, a') = u(wz + (1 + r)a - a') + \beta \sum_{z'} v(a', z')\Pi(z, z')$$

for all (a, z, a').

```
@jax.jit
def B(v, household, prices):
    # Unpack
```

(continues on next page)

(continued from previous page)

```
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)
    r, w = prices

    # Compute current consumption as array c[i, j, ip]
    a  = jnp.reshape(a_grid, (a_size, 1, 1))    # a[i]   ->  a[i, j, ip]
    z  = jnp.reshape(z_grid, (1, z_size, 1))    # z[j]   ->  z[i, j, ip]
    ap = jnp.reshape(a_grid, (1, 1, a_size))    # ap[ip] -> ap[i, j, ip]
    c = w * z + (1 + r) * a - ap

    # Calculate continuation rewards at all combinations of (a, z, ap)
    v = jnp.reshape(v, (1, 1, a_size, z_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Π = jnp.reshape(Π, (1, z_size, 1, z_size))  # Π[j, jp]  -> Π[i, j, ip, jp]
    EV = jnp.sum(v * Π, axis=-1)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, u(c) + β * EV, -jnp.inf)
```

The next function computes greedy policies.

```
@jax.jit
def get_greedy(v, household, prices):
    """
    Computes a v-greedy policy σ, returned as a set of indices.  If
    σ[i, j] equals ip, then a_grid[ip] is the maximizer at i, j.

    """
    return jnp.argmax(B(v, household, prices), axis=-1) # argmax over ap
```

The following function computes the array r_σ which gives current rewards given policy σ.

```
@jax.jit
def compute_r_σ(σ, household, prices):
    """
    Compute current rewards at each i, j under policy σ.  In particular,

        r_σ[i, j] = u((1 + r)a[i] + wz[j] - a'[ip])

    when ip = σ[i, j].

    """
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)
    r, w = prices

    # Compute r_σ[i, j]
    a = jnp.reshape(a_grid, (a_size, 1))
    z = jnp.reshape(z_grid, (1, z_size))
    ap = a_grid[σ]
    c = (1 + r) * a + w * z - ap
    r_σ = u(c)

    return r_σ
```

The value v_σ of a policy σ is defined as

$$v_\sigma = (I - \beta P_\sigma)^{-1} r_\sigma$$

(See Ch 5 of Dynamic Programming for notation and background on Howard policy iteration.)

To compute this vector, we set up the linear map $v \to R_\sigma v$, where $R_\sigma := I - \beta P_\sigma$.

This map can be expressed as

$$(R_\sigma v)(a, z) = v(a, z) - \beta \sum_{z'} v(\sigma(a, z), z')\Pi(z, z')$$

(Notice that R_σ is expressed as a linear operator rather than a matrix – this is much easier and cleaner to code, and also exploits sparsity.)

```
@jax.jit
def R_σ(v, σ, household):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    # Set up the array v[σ[i, j], jp]
    zp_idx = jnp.arange(z_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, z_size))
    σ = jnp.reshape(σ, (a_size, z_size, 1))
    V = v[σ, zp_idx]

    # Expand Π[j, jp] to Π[i, j, jp]
    Π = jnp.reshape(Π, (1, z_size, z_size))

    # Compute and return v[i, j] - β Σ_jp v[σ[i, j], jp] * Π[j, jp]
    return v - β * jnp.sum(V * Π, axis=-1)
```

The next function computes the lifetime value of a given policy.

```
@jax.jit
def get_value(σ, household, prices):
    """
    Get the lifetime value of policy σ by computing

        v_σ = R_σ^{-1} r_σ

    """
    r_σ = compute_r_σ(σ, household, prices)
    # Reduce R_σ to a function in v
    _R_σ = lambda v: R_σ(v, σ, household)
    # Compute v_σ = R_σ^{-1} r_σ using an iterative routing.
    return jax.scipy.sparse.linalg.bicgstab(_R_σ, r_σ)[0]
```

Here's the Howard policy iteration.

```
def howard_policy_iteration(household, prices,
                            tol=1e-4, max_iter=10_000, verbose=False):
    """
    Howard policy iteration routine.

    """
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)
    σ = jnp.zeros((a_size, z_size), dtype=int)

    v_σ = get_value(σ, household, prices)
```

(continues on next page)

(continued from previous page)

```
    i = 0
    error = tol + 1
    while error > tol and i < max_iter:
        σ_new = get_greedy(v_σ, household, prices)
        v_σ_new = get_value(σ_new, household, prices)
        error = jnp.max(jnp.abs(v_σ_new - v_σ))
        σ = σ_new
        v_σ = v_σ_new
        i = i + 1
        if verbose:
            print(f"Concluded loop {i} with error {error}.")
    return σ
```

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

```
# Create an instance of Household
household = create_household()
prices = create_prices()
```

```
r, w = prices
```

```
r, w
```

```
(0.01, 1.0)
```

```
%time σ_star = howard_policy_iteration(household, prices, verbose=True)
```

```
Concluded loop 1 with error 11.366831579022996.
Concluded loop 2 with error 9.574522771860245.
Concluded loop 3 with error 3.9654760004604777.
Concluded loop 4 with error 1.1207075306313232.
Concluded loop 5 with error 0.2524013153055833.
Concluded loop 6 with error 0.12172293662906064.
Concluded loop 7 with error 0.043395682867316765.
Concluded loop 8 with error 0.012132319676439351.
Concluded loop 9 with error 0.005822155404443308.
Concluded loop 10 with error 0.002863165320343697.
Concluded loop 11 with error 0.0016657175376657563.
Concluded loop 12 with error 0.0004143776102245589.
Concluded loop 13 with error 0.0.
CPU times: user 859 ms, sys: 37.9 ms, total: 897 ms
Wall time: 753 ms
```

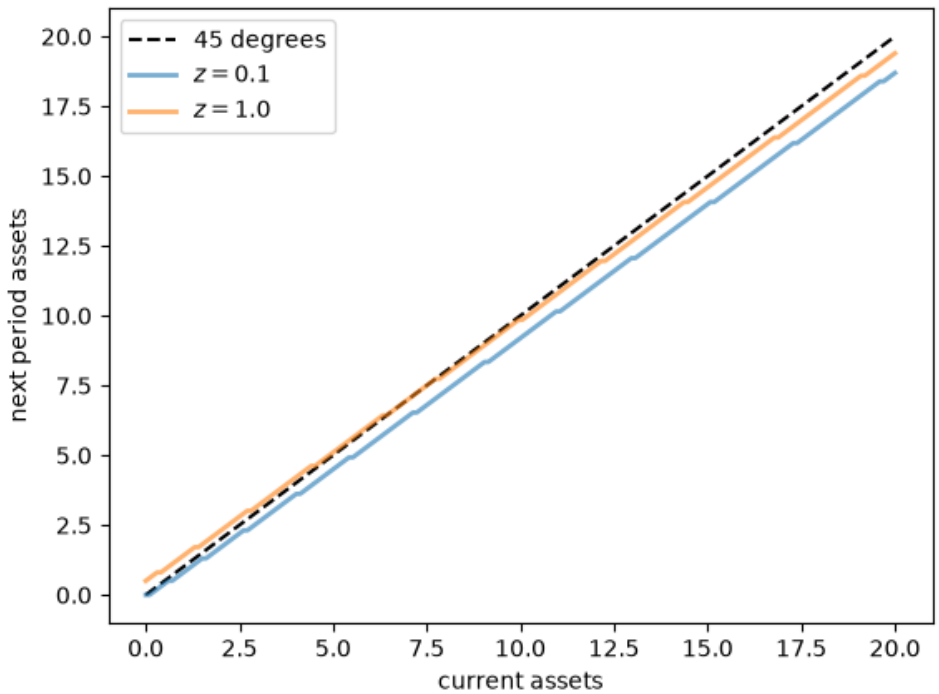
The next plot shows asset accumulation policies at different values of the exogenous state.

```
β, a_grid, z_grid, Π = household

fig, ax = plt.subplots()
ax.plot(a_grid, a_grid, 'k--', label="45 degrees")
for j, z in enumerate(z_grid):
    lb = f'$z = {z:.2}$'
    policy_vals = a_grid[σ_star[:, j]]
    ax.plot(a_grid, policy_vals, lw=2, alpha=0.6, label=lb)
    ax.set_xlabel('current assets')
    ax.set_ylabel('next period assets')
```

(continues on next page)

(continued from previous page)

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



18.3.2 Capital Supply

To start thinking about equilibrium, we need to know how much capital households supply at a given interest rate r.

This quantity can be calculated by taking the stationary distribution of assets under the optimal policy and computing the mean.

The next function computes the stationary distribution for a given policy σ via the following steps:

- compute the stationary distribution $\psi = (\psi(a, z))$ of P_σ, which defines the Markov chain of the state (a_t, z_t) under policy σ.
- sum out z_t to get the marginal distribution for a_t.

```
@jax.jit
def compute_asset_stationary(σ, household):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    # Construct P_σ as an array of the form P_σ[i, j, ip, jp]
    ap_idx = jnp.arange(a_size)
    ap_idx = jnp.reshape(ap_idx, (1, 1, a_size, 1))
    σ = jnp.reshape(σ, (a_size, z_size, 1, 1))
    A = jnp.where(σ == ap_idx, 1, 0)
    Π = jnp.reshape(Π, (1, z_size, 1, z_size))
    P_σ = A * Π
```

(continues on next page)

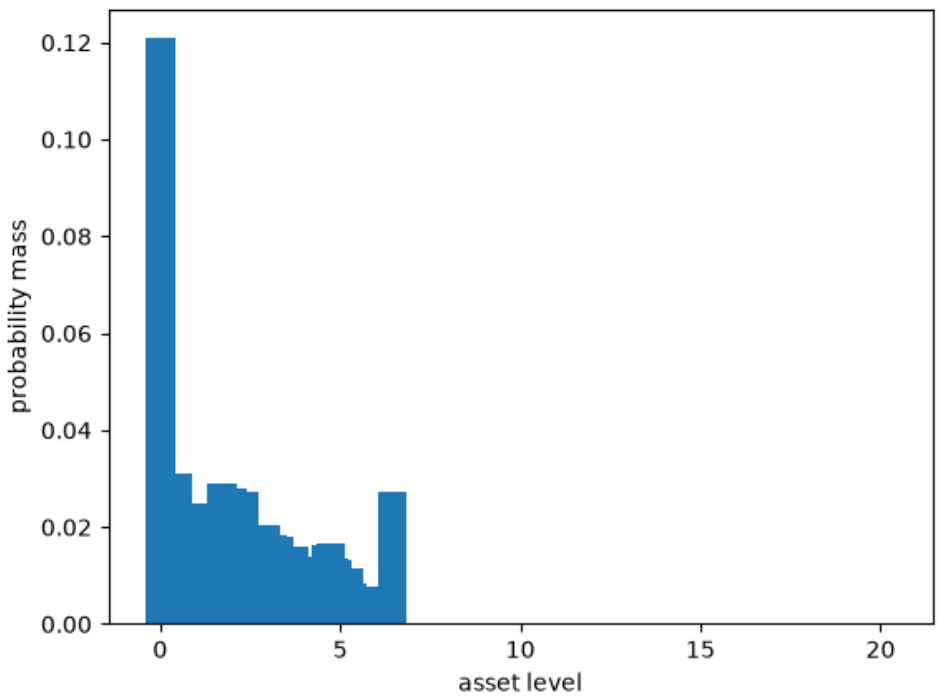
(continued from previous page)

```
    # Reshape P_σ into a matrix
    n = a_size * z_size
    P_σ = jnp.reshape(P_σ, (n, n))

    # Get stationary distribution and reshape back onto [i, j] grid
    ψ = compute_stationary(P_σ)
    ψ = jnp.reshape(ψ, (a_size, z_size))

    # Sum along the rows to get the marginal distribution of assets
    ψ_a = jnp.sum(ψ, axis=1)
    return ψ_a
```

Let's give this a test run.

```
ψ_a = compute_asset_stationary(σ_star, household)
```

```
fig, ax = plt.subplots()
ax.bar(household.a_grid, ψ_a)
ax.set_xlabel("asset level")
ax.set_ylabel("probability mass")
plt.show()
```



The distribution should sum to one:

```
ψ_a.sum()
```

```
Array(1., dtype=float64)
```

The next function computes aggregate capital supply by households under policy σ, given wages and interest rates.

```
def capital_supply(σ, household):
    """
    Induced level of capital stock under the policy, taking r and w as given.

    """
    β, a_grid, z_grid, Π = household
    ψ_a = compute_asset_stationary(σ, household)
    return float(jnp.sum(ψ_a * a_grid))
```

18.4 Equilibrium

We compute a **stationary rational expectations equilibrium** (SREE) as follows:

1. set $n = 0$, start with initial guess K_0 for aggregate capital
2. determine prices r, w from the firm decision problem, given K_n
3. compute the optimal savings policy of the households given these prices
4. compute aggregate capital K_{n+1} as the mean of steady state capital given this savings policy
5. if $K_{n+1} \approx K_n$ stop, otherwise go to step 2.

We can write the sequence of operations in steps 2-4 as

$$K_{n+1} = G(K_n)$$

If K_{n+1} agrees with K_n then we have a SREE.

In other words, our problem is to find the fixed-point of the one-dimensional map G.

Here's G expressed as a Python function:

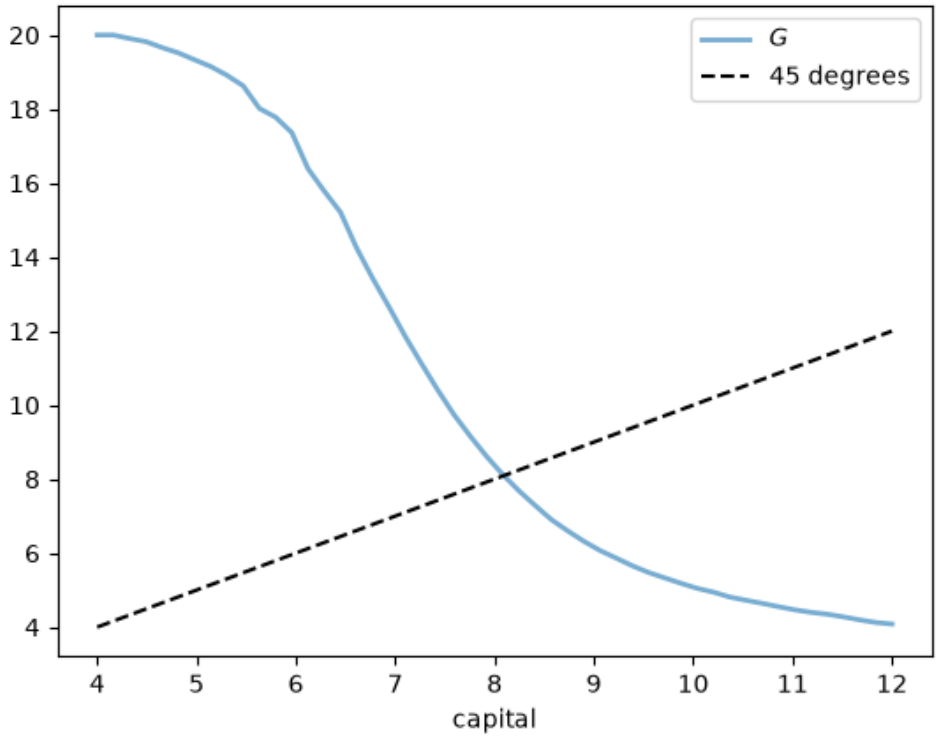
```
def G(K, firm, household):
    # Get prices r, w associated with K
    r = r_given_k(K, firm)
    w = r_to_w(r, firm)
    # Generate a household object with these prices, compute
    # aggregate capital.
    prices = create_prices(r=r, w=w)
    σ_star = howard_policy_iteration(household, prices)
    return capital_supply(σ_star, household)
```

18.4.1 Visual inspection

Let's inspect visually as a first pass.

```
num_points = 50
firm = create_firm()
household = create_household()
k_vals = np.linspace(4, 12, num_points)
out = [G(k, firm, household) for k in k_vals]
```

```
fig, ax = plt.subplots()
ax.plot(k_vals, out, lw=2, alpha=0.6, label='$G$')
ax.plot(k_vals, k_vals, 'k--', label="45 degrees")
ax.set_xlabel('capital')
ax.legend()
plt.show()
```



18.4.2 Computing the equilibrium

Now let's compute the equilibrium.

Looking at the figure above, we see that a simple iteration scheme $K_{n+1} = G(K_n)$ will cycle from high to low values, leading to slow convergence.

As a result, we use a damped iteration scheme of the form

$$K_{n+1} = \alpha K_n + (1 - \alpha) G(K_n)$$

```
def compute_equilibrium(firm, household,
                        K0=6, α=0.99, max_iter=1_000, tol=1e-4,
                        print_skip=10, verbose=False):
    n = 0
    K = K0
    error = tol + 1
    while error > tol and n < max_iter:
        new_K = α * K + (1 - α) * G(K, firm, household)
        error = abs(new_K - K)
        K = new_K
        n += 1
```

(continues on next page)

(continued from previous page)

```
        if verbose and n % print_skip == 0:
            print(f"At iteration {n} with error {error}")
    return K, n
```

```
firm = create_firm()
household = create_household()
print("\nComputing equilibrium capital stock")
start = time.time()
K_star, n = compute_equilibrium(firm, household, K0=6.0, verbose=True)
elapsed = time.time() - start
print(f"Computed equilibrium {K_star:.5} in {n} iterations and {elapsed} seconds")
```

```
Computing equilibrium capital stock
At iteration 10 with error 0.06490545798825043
At iteration 20 with error 0.03626685465742874
At iteration 30 with error 0.021452703682580676
At iteration 40 with error 0.013383829253038826
At iteration 50 with error 0.008397515264451982
At iteration 60 with error 0.005656370053102933
At iteration 70 with error 0.003750107819944226
At iteration 80 with error 0.0024816210955460605
At iteration 90 with error 0.0017352118525710836
At iteration 100 with error 0.0009559876839571047
At iteration 110 with error 0.0006843412658508186
At iteration 120 with error 0.00046762845631675987
At iteration 130 with error 0.00038193101011607666
At iteration 140 with error 0.00019421626809901227
At iteration 150 with error 0.00017041810881579522
At iteration 160 with error 0.00015412308287032772
At iteration 170 with error 0.00013938615349395889
Computed equilibrium 8.0918 in 176 iterations and 40.20579504966736 seconds
```

This is not very fast, given how quickly we can solve the household problem.

You can try varying α, but usually this parameter is hard to set a priori.

In the exercises below you will be asked to use bisection instead, which generally performs better.

18.5 Exercises

Exercise 18.5.1

Write a new version of `compute_equilibrium` that uses `bisect` from `scipy.optimize` instead of damped iteration.

See if you can make it faster that the previous version.

In `bisect`,

- you should set `xtol=1e-4` to have the same error tolerance as the previous version.
- for the lower and upper bounds of the bisection routine try `a = 1.0` and `b = 20.0`.

Solution

```
from scipy.optimize import bisect
```

We use bisection to find the zero of the function $h(k) = k - G(k)$.

```
def compute_equilibrium(firm, household, a=1.0, b=20.0):
    K = bisect(lambda k: k - G(k, firm, household), a, b, xtol=1e-4)
    return K
```

```
firm = create_firm()
household = create_household()
print("\nComputing equilibrium capital stock")
start = time.time()
K_star = compute_equilibrium(firm, household)
elapsed = time.time() - start
print(f"Computed equilibrium capital stock {K_star:.5} in {elapsed} seconds")
```

```
Computing equilibrium capital stock
Computed equilibrium capital stock 8.0938 in 0.9165403842926025 seconds
```

Bisection seems to be faster than the damped iteration scheme.

Exercise 18.5.2

Show how equilibrium capital stock changes with β.

Use the following values of β and plot the relationship you find.

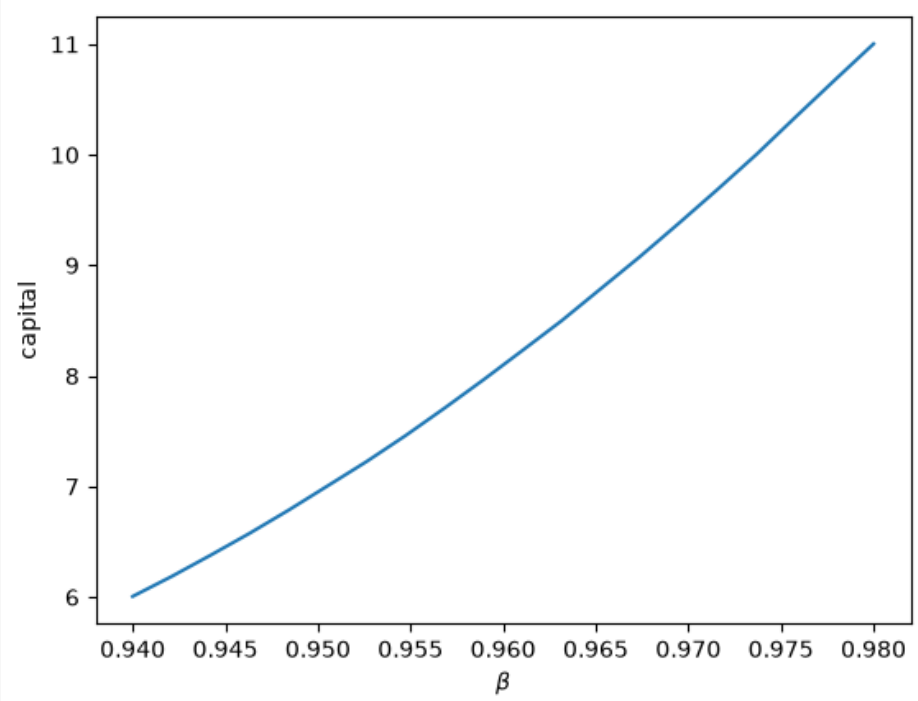
```
β_vals = np.linspace(0.94, 0.98, 20)
```

Solution

```
K_vals = np.empty_like(β_vals)
K = 6.0  # initial guess

for i, β in enumerate(β_vals):
    household = create_household(β=β)
    K = compute_equilibrium(firm, household, 0.5 * K, 1.5 * K)
    print(f"Computed equilibrium {K:.4} at β = {β}")
    K_vals[i] = K
```

```
Computed equilibrium 6.006 at β = 0.94
Computed equilibrium 6.186 at β = 0.9421052631578947
Computed equilibrium 6.379 at β = 0.9442105263157894
Computed equilibrium 6.577 at β = 0.9463157894736841
Computed equilibrium 6.786 at β = 0.9484210526315789
Computed equilibrium 7.005 at β = 0.9505263157894737
Computed equilibrium 7.226 at β = 0.9526315789473684
Computed equilibrium 7.461 at β = 0.9547368421052631
Computed equilibrium 7.709 at β = 0.9568421052631578
Computed equilibrium 7.966 at β = 0.9589473684210525
Computed equilibrium 8.231 at β = 0.9610526315789474
Computed equilibrium 8.499 at β = 0.9631578947368421
Computed equilibrium 8.787 at β = 0.9652631578947368
Computed equilibrium 9.076 at β = 0.9673684210526315
Computed equilibrium 9.378 at β = 0.9694736842105263
Computed equilibrium 9.687 at β = 0.971578947368421
Computed equilibrium 10.0 at β = 0.9736842105263157
Computed equilibrium 10.34 at β = 0.9757894736842105
Computed equilibrium 10.67 at β = 0.9778947368421053
Computed equilibrium 11.0 at β = 0.98
```

```
fig, ax = plt.subplots()
ax.plot(β_vals, K_vals, ms=2)
ax.set_xlabel(r'$\beta$')
ax.set_ylabel('capital')
plt.show()
```



CHAPTER

NINETEEN

THE HOPENHAYN ENTRY-EXIT MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

19.1 Outline

The Hopenhayn (1992, ECMA) entry-exit model is a highly influential (partial equilibrium) heterogeneous agent model where

- the agents receiving idiosyncratic shocks are firms, and
- the model produces a non-trivial firm size distribution and a range of predictions for firm dynamics.

We are going to study an extension of the basic model that has unbounded productivity.

Among other things, this allows us to match the heavy tail observed in the firm size distribution data.

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

```
!pip install quantecon
```

We will use the following imports.

```
import jax.numpy as jnp
import jax
import quantecon as qe
import matplotlib.pyplot as plt
from collections import namedtuple
```

19.2 The Model

19.2.1 Environment

There is a single good produced by a continuum of competitive firms.

The productivity $\varphi_t = \varphi_t^i$ of each firm evolves randomly on $\mathbb{R}_+$.

- the i superscript indicates the i-th firm and we drop it in what follows

A firm with productivity φ facing output price p and wage rate w earns current profits

$$\pi(\varphi, p) = p\varphi n^\theta - c - wn$$

Here

- $\theta \in (0, 1)$ is a productivity parameter,
- c is a fixed cost and
- n is labor input.

Maximizing over n leads to

$$\pi(\varphi, p) = (1-\theta)(p\varphi)^\eta \left(\frac{\theta}{w}\right)^{\theta/\eta} - c$$

where $\eta := 1/(1-\theta)$.

Output is

$$q(\varphi, p) = \varphi^\eta \left(\frac{p\theta}{w}\right)^{\theta/\eta}$$

Productivity of incumbents is independent across firms and grows according to

$$\varphi_{t+1} \sim \Gamma(\varphi_t, d\varphi')$$

Here Γ is a Markov transition kernel, so that $\Gamma(\varphi, d\varphi')$ is the distribution over $\mathbb{R}_+$ given $\varphi \in \mathbb{R}_+$.

For example,

$$\int \pi(\varphi', p)\Gamma(\varphi, d\varphi') = \mathbb{E}[\pi(\varphi_{t+1}, p) \,|\, \varphi_t = \varphi]$$

New entrants obtain their productivity draw from a fixed distribution γ.

Demand is

$$D(p) = 1/p$$

19.2.2 Firm decisions

The intertemporal firm decision problem is choosing when to exit.

The timing is:

1. produce and receive current profits $\pi(\varphi, p)$
2. decide whether to continue or exit.

Scrap value is set to zero and the discount rate is $\beta \in (0, 1)$.

The firm makes their stop-continue choice by first solving for the value function, which is the unique solution in a class of continuous functions $\mathcal{C}$ (details omitted) to

$$v(\varphi, p) = \pi(\varphi, p) + \beta \max \left\{ 0, \int v(\varphi', p) \Gamma(\varphi, d\varphi') \right\}. \tag{19.1}$$

Let $\bar{v}$ be the unique solution to this functional equation.

Given $\bar{v}$, we let $\bar{\varphi}$ be the exit threshold function defined by

$$\bar{\varphi}(p) := \min \left\{ \varphi \geq 0 \ \Big| \ \int \bar{v}(\varphi', p) \, \Gamma(\varphi, d\varphi') \geq 0 \right\}. \tag{19.2}$$

With the convention that incumbents who are indifferent remain rather than exit, an incumbent with productivity φ exits if and only if $\varphi < \bar{\varphi}(p)$.

19.2.3 Equilibrium

Now we describe a stationary recursive equilibrium in the model, where

- the goods market clears (supply equals demand)
- the firm size distribution is stationary over time (an invariance condition)
- the mass of entrants equals the mass of exiting firms, and
- entering firms satisfy a break-even condition.

The break-even condition means that expected lifetime value given an initial draw from γ is just equal to the fixed cost of entry, which we denote by c_e.

Let's now state this more formally

To begin, let $\mathcal{B}$ be the Borel subsets of $\mathbb{R}_+$ and $\mathcal{M}$ be all measures on $\mathcal{B}$.

Taking $\bar{v}$ and $\bar{\varphi}$ as defined in the previous section, a **stationary recursive equilibrium** is a triple

$$(p, M, \mu) \quad \text{in} \quad \mathcal{E} := (0, \infty) \times (0, \infty) \times \mathcal{M},$$

with p understood as price, M as mass of entrants, and μ as a distribution of firms over productivity levels, such that the goods market clears, or

$$\int q(\varphi, p) \mu(d\varphi) = D(p), \tag{19.3}$$

the *invariance condition* over the firm distribution

$$\mu(B) = \int \Gamma(\varphi, B) \, \mathbf{1}\{\varphi \geq \bar{\varphi}(p)\} \, \mu(d\varphi) + M \, \gamma(B) \text{ for all } B \in \mathcal{B}, \tag{19.4}$$

holds (see below for intuition), the *equilibrium entry condition*

$$\int \bar{v}(\varphi, p) \, \gamma(d\varphi) = c_e \tag{19.5}$$

holds, and the *balanced entry and exit condition*

$$M = \mu\{\varphi < \bar{\varphi}(p)\} \tag{19.6}$$

is verified.

The invariance condition says that, for any subset B of the state space, the mass of firms with productivity in set B today (the left-hand side) is equal to the mass of firms with productivity in set B tomorrow (the right-hand side).

19.2.4 Computing equilibrium

We compute the equilibrium as follows:

For the purposes of this section, we insert balanced entry and exit into the time invariance condition, yielding

$$\mu(B) = \int \Pi(\varphi, B)\mu(d\varphi) \quad \text{for all } B \in \mathcal{B}, \tag{19.7}$$

where Π is the transition kernel on $\mathbb{R}_+$ defined by

$$\Pi(\varphi, B) = \Gamma(\varphi, B)\mathbf{1}\{\varphi \geq \bar{\varphi}(p)\} + \mathbf{1}\{\varphi < \bar{\varphi}(p)\}\gamma(B). \tag{19.8}$$

Under our assumptions, for each $p > 0$, there exists a unique μ satisfying this invariance law.

Let $\mathcal{P}$ be the probability measures on $\mathbb{R}_+$.

The unique stationary equilibrium can be computed as follows:

1. Obtain $\bar{v}$ as the unique solution to the Bellman equation, and then calculate the exit threshold function $\bar{\varphi}$.
2. Solve for the equilibrium entry price p^* by solving the entry condition.
3. Define Π as above, using p^* as the price, and compute μ as the unique solution to the invariance condition.
4. Rescale μ by setting $s := D(p^*)/\int q(\varphi, p^*)\mu(d\varphi)$ and then $\mu^* := s\,\mu$.
5. Obtain the mass of entrants via $M^* = \mu^*\{\varphi < \bar{\varphi}(p^*)\}$.

When we compute the distribution μ in step 3, we will use the fact that it is the stationary distribution of the Markov transition kernel $\Pi(\varphi, d\varphi)$.

This transition kernel turns out to be *ergodic*, which means that if we simulate a cross-section of firms according to Π for a large number of periods, the resulting sample (cross-section of productivities) will approach a set of IID draws from μ.

This allows us to

1. compute integrals with respect to the distribution using Monte Carlo, and
2. investigate the shape and properties of the stationary distribution.

19.2.5 Specification of dynamics

Before solving the model we need to specify Γ and γ.

We assume $\Gamma(\varphi, d\varphi')$ is given by

$$\varphi_{t+1} = A_{t+1}\varphi_t$$

We assume that (A_t) is IID over time, independent across firms, and lognormal $LN(m_a, \sigma_a)$.

(This means that incumbents follow Gibrat's law, which is a reasonable assumption for medium to large firms – and hence for incumbents.)

New entrants are drawn from a lognormal distribution $LN(m_e, \sigma_e)$.

19.3 Code

We use 64 bit floats for extra precision.

```
jax.config.update("jax_enable_x64", True)
```

We store the parameters, grids and Monte Carlo draws in a `namedtuple`.

```
Parameters = namedtuple("Parameters",
    ("β",              # discount factor
     "θ",              # labor productivity
     "c",              # fixed cost in production
     "c_e",            # entry cost
     "w",              # wages
     "m_a",            # productivity shock location parameter
     "σ_a",            # productivity shock scale parameter
     "m_e",            # new entrant location parameter
     "σ_e"))           # new entrant scale parameter
```

```
Grids = namedtuple("Grids",
    ("ψ_grid",         # productivity grid
     "E_draws",        # entry size draws for Monte Carlo
     "A_draws"))       # productivity shock draws for Monte Carlo
```

```
Model = namedtuple("Model",
    ("parameters",     # instance of Parameters
     "grids"))         # instance of Grids
```

```
def create_model(β=0.95,              # discount factor
                 θ=0.3,               # labor productivity
                 c=4.0,               # fixed cost in production
                 c_e=1.0,             # entry cost
                 w=1.0,               # wages
                 m_a=-0.012,          # productivity shock location parameter
                 σ_a=0.1,             # productivity shock scale parameter
                 m_e=1.0,             # new entrant location parameter
                 σ_e=0.2,             # new entrant scale parameter
                 ψ_grid_max=5,        # productivity grid max
                 ψ_grid_size=100,     # productivity grid size
                 E_draw_size=200,     # entry MC integration size
                 A_draw_size=200,     # prod shock MC integration size
                 seed=1234):          # Seed for MC draws
    """
    Create an instance of the `namedtuple` Model using default parameter values.
    """

    # Test stability
    assert m_a + σ_a**2 / (2 * (1 - θ)) < 0, "Stability condition fails"
    # Build grids and initialize random number generator
    ψ_grid = jnp.linspace(0, ψ_grid_max, ψ_grid_size)
    key, subkey = jax.random.split(jax.random.key(seed))
    # Generate a sample of draws of A for Monte Carlo integration
    A_draws = jnp.exp(m_a + σ_a * jax.random.normal(key, (A_draw_size,)))
    # Generate a sample of draws from γ for Monte Carlo
    E_draws = jnp.exp(m_e + σ_e * jax.random.normal(subkey, (E_draw_size,)))
    # Build namedtuple and return
```

(continues on next page)

(continued from previous page)

```
    parameters = Parameters(β, θ, c, c_e, w, m_a, σ_a, m_e, σ_e)
    grids = Grids(ψ_grid, E_draws, A_draws)
    model = Model(parameters, grids)
    return model
```

Let us write down functions for profits and output.

```
@jax.jit
def π(ψ, p, parameters):
    """ Profits. """
    # Unpack
    β, θ, c, c_e, w, m_a, σ_a, m_e, σ_e = parameters
    # Compute profits
    return (1 - θ) * (p * ψ)**(1/(1 - θ)) * (θ/w)**(θ/(1 - θ)) - c
```

```
@jax.jit
def q(ψ, p, parameters):
    """ Output. """
    # Unpack
    β, θ, c, c_e, w, m_a, σ_a, m_e, σ_e = parameters
    # Compute output
    return ψ**(1/(1 - θ)) * (p * θ/w)**(θ/(1 - θ))
```

Let's write code to simulate a cross-section of firms given a particular value for the exit threshold (rather than an exit threshold function).

Firms that exit are immediately replaced by a new entrant, drawn from γ.

Our first function updates by one step

```
def update_cross_section(ψ_bar, ψ_vec, key, parameters, num_firms):
    # Unpack
    β, θ, c, c_e, w, m_a, σ_a, m_e, σ_e = parameters
    # Update
    Z = jax.random.normal(key, (2, num_firms))  # long rows for row-major arrays
    incumbent_draws = ψ_vec * jnp.exp(m_a + σ_a * Z[0, :])
    new_firm_draws = jnp.exp(m_e + σ_e * Z[1, :])
    return jnp.where(ψ_vec >= ψ_bar, incumbent_draws, new_firm_draws)
```

```
update_cross_section = jax.jit(update_cross_section, static_argnums=(4,))
```

Our next function runs the cross-section forward in time `sim_length` periods.

```
def simulate_firms(ψ_bar, parameters, grids,
                   sim_length=200, num_firms=1_000_000, seed=12):
    """
    Simulate a cross-section of firms when the exit threshold is ψ_bar.

    """
    # Set initial conditions to the threshold value
    ψ_vec = jnp.ones((num_firms,)) * ψ_bar
    key = jax.random.key(seed)
    # Iterate forward in time
    for t in range(sim_length):
        key, subkey = jax.random.split(key)
        ψ_vec = update_cross_section(ψ_bar, ψ_vec, subkey, parameters, num_firms)
    return ψ_vec
```

Here's a utility function to compute the expected value

$$\int v(\varphi')\Gamma(\varphi, d\varphi') = \mathbb{E}v(A_{t+1}\varphi)$$

given φ

```
@jax.jit
def _compute_exp_value_at_phi(v, ψ, grids):
    """
    Compute

        E[v(ψ')| ψ] = Ev(A ψ)

    using linear interpolation and Monte Carlo.
    """
    # Unpack
    ψ_grid, E_draws, A_draws = grids
    # Set up V
    Aψ = A_draws * ψ
    vAψ  = jnp.interp(Aψ, ψ_grid, v)  # v(A_j ψ) for all j
    # Return mean
    return jnp.mean(vAψ)      # (1/n) Σ_j v(A_j ψ)
```

Now let's vectorize this function in φ and then write another function that computes the expected value across all φ in `ψ_grid`

```
compute_exp_value_at_phi = jax.vmap(_compute_exp_value_at_phi, (None, 0, None))
```

```
@jax.jit
def compute_exp_value(v, grids):
    """
    Compute

        E[v(ψ_prime)| ψ] = Ev(A ψ) for all ψ, as a vector

    """
    # Unpack
    ψ_grid, E_draws, A_draws = grids
    return compute_exp_value_at_phi(v, ψ_grid, grids)
```

Here is the Bellman operator T.

```
@jax.jit
def T(v, p, parameters, grids):
    """ Bellman operator. """
    # Unpack
    β, θ, c, c_e, w, m_a, σ_a, m_e, σ_e = parameters
    ψ_grid, E_draws, A_draws = grids
    # Compute Tv
    EvAψ = compute_exp_value(v, grids)
    return π(ψ_grid, p, parameters) + β * jnp.maximum(0.0, EvAψ)
```

The next function takes v, p as inputs and, conditional on the value function v, computes the value $\bar{\varphi}(p)$ that corresponds to the exit value.

```
@jax.jit
def get_threshold(v, grids):
```

(continues on next page)

(continued from previous page)

```
    """ Compute the exit threshold. """
    # Unpack
    ψ_grid, E_draws, A_draws = grids
    # Compute exit threshold: ψ such that E v(A ψ) = 0
    EvAψ = compute_exp_value(v, grids)
    i = jnp.searchsorted(EvAψ, 0.0)
    return ψ_grid[i]
```

We use value function iteration (VFI) to compute the value function $\bar{v}(\cdot, p)$, taking p as given.

VFI is relatively cheap and simple in this setting.

```
@jax.jit
def vfi(p, v_init, parameters, grids, tol=1e-6, max_iter=10_000):
    """
    Implement value function iteration to solve for the value function.
    """
    # Unpack
    ψ_grid, E_draws, A_draws = grids
    # Set up
    def cond_function(state):
        i, v, error = state
        return jnp.logical_and(i < max_iter, error > tol)

    def body_function(state):
        i, v, error = state
        new_v = T(v, p, parameters, grids)
        error = jnp.max(jnp.abs(v - new_v))
        i += 1
        return i, new_v, error

    # Loop till convergence
    init_state = 0, v_init, tol + 1
    state = jax.lax.while_loop(cond_function, body_function, init_state)
    i, v, error = state
    return v
```

```
@jax.jit
def compute_net_entry_value(p, v_init, parameters, grids):
    """
    Returns the net value of entry, which is

         \int v_bar(ψ, p) γ(d ψ) - c_e

    This is the break-even condition for new entrants.  The argument
    v_init is used as an initial condition when computing v_bar for VFI.
    """
    c_e = parameters.c_e
    ψ_grid = grids.ψ_grid
    E_draws = grids.E_draws
    v_bar = vfi(p, v_init, parameters, grids)
    v_ψ = jnp.interp(E_draws, ψ_grid, v_bar)
    Ev_ψ = jnp.mean(v_ψ)
    return Ev_ψ - c_e, v_bar
```

We need to solve for the equilibrium price, which is the p satisfying

$$\int \bar{v}(\varphi', p)\gamma(d\varphi') = c_e$$

At each price p, we need to recompute $\bar{v}(\cdot, p)$ and then take the expectation.

The technique we will use is bisection.

We will write our own bisection routine because, when we shift to a new price, we want to update the initial condition for value function iteration to the value function from the previous price.

```
def compute_p_star(parameters, grids, p_min=1.0, p_max=2.0, tol=10e-5):
    """
    Compute the equilibrium entry price p^* via bisection.

    Return both p* and the corresponding value function v_bar, which is
    computed as a byproduct.

    Implements the bisection root finding algorithm to find p_star

    """
    φ_grid, E_draws, A_draws = grids
    lower, upper = p_min, p_max
    v_bar = jnp.zeros_like(φ_grid)  # Initial condition at first price guess

    while upper - lower > tol:
        mid = 0.5 * (upper + lower)
        entry_val, v_bar = compute_net_entry_value(mid, v_bar, parameters, grids)
        if entry_val > 0:   # Root is between lower and mid
            lower, upper = lower, mid
        else:               # Root is between mid and upper
            lower, upper = mid, upper

    p_star = 0.5 * (upper + lower)
    return p_star, v_bar
```

We are now ready to compute all elements of the stationary recursive equilibrium.

```
def compute_equilibrium_prices_and_quantities(model):
    """
    Compute

        1. The equilibrium outcomes for p*, v* and φ*, where φ* is the
           equilibrium exit threshold φ_bar(p*).
        1. The scaling factor necessary to convert the stationary probability
           distribution μ into the equilibrium firm distribution μ* = s μ.
        2. The equilibrium mass of entrants M* = μ*{ φ < φ*}

    """
    # Unpack
    parameters, grids = model
    # Compute prices and values
    p_star, v_bar = compute_p_star(parameters, grids)
    # Get φ_star = φ_bar(p_star), the equilibrium exit threshold
    φ_star = get_threshold(v_bar, grids)
    # Generate an array of draws from μ, the normalized stationary distribution.
    φ_sample = simulate_firms(φ_star, parameters, grids)
    # Compute s to scale μ
```

(continues on next page)

(continued from previous page)

```
    demand = 1 / p_star
    pre_normalized_supply = jnp.mean(q(ψ_sample, p_star, parameters))
    s = demand / pre_normalized_supply
    # Compute M* = μ*{ ψ < ψ_star}
    m_star = s * jnp.mean(ψ_sample < ψ_star)
    # return computed objects
    return p_star, v_bar, ψ_star, ψ_sample, s, m_star
```

19.4 Solving the model

19.4.1 Preliminary calculations

Let's create an instance of the model.

```
model = create_model()
parameters, grids = model
```

Let's see how long it takes to compute the value function at a given price from a cold start.

```
p = 2.0
v_init = jnp.zeros_like(grids.ψ_grid)             # Initial condition
%time v_bar = vfi(p, v_init, parameters, grids).block_until_ready()
```

```
CPU times: user 628 ms, sys: 24.5 ms, total: 653 ms
Wall time: 485 ms
```

Let's run the code again to eliminate compile time.

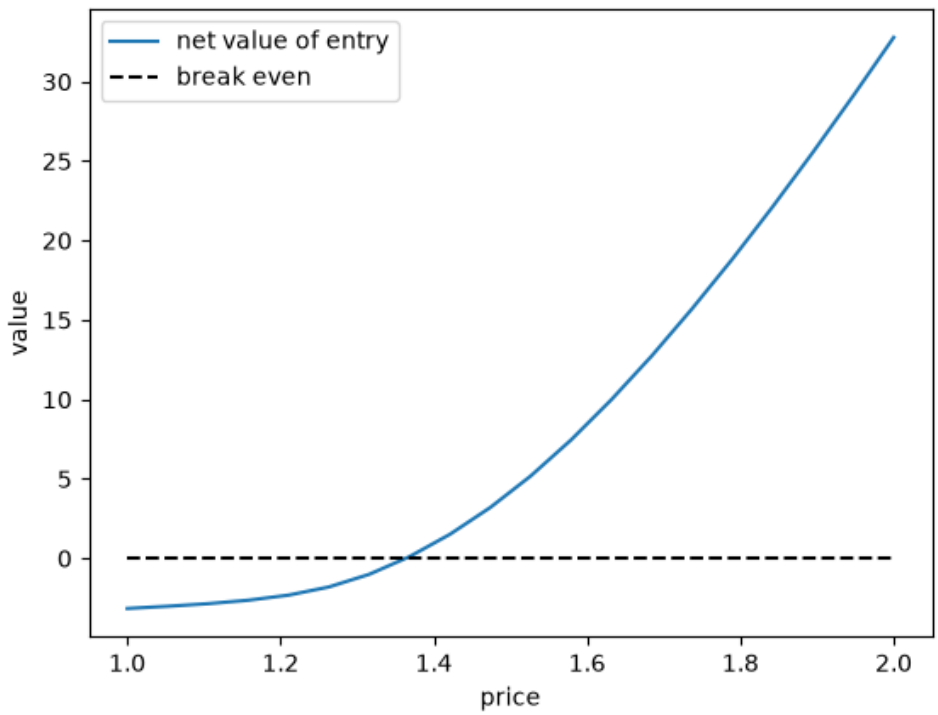
```
%time v_bar = vfi(p, v_init, parameters, grids).block_until_ready()
```

```
CPU times: user 47.7 ms, sys: 235 µs, total: 48 ms
Wall time: 46.3 ms
```

Let's have a look at the net entry value as a function of price

The root is the equilibrium price at the given parameters

```
p_min, p_max, p_size = 1.0, 2.0, 20
p_vec = jnp.linspace(p_min, p_max, p_size)
entry_vals = []
v_bar = jnp.zeros_like(grids.ψ_grid)  # Initial condition at first price guess
for i, p in enumerate(p_vec):
    entry_val, v_bar = compute_net_entry_value(p, v_bar, parameters, grids)
    entry_vals.append(entry_val)
fig, ax = plt.subplots()
ax.plot(p_vec, entry_vals, label="net value of entry")
ax.plot(p_vec, jnp.zeros_like(p_vec), 'k', ls='--', label="break even")
ax.legend()
ax.set_xlabel("price")
ax.set_ylabel("value")
plt.show()
```



Below we solve for the zero of this function to calculate $p*$.

From the figure it looks like p^* will be close to 1.5.

19.4.2 Computing the equilibrium

Now let's try computing the equilibrium

```
%%time
p_star, v_bar, ψ_star, ψ_sample, s, m_star = \
        compute_equilibrium_prices_and_quantities(model)
```

```
CPU times: user 2.5 s, sys: 106 ms, total: 2.6 s
Wall time: 2.58 s
```

Let's run the code again to get rid of compile time.

```
%%time
p_star, v_bar, ψ_star, ψ_sample, s, m_star = \
        compute_equilibrium_prices_and_quantities(model)
```

```
CPU times: user 460 ms, sys: 25 ms, total: 485 ms
Wall time: 790 ms
```

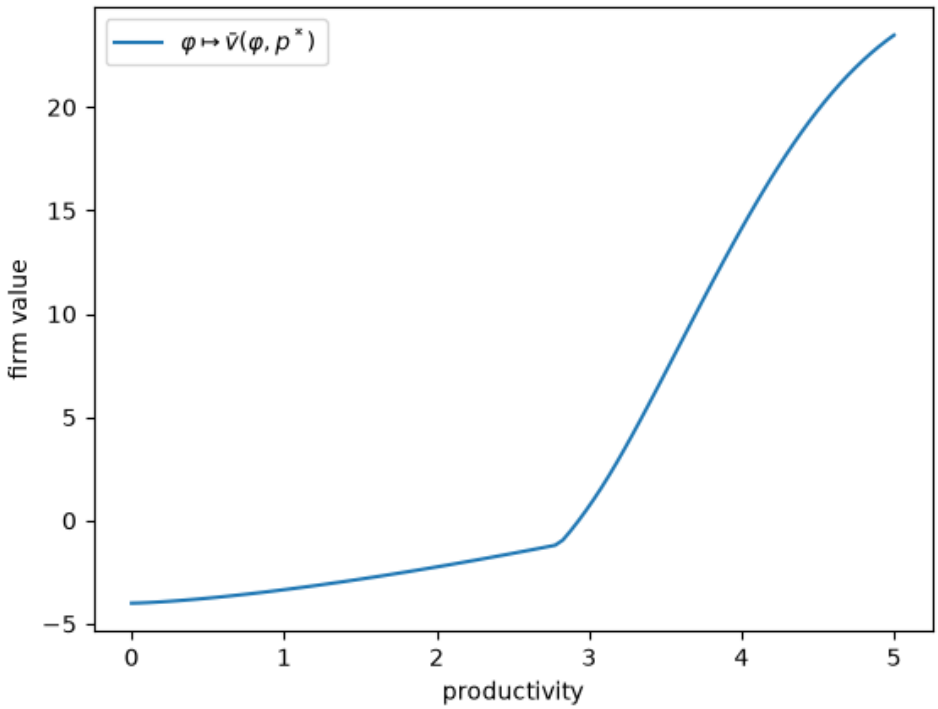
Let's check that p^* is close to 1.5

```
p_star
```

```
1.363800048828125
```

Here is a plot of the value function $\bar{v}(\cdot, p^*)$.

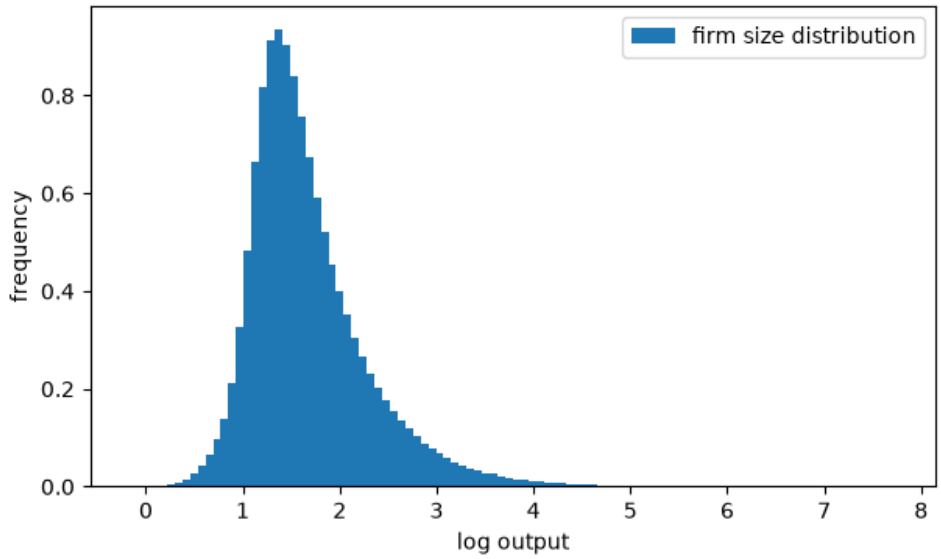
```
fig, ax = plt.subplots()
ax.plot(grids.φ_grid, v_bar, label=r'$\varphi \mapsto \bar v(\varphi, p^*)$')
ax.set_xlabel("productivity")
ax.set_ylabel("firm value")
ax.legend()
plt.show()
```



Let's have a look at the firm size distribution, with firm size measured by output.

```
output_dist = q(φ_sample, p_star, parameters)
```

```
fig, ax = plt.subplots(figsize=(7, 4))
ax.hist(jnp.log(output_dist), bins=100, density=True,
        label="firm size distribution")
ax.set_xlabel("log output")
ax.set_ylabel("frequency")
ax.legend()
plt.show()
```



19.5 Pareto tails

The firm size distribution shown above appears to have a long right tail.

This matches the observed firm size distribution.

In fact the firm size distribution obeys a **power law**.

More mathematically, the distribution of firm size has a Pareto right hand tail, so that there exist constants $k, \alpha > 0$ with

$$\mu((x, \infty)) \approx kx^{-\alpha} \text{ when } x \text{ is large}$$

Here α is called the tail index.

Does the model replicate this feature?

One option is to look at the empirical counter CDF (cumulative distribution).

The idea is as follows: The counter CDF of a random variable X is

$$G(x) := \mathbb{P}\{X > x\}$$

In the case of a Pareto tailed distribution we have $\mathbb{P}\{X > x\} \approx kx^{-\alpha}$ for large x.

Hence, for large x,

$$\ln G(x) \approx \ln k - \alpha \ln x$$

The empirical counterpart of G given sample $X_1, \dots, X_n$ is

$$G_n(x) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{X_i > x\}$$

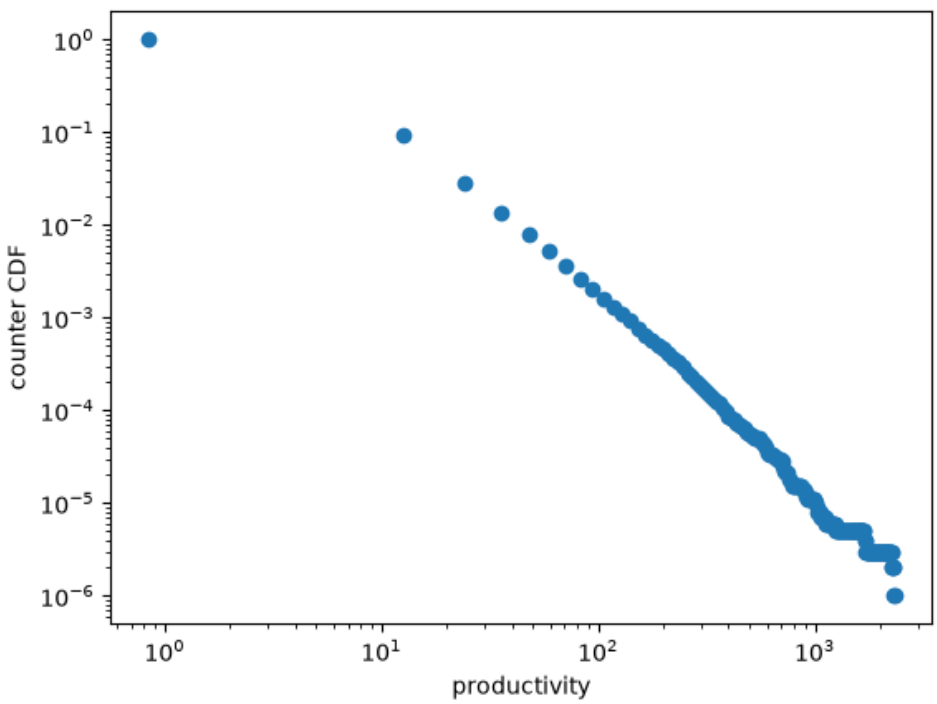
For large k (implying $G_n \approx G$) and large x, we expect that, for a Pareto-tailed sample, $\ln G_n$ is approximately linear.

Here's a function to compute the empirical counter CDF:

```
def ECCDF(data):
    """
    Return a function that implements the ECCDF given the data.
    """
    def eccdf(x):
        return jnp.mean(data > x)
    return eccdf
```

Let's plot the empirical counter CDF of the output distribution.

```
eccdf = ECCDF(output_dist)

ϵ = 10e-10
x_grid = jnp.linspace(output_dist.min() + ϵ, output_dist.max() - ϵ, 200)
y = [eccdf(x) for x in x_grid]

fix, ax = plt.subplots()
ax.loglog(x_grid, y, 'o', label="ECCDF")
ax.set_xlabel("productivity")
ax.set_ylabel("counter CDF")
plt.show()
```



19.6 Exercise

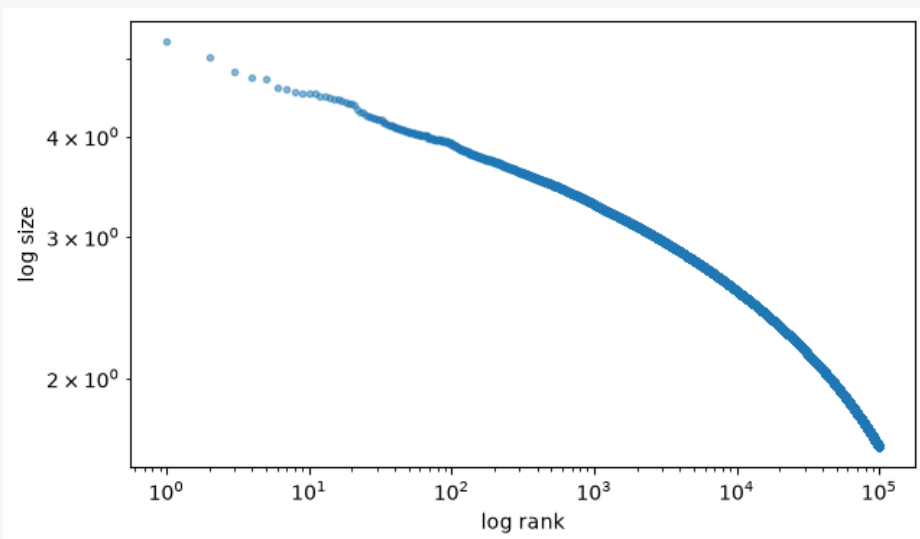
Exercise 19.6.1

Plot the same output distribution, but this time using a rank-size plot.

If the rank-size plot is approximately linear, the data suggests a Pareto tail.

You can use QuantEcon's `rank_size` function — here's an example of usage.

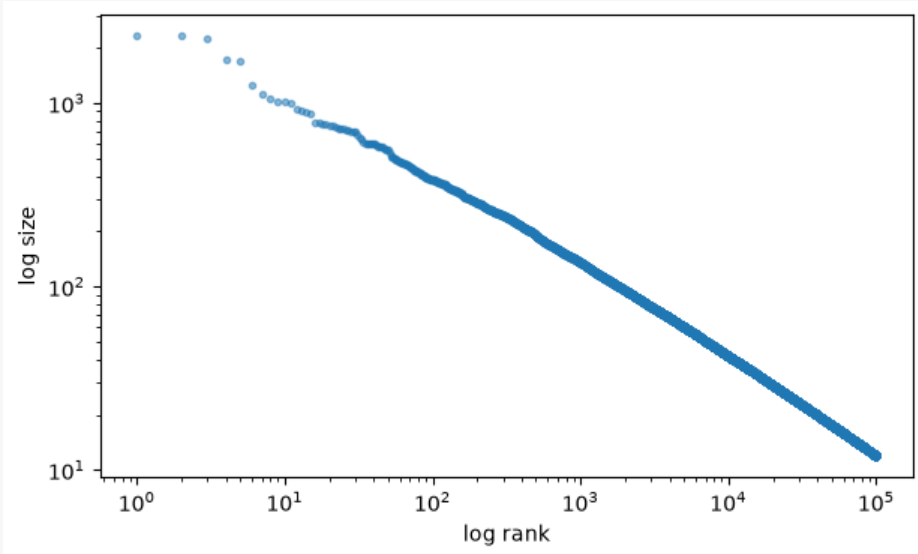
```
x = jnp.abs(jax.random.normal(jax.random.key(42), (1_000_000,)))
rank_data, size_data = qe.rank_size(x, c=0.1)
fig, ax = plt.subplots(figsize=(7,4))
ax.loglog(rank_data, size_data, "o", markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")
plt.show()
```



This plot is not linear — as expected, since we are using a folded normal distribution.

Solution

```
rank_data, size_data = qe.rank_size(output_dist, c=0.1)
fig, ax = plt.subplots(figsize=(7,4))
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()
```



This looks very linear — the model generates Pareto tails.

(In fact it's possible to prove this.)

Exercise 19.6.2

As an exercise, let's look at the fixed cost paid by incumbents each period and how it relates to the equilibrium price.

We expect that a higher fixed cost will reduce supply and hence increase the market price.

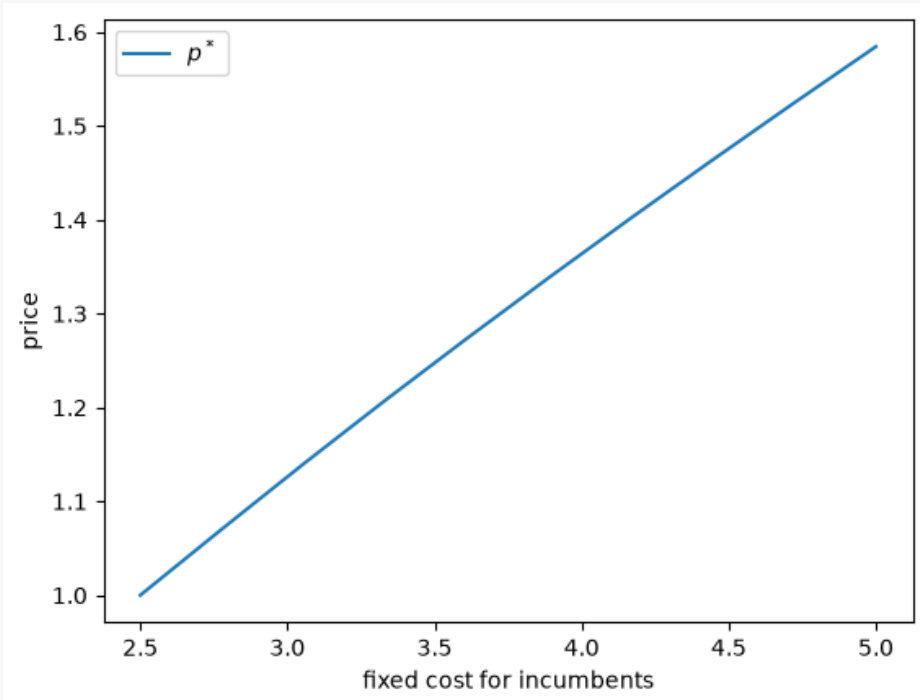
For the fixed costs, use

```
c_values = jnp.linspace(2.5, 5.0, 10)
```

Solution

```
eq_prices = []
for i, c in enumerate(c_values):
    model = create_model(c=c)
    p_star, v_bar, ψ_star, ψ_sample, s, m_star = \
        compute_equilibrium_prices_and_quantities(model)
    eq_prices.append(p_star)
    print(f"Equilibrium price when c = {c:.2} is {p_star:.2}")

fig, ax = plt.subplots()
ax.plot(c_values, eq_prices, label="$p^*$")
ax.set_xlabel("fixed cost for incumbents")
ax.set_ylabel("price")
ax.legend()
plt.show()
```

```
Equilibrium price when c = 2.5 is 1.0
Equilibrium price when c = 2.8 is 1.1
Equilibrium price when c = 3.1 is 1.1
Equilibrium price when c = 3.3 is 1.2
Equilibrium price when c = 3.6 is 1.3
Equilibrium price when c = 3.9 is 1.3
Equilibrium price when c = 4.2 is 1.4
Equilibrium price when c = 4.4 is 1.5
Equilibrium price when c = 4.7 is 1.5
Equilibrium price when c = 5.0 is 1.6
```



CHAPTER

TWENTY

BIANCHI OVERBORROWING MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

This lecture provides a JAX implementation of "Overborrowing and Systemic Externalities" [Bianchi, 2011] by Javier Bianchi.

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

```
!pip install quantecon
```

We use the following imports.

```
import time
import jax
import numba
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
import scipy as sp
import matplotlib.pyplot as plt
from collections import namedtuple
```

20.1 Markov dynamics

Before studying Bianchi (2011), we develop some functions for working with the bivariate VAR process

$$\ln y' = A \ln y + u'$$

where

- prime indicates next period value
- $y = (y_t, y_n) =$ output of (tradables, nontradables)

- $u' \sim N(0, \Omega)$ and Ω is positive definite
- the log function is applied pointwise

We use the following estimated values, reported on p. 12 of Yamada (2023).

```
A = [[0.2425,   0.3297],
     [-0.1984,  0.7576]]

Ω = [[0.0052, 0.002],
     [0.002,  0.0059]]

A, Ω = np.array(A), np.array(Ω)
```

We'll store the data in Ω using its square root:

```
C = sp.linalg.sqrtm(Ω)
```

20.1.1 Simulating the VAR

Here's code for generating the original VAR process, which can be used for testing.

```
@numba.jit
def generate_var_process(A=A, C=C, ts_length=1_000_000):
    """
    Generate the original VAR process.

    """
    y_series = np.empty((ts_length, 2))
    y_series[0, :] = np.zeros(2)
    for t in range(ts_length-1):
        y_series[t+1, :] = A @ y_series[t, :] + C @ np.random.randn(2)
    y_t_series = np.exp(y_series[:, 0])
    y_n_series = np.exp(y_series[:, 1])
    return y_t_series, y_n_series
```

20.1.2 Discretizing the VAR

Here's a function to convert the VAR process to a Markov chain evolving on a rectilinear grid of points in $\mathbb{R}^2$.

The function returns arrays `y_t`, `y_n` and `Q`

- `Q[i, j, i', j']` is the probability of moving from `(y_t[i], y_n[j])` to `(y_t[i'], y_n[j'])`.

Under the hood, this function uses the QuantEcon function `discrete_var`.

```
def discretize_income_var(A=A, C=C, n=4, seed=1234):
    """
    Discretize the VAR model, returning

        y_t, an n-grid of y_t values
        y_n, an n-grid of y_n values
        Q, a Markov operator

    The format is that Q is n x n x n x n, with
```

(continues on next page)

(continued from previous page)

```
        Q[i, j, i', j'] = one step transition prob from
        (y_t[i], y_n[j]) to (y_t[i'], y_n[j'])

    """
    rng = np.random.default_rng(seed)
    mc = qe.markov.discrete_var(A, C, (n, n),
                                sim_length=1_000_000,
                                std_devs=np.sqrt(3),
                                random_state=rng)
    y, Q = np.exp(mc.state_values), mc.P
    # The array y is currently an array listing all bivariate state pairs
    # (y_t, y_n), so that y[i] is the i-th such pair, while Q[l, m]
    # is the probability of transitioning from state l to state m in one step.
    # We switch the representation to the one described in the docstring.
    y_t = [y[n*i, 0] for i in range(n)]
    y_n = y[0:4, 1]
    Q = np.reshape(Q, (n, n, n, n))
    return y_t, y_n, Q
```

Here's code for sampling from the Markov chain.

```
def generate_discrete_var(A=A, C=C, n=4, seed=1234,
                          ts_length=1_000_000,
                          indices=False):
    """
    Generate a time series from the discretized model, returning y_t_series and
    y_n_series.  If `indices=True`, then these series are returned as grid
    indices.
    """


    rng = np.random.default_rng(seed)
    mc = qe.markov.discrete_var(A, C, (n, n),
                                sim_length=1_000_000,
                                std_devs=np.sqrt(3),
                                random_state=rng)
    if indices:
        y_series = mc.simulate_indices(ts_length=ts_length)
        y_t_series, y_n_series = y_series % n, y_series // n
    else:
        y_series = np.exp(mc.simulate(ts_length=ts_length))
        y_t_series, y_n_series = y_series[:, 0], y_series[:, 1]
    return y_t_series, y_n_series
```

20.1.3 Testing the discretization

Let's check some statistics for both the original and the discretized processes, to see if they match up.

```
def corr(x, y):
    m_x, m_y = x.mean(), y.mean()
    s_xy = np.sqrt(np.sum((x - m_x)**2) * np.sum((y - m_y)**2))
    return np.sum((x - m_x) * (y - m_y)) / (s_xy)
```

```
def print_stats(y_t_series, y_n_series):
    print(f"Std dev of y_t is {y_t_series.std():.3}")
```

(continues on next page)

(continued from previous page)

```
    print(f"Std dev of y_n is {y_n_series.std():.3}")
    print(f"corr(y_t, y_n) is {corr(y_t_series, y_n_series):.3}")
    print(f"auto_corr(y_t) is {corr(y_t_series[:-1], y_t_series[1:]):.3}")
    print(f"auto_corr(y_n) is {corr(y_n_series[:-1], y_n_series[1:]):.3}")
    print("\n")
```

```
print("Statistics for original process.\n")
print_stats(*generate_var_process())
```

```
Statistics for original process.
Std dev of y_t is 0.0879
Std dev of y_n is 0.106
corr(y_t, y_n) is 0.541
auto_corr(y_t) is 0.456
auto_corr(y_n) is 0.668
```

```
print("Statistics for discretized process.\n")
print_stats(*generate_discrete_var())
```

```
Statistics for discretized process.
Std dev of y_t is 0.0876
Std dev of y_n is 0.105
corr(y_t, y_n) is 0.478
auto_corr(y_t) is 0.401
auto_corr(y_n) is 0.588
```

20.2 Description of the model

The Bianchi (2011) model seeks to explain sudden stops in emerging market economies.

A representative household chooses how much to borrow on international markets and how much to consume.

The household is credit constrained, with the constraint depending on both current income and the real exchange rate.

Household "overborrow" (relative to a planner) because they do not internalize the effect of borrowing on the credit constraint.

This overborrowing leaves them vulnerable to bad income shocks.

In essence, the model works as follows

1. During good times, households borrow more and consume more
2. Increased consumption pushes up the price of nontradables and hence the real exchange rate
3. A rising real exchange rate loosens the credit constraint and encourages more borrowing
4. This leads to excessive borrowing relative to a planner.

This overborrowing leads to vulnerability vis-a-vis bad shocks.

1. When a bad shock hits, borrowing is restricted.
2. Consumption now falls, pushing down the real exchange rate.
3. This fall in the exchange rate further tightens the borrowing constraint, amplifying the shock

20.2.1 Decentralized equilibrium

The model contains a representative household that seeks to maximize an expected sum of discounted utility with

$$u(c) = \frac{c^{1-\sigma}}{1-\sigma} \quad \text{where} \quad c = (\omega c_t^{-\eta} + (1-\omega) c_n^{-\eta})^{-1/\eta}$$

Here c_t (resp., c_n) is consumption of tradables (resp., nontradables).

The household maximizes subject to the budget constraint

$$b' + c_t + p_n c_n = b(1+r) + y_t + p_n y_n$$

where

- b is bond holdings (positive values denote assets!)
- primes denote next period values
- the interest rate r is exogenous
- p_n is the price of nontradables, while the price of tradables is normalized to 1
- y_t and y_n are current tradable and nontradable income

The process for $y := (y_t, y_n)$ is first-order Markov.

The household also faces the credit constraint

$$b' \geq -\kappa(y_t + p_n y_n)$$

Market clearing implies

$$c_n = y_n \quad \text{and} \quad c_t = y_t + (1+r)b - b'$$

The household takes the aggregate timepath for bonds as given by

$$B' = H(B, y)$$

and solves

$$v(b, B, y) = \max_{c, b'} \left\{ u(c) + \beta \, \mathbb{E}_y \, v(b', B', y') \right\}$$

subject to the budget and credit constraints.

Let the solution to the dynamic program be the policy $b' = h(b, B, y) =$ savings decision in state (b, B, y).

A **decentralized equilibrium** is a map H such that

$$h(B, B, y) = H(B, y) \quad \text{for all } B, y$$

20.2.2 Notation

Let

$$w(c_t, y_n) = \frac{c^{1-\sigma}}{1-\sigma} \quad \text{where} \quad c = [\omega c_t^{-\eta} + (1-\omega) y_n^{-\eta}]^{-1/\eta}$$

Using the market clearing conditions, we can write the household problem as

$$v(b, B, y) = \max_{b'} \left\{ w((1+r)b + y_t - b', y_n) + \beta \, \mathbb{E}_y \, v(b', H(B, y), y') \right\}$$

subject to

$$-\kappa(p_n y_n + y_t) \leq b' \leq (1+r)b + y_t$$

where p_n is given by

$$p_n = \frac{1-\omega}{\omega}\left(\frac{C}{y_n}\right)^{\eta+1} \quad \text{with} \quad C := (1+r)B + y_t - H(B,y)$$

We will make use of the policy operator that maps h into

$$(T_h v)(b,B,y) = \left\{w((1+r)b + y_t - h(b,B,y), y_n) + \beta\,\mathbb{E}_y\, v(h(b,B,y), H(B,y), y')\right\}$$

Our algorithm is

1. initialize v and H
2. get a greedy policy h given v and H
3. update H via $H = \alpha h + (1-\alpha)H$
4. iterate m times with the policy operator T_h to get $v = T_h^m v$
5. go to step 2

In other words, we use optimistic policy iteration, updating our guess of the aggregate law of motion every time we update the household policy function.

20.3 Overborrowing model in Python / JAX

In what follows

- `y = (y_t, y_n)` is the exogenous state process

Individual states and actions are

- `c` = consumption of tradables (`c` rather than `c_t`)
- `b` = household savings (bond holdings)
- `bp` = household savings decision (next period bond holdings)

Aggregate quantities and prices are

- `p` = price of nontradables (`p` rather than `p_n`)
- `B` = aggregate savings (bond holdings)
- `C` = aggregate consumption
- `H` = current guess of update rule as an array of the form $H(B,y)$

Here's code to create three tuples that store model data relevant for computation.

```
Model = namedtuple('Model',
    ('σ', 'η', 'β', 'ω', 'κ', 'r', 'b_grid', 'y_t_nodes', 'y_n_nodes', 'Q'))
```

```
def create_overborrowing_model(
        σ=2.0,                  # CRRA utility parameter
        η=(1 / 0.83) - 1,       # Elasticity = 0.83, η = 0.2048
        β=0.91,                 # Discount factor
```

(continues on next page)

(continued from previous page)

```
        ω=0.31,               # Aggregation constant
        κ=0.3235,             # Constraint parameter
        r=0.04,               # Interest rate
        b_size=800,           # Bond grid size
        b_grid_min=-1.02,     # Bond grid min
        b_grid_max=-0.2       # Bond grid max (originally -0.6 to match fig)
    ):
    """
    Creates an instance of the overborrowing model using

        * default parameter values from Bianchi (2011)
        * Markov dynamics from Yamada (2023)

    The Markov kernel Q has the interpretation

        Q[i, j, ip, jp] = one step prob of moving

            (y_t[i], y_n[j]) -> (y_t[ip], y_n[jp])

    """
    # Read in Markov data and shift to JAX arrays
    data = discretize_income_var()
    y_t_nodes, y_n_nodes, Q = [jnp.array(d) for d in data]
    # Set up grid for bond holdings
    b_grid = jnp.linspace(b_grid_min, b_grid_max, b_size)
    # Pack and return
    return Model(σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q)
```

Default parameter values are from Bianchi.

Notice that β is quite small (too small?), so value function iteration will be relatively quick.

Here's flow utility.

```
@jax.jit
def w(model, c, y_n):
    """
    Current utility when c_t = c and c_n = y_n.

        a = [ω c^(- η) + (1 - ω) y_n^(- η)]^(-1/η)

        w(c, y_n) := a^(1 - σ) / (1 - σ)

    """
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    a = (ω * c**(-η) + (1 - ω) * y_n**(-η))**(-1/η)
    return a**(1 - σ) / (1 - σ)
```

We need code to generate an initial guess of H.

```
@jax.jit
def generate_initial_H(model, at_constraint=False):
    """
    Compute an initial guess for H. Repeat the indices for b_grid over y_t and
    y_n axes.

    """
```

(continues on next page)

(continued from previous page)

```
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    b_size, y_size = len(b_grid), len(y_t_nodes)
    b_indices = jnp.arange(b_size)
    O = jnp.ones((b_size, y_size, y_size), dtype=int)
    return  O * jnp.reshape(b_indices, (b_size, 1, 1))
```

We need to construct the Bellman operator for the household.

Our first function returns the (unmaximized) RHS of the Bellman equation.

```
@jax.jit
def BellmanRHS(model, v, H, i_b, i_B, i_y_t, i_y_n, i_bp):
    """
    Given current state (b, B, y_t, y_n) with indices (i_b, i_B, i_y_t, i_y_n),
    compute the unmaximized right hand side (RHS) of the Bellman equation as a
    function of the next period choice bp = b', with index i_bp.
    """
    # Unpack
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    # Compute next period aggregate bonds given H
    i_Bp = H[i_B, i_y_t, i_y_n]
    # Evaluate states and actions at indices
    B, Bp, b, bp = b_grid[i_B], b_grid[i_Bp], b_grid[i_b], b_grid[i_bp]
    y_t = y_t_nodes[i_y_t]
    y_n = y_n_nodes[i_y_n]
    # Compute price of nontradables using aggregates
    C = (1 + r) * B + y_t - Bp
    p = ((1 - ω) / ω) * (C / y_n)**(η + 1)
    # Compute household flow utility
    c = (1 + r) * b + y_t - bp
    utility = w(model, c, y_n)
    # Compute expected value Σ_{y'} v(b', B', y') Q(y, y')
    EV = jnp.sum(v[i_bp, i_Bp, :, :] * Q[i_y_t, i_y_n, :, :])
    # Set up constraints
    credit_constraint_holds = - κ * (p * y_n + y_t) <= bp
    budget_constraint_holds = bp <= (1 + r) * b + y_t
    constraints_hold = jnp.logical_and(credit_constraint_holds,
                                       budget_constraint_holds)
    # Compute and return
    return jnp.where(constraints_hold, utility + β * EV, -jnp.inf)
```

Let's now vectorize and jit-compile this map.

```
# Vectorize over the control bp and all the current states
BellmanRHS = jax.vmap(BellmanRHS,
    in_axes=(None, None, None, None, None, None, None, 0))
BellmanRHS = jax.vmap(BellmanRHS,
    in_axes=(None, None, None, None, None, None, 0, None))
BellmanRHS = jax.vmap(BellmanRHS,
    in_axes=(None, None, None, None, None, 0, None, None))
BellmanRHS = jax.vmap(BellmanRHS,
    in_axes=(None, None, None, None, 0, None, None, None))
BellmanRHS = jax.vmap(BellmanRHS,
    in_axes=(None, None, None, 0, None, None, None, None))
```

Here's a function that computes a greedy policy (best response to v).

```
@jax.jit
def get_greedy(model, v, H):
    """
    Compute the greedy policy for the household, which maximizes the right hand
    side of the Bellman equation given v and H.  The greedy policy is recorded
    as an array giving the index i in b_grid such that b_grid[i] is the optimal
    choice, for every state.

    Return

        * bp_policy as an array of shape (b_size, b_size, y_size, y_size).

    """
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    b_size, y_size = len(b_grid), len(y_t_nodes)
    b_indices, y_indices = jnp.arange(b_size), jnp.arange(y_size)
    val = BellmanRHS(model, v, H,
                     b_indices, b_indices, y_indices, y_indices, b_indices)
    return jnp.argmax(val, axis=-1)
```

Here's the policy operator

```
@jax.jit
def _T_h(model, h, v, H, i_b, i_B, i_y_t, i_y_n):
    """
    Evaluate the RHS of the policy operator associated with individual
    policy h and aggregate policy H.


    """
    # Unpack
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    # Compute next period states
    i_bp = h[i_b, i_B, i_y_t, i_y_n]
    i_Bp = H[i_B, i_y_t, i_y_n]
    # Convert indices into values
    B, Bp, b, bp = b_grid[i_B], b_grid[i_Bp], b_grid[i_b], b_grid[i_bp]
    y_t = y_t_nodes[i_y_t]
    y_n = y_n_nodes[i_y_n]
    # Compute household flow utility
    c = (1 + r) * b + y_t - bp
    utility = w(model, c, y_n)
    # Compute expected value Σ_{y'} v(b', B', y') Q(y, y')
    EV = jnp.sum(v[i_bp, i_Bp, :, :] * Q[i_y_t, i_y_n, :, :])
    val = utility + β * EV
    return val
```

```
# Vectorize over the control bp and all the current states
_T_h = jax.vmap(_T_h,
    in_axes=(None, None, None, None, None, None, None, 0))
_T_h = jax.vmap(_T_h,
    in_axes=(None, None, None, None, None, None, 0, None))
_T_h = jax.vmap(_T_h,
    in_axes=(None, None, None, None, None, 0, None, None))
_T_h = jax.vmap(_T_h,
    in_axes=(None, None, None, None, 0, None, None, None))
```

```
@jax.jit
def T_h(model, h, v, H):
    """
    Vectorized version of the policy operator.

    """
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    b_size, y_size = len(b_grid), len(y_t_nodes)
    b_indices, y_indices = jnp.arange(b_size), jnp.arange(y_size)
    val = _T_h(model, h, v, H,
                     b_indices, b_indices, y_indices, y_indices)
    return val
```

```
@jax.jit
def iterate_policy_operator(model, h, v, H, m):

    def update(i, v):
        v = T_h(model, h, v, H)
        return v
    v = jax.lax.fori_loop(0, m, update, v)
    return v
```

This is how we update our guess of H, using the current policy b' and a damped fixed point iteration scheme.

```
@jax.jit
def update_H(model, H, h, α):
    """
    Update guess of the aggregate update rule.

    """
    # Set up
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    b_size, y_size = len(b_grid), len(y_t_nodes)
    b_indices = jnp.arange(b_size)
    # Switch policy arrays to values rather than indices
    H_vals = b_grid[H]
    bp_vals = b_grid[h]
    # Update guess
    new_H_vals = α * bp_vals[b_indices, b_indices, :, :] + (1 - α) * H_vals
    # Switch back to indices
    new_H = jnp.searchsorted(b_grid, new_H_vals)
    return new_H
```

Now we can write code to compute an equilibrium law of motion H.

```
def compute_equilibrium(model, m=50,
                        α=0.5, tol=0.005, max_iter=500):
    """
    Compute the equilibrium law of motion.

    """
    H = generate_initial_H(model)
    v = jnp.ones((b_size, b_size, y_size, y_size))
    h = get_greedy(model, v, H)
    error = tol + 1
    i = 0
    while error > tol and i < max_iter:
```

(continues on next page)

(continued from previous page)

```
        new_H = update_H(model, H, h, α)
        new_v = iterate_policy_operator(model, h, v, new_H, m)
        new_h = get_greedy(model, new_v, new_H)
        error = jnp.max(jnp.abs(b_grid[H] - b_grid[new_H]))
        print(f"Updated H at iteration {i} with error {error}.")
        H = new_H
        v = new_v
        h = new_h
        i += 1
    if i == max_iter:
        print("Warning: Equilibrium search iteration hit upper bound.")
    return H
```

20.4 Planner problem

Now we switch to the planner problem.

The constrained planner solves

$$V(b, B, y) = \max_{c, b'} \left\{ u(c) + \beta \mathbb{E}_y v(b', B', y') \right\}$$

subject to the market clearing conditions and the same constraint

$$-\kappa(y_t + p_n y_n) \leq b' \leq (1 + r)b + y_t$$

although the price of nontradable is now given by

$$p_n = ((1 - \omega)/\omega)(c_t/y_n)^{\eta+1} \quad \text{with} \quad c_t := (1 + r)b + y_t - b'$$

We see that the planner internalizes the impact of the savings choice b' on the price of nontradables and hence the credit constraint.

Our first function returns the (unmaximized) RHS of the Bellman equation.

```
@jax.jit
def planner_T_generator(v, model, i_b, i_y_t, i_y_n, i_bp):
    """
    Given current state (b, y_t, y_n) with indices (i_b, i_y_t, i_y_n),
    compute the unmaximized right hand side (RHS) of the Bellman equation as a
    function of the next period choice bp = b'.
    """
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    y_t = y_t_nodes[i_y_t]
    y_n = y_n_nodes[i_y_n]
    b, bp = b_grid[i_b], b_grid[i_bp]
    # Compute price of nontradables using aggregates
    c = (1 + r) * b + y_t - bp
    p = ((1 - ω) / ω) * (c / y_n)**(η + 1)
    # Compute household flow utility
    utility = w(model, c, y_n)
    # Compute expected value (continuation)
    EV = jnp.sum(v[i_bp, :, :] * Q[i_y_t, i_y_n, :, :])
    # Set up constraints and evaluate
    credit_constraint_holds = - κ * (p * y_n + y_t) <= bp
```

(continues on next page)

(continued from previous page)

```
    budget_constraint_holds = bp <= (1 + r) * b + y_t
    return jnp.where(jnp.logical_and(credit_constraint_holds,
                                     budget_constraint_holds),
                     utility + β * EV,
                     -jnp.inf)
```

```
# Vectorize over the control bp and all the current states
planner_T_generator = jax.vmap(planner_T_generator,
        in_axes=(None, None, None, None, None, 0))
planner_T_generator = jax.vmap(planner_T_generator,
        in_axes=(None, None, None, None, 0, None))
planner_T_generator = jax.vmap(planner_T_generator,
        in_axes=(None, None, None, 0, None, None))
planner_T_generator = jax.vmap(planner_T_generator,
        in_axes=(None, None, 0, None, None, None))
```

Now we construct the Bellman operator.

```
@jax.jit
def planner_T(model, v):
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    b_size, y_size = len(b_grid), len(y_t_nodes)
    b_indices, y_indices = jnp.arange(b_size), jnp.arange(y_size)
    # Evaluate RHS of Bellman equation at all states and actions
    val = planner_T_generator(v, model,
                     b_indices, y_indices, y_indices, b_indices)
    # Maximize over bp
    return jnp.max(val, axis=-1)
```

Here's a function that computes a greedy policy (best response to v).

```
@jax.jit
def planner_get_greedy(model, v):
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    b_size, y_size = len(b_grid), len(y_t_nodes)
    b_indices, y_indices = jnp.arange(b_size), jnp.arange(y_size)
    # Evaluate RHS of Bellman equation at all states and actions
    val = planner_T_generator(v, model,
                     b_indices, y_indices, y_indices, b_indices)
    # Maximize over bp
    return jnp.argmax(val, axis=-1)
```

```
def vfi(T, v_init, max_iter=10_000, tol=1e-5):
    """
    Use successive approximation to compute the fixed point of T, starting from
    v_init.

    """
    v = v_init

    def cond_fun(state):
        error, i, v = state
        return (error > tol) & (i < max_iter)

    def body_fun(state):
        error, i, v = state
```

(continues on next page)

(continued from previous page)

```
        v_new = T(v)
        error = jnp.max(jnp.abs(v_new - v))
        return error, i+1, v_new

    error, i, v_new = jax.lax.while_loop(cond_fun, body_fun,
                                                 (tol+1, 0, v))
    return v_new, i
```

```
vfi = jax.jit(vfi, static_argnums=(0,))
```

Computing the planner solution is straightforward value function iteration:

```
def compute_planner_solution(model):
    """
    Compute the constrained planner solution.

    """
    σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
    b_size, y_size = len(b_grid), len(y_t_nodes)
    b_indices = jnp.arange(b_size)
    v_init = jnp.ones((b_size, y_size, y_size))
    _T = lambda v: planner_T(model, v)
    # Compute household response to current guess H
    v, vfi_num_iter = vfi(_T, v_init)
    bp_policy = planner_get_greedy(model, v)
    return v, bp_policy, vfi_num_iter
```

20.5 Numerical solution

Let's now solve the model and compare the decentralized and planner solutions

20.5.1 Generating solutions

Here we compute the two solutions.

```
model = create_overborrowing_model()
σ, η, β, ω, κ, r, b_grid, y_t_nodes, y_n_nodes, Q = model
b_size, y_size = len(b_grid), len(y_t_nodes)
```

```
print("Computing decentralized solution.")
in_time = time.time()
H_eq = compute_equilibrium(model)
out_time = time.time()
diff = out_time - in_time
print(f"Computed decentralized equilibrium in {diff} seconds")
```

```
Computing decentralized solution.
Updated H at iteration 0 with error 0.4094868302345276.
Updated H at iteration 1 with error 0.08928662538528442.
Updated H at iteration 2 with error 0.14060074090957642.
Updated H at iteration 3 with error 0.11083859205245972.
```

(continues on next page)

(continued from previous page)

```
Updated H at iteration 4 with error 0.0615769624710083.
Updated H at iteration 5 with error 0.05131411552429199.
Updated H at iteration 6 with error 0.043103933334350586.
Updated H at iteration 7 with error 0.03797250986099243.
Updated H at iteration 8 with error 0.0328410267829895.
Updated H at iteration 9 with error 0.0287359356880188.
Updated H at iteration 10 with error 0.024630755186080933.
Updated H at iteration 11 with error 0.020525693893432617.
Updated H at iteration 12 with error 0.01744687557220459.
Updated H at iteration 13 with error 0.015394270420074463.
Updated H at iteration 14 with error 0.012315452098846436.
Updated H at iteration 15 with error 0.011289149522781372.
Updated H at iteration 16 with error 0.009236574172973633.
Updated H at iteration 17 with error 0.008210301399230957.
Updated H at iteration 18 with error 0.0071839988231658936.
Updated H at iteration 19 with error 0.00615769624710083.
Updated H at iteration 20 with error 0.00513148307800293.
Updated H at iteration 21 with error 0.005131423473358154.
Updated H at iteration 22 with error 0.0041051506996154785.
Computed decentralized equilibrium in 8.641213417053223 seconds
```

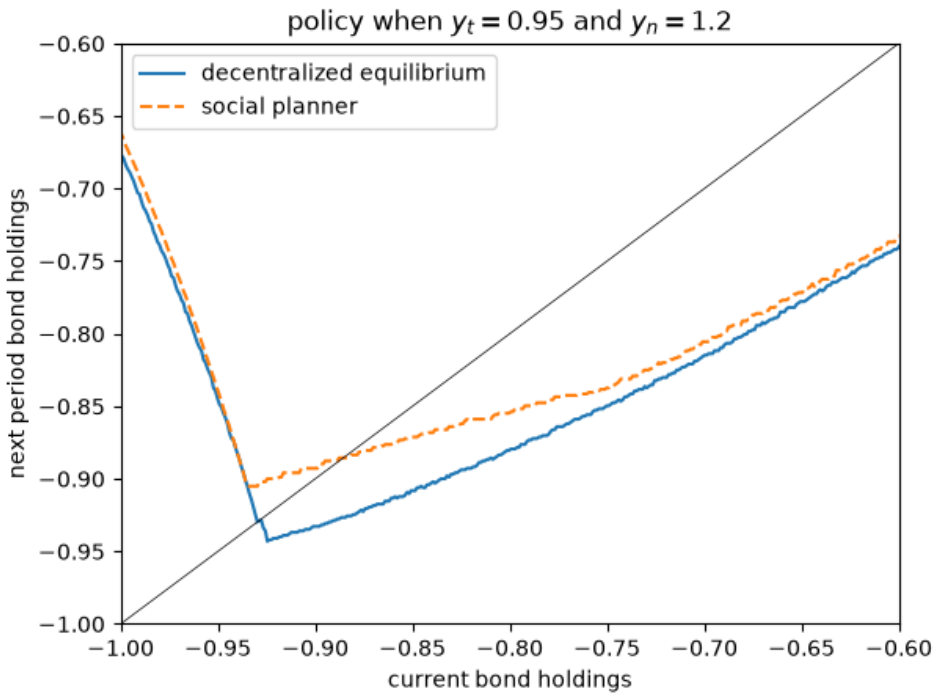
```
print("Computing planner's solution.")
in_time = time.time()
planner_v, H_plan, vfi_num_iter = compute_planner_solution(model)
out_time = time.time()
diff = out_time - in_time
print(f"Computed planner's solution in {diff} seconds")
```

```
Computing planner's solution.
Computed planner's solution in 1.0042343139648438 seconds
```

20.5.2 Policy plots

We produce a policy plot that is similar to Figure 1 in Bianchi (2011).

```
i, j = 1, 3
y_t, y_n = y_t_nodes[i], y_n_nodes[j]
fig, ax = plt.subplots()
ax.plot(b_grid, b_grid[H_eq[:, i, j]], label='decentralized equilibrium')
ax.plot(b_grid, b_grid[H_plan[:, i, j]], ls='--', label='social planner')
ax.plot(b_grid, b_grid, color='black', lw=0.5)
ax.set_ylim((-1.0, -0.6))
ax.set_xlim((-1.0, -0.6))
ax.set_xlabel("current bond holdings")
ax.set_ylabel("next period bond holdings")
ax.set_title(f"policy when $y_t = {y_t:.2}$ and $y_n = {y_n:.2}$")
ax.legend()
plt.show()
```



The match is not exact because we use a different estimate of the Markov dynamics for income.

Nonetheless, it is qualitatively similar.

20.6 Exercise

Exercise 20.6.1

Your task is to examine the ergodic distribution of borrowing in the decentralized and planner equilibria.

We recommend that you use simulation and a kernel density estimate.

Here's a function for generating the borrowing sequence.

We use Numba because we want to compile a long for loop.

```
@numba.jit
def generate_borrowing_sequence(H, y_t_series, y_n_series):
    """
    Generate the borrowing sequence B' = H(B, y_t, y_n).

    * H is a policy array
    * y_t_series and y_n_series are simulated time paths

    Both y_t_series and y_n_series are stored as indices rather than values.

    """
    B = np.empty_like(y_t_series)
    B[0] = 0
    for t in range(len(y_t_series)-1):
        B[t+1] = H[B[t], y_t_series[t], y_n_series[t]]
    return B
```

Note that you will have to convert JAX arrays into NumPy arrays if you want to use this function.

From here you will need to

- generate a time path for income $y = (y_t, y_n)$ using one of the functions provided above.
- use the function `generate_borrowing_sequence` plus `H_eq` and `H_plan` to calculate bond holdings for the planner and the decentralized equilibrium
- produce a kernel density plot for each of these data sets

If you are successful, your plot should look something like Fig 2 of Bianchi (2011) — although not exactly the same, due to the alternative specification of the Markov process.

To generate a kernel density plot, we recommend that you use `kdeplot` from the package `seaborn`, which is included in Anaconda.

Solution

```
import seaborn  # For kernel density plots
```

```
sim_length = 100_000
y_t_series, y_n_series = generate_discrete_var(ts_length=sim_length,
                                               indices=True)
```

We convert JAX arrays to NumPy arrays in order to use Numba.

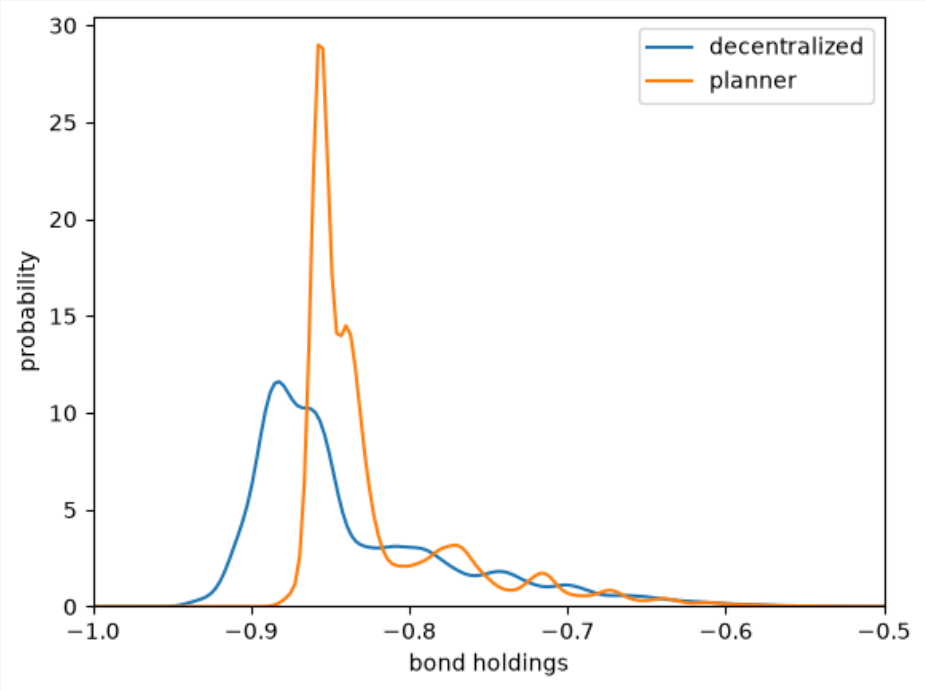
```
y_t_series, y_n_series, H_eq, H_plan = \
        [np.array(v) for v in (y_t_series, y_n_series, H_eq, H_plan)]
```

Now let's compute borrowing for the decentralized equilibrium and the planner.

```
B_eq = generate_borrowing_sequence(H_eq, y_t_series, y_n_series)
eq_b_sequence = b_grid[B_eq]
B_plan = generate_borrowing_sequence(H_plan, y_t_series, y_n_series)
plan_b_sequence = b_grid[B_plan]
```

Now let's plot the distributions using a kernel density estimator.

```
fig, ax = plt.subplots()
seaborn.kdeplot(eq_b_sequence, ax=ax, label='decentralized')
seaborn.kdeplot(plan_b_sequence, ax=ax, label='planner')
ax.legend()
ax.set_xlim((-1, -0.5))
ax.set_xlabel("bond holdings")
ax.set_ylabel("probability")
plt.show()
```



This corresponds to Figure 2 in Bianchi.

Again, the match is not exact but it is qualitatively similar.

Asset holding has a longer left hand tail under the decentralized equilibrium, leaving the economy more vulnerable to bad shocks.

Part VI

Data and Empirics

CHAPTER

TWENTYONE

MAXIMUM LIKELIHOOD ESTIMATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

21.1 Overview

This lecture is the extended JAX implementation of this section of this lecture.

Please refer that lecture for all background and notation.

Here we will exploit the automatic differentiation capabilities of JAX rather than calculating derivatives by hand.

We'll require the following imports:

```
import matplotlib.pyplot as plt
from collections import namedtuple
import jax.numpy as jnp
import jax
from statsmodels.api import Poisson
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
Fri Oct  9 00:25:14 2026
+-----------------------------------------------------------------------------------------
↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
↪13.0     |
+-----------------------------------------+------------------------+----------------
↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
```

(continues on next page)

(continued from previous page)

```
↪Compute M. |
|                                         |                         |              ␣
↪MIG M. |
|=========================================+=========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off  |              ␣
↪     0 |
| N/A   35C    P8             14W /   70W |       0MiB /  15360MiB  |      0%      ␣
↪Default |
|                                         |                         |              ␣
↪   N/A |
+-----------------------------------------+-------------------------+---------------
↪-------+

+-----------------------------------------------------------------------------------
↪-------+
| Processes:                                                                        ␣
↪       |
|  GPU   GI   CI              PID   Type   Process name                        GPU␣
↪Memory |
|        ID   ID                                                               ␣
↪Usage      |
|===================================================================================|
|  No running processes found                                                       ␣
↪       |
+-----------------------------------------------------------------------------------
↪-------+
```

We will use 64 bit floats with JAX in order to increase the precision.

```
jax.config.update("jax_enable_x64", True)
```

21.2 MLE with numerical methods (JAX)

Many distributions do not have nice, analytical solutions and therefore require numerical methods to solve for parameter estimates.

One such numerical method is the Newton-Raphson algorithm.

Let's start with a simple example to illustrate the algorithm.

21.2.1 A toy model

Our goal is to find the maximum likelihood estimate $\hat{\beta}$.

At $\hat{\beta}$, the first derivative of the log-likelihood function will be equal to 0.

Let's illustrate this by supposing

$$\log \mathcal{L}(\beta) = -(\beta - 10)^2 - 10$$

Define the function `logL`.

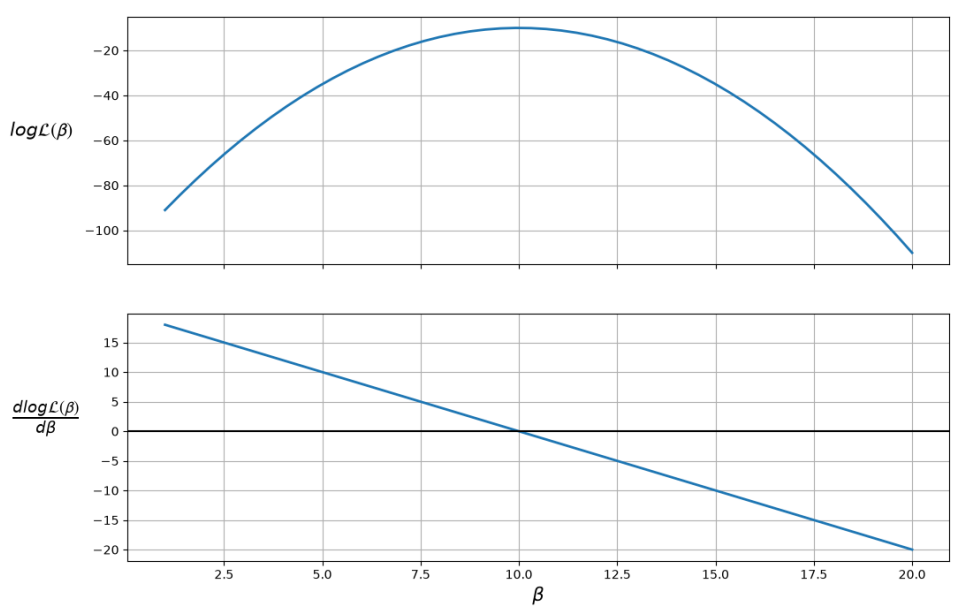
```
@jax.jit
def logL(β):
    return -(β - 10) ** 2 - 10
```

To find the value of $\frac{d \log \mathcal{L}(\beta)}{d\beta}$, we can use jax.grad which auto-differentiates the given function.

We further use jax.vmap which vectorizes the given function i.e. the function acting upon scalar inputs can now be used with vector inputs.

```
dlogL = jax.vmap(jax.grad(logL))
```

```
β = jnp.linspace(1, 20)

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(β, logL(β), lw=2)
ax2.plot(β, dlogL(β), lw=2)

ax1.set_ylabel(r'$log \mathcal{L(\beta)}$',
               rotation=0,
               labelpad=35,
               fontsize=15)
ax2.set_ylabel(r'$\frac{dlog \mathcal{L(\beta)}}{d \beta}$ ',
               rotation=0,
               labelpad=35,
               fontsize=19)

ax2.set_xlabel(r'$\beta$', fontsize=15)
ax1.grid(), ax2.grid()
plt.axhline(c='black')
plt.show()
```



The plot shows that the maximum likelihood value (the top plot) occurs when $\frac{d \log \mathcal{L}(\beta)}{d\beta} = 0$ (the bottom plot).

Therefore, the likelihood is maximized when $\beta = 10$.

We can also ensure that this value is a *maximum* (as opposed to a minimum) by checking that the second derivative (slope

of the bottom plot) is negative.

The Newton-Raphson algorithm finds a point where the first derivative is 0.

To use the algorithm, we take an initial guess at the maximum value, β_0 (the OLS parameter estimates might be a reasonable guess).

Then we use the updating rule involving gradient information to iterate the algorithm until the error is sufficiently small or the algorithm reaches the maximum number of iterations.

Please refer to this section for the detailed algorithm.

21.2.2 A Poisson model

Let's have a go at implementing the Newton-Raphson algorithm to calculate the maximum likelihood estimations of a Poisson regression.

The Poisson regression has a joint pmf:

$$f(y_1, y_2, \dots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

$$\text{where } \mu_i = \exp(\mathbf{x}_i' \beta) = \exp(\beta_0 + \beta_1 x_{i1} + \dots + \beta_k x_{ik})$$

We create a `namedtuple` to store the observed values

```
RegressionModel = namedtuple('RegressionModel', ['X', 'y'])

def create_regression_model(X, y):
    n, k = X.shape
    # Reshape y as a n_by_1 column vector
    y = y.reshape(n, 1)
    X, y = jax.device_put((X, y))
    return RegressionModel(X=X, y=y)
```

The log likelihood function of the Poisson regression is

$$\max_{\beta} \Big(\sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y! \Big)$$

The full derivation can be found here.

The log likelihood function involves factorial, but JAX doesn't have a readily available implementation to compute factorial directly.

In order to compute the factorial efficiently such that we can JIT it, we use

$$n! = e^{\log(\Gamma(n+1))}$$

since jax.lax.lgamma and jax.lax.exp are available.

The following function `jax_factorial` computes the factorial using this idea.

Let's define this function in Python

```
@jax.jit
def _factorial(n):
    return jax.lax.exp(jax.lax.lgamma(n + 1.0)).astype(int)

jax_factorial = jax.vmap(_factorial)
```

Now we can define the log likelihood function in Python

```
@jax.jit
def poisson_logL(β, model):
    y = model.y
    μ = jnp.exp(model.X @ β)
    return jnp.sum(model.y * jnp.log(μ) - μ - jnp.log(jax_factorial(y)))
```

To find the gradient of the `poisson_logL`, we again use jax.grad.

According to the documentation,

- `jax.jacfwd` uses forward-mode automatic differentiation, which is more efficient for "tall" Jacobian matrices, while
- `jax.jacrev` uses reverse-mode, which is more efficient for "wide" Jacobian matrices.

(The documentation also states that when matrices that are near-square, `jax.jacfwd` probably has an edge over `jax.jacrev`.)

Therefore, to find the Hessian, we can directly use `jax.jacfwd`.

```
G_poisson_logL = jax.grad(poisson_logL)
H_poisson_logL = jax.jacfwd(G_poisson_logL)
```

Our function `newton_raphson` will take a `RegressionModel` object that has an initial guess of the parameter vector β_0.

The algorithm will update the parameter vector according to the updating rule, and recalculate the gradient and Hessian matrices at the new parameter estimates.

```
def newton_raphson(model, β, tol=1e-3, max_iter=100, display=True):

    i = 0
    error = 100  # Initial error value

    # Print header of output
    if display:
        header = f'{"Iteration_k":<13}{"Log-likelihood":<16}{"θ":<60}'
        print(header)
        print("-" * len(header))

    # While loop runs while any value in error is greater
    # than the tolerance until max iterations are reached
    while jnp.any(error > tol) and i < max_iter:
        H, G = jnp.squeeze(H_poisson_logL(β, model)), G_poisson_logL(β, model)
        β_new = β - (jnp.dot(jnp.linalg.inv(H), G))
        error = jnp.abs(β_new - β)
        β = β_new

        if display:
            β_list = [f'{t:.3}' for t in list(β.flatten())]
            update = f'{i:<13}{poisson_logL(β, model):<16.8}{β_list}'
            print(update)

        i += 1

    print(f'Number of iterations: {i}')
    print(f'β_hat = {β.flatten()}')
```

(continues on next page)

(continued from previous page)

```
    return β
```

Let's try out our algorithm with a small dataset of 5 observations and 3 variables in **X**.

```
X = jnp.array([[1, 2, 5],
               [1, 1, 3],
               [1, 4, 2],
               [1, 5, 2],
               [1, 3, 1]])

y = jnp.array([1, 0, 1, 1, 0])

# Take a guess at initial βs
init_β = jnp.array([0.1, 0.1, 0.1]).reshape(X.shape[1], 1)

# Create an object with Poisson model values
poi = create_regression_model(X, y)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, init_β, display=True)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
 ↪------
0            -4.3447622      ['-1.49', '0.265', '0.244']
1            -3.5742413      ['-3.38', '0.528', '0.474']
2            -3.3999526      ['-5.06', '0.782', '0.702']
3            -3.3788646      ['-5.92', '0.909', '0.82']
4            -3.3783559      ['-6.07', '0.933', '0.843']
5            -3.3783555      ['-6.08', '0.933', '0.843']
6            -3.3783555      ['-6.08', '0.933', '0.843']
Number of iterations: 7
β_hat = [-6.07848573  0.9334028   0.84329677]
```

As this was a simple model with few observations, the algorithm achieved convergence in only 7 iterations.

The gradient vector should be close to 0 at $\hat{\beta}$

```
G_poisson_logL(β_hat, poi)
```

```
Array([[-2.53297383e-13],
       [-6.40432152e-13],
       [-4.93882713e-13]], dtype=float64)
```

21.3 MLE with statsmodels

We'll use the Poisson regression model in `statsmodels` to verify the results obtained using JAX.

`statsmodels` uses the same algorithm as above to find the maximum likelihood estimates.

Now, as `statsmodels` accepts only NumPy arrays, we can use the `__array__` method of JAX arrays to convert it to NumPy arrays.

```
X_numpy = X.__array__()
y_numpy = y.__array__()
```

```
stats_poisson = Poisson(y_numpy, X_numpy).fit()
print(stats_poisson.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.675671
         Iterations 7
                          Poisson Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                        Poisson   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Fri, 09 Oct 2026   Pseudo R-squ.:                  0.2546
Time:                        00:25:17   Log-Likelihood:                -3.3784
converged:                       True   LL-Null:                       -4.5325
Covariance Type:            nonrobust   LLR p-value:                    0.3153
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.0785      5.279     -1.151      0.250     -16.425       4.268
x1             0.9334      0.829      1.126      0.260      -0.691       2.558
x2             0.8433      0.798      1.057      0.291      -0.720       2.407
==============================================================================
```

The benefit of writing our own procedure, relative to statsmodels is that

- we can exploit the power of the GPU and
- we learn the underlying methodology, which can be extended to complex situations where no existing routines are available.

Exercise 21.3.1

We define a quadratic model for a single explanatory variable by

$$\log(\lambda_t) = \beta_0 + \beta_1 x_t + \beta_2 x_t^2$$

We calculate the mean on the original scale instead of the log scale by exponentiating both sides of the above equation, which gives

$$\lambda_t = \exp(\beta_0 + \beta_1 x_t + \beta_2 x_t^2) \tag{21.1}$$

Simulate the values of x_t by sampling from a normal distribution and λ_t by using (21.1) and the following constants:

$$\beta_0 = -2.5, \quad \beta_1 = 0.25, \quad \beta_2 = 0.5$$

Try to obtain the approximate values of $\beta_0, \beta_1, \beta_2$, by simulating a Poisson Regression Model such that

$$y_t \sim \text{Poisson}(\lambda_t) \quad \text{for all } t.$$

Using our `newton_raphson` function on the data set $X = [1, x_t, x_t^2]$ and y, obtain the maximum likelihood estimates of $\beta_0, \beta_1, \beta_2$.

With a sufficient large sample size, you should approximately recover the true values of of these parameters.

Solution

Let's start by defining "true" parameter values.

```
β_0 = -2.5
β_1 = 0.25
β_2 = 0.5
```

To simulate the model, we sample 500,000 values of x_t from the standard normal distribution.

```
seed = 32
shape = (500_000, 1)
key = jax.random.key(seed)
x = jax.random.normal(key, shape)
```

We compute λ using (21.1)

```
λ = jnp.exp(β_0 + β_1 * x + β_2 * x**2)
```

Let's define y_t by sampling from a Poisson distribution with mean as λ_t.

```
y = jax.random.poisson(key, λ, shape)
```

Now let's try to recover the true parameter values using the Newton-Raphson method described above.

```
X = jnp.hstack((jnp.ones(shape), x, x**2))

# Take a guess at initial βs
init_β = jnp.array([0.1, 0.1, 0.1]).reshape(X.shape[1], 1)

# Create an object with Poisson model values
poi = create_regression_model(X, y)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, init_β, tol=1e-5, display=True)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
 ↪---------
0            -2.2775261e+08  ['-1.58', '0.338', '0.87']
1            -8.3204829e+07  ['-2.55', '0.338', '0.869']
2            -3.0152953e+07  ['-3.48', '0.338', '0.866']
3            -1.075084e+07   ['-4.29', '0.336', '0.856']
4            -3704112.6      ['-4.81', '0.333', '0.832']
5            -1155469.9      ['-4.71', '0.324', '0.778']
6            -213241.4       ['-3.81', '0.307', '0.684']
7            124155.52       ['-2.99', '0.284', '0.596']
8            220518.05       ['-2.67', '0.266', '0.54']
9            236821.42       ['-2.54', '0.255', '0.509']
10           237651.75       ['-2.51', '0.252', '0.5']
11           237654.97       ['-2.5', '0.252', '0.5']
12           237654.97       ['-2.5', '0.252', '0.5']
Number of iterations: 13
β_hat = [-2.50352915  0.25171711  0.49977956]
```

The maximum likelihood estimates are similar to the true parameter values.

CHAPTER

TWENTYTWO

SIMPLE NEURAL NETWORK REGRESSION WITH KERAS AND JAX

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In this lecture we show how to implement one-dimensional nonlinear regression using a neural network.

We will use Keras, a popular and relatively simple library for deep learning.

The emphasis in Keras is on providing an intuitive API, while the heavy lifting is done by one of several possible backends.

Currently the backend library options are TensorFlow, PyTorch, and JAX.

In this lecture we will use JAX.

Our main task is to provide an elementary introduction to deep learning in a regression setting.

Later, in *a separate lecture*, we will investigate how to do the same learning task using JAX directly, rather than relying on Keras.

We begin this lecture with some standard imports.

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

Let's install Keras.

```
!pip install keras
```

Now we specify that the desired backend is JAX.

```
import os
os.environ['KERAS_BACKEND'] = 'jax'
```

Now we should be able to import some tools from Keras.

Note

Without setting the backend to JAX, the imports below might fail (unless you have PyTorch or TensorFlow set up).

If you have problems running the next cell in Jupyter, try

1. quitting
2. running `export KERAS_BACKEND="jax"`
3. starting Jupyter on the command line from the same terminal.

```
import keras
from keras import Sequential
from keras.layers import Dense
```

22.1 Data

First let's write a function to generate some data.

The data has the form

$$y_i = f(x_i) + \epsilon_i, \qquad i = 1, \dots, n,$$

where

- the input sequence (x_i) is an evenly-spaced grid,
- f is a nonlinear transformation, and
- each ϵ_i is independent white noise.

Here's the function that creates vectors `x` and `y` according to the rule above.

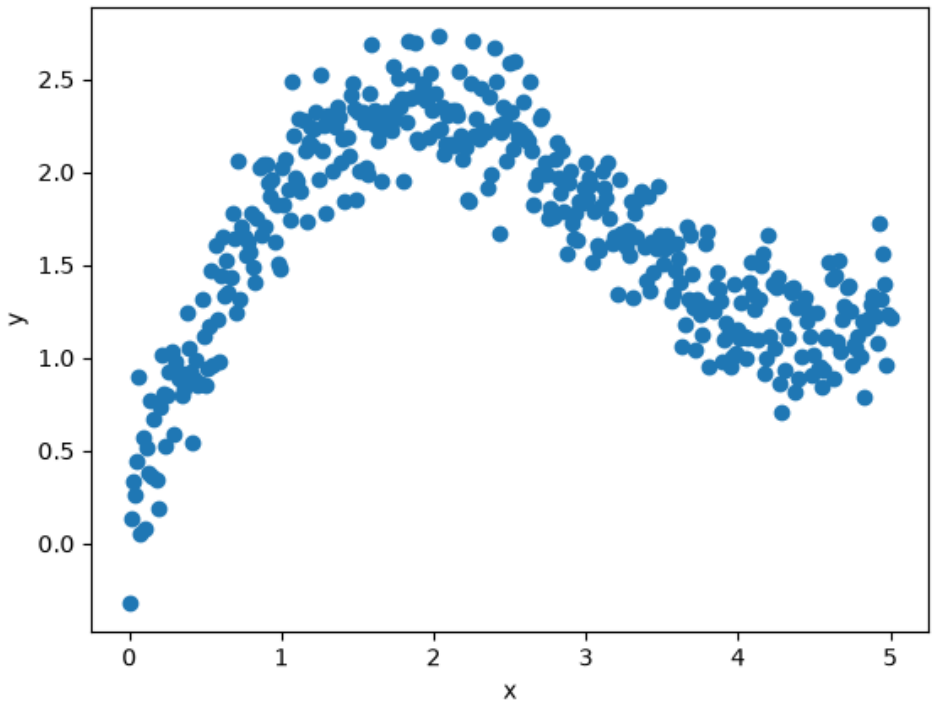
```
def generate_data(x_min=0,            # Minimum x value
                  x_max=5,            # Max x value
                  data_size=400,      # Default size for dataset
                  seed=1234):
    rng = np.random.default_rng(seed)
    x = np.linspace(x_min, x_max, num=data_size)

    ϵ = 0.2 * rng.standard_normal(data_size)
    y = x**0.5 + np.sin(x) + ϵ
    # Keras expects two dimensions, not flat arrays
    x, y = [np.reshape(z, (data_size, 1)) for z in (x, y)]
    return x, y
```

Now we generate some data to train the model.

```
x, y = generate_data()
```

Here's a plot of the training data.

```
fig, ax = plt.subplots()
ax.scatter(x, y)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```



We'll also use data from the same process for validation.

```
x_validate, y_validate = generate_data(seed=5678)
```

22.2 Models

We supply functions to build two types of models.

22.2.1 Regression model

The first implements linear regression.

This is achieved by constructing a neural network with just one layer, that maps to a single dimension (since the prediction is real-valued).

The object `model` will be an instance of `keras.Sequential`, which is used to stack layers into a single prediction model.

```
def build_regression_model():
    # Generate an instance of Sequential, to store layers and training attributes
    model = Sequential()
    # Add a single layer with scalar output
    model.add(Dense(units=1))
    # Configure the model for training
    model.compile(optimizer=keras.optimizers.SGD(),
                  loss='mean_squared_error')
    return model
```

In the function above you can see that

- we use stochastic gradient descent to train the model, and
- the loss is mean squared error (MSE).

The call `model.add` adds a single layer with the activation function equal to the identity map.

MSE is the standard loss function for ordinary least squares regression.

22.2.2 Deep Network

The second function creates a dense (i.e., fully connected) neural network with 3 hidden layers, where each hidden layer maps to a 10-dimensional output space.

```
def build_nn_model(output_dim=10, num_layers=3, activation_function='tanh'):
    model = Sequential()
    # Add layers to the network sequentially, from inputs towards outputs
    for i in range(num_layers):
        model.add(Dense(units=output_dim, activation=activation_function))
    # Add a final layer that maps to a scalar value, the prediction.
    model.add(Dense(units=1))
    # Embed training configurations
    model.compile(optimizer=keras.optimizers.SGD(),
                  loss='mean_squared_error')
    return model
```

22.2.3 Tracking errors

The following function will be used to plot the MSE of the model during the training process.

The MSE should fall as the parameters are adjusted to better fit the data.

```
def plot_loss_history(training_history, ax):
    # Plot MSE of training data against epoch
    epochs = training_history.epoch
    ax.plot(epochs,
            np.array(training_history.history['loss']),
            label='training loss')
    # Plot MSE of validation data against epoch
    ax.plot(epochs,
            np.array(training_history.history['val_loss']),
            label='validation loss')
    # Add labels
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss (Mean squared error)')
    ax.legend()
```

22.3 Training

Now let's go ahead and train our models.

22.3.1 Linear regression

We'll start with linear regression.

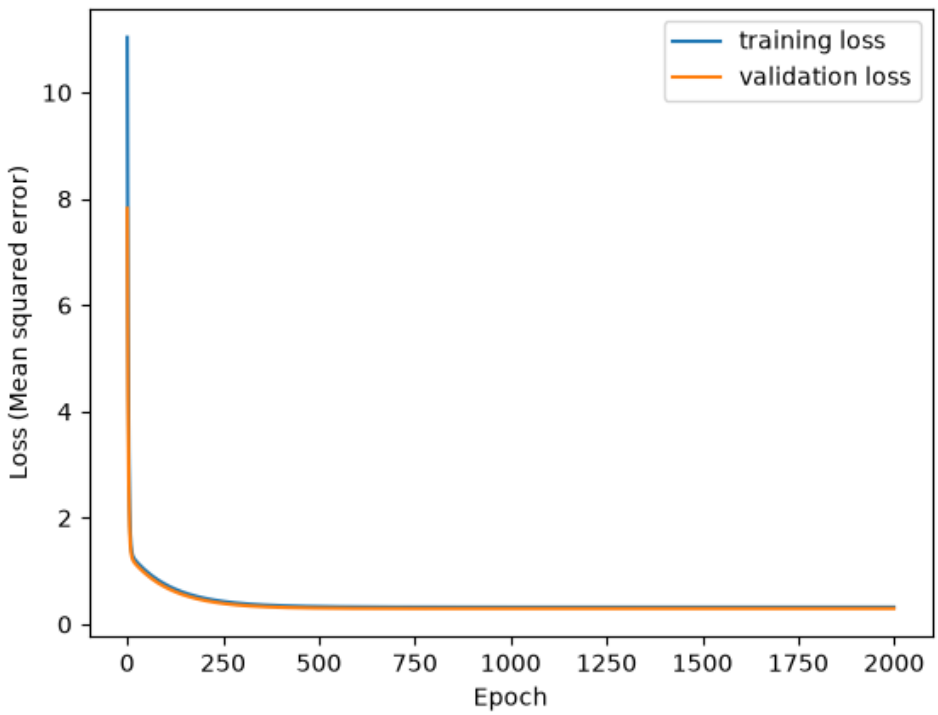
```
regression_model = build_regression_model()
```

Now we train the model using the training data.

```
training_history = regression_model.fit(
    x, y, batch_size=x.shape[0], verbose=0,
    epochs=2000, validation_data=(x_validate, y_validate))
```

Let's have a look at the evolution of MSE as the model is trained.

```
fig, ax = plt.subplots()
plot_loss_history(training_history, ax)
plt.show()
```



Let's print the final MSE on the validation data.

```
print("Testing loss on the validation set.")
regression_model.evaluate(x_validate, y_validate, verbose=2)
```

```
Testing loss on the validation set.
13/13 - 0s - 17ms/step - loss: 0.2954
```

```
0.29537487030029297
```

Here's our output predictions on the validation data.

```
y_predict = regression_model.predict(x_validate, verbose=2)
```

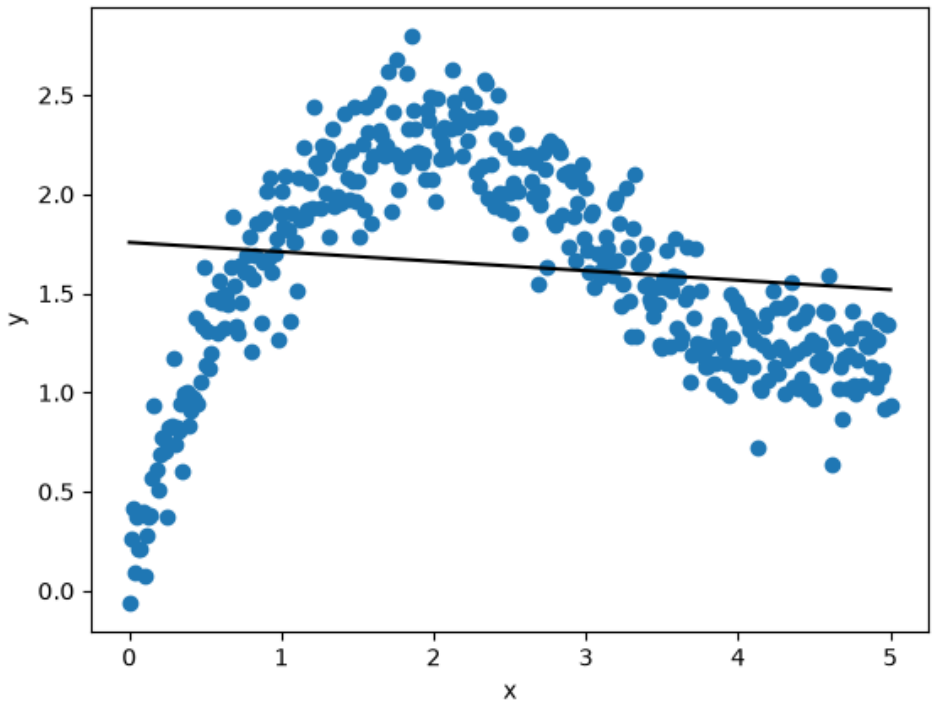
```
13/13 - 0s - 7ms/step
```

We use the following function to plot our predictions along with the data.

```
def plot_results(x, y, y_predict, ax):
    ax.scatter(x, y)
    ax.plot(x, y_predict, label="fitted model", color='black')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
```

Let's now call the function on the validation data.

```
fig, ax = plt.subplots()
plot_results(x_validate, y_validate, y_predict, ax)
plt.show()
```



22.3.2 Deep learning

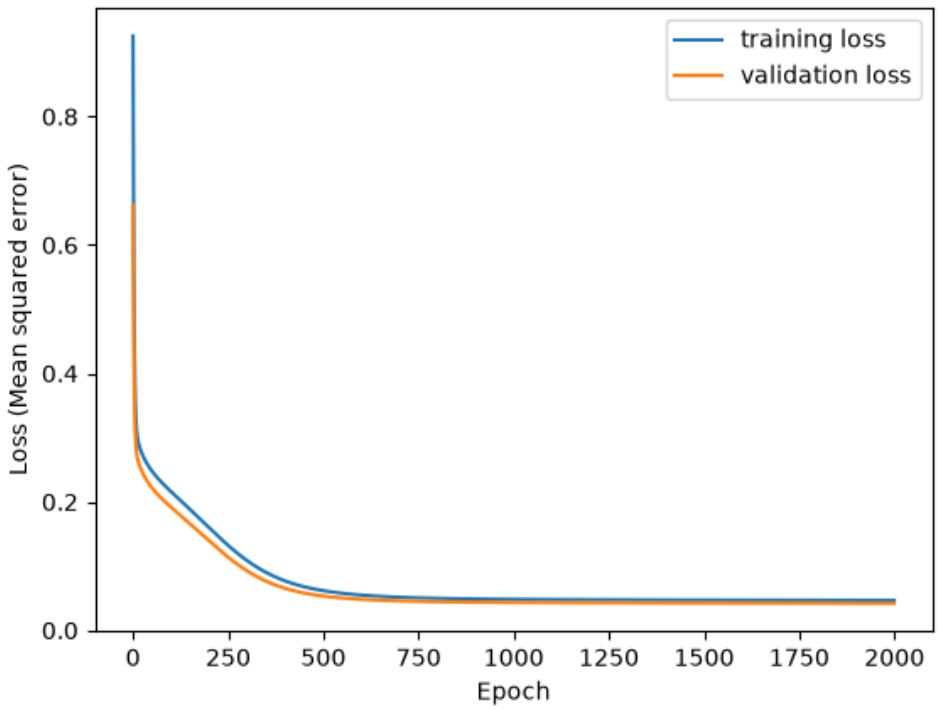
Now let's switch to a neural network with multiple layers.

We implement the same steps as before.

```
nn_model = build_nn_model()
```

```
training_history = nn_model.fit(
    x, y, batch_size=x.shape[0], verbose=0,
    epochs=2000, validation_data=(x_validate, y_validate))
```

```
fig, ax = plt.subplots()
plot_loss_history(training_history, ax)
plt.show()
```



Here's the final MSE for the deep learning model.

```
print("Testing loss on the validation set.")
nn_model.evaluate(x_validate, y_validate, verbose=2)
```

```
Testing loss on the validation set.
13/13 - 0s - 30ms/step - loss: 0.0420
```

```
0.041977375745773315
```

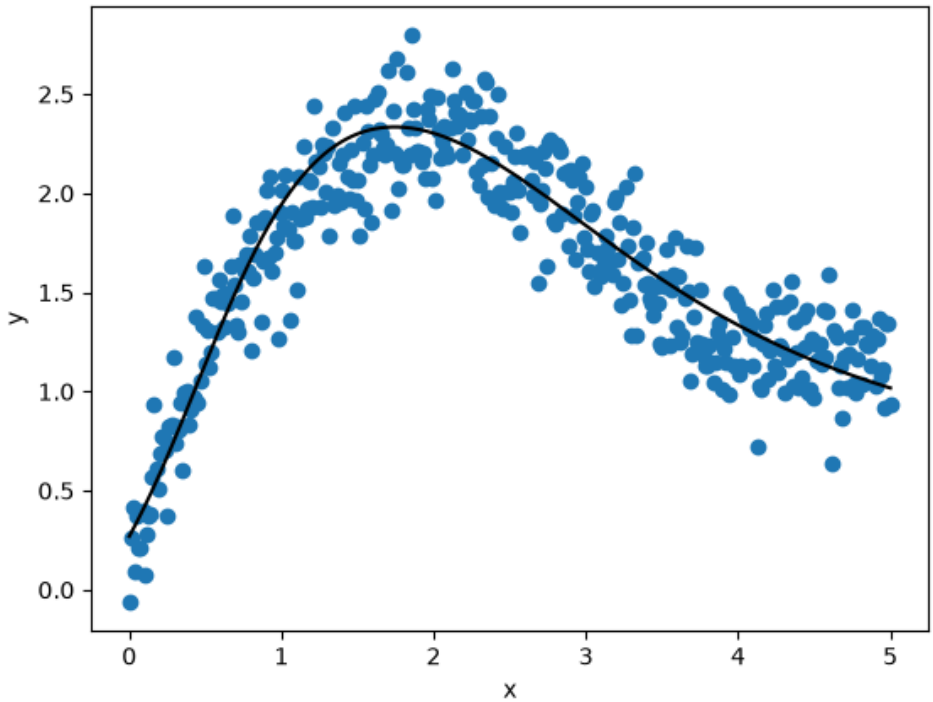
You will notice that this loss is much lower than the one we achieved with linear regression, suggesting a better fit.

To confirm this, let's look at the fitted function.

```
y_predict = nn_model.predict(x_validate, verbose=2)
```

```
13/13 - 0s - 24ms/step
```

```
fig, ax = plt.subplots()
plot_results(x_validate, y_validate, y_predict, ax)
plt.show()
```



Not surprisingly, the multilayer neural network does a much better job of fitting the data.

In *a follow-up lecture*, we will try to achieve the same fit using pure JAX, rather than relying on the Keras front-end.

CHAPTER

TWENTYTHREE

NEURAL NETWORK REGRESSION WITH JAX

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

23.1 Outline

In a *previous lecture*, we showed how to implement regression using a neural network via the deep learning library Keras.

In this lecture, we solve the same problem directly, using JAX operations rather than relying on the Keras frontend.

The objectives are

- Understand the nuts and bolts of the exercise better
- Explore more features of JAX
- Observe how using JAX directly allows us to greatly improve performance.

The lecture proceeds in three stages:

1. We solve the problem using Keras, to give ourselves a benchmark.
2. We solve the same problem in pure JAX, using pytree operations and gradient descent.
3. We solve the same problem using a combination of JAX and Optax, an optimization library built for JAX.

We begin with imports and installs.

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import os
from time import time
from typing import NamedTuple
from functools import partial
```

```
!pip install keras optax
```

```
os.environ['KERAS_BACKEND'] = 'jax'
```

```
import keras
from keras import Sequential
from keras.layers import Dense
import optax
```

23.2 Set up

Here we briefly describe the problem and generate synthetic data.

23.2.1 Flow

We use the routine from *Simple Neural Network Regression with Keras and JAX* to generate data for one-dimensional nonlinear regression.

Then we will create a dense (i.e., fully connected) neural network with 4 layers, where the input and hidden layers map to k-dimensional output space.

The inputs and outputs are scalar (for one-dimensional nonlinear regression), so the overall mapping is

$$\mathbb{R} \to \mathbb{R}^k \to \mathbb{R}^k \to \mathbb{R}^k \to \mathbb{R}$$

Here's a class to store the learning-related constants we'll use across all implementations.

Our default value of k will be 10.

```
class Config(NamedTuple):
    epochs: int = 4000              # Number of passes through the data set
    learning_rate: float = 0.001    # Learning rate for gradient descent
    layer_sizes: tuple = (1, 10, 10, 10, 1)   # Layer sizes
    seed: int = 14                  # Random seed for data generation
```

23.2.2 Data

Here's the function to generate the data for our regression analysis.

```
def generate_data(
        key: jax.Array,          # JAX random key
        data_size: int = 400,    # Sample size
        x_min: float = 0.0,      # Minimum x value
        x_max: float = 5.0       # Maximum x value
    ):
    """
    Generate synthetic regression data.
    """
    x = jnp.linspace(x_min, x_max, num=data_size)
    ϵ = 0.2 * jax.random.normal(key, shape=(data_size,))
    y = x**0.5 + jnp.sin(x) + ϵ
```

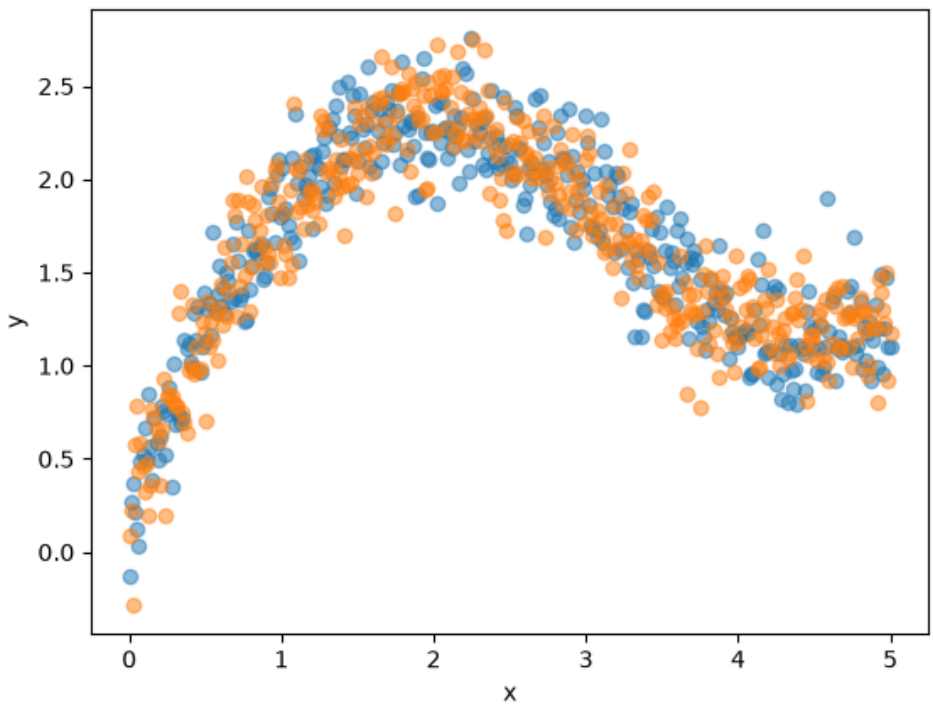
(continues on next page)

(continued from previous page)

```
    # Return observations as column vectors
    x = jnp.reshape(x, (data_size, 1))
    y = jnp.reshape(y, (data_size, 1))
    return x, y
```

Here's a plot of the data.

```
config = Config()
key = jax.random.key(config.seed)
key_train, key_validate = jax.random.split(key)
x_train, y_train = generate_data(key_train)
x_validate, y_validate = generate_data(key_validate)
keras.utils.set_random_seed(config.seed)
fig, ax = plt.subplots()
ax.scatter(x_train, y_train, alpha=0.5)
ax.scatter(x_validate, y_validate, alpha=0.5)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```



23.3 Training with Keras

We build a Keras model that can fit a nonlinear function to the generated data using an ANN.

We will use this fit as a benchmark to test our JAX code.

Since its role is only a benchmark, we refer readers to the *previous lecture* for details on the Keras interface.

We start with a function to build the model.

```
def build_keras_model(
        config: Config,                     # contains configuration data
        activation_function: str = 'tanh'   # activation with default
    ):
    model = Sequential()
    layer_sizes = config.layer_sizes

    # Add layers to the network sequentially, from inputs towards outputs
    for in_dim, out_dim in zip(layer_sizes[:-2], layer_sizes[1:-1]):
        model.add(
           Dense(
               units=out_dim,
               activation=activation_function,
               kernel_initializer=keras.initializers.HeNormal(),
               bias_initializer='ones',
           )
        )
    # Add a final layer that maps to a scalar value, for regression.
    model.add(
        Dense(
            units=layer_sizes[-1],
            kernel_initializer=keras.initializers.HeNormal(),
            bias_initializer='ones',
        )
    )
    # Embed training configurations
    model.compile(
        optimizer=keras.optimizers.SGD(
            learning_rate=config.learning_rate
        ),
        loss='mean_squared_error'
    )
    return model
```

Notice that we've set the optimizer to use stochastic gradient descent and a mean square loss.

Here is a function to train the model.

```
def train_keras_model(
        model,          # Instance of Keras Sequential model
        x,              # Training data, inputs
        y,              # Training data, outputs
        x_validate,     # Validation data, inputs
        y_validate,     # Validation data, outputs
        config: Config  # contains configuration data
    ):
    print("Training NN using Keras.")
    start_time = time()
    training_history = model.fit(
```

(continues on next page)

(continued from previous page)

```
        x, y,
        batch_size=max(x.shape),
        verbose=0,
        epochs=config.epochs,
        validation_data=(x_validate, y_validate)
    )
    elapsed = time() - start_time
    mse = model.evaluate(x_validate, y_validate, verbose=2)
    print(f"Trained in {elapsed:.2f} seconds, validation data MSE = {mse}")
    return model, training_history, elapsed, mse
```

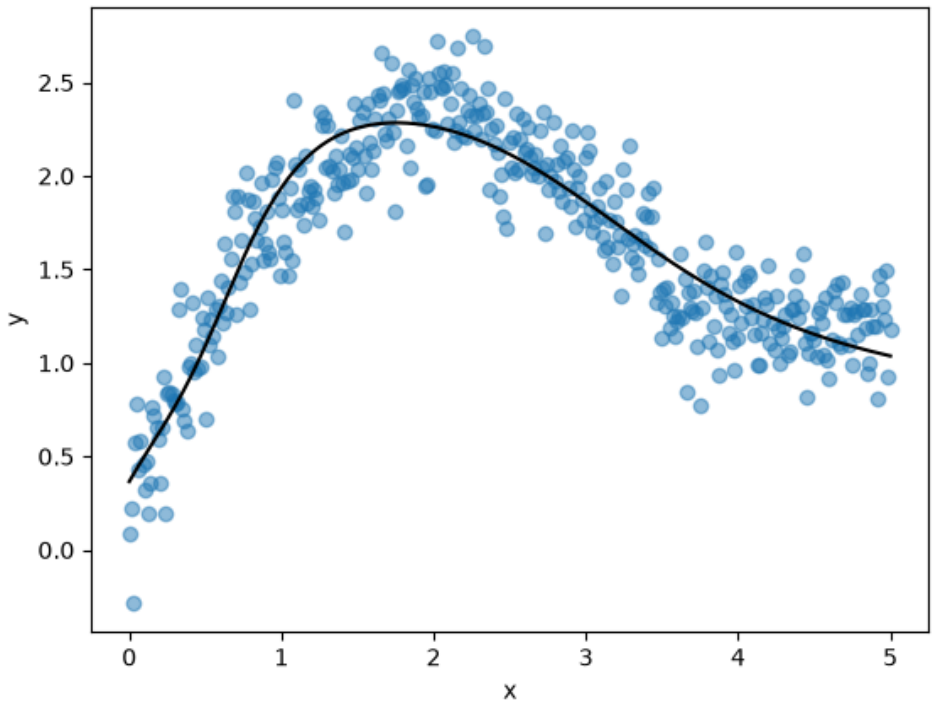
The next function extracts and visualizes a prediction from the trained Keras model.

```
def plot_keras_output(model, x_validate, y_validate):
    y_predict = model.predict(x_validate, verbose=2)
    fig, ax = plt.subplots()
    ax.scatter(x_validate, y_validate, alpha=0.5)
    ax.plot(x_validate, y_predict, label="fitted model", color='black')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    plt.show()
```

Let's run the Keras training:

```
config = Config()
model = build_keras_model(config)
model, training_history, keras_runtime, keras_mse = train_keras_model(
    model, x_train, y_train, x_validate, y_validate, config
)
plot_keras_output(model, x_validate, y_validate)
```

```
Training NN using Keras.
13/13 - 0s - 30ms/step - loss: 0.0464
Trained in 21.80 seconds, validation data MSE = 0.0464036799967289
13/13 - 0s - 24ms/step
```



The fit is good and we note the relatively low final MSE.

23.4 Training with JAX

For the JAX implementation, we need to construct the network ourselves, as a map from inputs to outputs.

We'll use the same network structure we used for the Keras implementation.

23.4.1 Background and set up

The neural network has the form

$$f(\theta, x) = (A_3 \circ \sigma \circ A_2 \circ \sigma \circ A_1 \circ \sigma \circ A_0)(x)$$

Here

- x is a scalar input – a point on the horizontal axis in the Keras estimation above,
- $\circ$ means composition of maps,
- σ is the activation function – in our case, tanh, and
- A_i represents the affine map $A_i x = W_i x + b_i$.

Each matrix W_i is called a **weight matrix** and each vector b_i is called a **bias** term.

The symbol θ represents the entire collection of parameters:

$$\theta = (W_0, b_0, W_1, b_1, W_2, b_2, W_3, b_3)$$

In fact, when we implement the affine map $A_i x = W_i x + b_i$, we will work with row vectors rather than column vectors, so that

- x and b_i are stored as row vectors, and
- the mapping is executed by JAX via the expression `x @ W + b`.

Here's a class to store parameters for one layer of the network.

```
class LayerParams(NamedTuple):
    """
    Stores parameters for one layer of the neural network.

    """
    W: jax.Array      # weights
    b: jax.Array      # biases
```

The following function initializes a single layer of the network using He initialization for weights and ones for biases.

```
def initialize_layer(in_dim, out_dim, key):
    """
    Initialize weights and biases for a single layer of the network.
    Use He initialization for weights and ones for biases.
    """
    W = jax.random.normal(key, shape=(in_dim, out_dim)) * jnp.sqrt(2 / in_dim)
    b = jnp.ones((1, out_dim))
    return LayerParams(W, b)
```

The next function builds an entire network, as represented by its parameters, by initializing layers and stacking them into a list.

```
def initialize_network(
        key: jax.Array,      # JAX random key
        config: Config       # contains configuration data
    ):
    """
    Build a network by initializing all of the parameters.
    A network is a list of LayerParams instances, each
    containing a weight-bias pair (W, b).
    """
    layer_sizes = config.layer_sizes
    params = []
    for i in range(len(layer_sizes) - 1):
        key, subkey = jax.random.split(key)
        layer = initialize_layer(
            layer_sizes[i],       # in dimension for layer
            layer_sizes[i + 1],   # out dimension for layer
            subkey
        )
        params.append(layer)
    return params
```

Wait, you say!

Shouldn't we concatenate the elements of θ into some kind of big array, so that we can do autodiff with respect to this array?

Actually we don't need to — we use the JAX PyTree approach discussed below.

23.4.2 Coding the network

Here's our implementation of the ANN f:

```
def f(
        θ: list,                        # Network parameters (pytree)
        x: jax.Array,                 # Input data (row vector)
        σ: callable = jnp.tanh          # Activation function
    ):
    """
    Perform a forward pass over the network to evaluate f(θ, x).
    """
    *hidden, last = θ
    for layer in hidden:
        x = σ(x @ layer.W + layer.b)
    x = x @ last.W + last.b
    return x
```

The function f is appropriately vectorized, so that we can pass in the entire set of input observations as `x` and return the predicted vector of outputs `y_hat = f(θ, x)` corresponding to each data point.

The loss function is mean squared error, the same as the Keras case.

```
def loss_fn(
        θ: list,            # Network parameters (pytree)
        x: jax.Array,     # Input data
        y: jax.Array      # Target data
    ):
    return jnp.mean((f(θ, x) - y)**2)
```

We'll also define a function to visualize predictions from our JAX models.

```
def plot_jax_output(θ, x_validate, y_validate):
    fig, ax = plt.subplots()
    ax.scatter(x_validate, y_validate, color='red', alpha=0.5)
    ax.plot(x_validate.flatten(), f(θ, x_validate).flatten(),
            label="fitted model", color='black')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    plt.show()
```

We'll use the gradient of the loss to do stochastic gradient descent.

(Technically, we will be doing gradient descent, rather than stochastic gradient descent, since we will not randomize over sample points when we evaluate the gradient.)

```
loss_gradient = jax.grad(loss_fn)
```

The gradient of `loss_fn` is with respect to the first argument `θ`.

The code above seems kind of magical, since we are differentiating with respect to a parameter "vector" stored as a list of namedtuples containing arrays.

How can we differentiate with respect to such a complex object?

The answer is that the list of namedtuples is treated internally as a pytree.

The JAX function `grad` understands how to

1. extract the individual arrays (the "leaves" of the tree),

2. compute derivatives with respect to each one, and
3. pack the resulting derivatives into a pytree with the same structure as the parameter vector.

23.4.3 Gradient descent

Using the above code, we can now write our rule for updating the parameters via gradient descent, which is the algorithm we covered in our lecture on autodiff.

In this case, to keep things as simple as possible, we'll use a fixed learning rate for every iteration.

```
def update_parameters(
        θ: list,            # Current parameters (pytree)
        x: jax.Array,     # Input data
        y: jax.Array,     # Target data
        config: Config      # contains configuration data
    ):
    """
    Update the parameter pytree using gradient descent.

    """
    λ = config.learning_rate

    # Specify the update rule
    def gradient_descent_step(p, g):
        """
        A rule for updating parameter vector p given gradient vector g.
        It will be applied to each leaf of the pytree of parameters.
        """
        return p - λ * g

    gradient = loss_gradient(θ, x, y)

    # Use tree.map to apply the update rule to the parameter vectors
    θ_new = jax.tree.map(gradient_descent_step, θ, gradient)
    return θ_new
```

Here `jax.tree.map` understands `θ` and `gradient` as pytrees of the same structure and executes `p - λ * g` on the corresponding leaves of the pair of trees.

Each weight matrix and bias vector is updated by gradient descent, exactly as required.

Here's code that puts this all together.

```
@partial(jax.jit, static_argnames=['config'])
def train_jax_model(
        θ: list,                    # Initial parameters (pytree)
        x: jax.Array,             # Training input data
        y: jax.Array,             # Training target data
        config: Config              # contains configuration data
    ):
    """
    Train model using gradient descent.
    """
    def update(_, θ):
        θ_new = update_parameters(θ, x, y, config)
        return θ_new
```

(continues on next page)

(continued from previous page)

```
    θ_final = jax.lax.fori_loop(0, config.epochs, update, θ)
    return θ_final
```

23.4.4 Execution

Let's run our code and see how it goes.

We'll reuse the data we generated earlier.

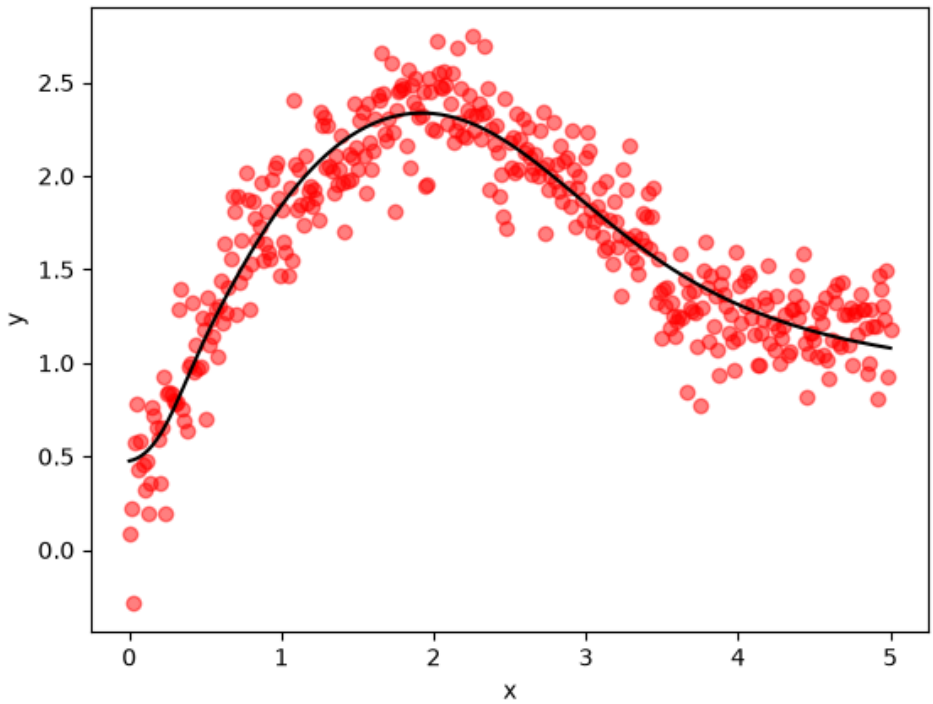
```
# Reset parameter vector
config = Config()
param_key = jax.random.key(1234)
θ = initialize_network(param_key, config)

# Warmup run to trigger JIT compilation
train_jax_model(θ, x_train, y_train, config)

# Reset and time the actual run
θ = initialize_network(param_key, config)
start_time = time()
θ = train_jax_model(θ, x_train, y_train, config)
θ[0].W.block_until_ready()  # Ensure computation completes
jax_runtime = time() - start_time

jax_mse = loss_fn(θ, x_validate, y_validate)
jax_train_mse = loss_fn(θ, x_train, y_train)
print(f"Trained model with JAX in {jax_runtime:.2f} seconds.")
print(f"Final MSE on validation data = {jax_mse:.6f}")
```

```
Trained model with JAX in 0.25 seconds.
Final MSE on validation data = 0.042119
```

Despite the simplicity of our implementation, we actually perform slightly better than Keras.

Here's a visualization of the quality of our fit.

```
plot_jax_output(θ, x_validate, y_validate)
```



23.5 JAX plus Optax

Our hand-coded optimization routine above was quite effective, but in practice we might wish to use an optimization library written for JAX.

One such library is Optax.

23.5.1 Optax with SGD

Here's a training routine using Optax's stochastic gradient descent solver.

```
@partial(jax.jit, static_argnames=['config'])
def train_jax_optax(
        θ: list,                    # Initial parameters (pytree)
        x: jax.Array,             # Training input data
        y: jax.Array,             # Training target data
        config: Config              # contains configuration data
    ):
    " Train model using Optax SGD optimizer. "
    epochs = config.epochs
    learning_rate = config.learning_rate
    solver = optax.sgd(learning_rate)
    opt_state = solver.init(θ)

    def update(_, loop_state):
        θ, opt_state = loop_state
        grad = loss_gradient(θ, x, y)
        updates, new_opt_state = solver.update(grad, opt_state, θ)
        θ_new = optax.apply_updates(θ, updates)
```

(continues on next page)

(continued from previous page)

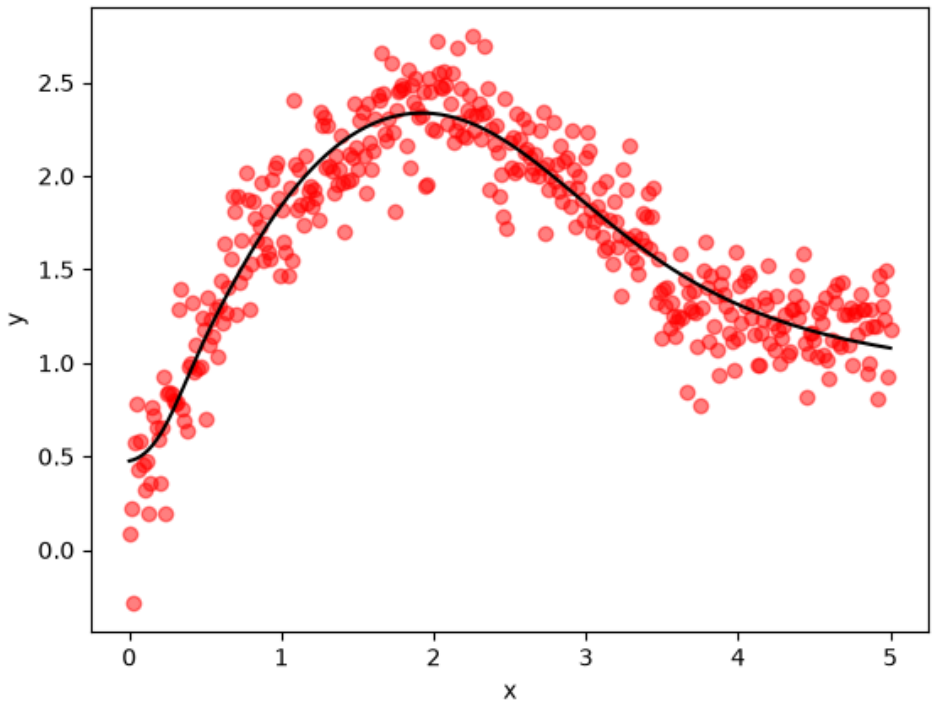
```
        new_loop_state = θ_new, new_opt_state
        return new_loop_state

    initial_loop_state = θ, opt_state
    final_loop_state = jax.lax.fori_loop(
        0, epochs, update, initial_loop_state
    )
    θ_final, _ = final_loop_state
    return θ_final
```

Let's try running it.

```
# Reset parameter vector
θ = initialize_network(param_key, config)

# Warmup run to trigger JIT compilation
train_jax_optax(θ, x_train, y_train, config)

# Reset and time the actual run
θ = initialize_network(param_key, config)
start_time = time()
θ = train_jax_optax(θ, x_train, y_train, config)
θ[0].W.block_until_ready()  # Ensure computation completes
optax_sgd_runtime = time() - start_time

optax_sgd_mse = loss_fn(θ, x_validate, y_validate)
optax_sgd_train_mse = loss_fn(θ, x_train, y_train)
print(
    "Trained model with JAX and Optax SGD "
    f"in {optax_sgd_runtime:.2f} seconds."
)
print(f"Final MSE on validation data = {optax_sgd_mse:.6f}")
```

```
Trained model with JAX and Optax SGD in 0.25 seconds.
Final MSE on validation data = 0.042119
```

```
plot_jax_output(θ, x_validate, y_validate)
```



23.5.2 Optax with Adam

We can also consider using a slightly more sophisticated gradient-based method, such as Adam.

You will notice that the syntax for using this alternative optimizer is very similar.

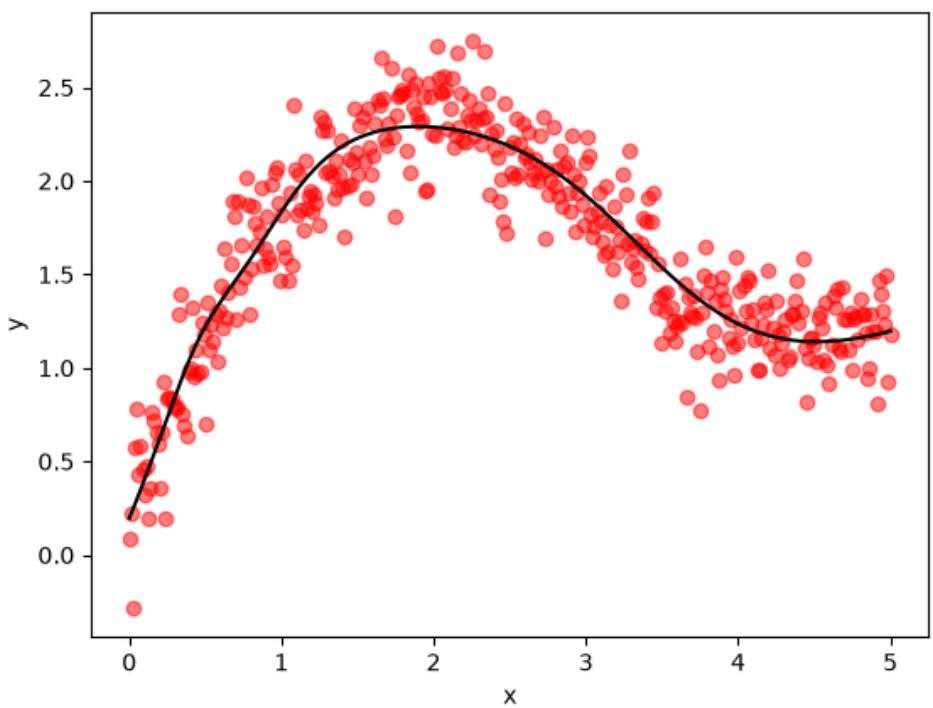
```
@partial(jax.jit, static_argnames=['config'])
def train_jax_optax_adam(
        θ: list,                    # Initial parameters (pytree)
        x: jax.Array,             # Training input data
        y: jax.Array,             # Training target data
        config: Config              # contains configuration data
    ):
    " Train model using Optax Adam optimizer. "
    epochs = config.epochs
    learning_rate = config.learning_rate
    solver = optax.adam(learning_rate)
    opt_state = solver.init(θ)

    def update(_, loop_state):
        θ, opt_state = loop_state
        grad = loss_gradient(θ, x, y)
        updates, new_opt_state = solver.update(grad, opt_state, θ)
        θ_new = optax.apply_updates(θ, updates)
        return (θ_new, new_opt_state)

    initial_loop_state = θ, opt_state
    θ_final, _ = jax.lax.fori_loop(
        0, epochs, update, initial_loop_state
    )
    return θ_final
```

```
# Reset parameter vector
θ = initialize_network(param_key, config)

# Warmup run to trigger JIT compilation
train_jax_optax_adam(θ, x_train, y_train, config)

# Reset and time the actual run
θ = initialize_network(param_key, config)
start_time = time()
θ = train_jax_optax_adam(θ, x_train, y_train, config)
θ[0].W.block_until_ready()  # Ensure computation completes
optax_adam_runtime = time() - start_time

optax_adam_mse = loss_fn(θ, x_validate, y_validate)
optax_adam_train_mse = loss_fn(θ, x_train, y_train)
print(f"Trained model with JAX and Optax Adam in {optax_adam_runtime:.2f} seconds.")
print(f"Final MSE on validation data = {optax_adam_mse:.6f}")
```

```
Trained model with JAX and Optax Adam in 0.25 seconds.
Final MSE on validation data = 0.040869
```

Here's a visualization of the result.

```
plot_jax_output(θ, x_validate, y_validate)
```



23.6 Summary

Here we compare the performance of the four different training approaches we explored in this lecture.

```
Summary of Training Methods:
                  Method  Runtime (s) Training MSE Validation MSE
                   Keras    21.799850     0.041501       0.046404
Pure JAX (hand-coded GD)     0.252983     0.039963       0.042119
         JAX + Optax SGD     0.248129     0.039963       0.042119
        JAX + Optax Adam     0.252384     0.035879       0.040869
```

All methods achieve similar validation MSE values (around 0.040-0.046).

At the time of writing, the MSEs from plain vanilla Optax and our own hand-coded SGD routine are identical.

The Adam optimizer achieves slightly better MSE by using adaptive learning rates.

Still, our hand-coded algorithm does very well compared to this high-quality optimizer.

Note also that the pure JAX implementations are significantly faster than Keras.

This is because JAX can JIT-compile the entire training loop.

Not surprisingly, Keras has more overhead from its abstraction layers.

23.7 Exercises

Exercise 23.7.1

Try to reduce the MSE on the validation data without significantly increasing the computational load.

You should hold constant both the number of epochs and the total number of parameters in the network.

Currently, the network has 4 layers with output dimension $k = 10$, giving a total of 251 parameters.

You can experiment with:

- Changing the network architecture
- Trying different activation functions
- Modifying the optimizer (e.g., learning rates, learning rate schedules, momentum, etc.)
- Experimenting with different weight initialization strategies
- Modifying the loss function (e.g., adding regularization)

Which combination gives you the lowest validation MSE?

Solution

Let's implement and test several strategies.

Strategy 1: LR schedule and L2 regularization

Let's keep the baseline network architecture and add a learning rate schedule with L2 regularization:

```
# Strategy 1: LR schedule and L2 regularization
```

```
def loss_fn_l2(
        θ: list,
        x: jnp.ndarray,
        y: jnp.ndarray,
        λ_l2: float
    ):
    " L2-regularized MSE loss. "
    mse = jnp.mean((f(θ, x) - y)**2)
    l2_penalty = 0.0
    for W, b in θ:
        l2_penalty += jnp.sum(W**2)
    return mse + λ_l2 * l2_penalty


loss_gradient_l2 = jax.grad(loss_fn_l2)


@partial(jax.jit, static_argnames=['config'])
def train_with_schedule_and_l2(
        θ: list,
        x: jax.Array,
        y: jax.Array,
        config: Config,
        λ_l2: float = 0.001
    ):
    """
    Train baseline network with learning rate schedule
    and L2 regularization.
    """
    epochs = config.epochs
    schedule = optax.exponential_decay(
        init_value=0.003,
        transition_steps=1000,
        decay_rate=0.5
    )

    solver = optax.adam(schedule)
    opt_state = solver.init(θ)

    def update(_, loop_state):
        θ_curr, opt_state_curr = loop_state
        grad = loss_gradient_l2(θ_curr, x, y, λ_l2)
        updates, opt_state_new = solver.update(
            grad, opt_state_curr, θ_curr
        )
        θ_new = optax.apply_updates(θ_curr, updates)
        return (θ_new, opt_state_new)

    initial_loop_state = θ, opt_state
    θ_final, _ = jax.lax.fori_loop(
        0, epochs, update, initial_loop_state
    )
    return θ_final

# Warmup
θ_l2 = initialize_network(param_key, config)
train_with_schedule_and_l2(θ_l2, x_train, y_train, config)
```

```
# Actual run
θ_l2 = initialize_network(param_key, config)
start_time = time()
θ_l2 = train_with_schedule_and_l2(θ_l2, x_train, y_train, config)
θ_l2[0].W.block_until_ready()
deep_l2_runtime = time() - start_time

deep_l2_mse = loss_fn(θ_l2, x_validate, y_validate)
print(f"Strategy 1 - LR schedule and L2 regularization")
print(f"  Runtime: {deep_l2_runtime:.2f}s")
print(f"  Validation MSE: {deep_l2_mse:.6f}")
print(f"  Improvement over Adam: {optax_adam_mse - deep_l2_mse:.6f}")
```

```
Strategy 1 - LR schedule and L2 regularization
  Runtime: 0.27s
  Validation MSE: 0.040850
  Improvement over Adam: 0.000018
```

Strategy 2: Baseline + Armijo line search

Let's implement gradient descent with Armijo line search for adaptive step size selection:

```
# Strategy 2: Baseline architecture + Armijo line search
# Line search parameters
line_search_init_value = 0.01
line_search_backtrack_factor = 0.5
line_search_armijo_constant = 0.001
max_backtrack_steps = 20

@partial(jax.jit, static_argnames=['config'])
def train_jax_armijo_ls(
        θ: list,                    # Initial parameters (pytree)
        x: jnp.ndarray,             # Training input data
        y: jnp.ndarray,             # Training target data
        config: Config              # contains configuration data
    ):
    """
    Train model using gradient descent with Armijo line search.

    The Armijo line search adaptively finds a suitable step size at each
    iteration by ensuring sufficient decrease in the loss function.
    """
    epochs = config.epochs

    # Line search parameters
    α_init = line_search_init_value
    backtrack_factor = line_search_backtrack_factor
    armijo_c = line_search_armijo_constant

    def update_step(θ_current, x_data, y_data):
        current_loss = loss_fn(θ_current, x_data, y_data)
        g = loss_gradient(θ_current, x_data, y_data)

        # Squared Euclidean norm of gradient for Armijo condition
        g_norm_sq = jax.tree.reduce(
            lambda a, b: a + jnp.sum(b**2),
            g,
            initializer=0.0,
```

```
        )

        # Define the condition for the while_loop
        def cond_fn(loop_args):
            (α_val,
             current_loss_val,
             g_sq_sum,
             θ_orig,
             x_in,
             y_in,
             step_count) = loop_args

            loss_threshold = (
                current_loss_val - armijo_c * α_val * g_sq_sum
            )

            θ_candidate = jax.tree.map(
                lambda p, g_leaf: p - α_val * g_leaf,
                θ_orig,
                g,
            )
            loss_candidate = loss_fn(θ_candidate, x_in, y_in)
            return ((loss_candidate > loss_threshold)
                    & (step_count < max_backtrack_steps))

        # Define the body for the while_loop
        def body_fn(loop_args):
            (α_val,
             current_loss_val,
             g_sq_sum,
             θ_orig,
             x_in,
             y_in,
             step_count) = loop_args
            α_new = α_val * backtrack_factor
            step_count_new = step_count + 1
            return (α_new,
                    current_loss_val,
                    g_sq_sum,
                    θ_orig,
                    x_in,
                    y_in,
                    step_count_new)

        # Execute the Armijo line search using jax.lax.while_loop
        loop_state = jax.lax.while_loop(
            cond_fn,
            body_fn,
            (α_init, current_loss, g_norm_sq, θ_current, x_data, y_data, 0),
        )
        α_final = loop_state[0]

        # Update parameters with the chosen step size
        θ_new = jax.tree.map(
            lambda p, g_leaf: p - α_final * g_leaf,
            θ_current,
            g,
```

```
        )
        return θ_new

    # Main training loop (epochs)
    def outer_update(_, θ_curr):
        return update_step(θ_curr, x, y)

    θ_final = jax.lax.fori_loop(0, epochs, outer_update, θ)
    return θ_final

# Warmup
θ = initialize_network(param_key, config)
train_jax_armijo_ls(θ, x_train, y_train, config)

# Actual run
θ = initialize_network(param_key, config)
start_time = time()
θ_armijo = train_jax_armijo_ls(θ, x_train, y_train, config)
θ_armijo[0].W.block_until_ready()
armijo_runtime = time() - start_time

armijo_mse = loss_fn(θ_armijo, x_validate, y_validate)
print(f"Strategy 2 - Baseline + Armijo line search")
print(f"  Runtime: {armijo_runtime:.2f}s")
print(f"  Validation MSE: {armijo_mse:.6f}")
print(f"  Improvement over Adam: {optax_adam_mse - armijo_mse:.6f}")
```

```
Strategy 2 - Baseline + Armijo line search
  Runtime: 0.41s
  Validation MSE: 0.040810
  Improvement over Adam: 0.000058
```

Results Summary

Let's compare all strategies:

```
Summary of Exercise Strategies:
                            Strategy  Runtime (s) Validation MSE  Improvement
              Baseline (Adam + tanh)     0.252384     0.04086869     0.000000
1. LR schedule and L2 regularization     0.267982     0.04085019     0.000018
    2. Baseline + Armijo line search     0.410513    0.040810395     0.000058
```

In terms of reducing loss on the validation data, the current winner is the Armijo line search strategy.

The Armijo backtracking line search is an adaptive step size method that dynamically adjusts the learning rate at each iteration to ensure sufficient decrease in the loss function.

Unlike fixed learning rates or predetermined schedules, it adapts to the local geometry of the loss landscape.

This strategy and its code were contributed by Matyas Farkas.

CHAPTER

TWENTYFOUR

POLICY GRADIENT-BASED OPTIMAL SAVINGS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

24.1 Introduction

In this notebook we solve infinite horizon optimal savings problems using deep learning and policy gradient ascent with JAX.

Each policy is represented as a fully connected feed forward neural network.

We begin with a cake eating problem with a known analytical solution.

Then we shift to an income fluctuation problem where we can compute an optimal policy easily with the endogenous grid method (EGM).

We do this first and then try to learn the same policy with deep learning.

The technique we will use is called policy gradient ascent.

This method is popular in the machine learning community for solving high-dimensional dynamic programming problems.

Since the income fluctuation problem is low-dimensional, the policy gradient method will not be superior to EGM.

However, by working through this lecture, we can learn the basic principles of policy gradient methods and see them work in practice.

We'll use the following libraries

```
!pip install optax
```

We'll use the following imports

```
import jax
import jax.numpy as jnp
from jax import grad, jit, random
```

(continues on next page)

(continued from previous page)

```
import optax
import matplotlib.pyplot as plt
from functools import partial
from typing import NamedTuple
```

24.2 Theory

Let's describe the income fluctuation problem and the ideas behind policy gradient ascent.

24.2.1 Household problem

A household chooses a consumption plan $\{c_t\}_{t \geq 0}$ to maximize

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

subject to

$$a_{t+1} = R(a_t - c_t) + Y_{t+1}, \quad c_t \geq 0, \quad a_t \geq 0, \quad t = 0, 1, \ldots$$

Here Y_t is labor income, which is IID and normally distributed:

$$Z_t \sim N(m, v), \quad Y_t = \exp(Z_t)$$

We assume:

1. $\beta R < 1$
2. u is CRRA with parameter γ

We will be interested in the value of alternative policy functions for this household.

Since the shocks are IID, and hence offer no predictive content for future shocks, optimal policies will depend only on current assets.

The next section discusses policies and their values in more detail.

24.2.2 Lifetime Value and Optimization

A **policy** is a function σ from $\mathbb{R}_+$ to itself, where $\sigma(a)$ is understood as the amount consumed under policy σ given current state a.

A **feasible policy** is a (measurable) policy satisfying $0 \leq \sigma(a) \leq a$ for all a (no borrowing).

We let Σ denote the set of all feasible policies.

We let $v_\sigma(a)$ be the lifetime value of following policy σ, given initial assets a.

That is,

$$v_\sigma(a) = \mathbb{E} \sum_{t \geq 0} \beta^t u(c_t)$$

where

- $c_t = \sigma(a_t)$
- $a_0 = a$
- $a_{t+1} = R(a_t - \sigma(a_t)) + Y_{t+1}$ for $t = 0, 1, \ldots$

A policy σ is called **optimal** if $v_s(a) \leq v_\sigma(a)$ for all asset levels a and all alternative policies $s \in \Sigma$.

The function v^* defined by $v^*(a) := \sup_{\sigma \in \Sigma} v_\sigma(a)$ is called the **value function**.

Using this definition, we can alternatively say that a policy σ is optimal if and only if $v_\sigma = v^*$.

We know that we can find an optimal policy using dynamic programming and, in particular, the endogenous grid method (EGM).

Now let's look at another method.

24.2.3 The policy gradient approach

The policy gradient approach starts by fixing an initial distribution F and trying to solve

$$\max_{\sigma \in \Sigma} \int v_\sigma(a) F(da)$$

Working with this alternative objective transforms a dynamic programming problem into a regular optimization with a real-valued objective (the last display).

Here we'll focus on the case where F concentrates on a single point a_0, so the objective becomes

$$\max_{\sigma \in \Sigma} M(\sigma) \quad \text{where} \quad M(\sigma) := v_\sigma(a_0)$$

> **Note**
>
> Does our choice of the initial condition a_0 matter in terms of delivering an optimal policy?
>
> The answer is, in general, yes.
>
> Essentially, we want the state to explore as much of the state space as possible.
>
> We'll try to engineer this outcome in our choice of a_0.

From here the approach is

1. Replace Σ with $\{\sigma(\cdot, \theta) \,:\, \theta \in \Theta\}$ where $\sigma(\cdot, \theta)$ is an ANN with parameter vector θ
2. Replace the objective function with $M(\theta) := v_{\sigma(\cdot,\theta)}(a_0)$
3. Replace M with a Monte Carlo approximation $\hat{M}$
4. Use gradient ascent to maximize $\hat{M}(\theta)$ over θ.

In the last step we do

$$\theta_{n+1} = \theta_n + \lambda_n \nabla \hat{M}(\theta_n)$$

We compute $\hat{M}$ via

$$\hat{M}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \beta^t u(\sigma(a_t^i, \theta))$$

Here a_0^i is fixed at the given value a_0 for all i and

$$a_{t+1}^i = R(a_t^i - \sigma(a_t^i, \theta)) + Y_{t+1}^i$$

24.3 Network

Before we get to policy gradient ascent, let's set up a generic deep learning environment.

The environment will work with an arbitrary loss function.

Below, in each optimal savings application, we will specify the loss function as $-\hat{M}$, where $\hat{M}$ is the approximation to lifetime value defined above.

Thus, minimizing loss in policy space means maximizing lifetime value (given fixed a_0).

We store some fixed values that form part of the network training configuration.

```
class Config(NamedTuple):
    """
    Configuration and parameters for training the neural network.

    """
    seed: int = 1234                          # Seed for network initialization
    epochs: int = 400                         # No of training epochs
    path_length: int = 200                    # Length of each consumption path
    layer_sizes: tuple = (1, 6, 6, 6, 1)      # Network layer sizes
    learning_rate: float = 0.001              # Constant learning rate
    num_paths: int = 100                      # Number of paths to average over
```

We use a class called `LayerParams` to store parameters representing a single layer of the neural network.

```
class LayerParams(NamedTuple):
    """
    Stores parameters for one layer of the neural network.

    """
    W: jnp.ndarray     # weights
    b: jnp.ndarray     # biases
```

The following function initializes a single layer of the network using Le Cun initialization.

```
def initialize_layer(
        in_dim: int,                      # Input dimension for the layer
        out_dim: int,                     # Output dimension for the layer
        key: jax.Array                    # Random key for initialization
    ):
    """
    Initialize weights and biases for a single layer of the network.
    Use LeCun initialization.

    """
    s = jnp.sqrt(1.0 / in_dim)
    W = jax.random.normal(key, (in_dim, out_dim)) * s
    b = jnp.zeros((out_dim,))
    return LayerParams(W, b)
```

The next function builds an entire network, as represented by its parameters, by initializing layers and stacking them into a list.

```
def initialize_network(
        key: jax.Array,                   # Random key for initialization
        layer_sizes: tuple                # Layer sizes (input, hidden..., output)
```

(continues on next page)

(continued from previous page)

```
    ):
    """
    Build a network by initializing all of the parameters.
    A network is a list of LayerParams instances, each
    containing a weight-bias pair (W, b).

    """
    params = []
    # For all layers but the output layer
    for i in range(len(layer_sizes) - 1):
        # Build the layer
        key, subkey = jax.random.split(key)
        layer = initialize_layer(
            layer_sizes[i],       # in dimension for layer
            layer_sizes[i + 1],   # out dimension for layer
            subkey
        )
        # And add it to the parameter list
        params.append(layer)

    return params
```

Next we write a function to train the network by gradient *descent*, given a generic loss function.

Note

We use gradient descent rather than ascent because we'll employ optax, which expects to be minimizing a loss function.

To make this work, we'll set the loss to $-\widehat{M}(\theta)$.

Here's the function.

```
@partial(jax.jit, static_argnames=('config', 'loss_fn'))
def train_network(
        config: Config,             # Configuration with training parameters
        loss_fn: callable           # Loss function taking params, returning loss
    ):
    """
    Train a neural network using policy gradient ascent.

    This is a generic training function that can be applied to different
    models by providing an appropriate loss function.

    """
    # Initialize network parameters
    key = random.key(config.seed)
    params = initialize_network(key, config.layer_sizes)

    # Set up optimizer
    optimizer = optax.chain(
        optax.clip_by_global_norm(1.0),  # Gradient clipping for stability
        optax.adam(learning_rate=config.learning_rate)
    )
    opt_state = optimizer.init(params)
```

(continues on next page)

(continued from previous page)

```
    # Training loop state
    def step(i, state):
        params, opt_state, best_value, best_params, value_history = state

        # Compute value and gradients at existing parameterization
        loss, grads = jax.value_and_grad(loss_fn)(params)
        lifetime_value = -loss

        # Update value history
        value_history = value_history.at[i].set(lifetime_value)

        # Track best parameters
        is_best = lifetime_value > best_value
        best_value = jnp.where(is_best, lifetime_value, best_value)
        best_params = jax.tree.map(
            lambda new, old: jnp.where(is_best, new, old),
            params, best_params
        )

        # Update parameters using optimizer
        updates, opt_state = optimizer.update(grads, opt_state)
        params = optax.apply_updates(params, updates)

        return params, opt_state, best_value, best_params, value_history

    # Run training loop
    value_history = jnp.zeros(config.epochs)
    initial_state = (params, opt_state, -jnp.inf, params, value_history)
    final_state = jax.lax.fori_loop(
        0, config.epochs, step, initial_state
    )

    # Extract results
    _, _, best_value, best_params, value_history = final_state
    return best_params, value_history, best_value
```

24.4 Cake Eating Case

We will start by tackling a very simple case, without any labor income, so that Y_t is always zero and

$$a_{t+1} = R(a_t - c_t) \qquad \text{for all } t$$

For this "cake-eating" model, the optimal policy is known to be

$$c = \kappa a, \quad \text{where} \quad \kappa := 1 - [\beta R^{1-\gamma}]^{1/\gamma}$$

We use this known exact solution to check our numerical methods.

24.4.1 Cake eating loss function

We use a class called `CakeEatingModel` to store model parameters.

```
class CakeEatingModel(NamedTuple):
    """
    Stores parameters for the model.

    """
    γ: float = 1.5
    β: float = 0.96
    R: float = 1.01
```

We use CRRA utility.

```
def u(c, γ):
    """ Utility function. """
    c = jnp.maximum(c, 1e-10)
    return c**(1 - γ) / (1 - γ)
```

Now we provide a function that implements a consumption policy as a neural network, given the parameters of the network.

```
def forward(
        params: list,          # Network parameters (LayerParams list)
        a: float               # Current asset level
    ):
    """
    Evaluate neural network policy: maps a given asset level a to
    consumption rate c/a by running a forward pass through the network.

    """
    σ = jax.nn.selu           # Activation function
    x = jnp.array((a,))       # Make state a 1D array
    # Forward pass through network, without the last step
    for W, b in params[:-1]:
        x = σ(x @ W + b)
    # Complete with sigmoid activation for consumption rate
    W, b = params[-1]
    # Direct output in [0, 0.99] range for stability
    x = jax.nn.sigmoid(x @ W + b) * 0.99
    # Extract and return consumption rate
    consumption_rate = x[0]
    return consumption_rate
```

The next function approximates lifetime value for the cake eating agent associated with a given policy, as represented by the parameters of a neural network.

```
@partial(jax.jit, static_argnames=('path_length'))
def compute_lifetime_value(
        params: list,                 # Network parameters
        cake_eating_model: tuple,     # Model parameters (γ, β, R)
        path_length: int              # Length of simulation path
    ):
    """
    Compute the lifetime value of a path generated from
    the policy embedded in params and the initial condition a_0 = 1.
```

(continues on next page)

(continued from previous page)

```
    """
    γ, β, R = cake_eating_model
    initial_a = 1.0

    def update(t, state):
        # Unpack and compute consumption given current assets
        a, value, discount = state
        consumption_rate = forward(params, a)
        c = consumption_rate * a
        # Update loop state and return it
        a = R * (a - c)
        value = value + discount * u(c, γ)
        discount = discount * β
        new_state = a, value, discount
        return new_state

    initial_value, initial_discount = 0.0, 1.0
    initial_state = initial_a, initial_value, initial_discount
    final_a, final_value, discount = jax.lax.fori_loop(
        0, path_length, update, initial_state
    )
    return final_value
```

Here's the loss function we will minimize.

```
def loss_function(params, cake_eating_model, path_length):
    """
    Loss is the negation of the lifetime value of the policy
    identified by `params`.

    """
    return -compute_lifetime_value(params, cake_eating_model, path_length)
```

24.4.2 Train and solve

First we create an instance of the model and unpack names

```
model = CakeEatingModel()
γ, β, R = model.γ, model.β, model.R
```

We test stability.

```
assert β * R**(1 - γ) < 1, "Parameters fail stability test."
```

We compute the optimal consumption rate and lifetime value from the analytical expressions.

```
κ = 1 - (β * R**(1 - γ))**(1/γ)
print(f"Optimal consumption rate = {κ:.4f}.\n")
v_max = κ**(-γ) * u(1.0, γ)
print(f"Theoretical maximum lifetime value = {v_max:.4f}.\n")
```

```
Optimal consumption rate = 0.0301.
Theoretical maximum lifetime value = -383.5558.
```

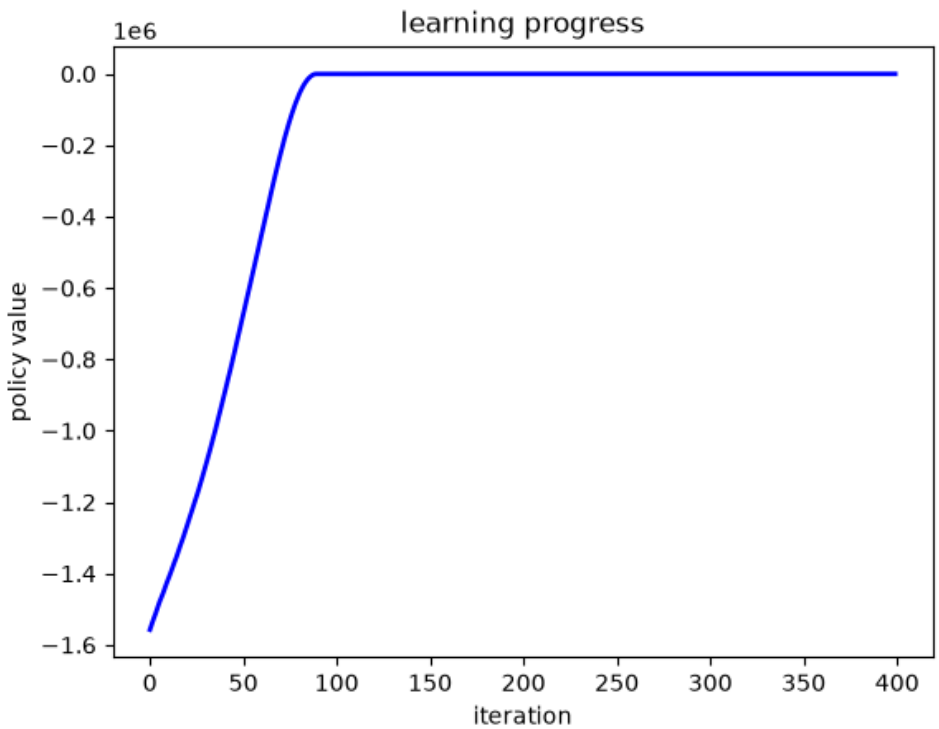
Now let's train the network.

```python
import time

config = Config(num_paths=1)

# Create a loss function that has params as the only argument
loss_fn = lambda params: loss_function(params, model, config.path_length)

# Warmup to trigger JIT compilation
print("Warming up JIT compilation...")
_ = train_network(config, loss_fn)

start_time = time.time()
params, value_history, best_value = train_network(config, loss_fn)
best_value.block_until_ready()
elapsed = time.time() - start_time

print(f"\nBest value: {best_value:.4f}")
print(f"Final value: {value_history[-1]:.4f}")
print(f"Training time: {elapsed:.2f} seconds")
```

```
Warming up JIT compilation...

Best value: -382.5453
Final value: -382.5459
Training time: 2.87 seconds
```

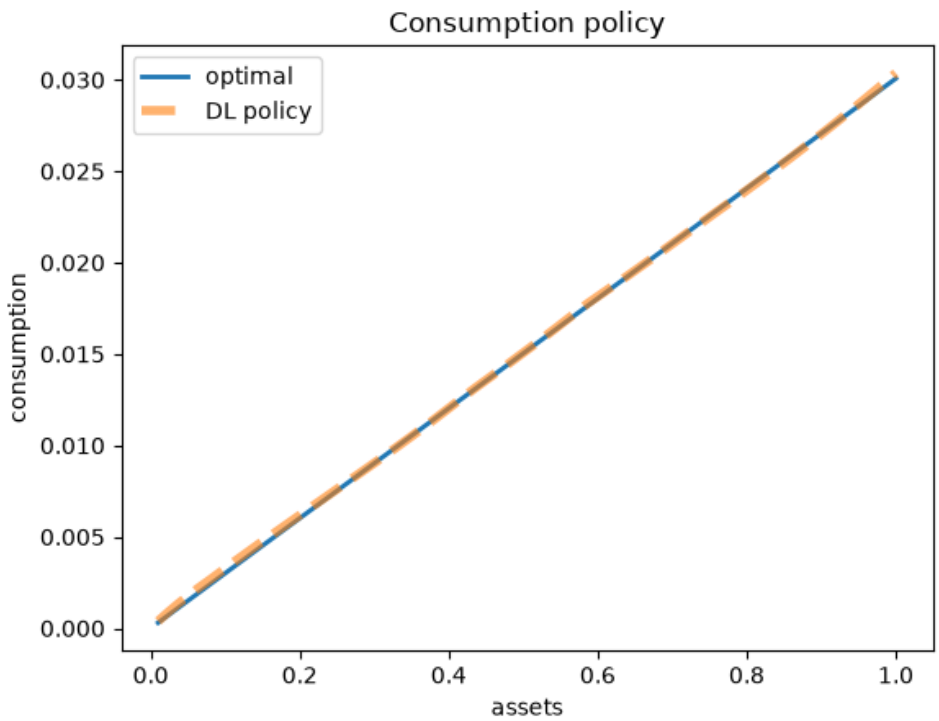
First we plot the evolution of lifetime value over the epochs.

```python
# Plot learning progress
fig, ax = plt.subplots()
ax.plot(value_history, 'b-', linewidth=2)
ax.set_xlabel('iteration')
ax.set_ylabel('policy value')
ax.set_title('learning progress')
plt.show()
```



Next we compare the learned and optimal policies.

```
a_grid = jnp.linspace(0.01, 1.0, 1000)
policy_vmap = jax.vmap(lambda a: forward(params, a))
consumption_rate = policy_vmap(a_grid)
# Compute actual consumption: c = (c/a) * a
c_learned = consumption_rate * a_grid
c_optimal = κ * a_grid

fig, ax = plt.subplots()
ax.plot(a_grid, c_optimal, lw=2, label='optimal')
ax.plot(a_grid, c_learned, linestyle='--', lw=4, alpha=0.6, label='DL policy')
ax.set_xlabel('assets')
ax.set_ylabel('consumption')
ax.set_title('Consumption policy')
ax.legend()
plt.show()
```



24.4.3 Simulation

Let's have a look at paths for consumption and assets under the learned and optimal policies.

The figures below show that the learned policies are close to optimal.

```
def simulate_consumption_path(
        params,    # ANN-based policy identified by params
        a_0,       # Initial condition
        T=120      # Simulation length
    ):
    # Simulate consumption and asset paths using ANN
    a = a_0
    a_sim, c_sim = [a], []
    for t in range(T):
        # Update policy path
        c = forward(params, a) * a
        c_sim.append(float(c))
        a = R * (a - c)
        a_sim.append(float(a))
        if a <= 1e-10:
            break

    # Simulate consumption and asset paths using optimal policy
    a = a_0
    a_opt, c_opt = [a], []
    for t in range(T):
        # Update optimal path
        c = κ * a
        c_opt.append(c)
        a = R * (a - c)
```

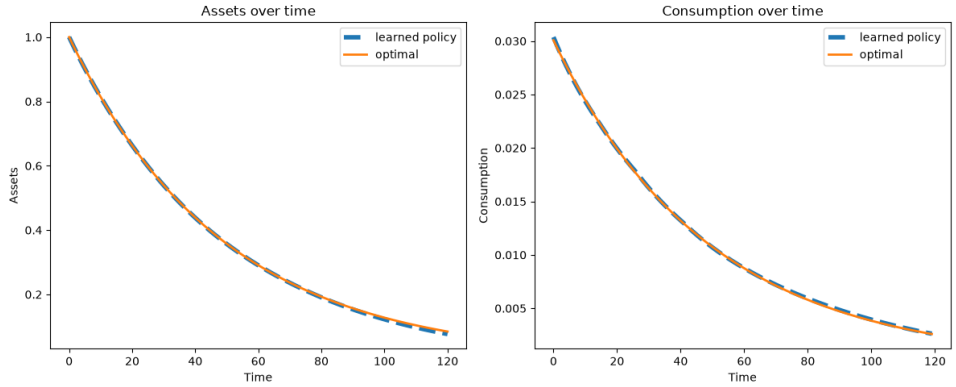
(continues on next page)

(continued from previous page)

```
        a_opt.append(a)
        if a <= 1e-10:
            break

    return a_sim, c_sim, a_opt, c_opt
```

Let's simulate and plot path

```
a_sim, c_sim, a_opt, c_opt = simulate_consumption_path(params, a_0=1.0)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

ax1.plot(a_sim, lw=4, linestyle='--', label='learned policy')
ax1.plot(a_opt, lw=2, label='optimal')
ax1.set_xlabel('Time')
ax1.set_ylabel('Assets')
ax1.set_title('Assets over time')
ax1.legend()

ax2.plot(c_sim, lw=4, linestyle='--', label='learned policy')
ax2.plot(c_opt, lw=2, label='optimal')
ax2.set_xlabel('Time')
ax2.set_ylabel('Consumption')
ax2.set_title('Consumption over time')
ax2.legend()

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



24.5 IFP Model

Now let's solve a model with IID stochastic labor income using deep learning.

The set up was described at the start of this lecture.

24.5.1 JAX Implementation

We start with a class called `IFP` that stores the model primitives.

```
class IFP(NamedTuple):
    R: float                  # Gross interest rate R = 1 + r
    β: float                  # Discount factor
    γ: float                  # Preference parameter
    z_mean: float             # Mean of log income shock
    z_std: float              # Std dev of log income shock
    z_samples: jnp.ndarray    # Std dev of log income shock


def create_ifp(
        r=0.01,
        β=0.96,
        γ=1.5,
        z_mean=0.1,
        z_std=0.1,
        n_shocks=200,
        seed=42
    ):
    R = 1 + r
    assert R * β < 1, "Stability condition violated."
    key = random.key(seed)
    z_samples = z_mean + z_std * jax.random.normal(key, n_shocks)
    return IFP(R, β, γ, z_mean, z_std, z_samples)
```

24.5.2 Solving the IID model using the EGM

Since the shocks are IID, the optimal policy depends only on current assets a.

For the IID normal case, we need to compute the expectation:

$$\mathbb{E}[u'(\sigma(Rs + Y))]$$

where $Z \sim N(m, v)$ and $Y = \exp(Z)$.

We approximate this expectation using Monte Carlo.

Here is the EGM operator K for the IID case:

```
def K(
        c_in: jnp.ndarray,        # Current consumption policy on endogenous grid
        a_in: jnp.ndarray,        # Current endogenous asset grid
        ifp: IFP,                 # IFP model instance
        s_grid: jnp.ndarray,      # Exogenous savings grid
        n_shocks: int = 50        # Number of points for Monte Carlo integration
    ):
```

(continues on next page)

(continued from previous page)

```
    """
    The Euler equation operator for the IFP model with IID shocks using EGM.

    """
    R, β, γ, z_mean, z_std, z_samples = ifp
    y_samples = jnp.exp(z_samples)
    u_prime = lambda c: c**(-γ)
    u_prime_inv = lambda c: c**(-1/γ)

    def compute_c_i(s_i):
        """Compute consumption for savings level s_i."""

        # For each income realization, compute next period assets and consumption
        def compute_mu_k(y_k):
            next_a = R * s_i + y_k
            # Interpolate to get consumption
            next_c = jnp.interp(next_a, a_in, c_in)
            return u_prime(next_c)

        # Compute expectation over income shocks (Monte Carlo average)
        mu_values = jax.vmap(compute_mu_k)(y_samples)
        expectation = jnp.mean(mu_values)

        # Invert to get consumption (handles s_i=0 case via smooth function)
        c = u_prime_inv(β * R * expectation)

        # For s_i = 0, consumption should be 0
        return jnp.where(s_i == 0, 0.0, c)

    # Compute consumption for all savings levels
    c_out = jax.vmap(compute_c_i)(s_grid)
    # Compute endogenous asset grid
    a_out = c_out + s_grid

    return c_out, a_out
```

Here's the solver using time iteration:

```
def solve_model(
        ifp: IFP,                # IFP model instance
        s_grid: jnp.ndarray,     # Exogenous savings grid
        n_shocks: int = 50,      # Number of income shock realizations
        tol: float = 1e-5,       # Convergence tolerance
        max_iter: int = 1000     # Maximum iterations
    ):
    """
    Solve the IID model using time iteration with EGM.

    """
    # Initialize with consumption = assets (consume everything)
    a_init = s_grid.copy()
    c_init = s_grid.copy()
    c_in, a_in = c_init, a_init

    for i in range(max_iter):
        c_out, a_out = K(c_in, a_in, ifp, s_grid, n_shocks)
```

(continues on next page)

(continued from previous page)

```
        error = jnp.max(jnp.abs(c_out - c_in))

        if error < tol:
            print(f"Converged in {i} iterations, error = {error:.2e}")
            break

        c_in, a_in = c_out, a_out

        if i % 100 == 0:
            print(f"Iteration {i}, error = {error:.2e}")

    return c_out, a_out
```

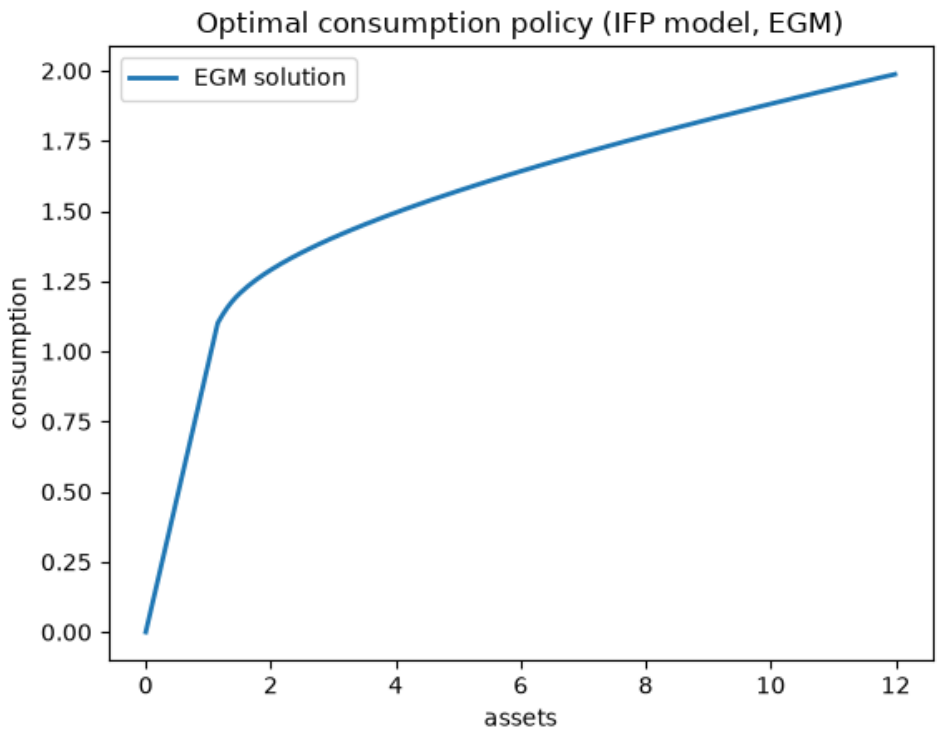
Let's solve the model and plot the optimal policy:

```
# Create model instance
ifp = create_ifp(z_mean=0.1, z_std=0.1)

# Create savings grid
s_grid = jnp.linspace(0, 10, 200)

# Solve using EGM
print("Solving IFP model using EGM...\n")
c_egm, a_egm = solve_model(ifp, s_grid, n_shocks=100)
```

```
Solving IFP model using EGM...
Iteration 0, error = 1.40e+00
Converged in 38 iterations, error = 6.79e-06
```

Plot the optimal consumption policy:

```
fig, ax = plt.subplots()
ax.plot(a_egm, c_egm, lw=2, label='EGM solution')
ax.set_xlabel('assets')
ax.set_ylabel('consumption')
ax.set_title('Optimal consumption policy (IFP model, EGM)')
ax.legend()
plt.show()
```



24.5.3 Solving the IID model with DL

Since the shocks are IID, the policy depends only on current assets a.

We use the same network architecture as the deterministic case.

The forward pass uses the `forward` function from the deterministic case.

Here we implement lifetime value computation.

The key is to simulate paths with IID normal income shocks.

```
@partial(jax.jit, static_argnames=('path_length', 'num_paths'))
def compute_lifetime_value_ifp(
        params: list,            # Neural network parameters
        ifp: IFP,                # IFP model instance
        path_length: int,        # Length of each simulated path
        num_paths: int,          # Number of paths to simulate for averaging
        key: jax.Array           # JAX random key for generating income shocks
    ):
    """
    Compute expected lifetime value by averaging over multiple
    simulated paths.

    """
    R, β, γ, z_mean, z_std, z_samples = ifp

    def simulate_path(subkey):
        """Simulate a single path and return its lifetime value."""
        z_shocks = z_mean + z_std * jax.random.normal(subkey, path_length)
        Y = jnp.exp(z_shocks)
```

(continues on next page)

(continued from previous page)

```
        def update(t, loop_state):
            a, value, discount = loop_state
            consumption_rate = forward(params, a)
            c = consumption_rate * a
            next_value = value + discount * u(c, γ)
            next_a = R * (a - c) + Y[t]
            next_discount = discount * β
            return next_a, next_value, next_discount

        initial_a = 10.0
        initial_value = 0.0
        initial_discount = 1.0
        initial_state = (initial_a, initial_value, initial_discount)
        final_a, final_value, final_discount = jax.lax.fori_loop(
            0, path_length, update, initial_state
        )

        return final_value

    # Generate keys for all paths
    path_keys = jax.random.split(key, num_paths)

    # Simulate all paths and average
    values = jax.vmap(simulate_path)(path_keys)
    return jnp.mean(values)
```

The loss function is the negation of the expected lifetime value.

```
def loss_function_ifp(params, ifp, path_length, num_paths, key):
    return -compute_lifetime_value_ifp(
        params, ifp, path_length, num_paths, key
    )
```

Now let's set up and train the network.

We use the same `ifp` instance that was created for the EGM solution above.

```
config = Config()
key = random.key(config.seed)

print("Training IFP model with deep learning...\n")

# Set up loss function to pass to train_network
ifp_loss_fn = lambda params: loss_function_ifp(
    params, ifp, config.path_length, config.num_paths, key
)

# Warmup to trigger JIT compilation
print("Warming up JIT compilation...")
_ = train_network(config, ifp_loss_fn)

start_time = time.time()
ifp_params, ifp_value_history, best_ifp_value = train_network(
    config, ifp_loss_fn
)
best_ifp_value.block_until_ready()
```

(continues on next page)

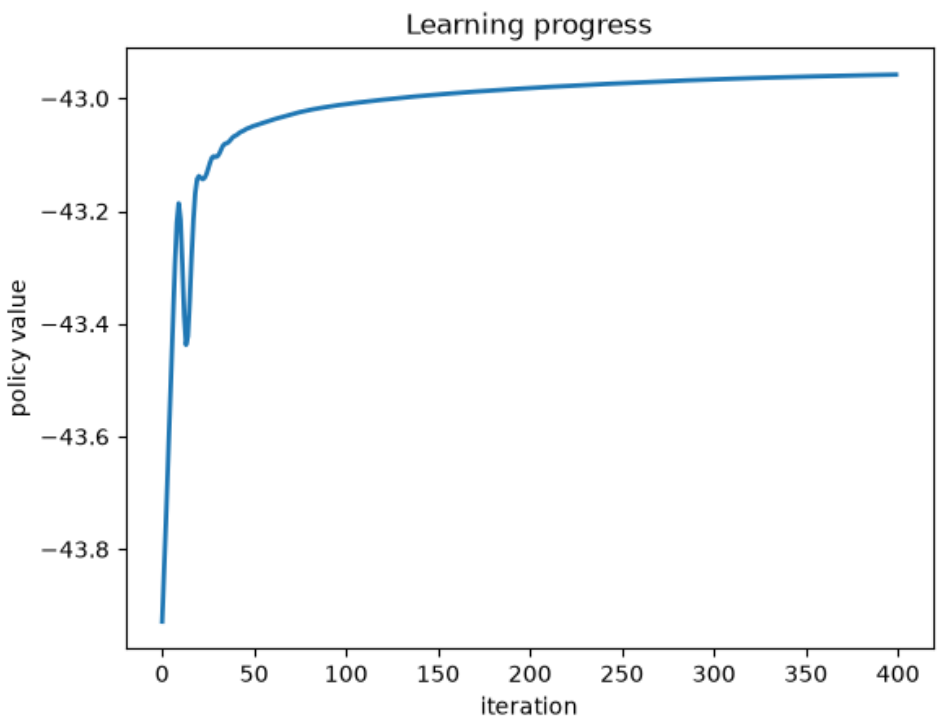
(continued from previous page)

```
elapsed = time.time() - start_time

print(f"\nBest value: {best_ifp_value:.4f}")
print(f"Final value: {ifp_value_history[-1]:.4f}")
print(f"Training time: {elapsed:.2f} seconds")
```

```
Training IFP model with deep learning...

Warming up JIT compilation...

Best value: -42.9578
Final value: -42.9578
Training time: 6.18 seconds
```

Plot the learning progress.

```
fig, ax = plt.subplots()
ax.plot(ifp_value_history, linewidth=2)
ax.set_xlabel('iteration')
ax.set_ylabel('policy value')
ax.set_title('Learning progress')
plt.show()
```



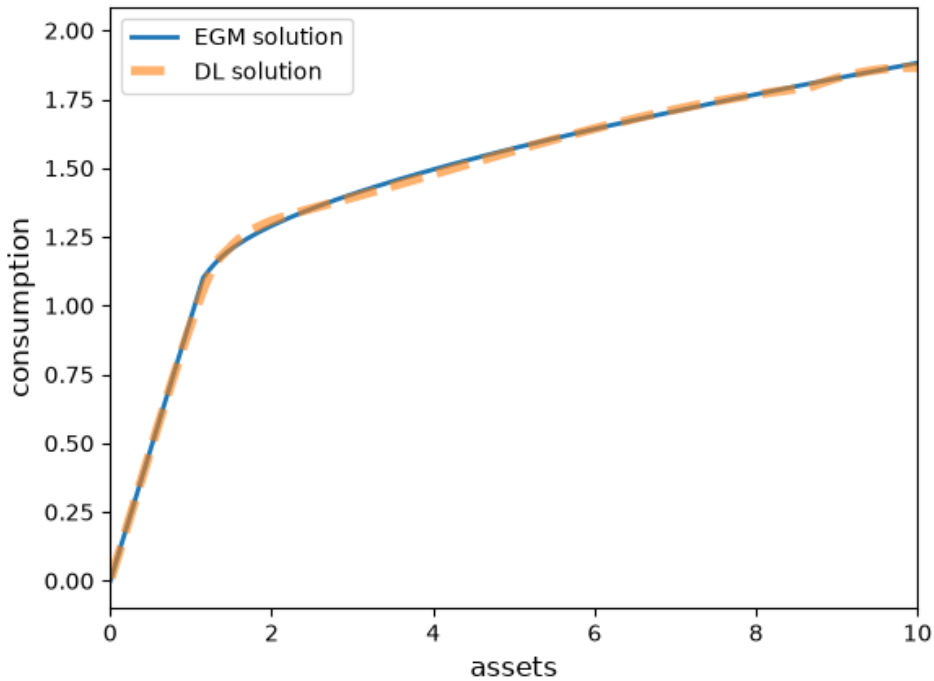
Compare EGM and DL solutions.

```
# Evaluate DL policy on asset grid
a_grid_dl = jnp.linspace(0.01, 10.0, 200)
policy_vmap = jax.vmap(lambda a: forward(ifp_params, a))
consumption_rate_dl = policy_vmap(a_grid_dl)
c_dl = consumption_rate_dl * a_grid_dl
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.plot(a_egm, c_egm, lw=2, label='EGM solution')
ax.plot(a_grid_dl, c_dl, linestyle='--', lw=4, alpha=0.6, label='DL solution')
ax.set_xlabel('assets', fontsize=12)
ax.set_ylabel('consumption', fontsize=12)
ax.set_xlim(0, min(a_grid_dl[-1], a_egm[-1]))
ax.legend()
plt.show()
```



The fit is quite good.

Part VII

Racial Segregation

CHAPTER

TWENTYFIVE

SCHELLING'S MODEL OF RACIAL SEGREGATION

25.1 Outline

Racial residential segregation has significant consequences across many dimensions of life in the US and other countries.

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation (Schelling, 1969).

His model studies the dynamics of racially mixed neighborhoods.

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

It studies a setting where agents (think of households) have relatively mild preference for neighbors of the same race.

For example, these agents might be comfortable with a mixed race neighborhood but uncomfortable when they feel "surrounded" by people from a different race.

Schelling illustrated the following surprising result: in such a setting, mixed race neighborhoods are likely to be unstable, tending to collapse over time.

In fact the model predicts strongly divided neighborhoods, with high levels of segregation.

In other words, extreme segregation outcomes arise even though people's preferences are not particularly extreme.

These extreme outcomes happen because of *interactions* between agents in the model (e.g., households in a city) that drive self-reinforcing dynamics in the model.

In recognition of his work on segregation and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences.

In this and the following lectures, we study Schelling's model using simulation.

We will start in this lecture with a relatively elementary version written in pure Python.

Once that simple version is in place, we will consider efficiency.

We will develop progressively more efficient versions of the model, allowing us to study dynamics with larger populations and more interesting features.

In the most efficient versions, JAX and parallelization will play a central role.

Let's start with some imports:

```
import matplotlib.pyplot as plt
from random import uniform, seed
from math import sqrt
import time
```

25.2 Background

Before jumping into the modelling process, let's look at some data

The maps below are from the Weldon Cooper Center for Public Service at the University of Virginia.

They illustrate the racial composition of several major US cities, based on census data.

Each dot represents a group of residents — we omit details on which color represents which group.

These maps reveal patterns of spatial separation between racial groups.

25.2.1 Columbus, Ohio

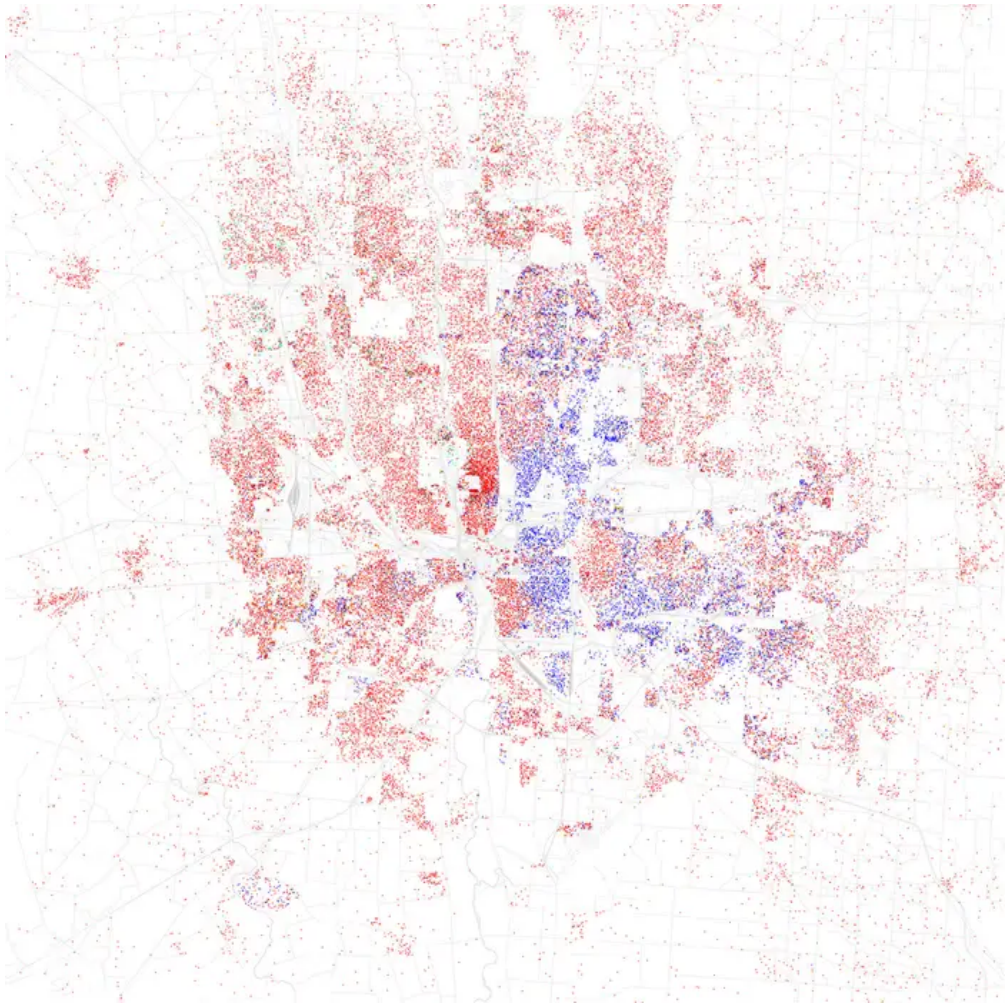


Fig. 25.1: Racial distribution in Columbus, Ohio.

25.2.2 Memphis, Tennessee

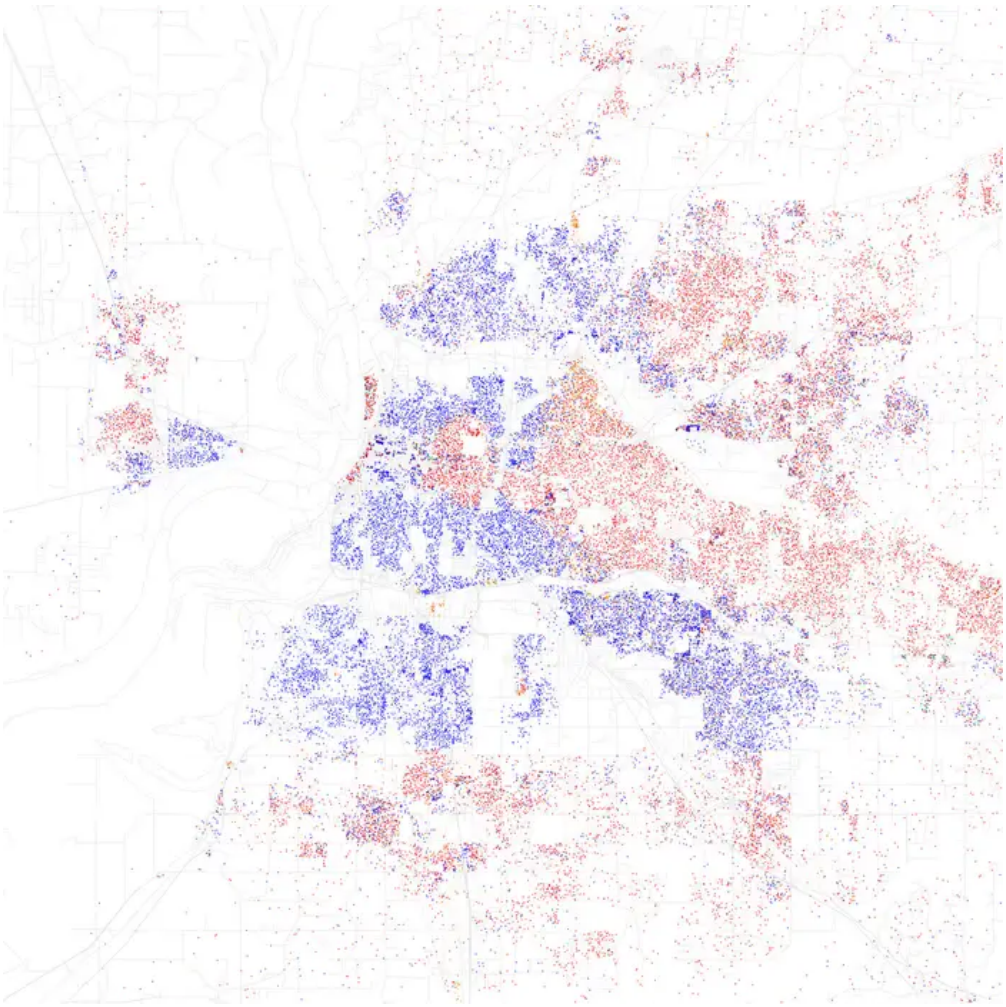


Fig. 25.2: Racial distribution in Memphis, Tennessee.

25.2.3 Washington, D.C.

25.2.4 Houston, Texas

25.2.5 Miami, Florida

Looking at these maps, one might assume that segregation persists because of strong preferences—that people simply want to live only with others of their own race.

But is this actually the case?

Let's now look at Schelling's segregation model, which demonstrates a surprising result: extreme segregation can emerge even when individuals have only mild preferences for same-race neighbors.

This insight has profound implications for understanding how segregation persists and what policies might effectively address it.

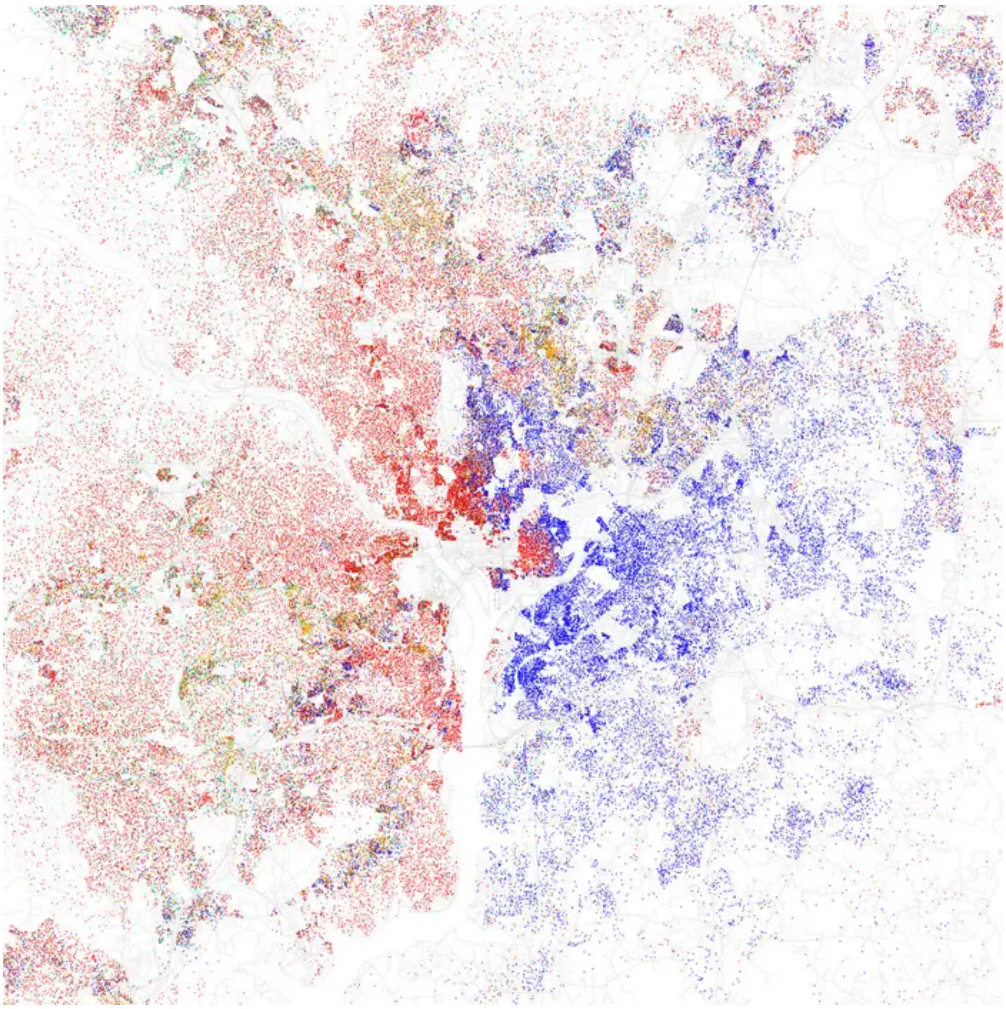


Fig. 25.3: Racial distribution in Washington, D.C.

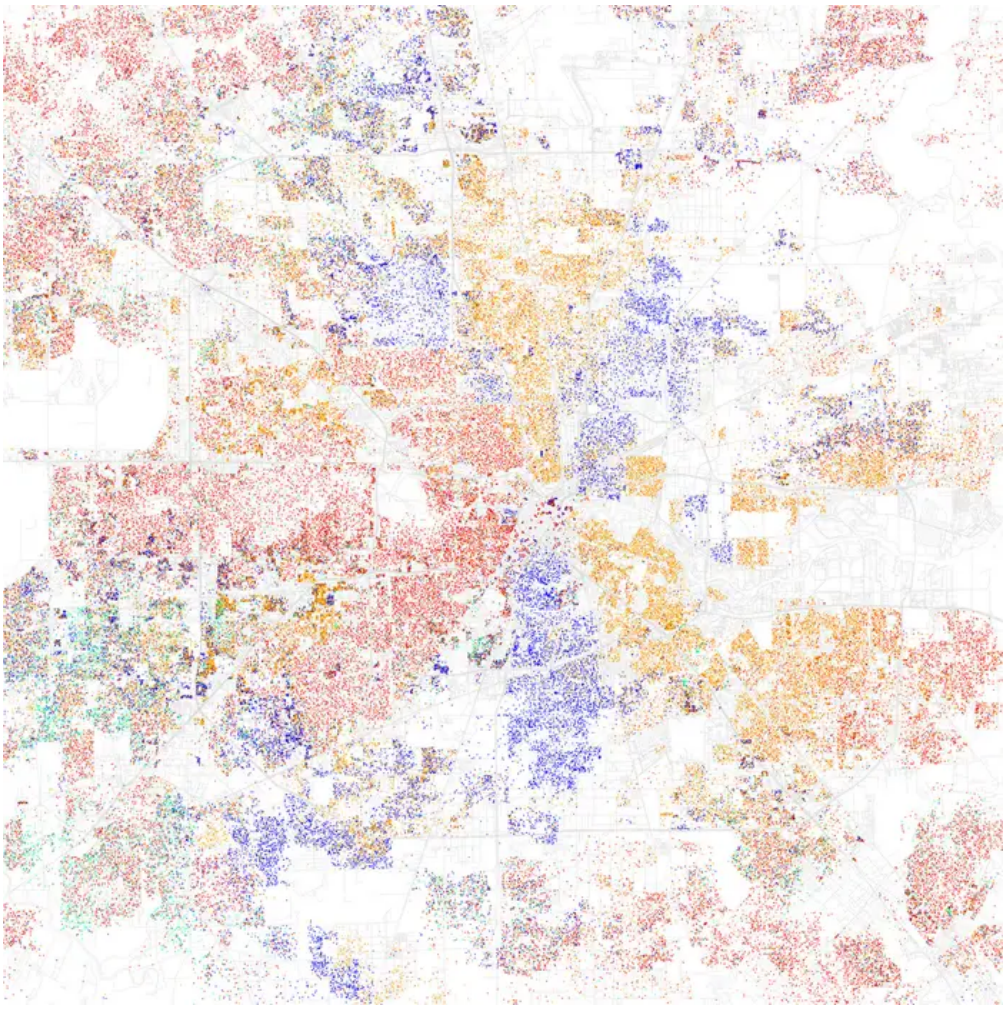


Fig. 25.4: Racial distribution in Houston, Texas.

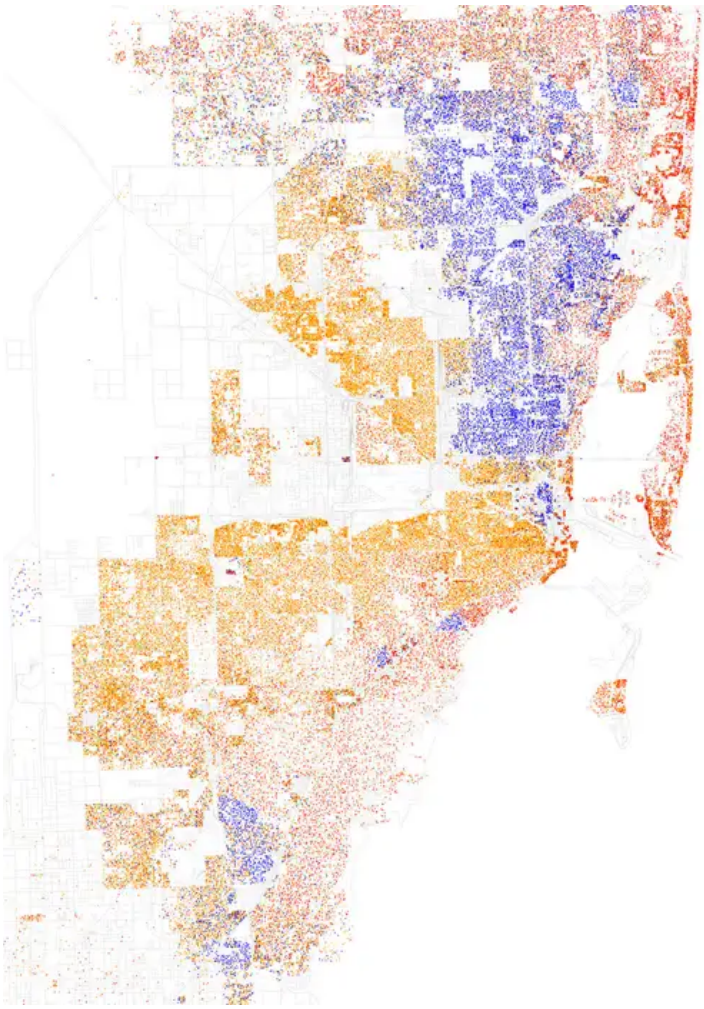


Fig. 25.5: Racial distribution in Miami, Florida.

25.3 The model

25.3.1 Set-Up

We will cover a variation of Schelling's model that is different from the original but also easy to program and, at the same time, captures his main idea.

For now our coding objective is clarity rather than efficiency.

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

Assume there are n of each type.

These agents all live on a single unit square.

Thus, the location (e.g, address) of an agent is just a point (x, y), where $0 < x, y < 1$.

- The set of all points (x, y) satisfying $0 < x, y < 1$ is called the **unit square**
- Below we denote the unit square by S

25.3.2 Preferences

We will say that an agent is

- **happy** if $k \leq 6$ of her 10 nearest neighbors are of a different type.
- **unhappy** if $k > 6$ of her 10 nearest neighbors are of a different type.

For example,

- if an agent is orange and 6 of her 10 nearest neighbors are green, then she is happy.
- if an agent is orange and 7 of her 10 nearest neighbors are green, then she is unhappy.

'Nearest' is in terms of Euclidean distance.

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

They are perfectly happy if 60% of their neighbors are of the other color.

Let's set up the parameters for our simulation:

```
seed(1234)              # set seed for reproducibility

num_of_type_0 = 1000    # number of agents of type 0 (orange)
num_of_type_1 = 1000    # number of agents of type 1 (green)
num_neighbors = 10      # number of agents viewed as neighbors
max_other_type = 6      # max number of different-type neighbors tolerated
```

25.3.3 Behavior

Initially, agents are mixed together (integrated).

In particular, we assume that the initial location of each agent is an independent draw from a bivariate uniform distribution on the unit square S.

- Their x coordinate is drawn from a uniform distribution on $(0, 1)$
- Their y coordinate is drawn independently from the same distribution.

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

Each agent stays if they are happy and moves if they are unhappy.

The algorithm for moving is as follows

Algorithm 25.3.1 (Relocation Algorithm)

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

We cycle continuously through the agents, each time allowing an unhappy agent to move.

We continue to cycle until no one wishes to move.

25.4 Main Loop

Let's write the code to run this simulation.

In what follows, agents are modeled as objects that store

```
    * type (green or orange)
    * location
```

Here's a class that we can use to instantiate agents from:

```
class Agent:

    def __init__(self, type):
        self.type = type
        self.location = uniform(0, 1), uniform(0, 1)
```

Here's a collection of functions that act on agents:

```
def get_distance(agent, other_agent):
    "Computes the Euclidean distance between self and other agent."
    a = agent.location[0] - other_agent.location[0]
    b = agent.location[1] - other_agent.location[1]
    return sqrt(a**2 + b**2)


def is_happy(agent, all_agents):
    """
```

(continues on next page)

(continued from previous page)

```
    True if agent has at most max_other_type different-type agents
    among its num_neighbors nearest neighbors.
    """
    # Collect all other agents, sorted by distance to agent
    others = [a for a in all_agents if a != agent]
    others.sort(key=lambda a: get_distance(agent, a))

    # Check the nearest neighbors
    neighbors = others[:num_neighbors]
    num_other_type = sum(neighbor.type != agent.type for neighbor in neighbors)
    return num_other_type <= max_other_type


def move_agent(agent, all_agents, max_attempts=10_000):
    "If not happy, then randomly choose new locations until happy."
    attempts = 0
    while not is_happy(agent, all_agents):
        agent.location = uniform(0, 1), uniform(0, 1)
        attempts += 1
        if attempts >= max_attempts:
            break
```

Here's some code that takes a list of agents and produces a plot showing their locations on the unit square.

Orange agents are represented by orange dots and green ones are represented by green dots.

The main loop cycles through all agents until no one wishes to move.

Algorithm 25.4.1 (Main Simulation Loop)

Input: Set of agents with initial random locations

Output: Final distribution of agents

1. Set `count` ← 1
2. While `count` < `max_iter`:
    1. Set `number_of_moves` ← 0
    2. For each agent:
        1. If agent is unhappy, relocate using *Algorithm 25.3.1* and increment `number_of_moves`
    3. Plot distribution
    4. Increment `count`
    5. If `number_of_moves` = 0, exit loop

The code is below

```
def run_simulation(all_agents, max_iter=100_000):

    # Initialize a counter
    count = 1

    # Loop until no agent wishes to move
    start_time = time.time()
```

(continues on next page)

(continued from previous page)

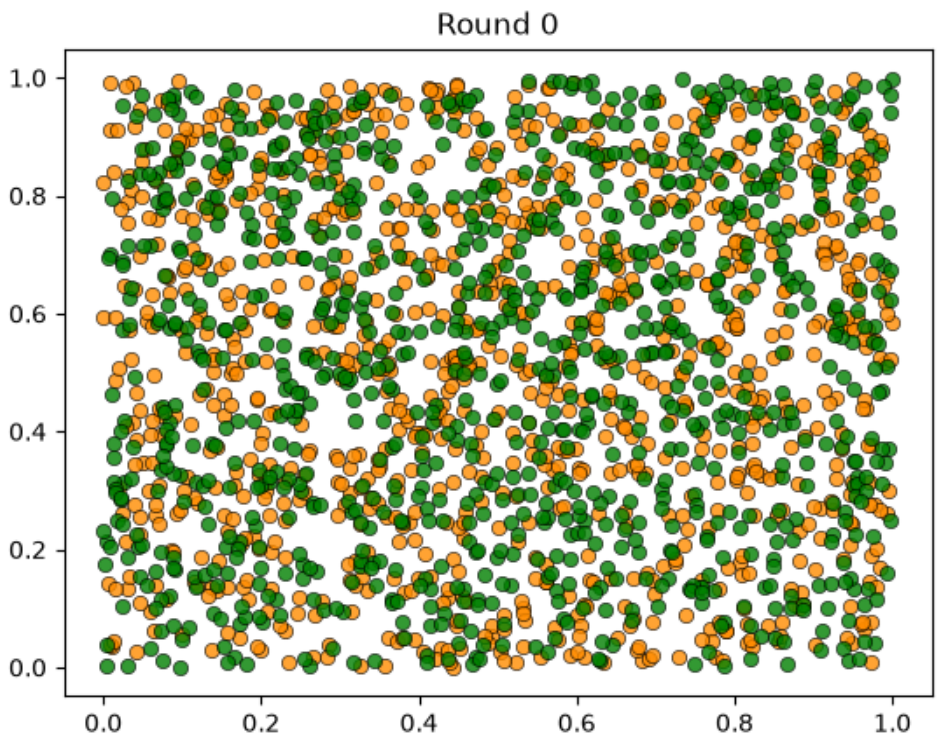
```
    while count < max_iter:
        number_of_moves = 0
        for agent in all_agents:
            if not is_happy(agent, all_agents):
                move_agent(agent, all_agents)
                number_of_moves += 1
        # Plot the distribution after this round
        plot_distribution(all_agents, count)
        # Print outcome and stop loop if no one moved
        print(f'Completed loop {count} with {number_of_moves} moves')
        count += 1
        if number_of_moves == 0:
            break
    elapsed = time.time() - start_time

    if count < max_iter:
        print(f'Converged in {elapsed:.2f} seconds after {count} iterations.')
    else:
        print('Hit iteration bound and terminated.')
```

25.5 Results

We are now ready to run our simulation.

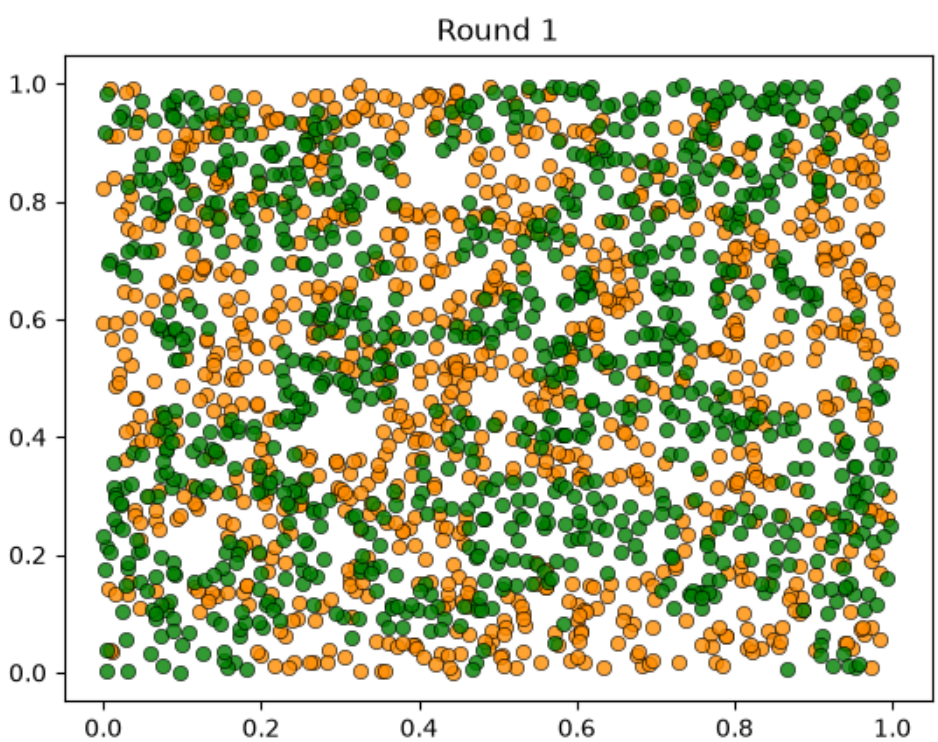
First we build a population of agents:

```
all_agents = []
for i in range(num_of_type_0):
    all_agents.append(Agent(0))
for i in range(num_of_type_1):
    all_agents.append(Agent(1))

plot_distribution(all_agents, 0)
```

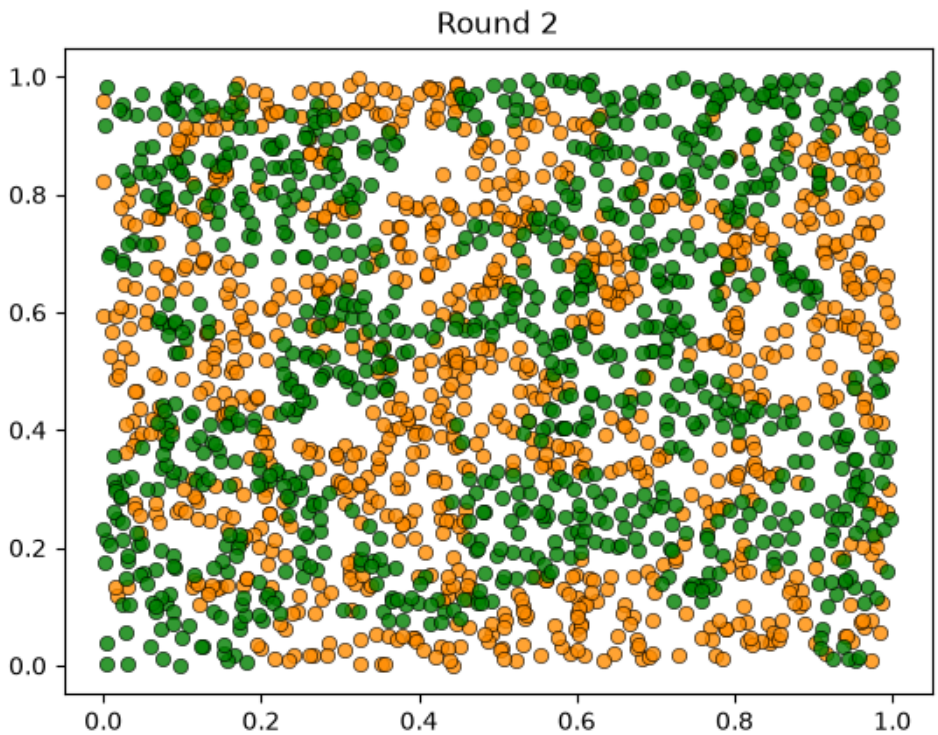


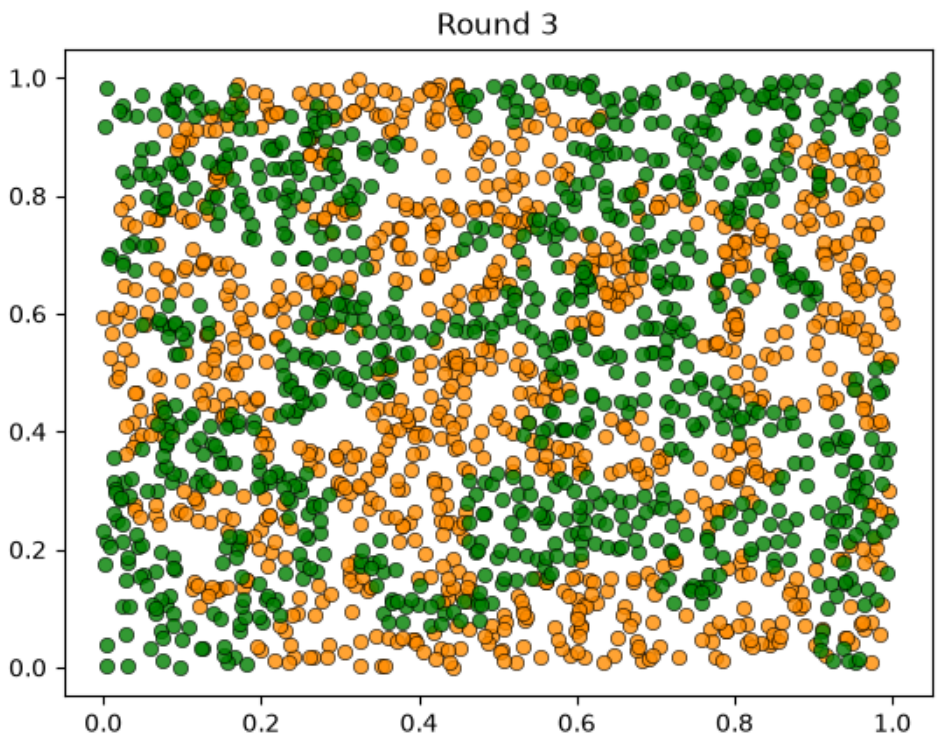
Now we run the simulation and look at the results.

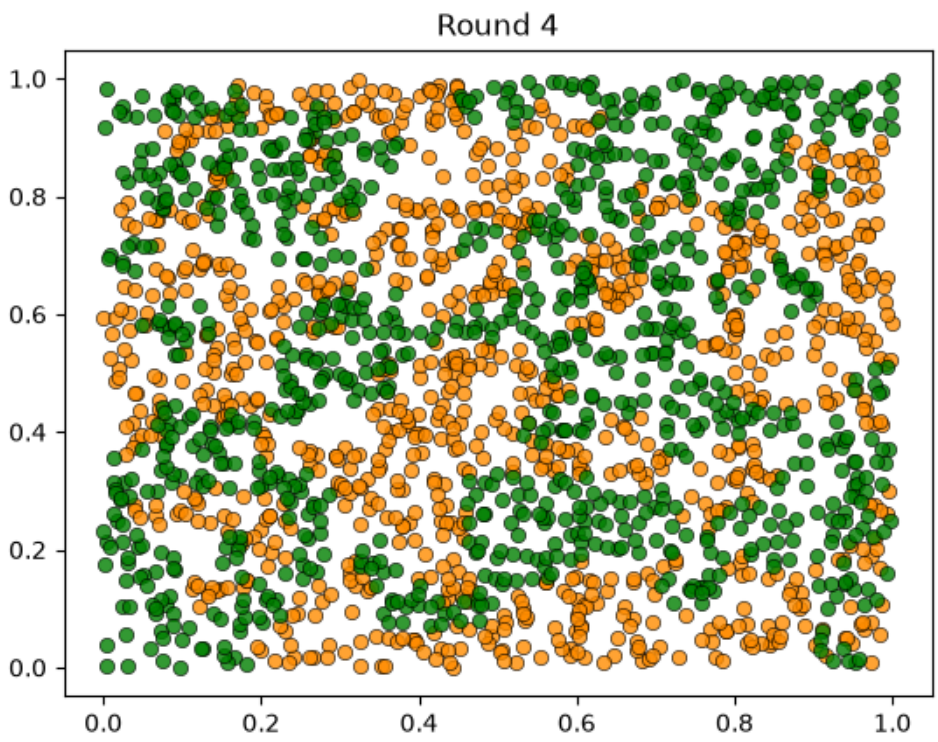
```
run_simulation(all_agents)
```

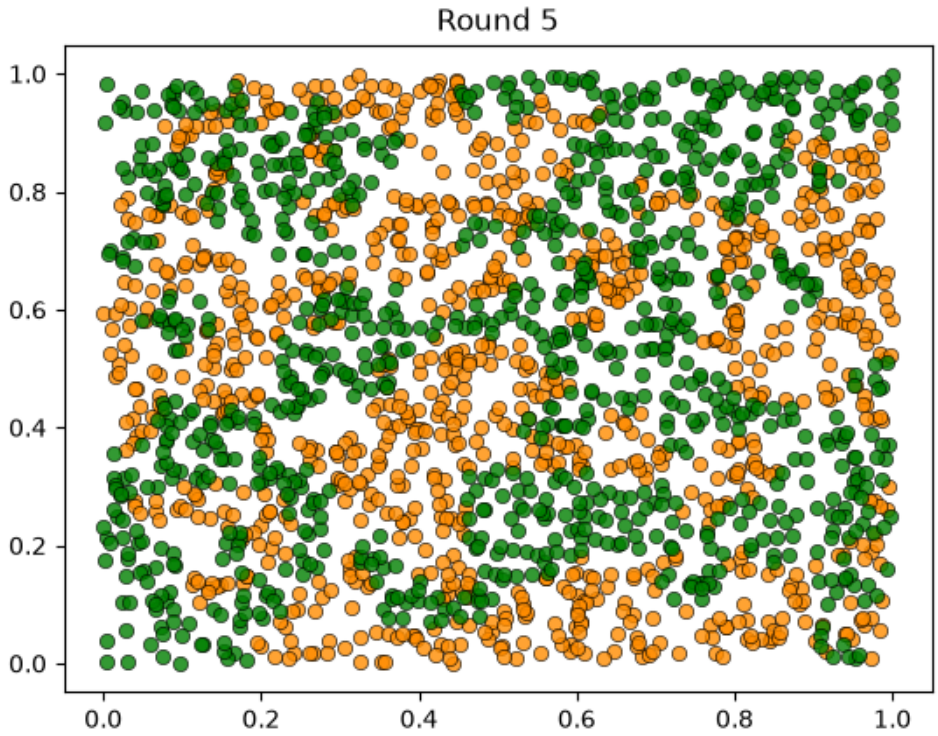


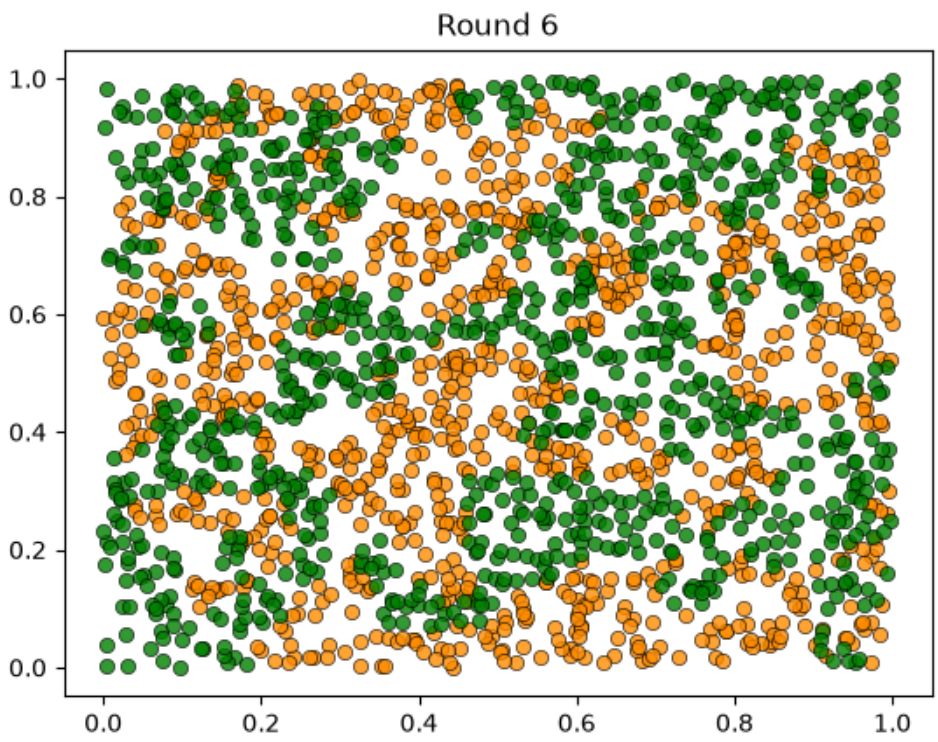
```
Completed loop 1 with 404 moves
Completed loop 2 with 193 moves
Completed loop 3 with 66 moves
Completed loop 4 with 9 moves
Completed loop 5 with 4 moves
Completed loop 6 with 1 moves
Completed loop 7 with 0 moves
Converged in 15.92 seconds after 8 iterations.
```

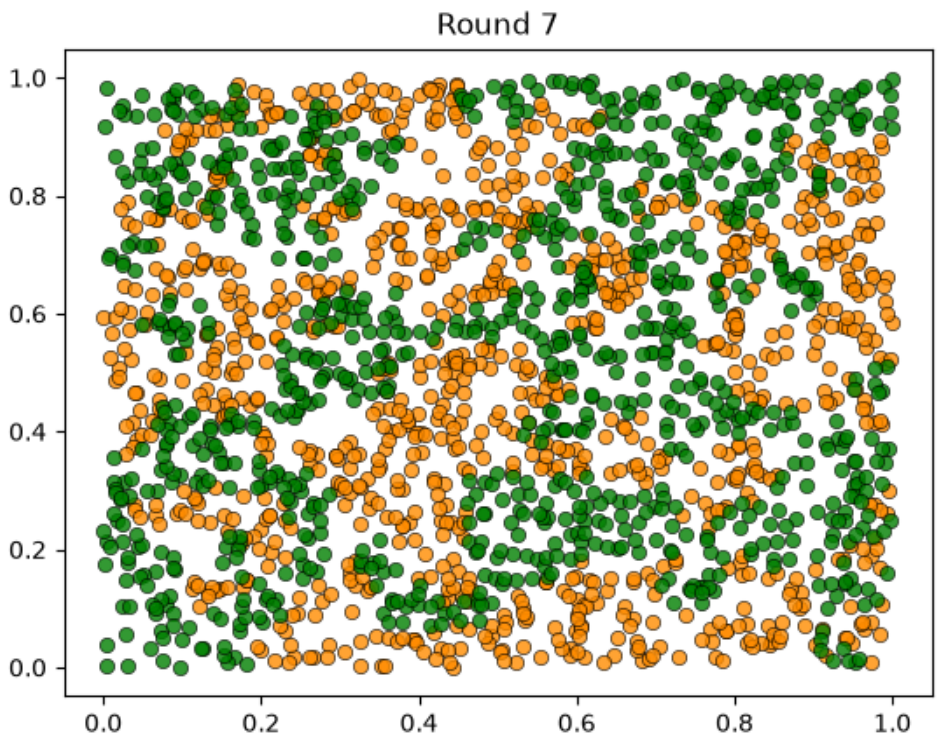



Round 3
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Round 4
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Round 5
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Round 6
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0



As discussed above, agents are initially mixed randomly together.

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

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

We notice that the fully mixed environment rapidly breaks down.

We get emergence of at least some segregation.

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

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

(Not a lot of segregation, but we'll see more segregation in later lectures, after some modifications.)

25.6 Performance

Our Python code was written for readability, not speed.

This is fine for very small simulations but not for bigger ones.

That's a problem for us because we want to experiment with some more ideas.

In the following lectures we'll look at strategies for making our code faster.

Then we'll investigate variations that might lead to even more segregation.

CHAPTER

TWENTYSIX

SCHELLING MODEL WITH NUMPY

26.1 Overview

In the *previous lecture*, we implemented the Schelling segregation model using pure Python and standard libraries.

In this lecture, we will rewrite the model using NumPy arrays and functions.

This is intended as a first step towards greater efficiency.

In later lectures, we'll improve execution speed further by adopting JAX and modifying algorithms.

We'll achieve greater speed — but at the cost of readability!

```
import matplotlib.pyplot as plt
import numpy as np
import time
```

26.2 Data Representation

In the class-based version, each agent was an object storing its own type and location.

Here we take a different approach: we store all agent data in NumPy arrays.

- `locations` — an $n \times 2$ array where row i holds the (x, y) coordinates of agent i
- `types` — an array of length n where entry i is 0 or 1, indicating agent i's type

Let's set up the parameters:

```
num_of_type_0 = 1000     # number of agents of type 0 (orange)
num_of_type_1 = 1000     # number of agents of type 1 (green)
n = num_of_type_0 + num_of_type_1  # total number of agents
num_neighbors = 10       # number of agents viewed as neighbors
max_other_type = 6       # max number of different-type neighbors tolerated
```

Here's a function to initialize the state with random locations and types:

```
def initialize_state(rng):
    locations = rng.random(size=(n, 2))
    types = np.zeros(n, dtype=int)
    types[num_of_type_0:] = 1
    return locations, types
```

Let's see what this looks like:

```
rng = np.random.default_rng(1234)
locations, types = initialize_state(rng)

print(f"locations shape: {locations.shape}")
print(f"First 5 locations:\n{locations[:5]}")
print(f"\ntypes shape: {types.shape}")
print(f"First 20 types: {types[:20]}")
```

```
locations shape: (2000, 2)
First 5 locations:
[[0.97669977 0.38019574]
 [0.92324623 0.26169242]
 [0.31909706 0.11809123]
 [0.24176629 0.31853393]
 [0.96407925 0.2636498 ]]

types shape: (2000,)
First 20 types: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

26.3 Helper Functions

Let's write some functions that compute what we need while operating on the arrays.

26.3.1 Checking Happiness

An agent is happy if at most `max_other_type` of their nearest neighbors are of a different type:

```
def is_happy(i, locations, types):
    " True if agent i has at most max_other_type neighbors of a different type. "
    # Compute distance from agent i to every other agent
    distances = np.linalg.norm(locations[i] - locations, axis=1)
    distances[i] = np.inf                              # exclude self
    neighbors = np.argsort(distances)[:num_neighbors]  # indices of nearest
    num_other = np.sum(types[neighbors] != types[i])
    return num_other <= max_other_type
```

26.3.2 Moving Unhappy Agents

When an agent is unhappy, they keep trying new random locations until they find one where they're happy:

```
def move_agent(i, locations, types, rng, max_attempts=10_000):
    " Move agent i to a new location where they are happy. "
    attempts = 0
    while not is_happy(i, locations, types) and attempts < max_attempts:
        locations[i, :] = rng.random(size=2)
        attempts += 1
```

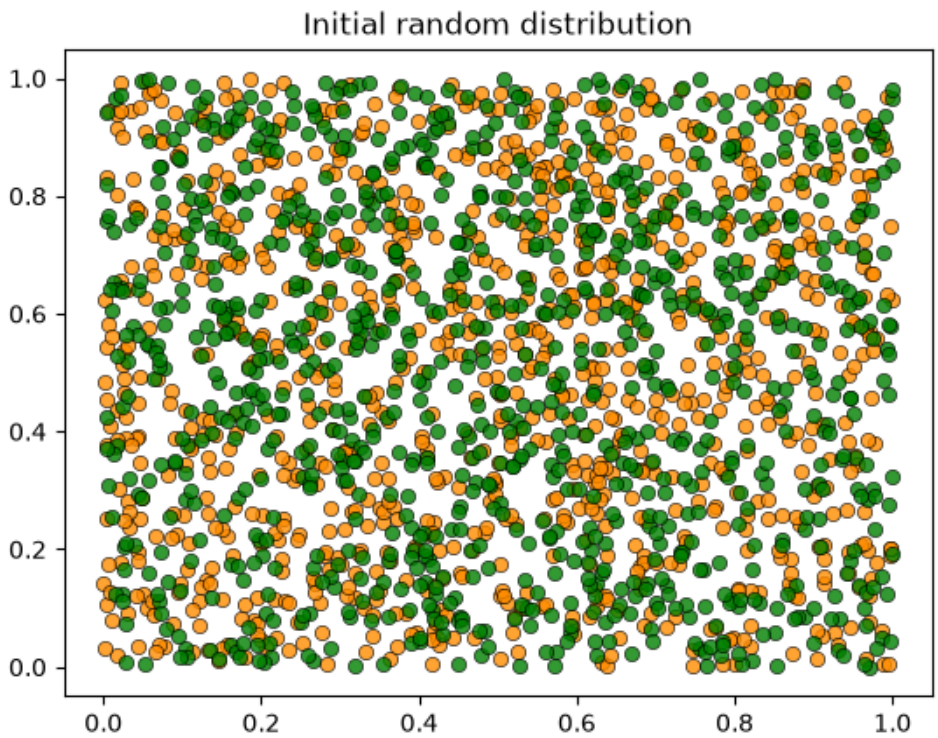
Note that `locations[i, :] = ...` modifies the array in place — the change is visible to all code that references `locations`.

26.4 Visualization

Here's some code for visualization — we'll skip the details

Let's visualize the initial random distribution:

```
rng = np.random.default_rng(1234)
locations, types = initialize_state(rng)
plot_distribution(locations, types, 'Initial random distribution')
```



26.5 The Simulation

Now we put it all together.

As in the first lecture, each iteration cycles through all agents in order, giving each the opportunity to move:

```
def run_simulation(max_iter=100_000, seed=42):
    """
    Run the Schelling simulation.

    Each iteration cycles through all agents, giving each a chance to move.
    """
    rng = np.random.default_rng(seed)
    locations, types = initialize_state(rng)

    plot_distribution(locations, types, 'Initial distribution')

    # Loop until no agent wishes to move
    start_time = time.time()
```

(continues on next page)

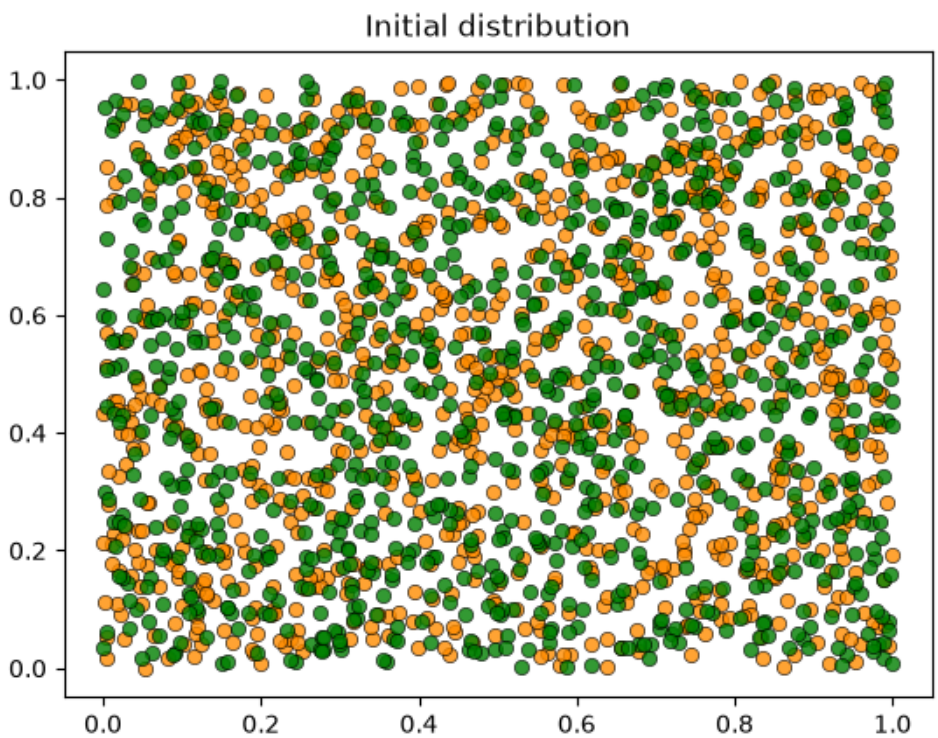
(continued from previous page)

```
    converged = False
    for iteration in range(1, max_iter + 1):
        print(f'Entering iteration {iteration}')
        someone_moved = False
        for i in range(n):
            if not is_happy(i, locations, types):
                move_agent(i, locations, types, rng)
                someone_moved = True
        if not someone_moved:
            converged = True
            break
    elapsed = time.time() - start_time

    plot_distribution(locations, types, f'Iteration {iteration}')

    if converged:
        print(f'Converged in {elapsed:.2f} seconds after {iteration} iterations.')
    else:
        print('Hit iteration bound and terminated.')

    return locations, types
```

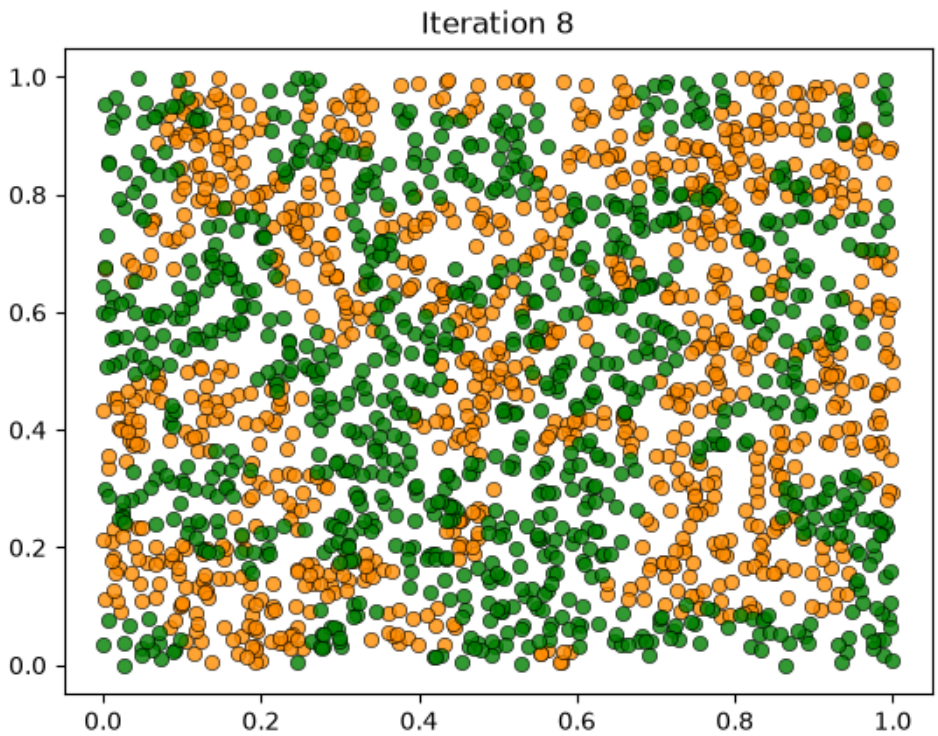
26.6 Results

Let's run the simulation:

```
locations, types = run_simulation()
```



```
Entering iteration 1
Entering iteration 2
Entering iteration 3
Entering iteration 4
Entering iteration 5
Entering iteration 6
Entering iteration 7
Entering iteration 8
Converged in 2.16 seconds after 8 iterations.
```



We see the same phenomenon as in the class-based version: starting from a random mixed distribution, agents self-organize into segregated clusters.

26.7 Performance

The NumPy version is faster than the pure Python version, but still slow for large populations.

In the *next lecture*, we'll rewrite the model using JAX, which offers just-in-time compilation, GPU acceleration, and faster nearest neighbor computations.

CHAPTER

TWENTYSEVEN

SCHELLING MODEL WITH JAX

27.1 Overview

In the *previous lecture*, we rewrote our Schelling model using NumPy arrays and functions.

In this lecture, we explore JAX, a library developed by Google for high-performance numerical computing.

JAX offers several powerful features, including automatic GPU/TPU acceleration and just-in-time compilation.

JAX is heavily used for AI workflows but we repurpose it to work with our simulation.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from jax import random, jit
from functools import partial
from typing import NamedTuple
import time
```

27.2 Setup

We use the same parameters as before. To keep our functions pure, we pack all parameters into a `NamedTuple` that gets passed to functions that need them:

```
class Params(NamedTuple):
    num_of_type_0: int = 1000     # number of agents of type 0 (orange)
    num_of_type_1: int = 1000     # number of agents of type 1 (green)
    num_neighbors: int = 10       # number of agents regarded as neighbors
    max_other_type: int = 6      # max number of different-type neighbors tolerated


params = Params()
```

Here's our initialization function. Note that we use `jax.random` instead of `numpy.random` — we pass a `key` argument to `random.uniform`, making random generation deterministic and reproducible:

```
def initialize_state(key, params):
    """
    Initialize agent locations and types.
```

(continues on next page)

(continued from previous page)

```
    """
    num_of_type_0, num_of_type_1 = params.num_of_type_0, params.num_of_type_1
    n = num_of_type_0 + num_of_type_1
    locations = random.uniform(key, (n, 2))
    types = jnp.concatenate([jnp.zeros(num_of_type_0, dtype=int),
                             jnp.ones(num_of_type_1, dtype=int)])
    return locations, types
```

27.3 JAX-Compiled Functions

Now let's rewrite our core functions for JAX.

We use `jit` to compile functions for faster execution.

```
@partial(jit, static_argnames=('params',))
def is_happy(loc, agent_idx, locations, types, params):
    " True if an agent at loc has at most max_other_type different-type neighbors. "
    # Squared distances from loc to every agent
    distances = jnp.sum((loc - locations)**2, axis=1)
    distances = distances.at[agent_idx].set(jnp.inf)  # exclude self
    # top_k finds the k smallest distances in O(n) (we negate to use top_k)
    _, neighbors = jax.lax.top_k(-distances, params.num_neighbors)
    num_other = jnp.sum(types[neighbors] != types[agent_idx])
    return num_other <= params.max_other_type
```

Compared to the NumPy version, there are a few differences worth noting.

We use `jax.lax.top_k` instead of `argsort` to find nearest neighbors — this is $O(n)$ rather than $O(n \log n)$.

We use `.at[].set()` rather than direct indexing to exclude the agent from its own neighbor set, since JAX arrays are immutable.

The function takes `loc` as an explicit argument rather than looking it up from the arrays, so we can test hypothetical locations without modifying the `locations` array.

The next function finds a location where a given agent would be happy.

Rather than updating the `locations` array on each iteration, it tests candidate locations directly and returns only the final location.

```
@partial(jit, static_argnames=('params',))
def move_agent(i, locations, types, key, params, max_attempts=10_000):
    """
    Find a location where agent i is happy.

    Returns the new location and updated random key. The calling code
    is responsible for updating the locations array if the agent moved.
    """
    loc = locations[i, :]

    # Continue while under max_attempts and not yet happy
    def while_test(state):
        loc, key, attempts = state
        return (attempts < max_attempts) & ~is_happy(loc, i, locations, types, params)
```

(continues on next page)

(continued from previous page)

```
    # Draw a new random location
    def update(state):
        _, key, attempts = state
        key, subkey = random.split(key)
        new_loc = random.uniform(subkey, 2)
        return new_loc, key, attempts + 1

    final_loc, key, _ = jax.lax.while_loop(while_test, update, (loc, key, 0))
    return final_loc, key
```

Here `jax.lax.while_loop` calls `update` repeatedly until `while_test` returns False.

27.4 The Simulation

```
def simulation_loop(locations, types, key, params, max_iter):
    """
    Run the simulation loop until convergence or max iterations.
    """
    n = params.num_of_type_0 + params.num_of_type_1
    converged = False
    for iteration in range(1, max_iter + 1):
        print(f'Entering iteration {iteration}')
        someone_moved = False
        for i in range(n):
            if not is_happy(locations[i], i, locations, types, params):
                new_loc, key = move_agent(i, locations, types, key, params)
                locations = locations.at[i, :].set(new_loc)
                someone_moved = True
        if not someone_moved:
            converged = True
            break

    return locations, iteration, converged, key
```

```
def run_simulation(params, max_iter=100_000, seed=1234):
    """
    Run the Schelling simulation using JAX.
    """
    key = random.key(seed)
    key, init_key = random.split(key)
    locations, types = initialize_state(init_key, params)

    plot_distribution(locations, types, 'Initial distribution')

    start_time = time.time()
    locations, iteration, converged, key = simulation_loop(locations, types, key,␣
↪params, max_iter)
    elapsed = time.time() - start_time

    plot_distribution(locations, types, f'Iteration {iteration}')

    if converged:
        print(f'Converged in {elapsed:.2f} seconds after {iteration} iterations.')
    else:
```

(continues on next page)

(continued from previous page)

```
        print('Hit iteration bound and terminated.')

    return locations, types
```

The simulation loop is similar to the NumPy version: it cycles through agents, checks each one for happiness, and moves the unhappy ones.

27.5 Results

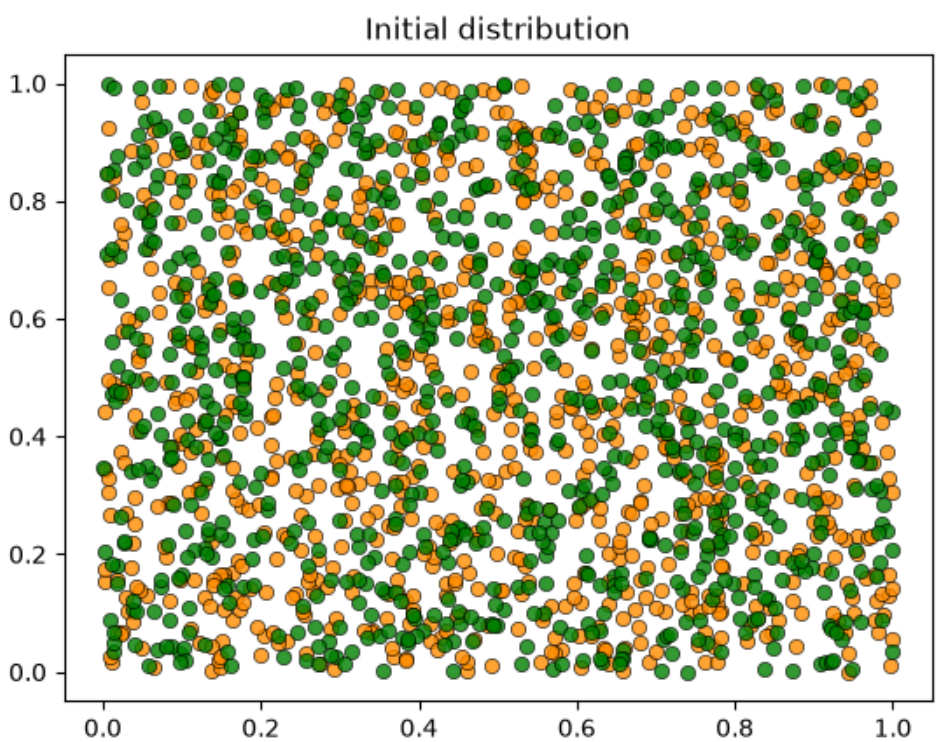
JAX compiles functions the first time they're called. Let's warm them up before timing the simulation:

```
key = random.key(42)
key, init_key = random.split(key)
test_locations, test_types = initialize_state(init_key, params)

_ = is_happy(test_locations[0], 0, test_locations, test_types, params)
key, subkey = random.split(key)
_, _ = move_agent(0, test_locations, test_types, subkey, params)

print("JAX functions compiled and ready!")
```

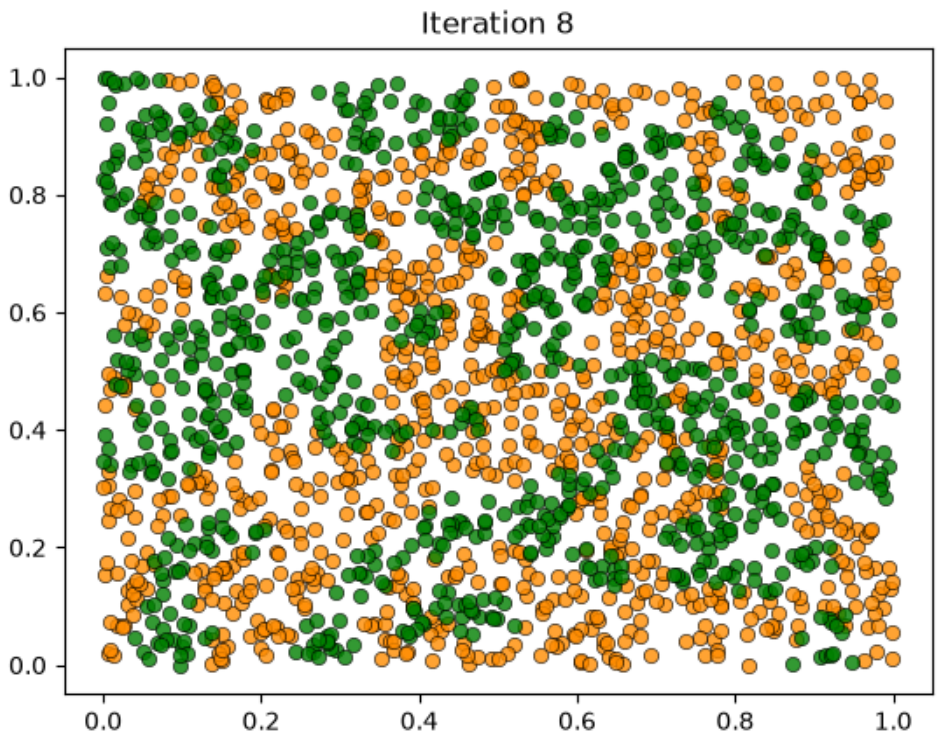
```
JAX functions compiled and ready!
```

Now let's run the simulation:

```
locations, types = run_simulation(params)
```



```
Entering iteration 1
Entering iteration 2
Entering iteration 3
Entering iteration 4
Entering iteration 5
Entering iteration 6
Entering iteration 7
Entering iteration 8
Converged in 14.32 seconds after 8 iterations.
```



27.6 Limitations of This Approach

While this lecture demonstrated JAX syntax and concepts, the algorithm itself doesn't fully leverage JAX's parallel capabilities. The original Schelling algorithm has inherent sequential dependencies:

- Agents update one at a time
- Each agent's move changes the state for subsequent agents
- The "move until happy" while loop has unpredictable length

These characteristics don't map well to parallel hardware like GPUs, which excel at performing the same operation on many data points simultaneously.

In the *next lecture*, we'll restructure the algorithm to better leverage JAX's parallel capabilities.

CHAPTER

TWENTYEIGHT

PARALLELIZING THE ALGORITHM

28.1 Overview

In the previous lectures, we implemented the Schelling segregation model using *NumPy* and *JAX*.

Both implementations are fundamentally sequential: agents update one at a time, and each agent's move changes the state for subsequent agents.

In this lecture, we introduce a **parallel algorithm** that fully leverages JAX's ability to perform vectorized operations across all agents simultaneously.

The algorithm is in some sense less elegant, but it still converges in a relatively small number of steps — and its parallel nature allows us to exploit the full power of JAX.

```
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from jax import random, jit, vmap
from functools import partial
from typing import NamedTuple
import time
```

28.2 Setup

We use the same parameters as before, with the addition of `num_candidates` — the number of candidate locations each agent considers per iteration:

```
class Params(NamedTuple):
    num_of_type_0: int = 1000     # number of agents of type 0 (orange)
    num_of_type_1: int = 1000     # number of agents of type 1 (green)
    num_neighbors: int = 10       # number of neighbors
    max_other_type: int = 6      # max number of different-type neighbors tolerated
    num_candidates: int = 3       # candidate locations per agent per iteration


params = Params()
```

The following functions are repeated from the *previous lecture*:

```
def initialize_state(key, params):
    n = params.num_of_type_0 + params.num_of_type_1
    locations = random.uniform(key, (n, 2))
    types = jnp.concatenate([jnp.zeros(params.num_of_type_0, dtype=int),
                             jnp.ones(params.num_of_type_1, dtype=int)])
    return locations, types


@partial(jit, static_argnames=('params',))
def is_happy(loc, agent_idx, locations, types, params):
    " True if an agent at loc has at most max_other_type different-type neighbors. "
    distances = jnp.sum((loc - locations)**2, axis=1)
    distances = distances.at[agent_idx].set(jnp.inf)
    _, neighbors = jax.lax.top_k(-distances, params.num_neighbors)
    num_other = jnp.sum(types[neighbors] != types[agent_idx])
    return num_other <= params.max_other_type


@partial(jit, static_argnames=('params',))
def get_unhappy_agents(locations, types, params):
    " Return a boolean array indicating which agents are unhappy. "
    n = params.num_of_type_0 + params.num_of_type_1

    def is_unhappy(i):
        return ~is_happy(locations[i], i, locations, types, params)

    return vmap(is_unhappy)(jnp.arange(n))
```

28.3 The Parallel Algorithm

Our aim is to update all agents at the same time, rather than sequentially.

To do this we

1. **Generate candidate locations** for all agents in parallel
2. **Update all agents** simultaneously — happy agents stay put, unhappy agents move if a happy candidate was found

We offer a fixed number of candidates to all agents, so that the parallel threads do the same amount of work and hence run at the same speed.

This approach is well-suited to GPU execution, where thousands of operations can run concurrently.

There are two trade-offs compared to the sequential algorithm.

First, the sequential algorithm guarantees that each agent finds a happy location before moving on.

The parallel algorithm instead proposes a fixed number of candidate locations per agent per iteration.

If none of the candidates make the agent happy, the agent stays put and tries again next iteration.

This means the parallel algorithm may need more iterations.

Second, because we update all agents at once, the agents have less information — they are predicting the next period distribution from the current one.

We hope that, nonetheless, the algorithm will converge.

The `update_agent_location` function below performs all computation (generating candidates for all agents, regardless of current status, checking happiness at each candidate) upfront before making the final decision about whether to move.

This may seem wasteful but it's actually optimal for parallel execution: on GPUs, all threads execute the same instructions in lockstep, so conditional branches don't skip work.

```
@partial(jit, static_argnames=('params',))
def update_agent_location(i, locations, types, key, params):
    """
    Consider current location and num_candidates random alternatives.
    Return the first happy one. Already happy agents stay put.
    """
    current_loc = locations[i, :]

    # Build candidate list: current location + num_candidates random ones
    random_locs = random.uniform(key, (params.num_candidates, 2))
    candidate_locations = jnp.vstack([current_loc[None, :], random_locs])

    # Check happiness at each candidate (in parallel)
    def check_candidate(loc):
        return is_happy(loc, i, locations, types, params)
    happy_at = vmap(check_candidate)(candidate_locations)

    # Return the first happy candidate location.
    # Already happy agents select candidate_locations[0] and stay put.
    # If no candidate is happy, argmax returns 0 and the agent stays put.
    return candidate_locations[jnp.argmax(happy_at)]
```

The code above is for one agent.

The next function uses `vmap` to apply this update rule to determine the new location array across all agents.

```
@partial(jit, static_argnames=('params',))
def parallel_update_step(locations, types, key, params):
    """
    One step of the parallel algorithm: for each agent, find a happy
    candidate location (in parallel). Happy agents stay put, unhappy
    agents search for new locations.
    """
    n = params.num_of_type_0 + params.num_of_type_1

    keys = random.split(key, n + 1)
    key = keys[0]
    agent_keys = keys[1:]

    # Closure: wraps update_agent_location so vmap can map over a single argument
    def update_one_agent(i):
        return update_agent_location(i, locations, types, agent_keys[i], params)
    new_locations = vmap(update_one_agent)(jnp.arange(n))

    return new_locations, key
```

Here's the outer loop, which updates the population until convergence.

By using `jax.lax.while_loop`, we can JIT-compile the entire simulation, avoiding Python loop overhead.

```
@partial(jit, static_argnames=('params', 'max_iter'))
def parallel_simulation_loop(locations, types, key, params, max_iter):
```

(continues on next page)

(continued from previous page)

```
    def while_test(state):
        locations, key, iteration = state
        unhappy = get_unhappy_agents(locations, types, params)
        return (iteration < max_iter) & jnp.any(unhappy)

    def update(state):
        locations, key, iteration = state
        locations, key = parallel_update_step(locations, types, key, params)
        return locations, key, iteration + 1

    locations, _, iteration = jax.lax.while_loop(
        while_test, update, (locations, key, 0)
    )
    converged = ~jnp.any(get_unhappy_agents(locations, types, params))
    return locations, iteration, converged


def run_parallel_simulation(params, max_iter=100_000, seed=42):
    key = random.key(seed)
    key, init_key = random.split(key)
    locations, types = initialize_state(init_key, params)

    plot_distribution(locations, types, 'Initial distribution')

    start_time = time.time()
    locations, iteration, converged = parallel_simulation_loop(locations, types, key,␣
↪params, max_iter)
    elapsed = time.time() - start_time

    plot_distribution(locations, types, f'Iteration {int(iteration)}')

    if converged:
        print(f'Converged in {elapsed:.2f} seconds after {int(iteration)} iterations.
↪')
    else:
        print('Hit iteration bound and terminated.')

    return locations, types
```

28.4 Results

Let's warm up the JIT-compiled functions and run the simulation:

```
key = random.key(0)
key, init_key = random.split(key)
test_locations, test_types = initialize_state(init_key, params)

_ = is_happy(test_locations[0], 0, test_locations, test_types, params)
_ = get_unhappy_agents(test_locations, test_types, params)
key, subkey = random.split(key)
_ = update_agent_location(0, test_locations, test_types, subkey, params)
key, subkey = random.split(key)
_, _ = parallel_update_step(test_locations, test_types, subkey, params)
```

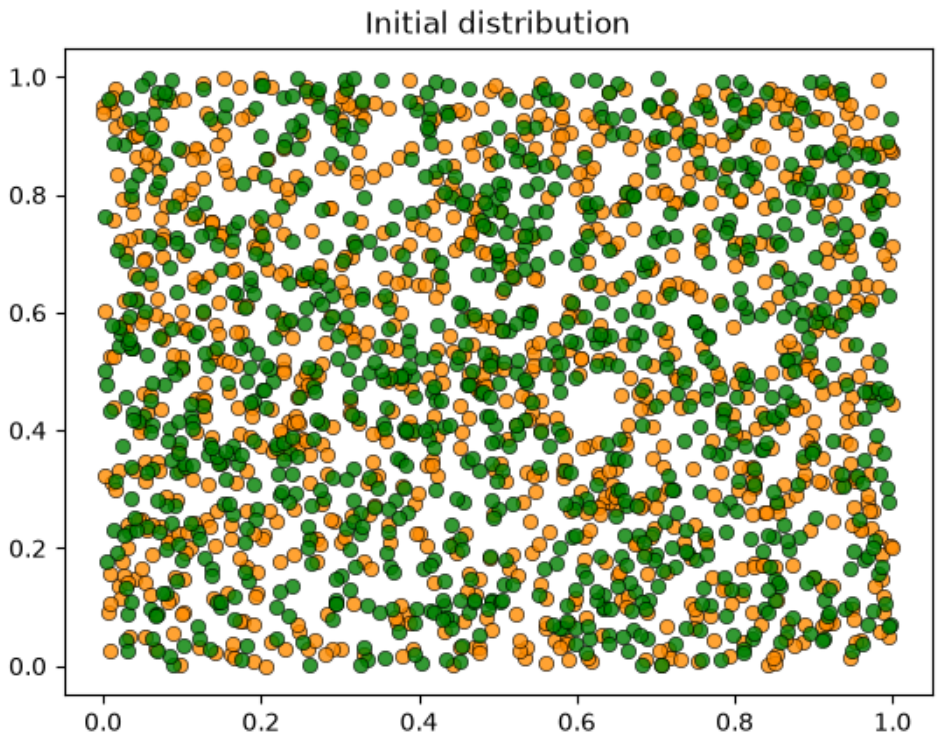
(continues on next page)

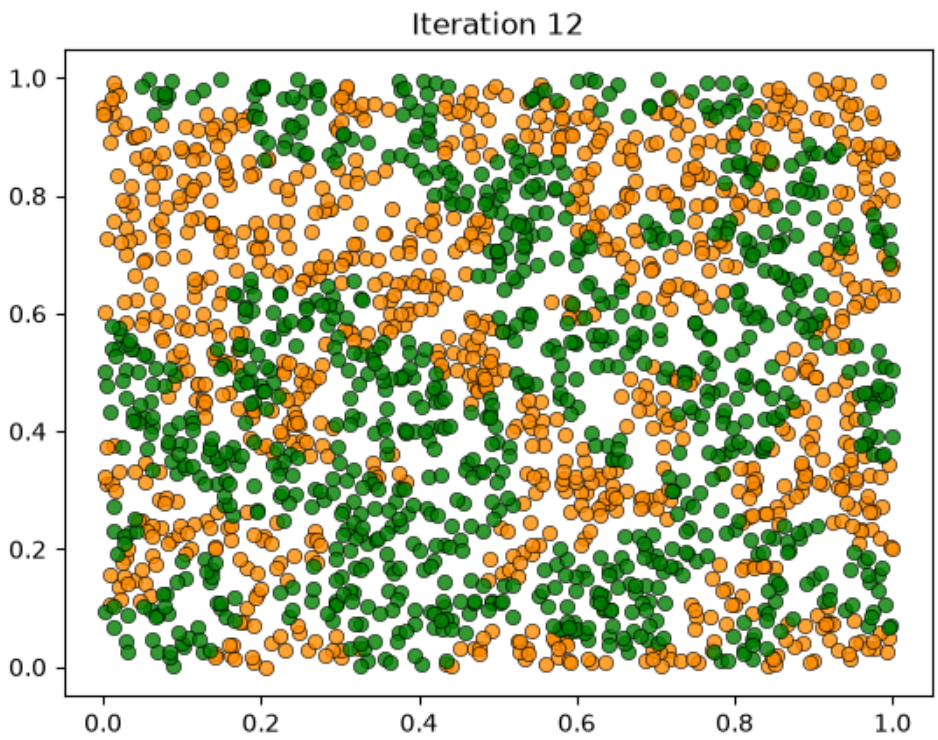
(continued from previous page)

```
print("JAX functions compiled and ready!")
```

```
JAX functions compiled and ready!
```

```
locations, types = run_parallel_simulation(params)
```






```
Converged in 0.68 seconds after 12 iterations.
```

You can compare the execution time with the *NumPy* and *JAX sequential* results from the previous lectures.

28.5 Discussion

The parallel approach processes all agents simultaneously each iteration.

While it may require more iterations (since agents get only a fixed number of candidate locations), each iteration leverages massive parallelism.

This trade-off strongly favors parallelism on GPUs, where thousands of threads can evaluate candidate locations concurrently.

The key lesson is that simply porting code to JAX doesn't automatically make it faster — algorithms often need to be restructured for parallelism.

CHAPTER

TWENTYNINE

SEGREGATION WITH PERSISTENT SHOCKS

29.1 Overview

In previous lectures, we saw that the Schelling model converges to a segregated equilibrium: agents relocate until everyone is happy, and then the system stops.

But real cities don't work this way. People move in and out, neighborhoods change, and the population is constantly being reshuffled by small shocks.

In this lecture, we explore what happens when we add this kind of persistent randomness to the model.

Specifically, after each iteration, we randomly flip the type of some agents with a small probability.

We can interpret this as agents occasionally moving away and being replaced by new agents whose type is randomly determined.

With persistent shocks, the system never converges, so the segregation dynamics keep operating indefinitely.

Because agents are constantly being nudged out of equilibrium, the forces that drive segregation never shut off.

The result is that segregation levels **continue to increase over time**, reaching levels **beyond what we see in the basic model**.

We use the parallel JAX implementation for efficiency, allowing us to run longer simulations with more agents.

```
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from jax import random, jit, vmap
from functools import partial
from typing import NamedTuple
import time
```

29.2 Parameters

We use 2000 agents of each type and add a `flip_prob` parameter that controls the probability of an agent's type being flipped after each iteration.

```
class Params(NamedTuple):
    num_of_type_0: int = 2000     # number of agents of type 0 (orange)
    num_of_type_1: int = 2000     # number of agents of type 1 (green)
    num_neighbors: int = 10       # number of neighbors
```

(continues on next page)

(continued from previous page)

```
    max_other_type: int = 6       # max number of different-type neighbors tolerated
    num_candidates: int = 3       # candidate locations per agent per iteration
    flip_prob: float = 0.01       # probability of flipping an agent's type


params = Params()
```

29.3 Setup

The following functions are repeated from the *previous lecture*:

29.4 Type Flipping

This is the key addition in this lecture. After each iteration, we randomly flip the type of each agent with probability `flip_prob`.

```
@partial(jit, static_argnames=('params',))
def flip_types(types, key, params):
    """
    Randomly flip agent types with probability flip_prob.
    """
    n = params.num_of_type_0 + params.num_of_type_1
    flip_prob = params.flip_prob

    # Generate random numbers for each agent
    random_vals = random.uniform(key, n)

    # Determine which agents get flipped
    should_flip = random_vals < flip_prob

    # Flip: 0 -> 1, 1 -> 0  (equivalent to 1 - type)
    flipped_types = 1 - types

    # Apply flips only where should_flip is True
    new_types = jnp.where(should_flip, flipped_types, types)

    return new_types
```

29.5 Simulation with Shocks

The simulation loop now includes type flipping after each iteration. We run for a fixed number of iterations rather than waiting for convergence, since the system will never fully converge with ongoing shocks.

```
def run_simulation_with_shocks(params, max_iter=1000, seed=42, plot_every=100):
    """
    Run the Schelling simulation with random type flips.

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

(continues on next page)

(continued from previous page)

```
    params : Params
        Model parameters including flip_prob.
    max_iter : int
        Number of iterations to run.
    seed : int
        Random seed.
    plot_every : int
        Plot the distribution every this many iterations.
    """
    key = random.key(seed)
    key, init_key = random.split(key)
    locations, types = initialize_state(init_key, params)
    n = locations.shape[0]

    print(f"Running simulation with {n} agents for {max_iter} iterations")
    print(f"Flip probability: {params.flip_prob}")
    print()

    plot_distribution(locations, types, 'Initial distribution')

    start_time = time.time()

    for iteration in range(1, max_iter + 1):
        # Update locations (agents try to find happy spots)
        locations, key = parallel_update_step(locations, types, key, params)

        # Apply random type flips
        key, flip_key = random.split(key)
        types = flip_types(types, flip_key, params)

        # Periodically report progress and plot
        if iteration % plot_every == 0:
            unhappy = get_unhappy_agents(locations, types, params)
            elapsed = time.time() - start_time
            print(f'Iteration {iteration}: {int(jnp.sum(unhappy))} unhappy agents,
↪{elapsed:.1f}s elapsed')
            plot_distribution(locations, types, f'Iteration {iteration}')

    elapsed = time.time() - start_time
    print(f'\nCompleted {max_iter} iterations in {elapsed:.2f} seconds.')

    return locations, types
```

29.6 Results

Let's warm up the JIT-compiled functions and run the simulation:

```
key = random.key(0)
key, init_key = random.split(key)
test_locations, test_types = initialize_state(init_key, params)

_ = is_happy(test_locations[0], 0, test_locations, test_types, params)
_ = get_unhappy_agents(test_locations, test_types, params)
key, subkey = random.split(key)
```

(continues on next page)

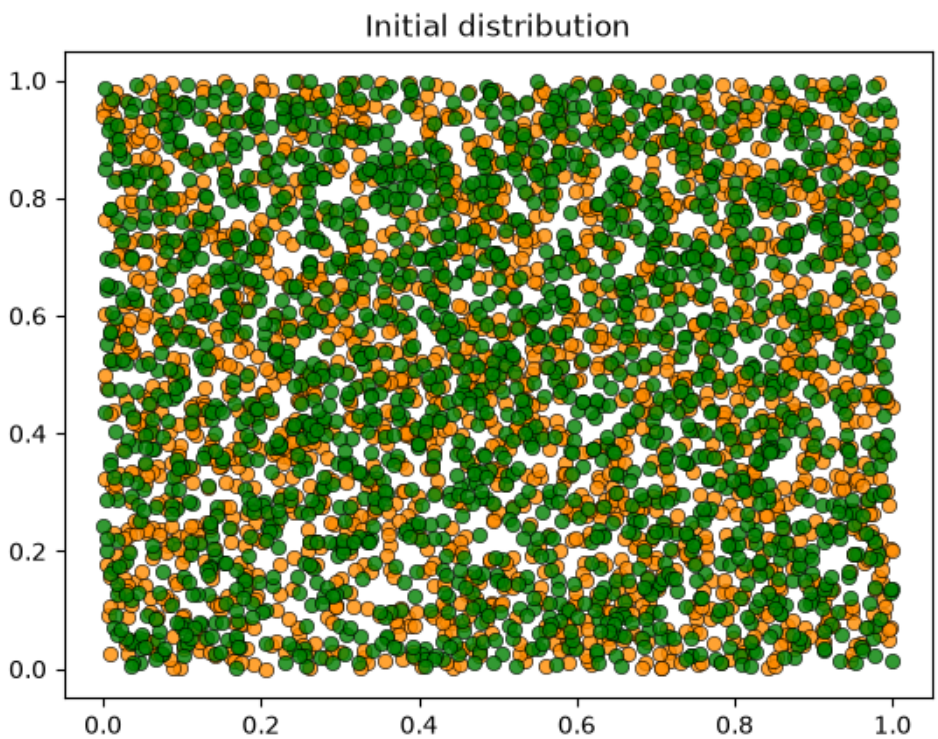
(continued from previous page)

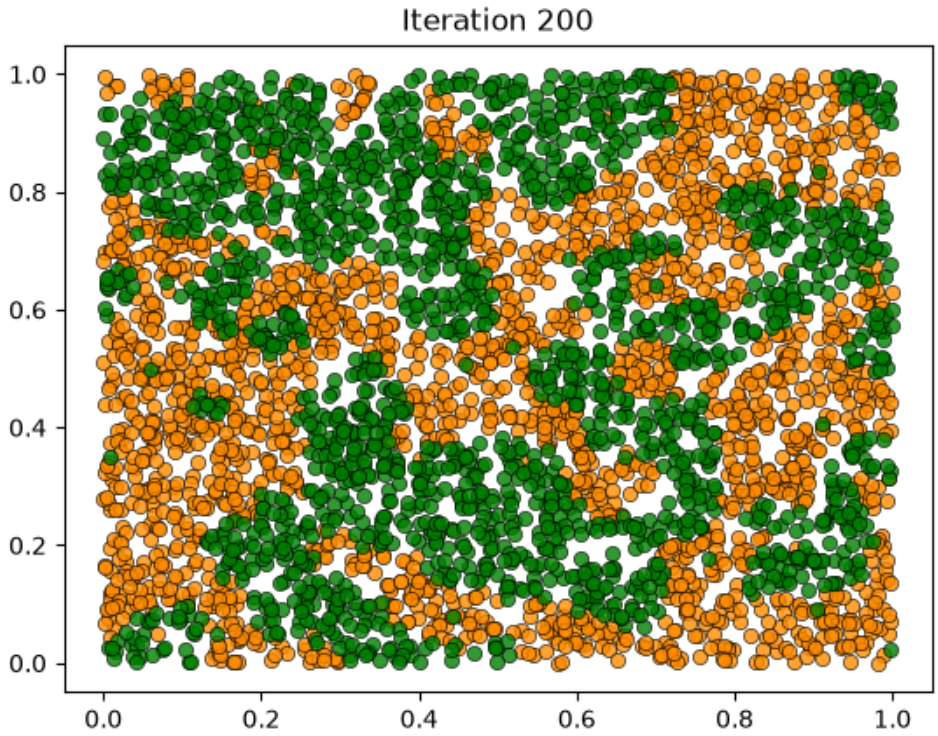
```
_ = update_agent_location(0, test_locations, test_types, subkey, params)
key, subkey = random.split(key)
_, _ = parallel_update_step(test_locations, test_types, subkey, params)
key, subkey = random.split(key)
_ = flip_types(test_types, subkey, params)

print("JAX functions compiled and ready!")
```

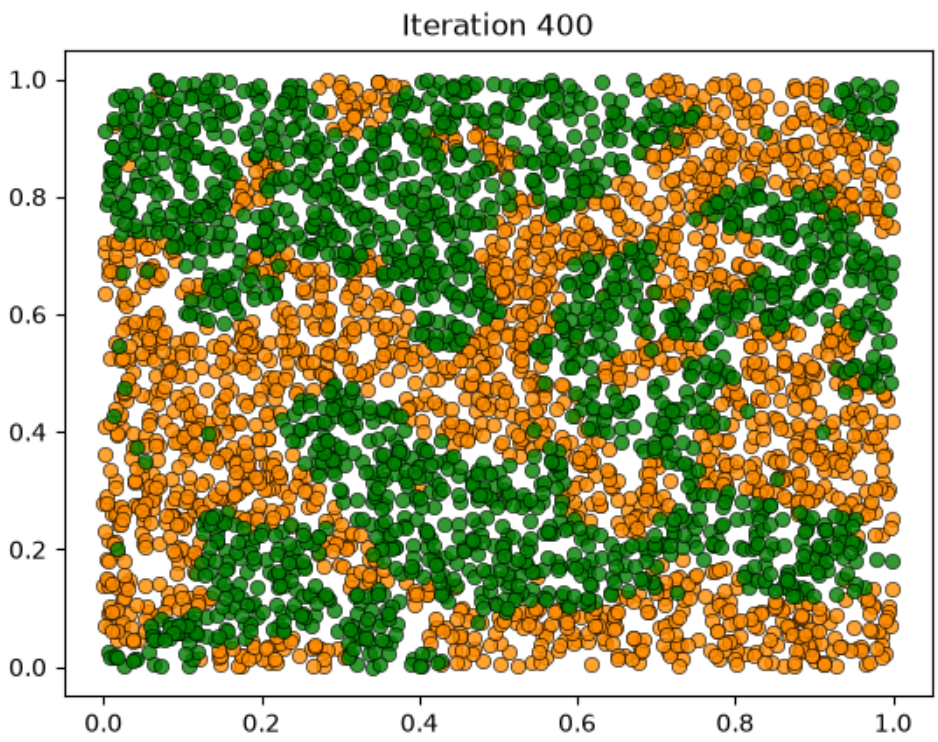
```
JAX functions compiled and ready!
```

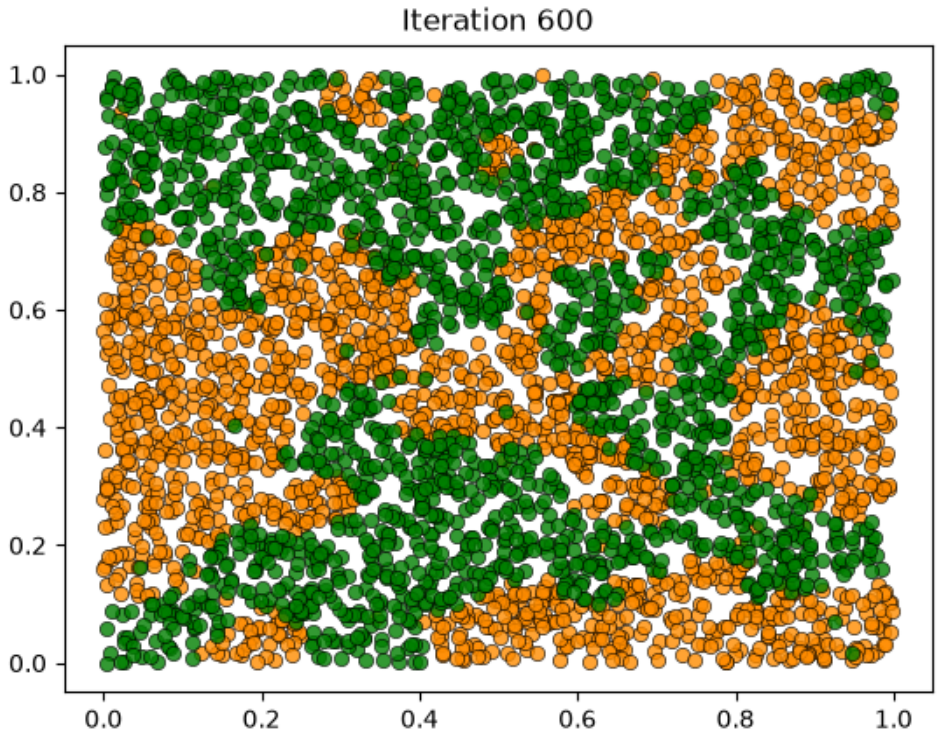
```
locations, types = run_simulation_with_shocks(params, max_iter=1000, plot_every=200)
```

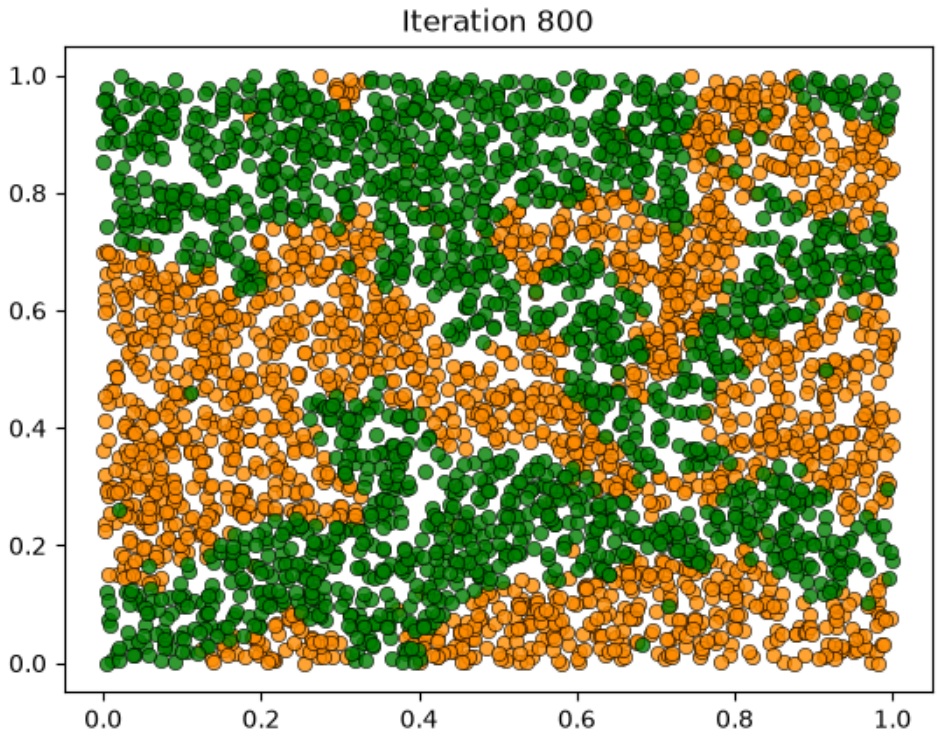
```
Running simulation with 4000 agents for 1000 iterations
Flip probability: 0.01
Iteration 200: 48 unhappy agents, 3.4s elapsed
Iteration 400: 57 unhappy agents, 6.8s elapsed
Iteration 600: 57 unhappy agents, 10.3s elapsed
Iteration 800: 48 unhappy agents, 13.8s elapsed
Iteration 1000: 45 unhappy agents, 17.2s elapsed

Completed 1000 iterations in 17.51 seconds.
```

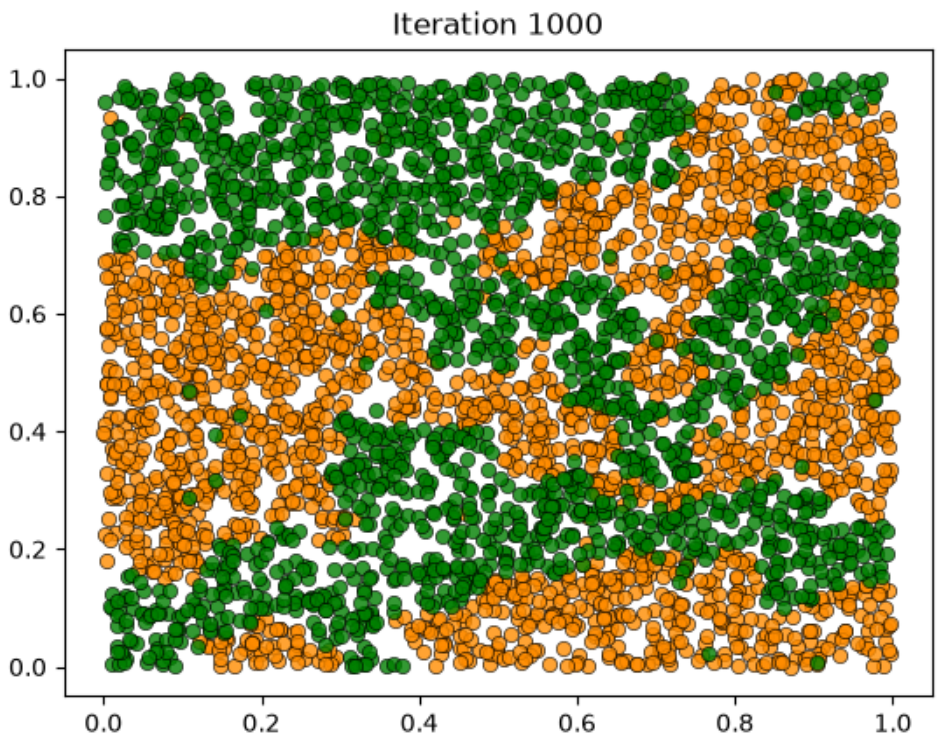



Iteration 200
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Iteration 400
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Iteration 600
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Iteration 800
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0



29.7 Discussion

The figures show an interesting result: segregation levels at the end of the simulation are much higher than in the basic model without shocks.

Why does this happen?

In the basic model, the system converges to an equilibrium where everyone is happy, and then the dynamics stop.

With persistent shocks, the system never converges — random type flips create local pockets of unhappiness, triggering relocations that can cascade through the population.

The key insight is that the segregation dynamics never shut off.

The result is that segregation continues to increase over time, reaching levels far beyond what we observe when the system is allowed to converge.

This is arguably more realistic than the static equilibrium of the basic model.

Real cities experience constant population turnover, and the Schelling dynamics operate continuously on the evolving population.

Part VIII

Other

CHAPTER

THIRTY

TROUBLESHOOTING

This page is for readers experiencing errors when running the code from the lectures.

30.1 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.

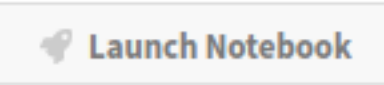
You also need to keep the external code libraries, such as QuantEcon.py up to date.

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.

30.2 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

THIRTYONE

REFERENCES

CHAPTER

THIRTYTWO

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
aiyagari_jax	2026-10-09 00:18	cache	105.43	✅
arellano_jax	2026-10-09 00:18	cache	20.82	✅
autodiff	2026-10-12 00:25	cache	18.07	✅
hopenhayn	2026-10-09 00:19	cache	23.57	✅
ifp_dl	2026-10-09 00:20	cache	32.51	✅
ifp_egm_jax	2026-10-09 00:22	cache	147.2	✅
intro	2026-10-09 00:22	cache	1.0	✅
inventory_dynamics_jax	2026-10-09 00:22	cache	8.89	✅
inventory_ssd	2026-10-09 00:22	cache	9.85	✅
jax_intro_jax	2026-10-09 00:23	cache	18.95	✅
jax_nn	2026-10-09 00:23	cache	38.61	✅
job_search	2026-10-09 00:23	cache	9.38	✅
keras	2026-10-09 00:24	cache	29.88	✅
kesten_processes_jax	2026-10-09 00:24	cache	11.82	✅
lucas_model_jax	2026-10-09 00:24	cache	19.43	✅
markov_asset_jax	2026-10-09 00:25	cache	11.25	✅
mle_jax	2026-10-09 00:25	cache	12.47	✅
newtons_method	2026-10-09 00:28	cache	187.28	✅
opt_invest	2026-10-09 00:28	cache	21.94	✅
opt_savings_1	2026-10-09 00:29	cache	42.95	✅
opt_savings_2	2026-10-09 00:29	cache	20.11	✅
overborrowing	2026-10-09 00:30	cache	23.91	✅
schelling_jax	2026-10-09 00:30	cache	18.72	✅
schelling_jax_parallel	2026-10-09 00:30	cache	5.67	✅
schelling_numpy	2026-10-09 00:30	cache	4.07	✅
schelling_python	2026-10-09 00:31	cache	17.76	✅
schelling_shocks	2026-10-09 00:31	cache	22.42	✅
short_path_jax	2026-10-09 00:31	cache	4.28	✅
status	2026-10-09 00:31	cache	6.69	✅
troubleshooting	2026-10-09 00:22	cache	1.0	✅
wealth_dynamics_jax	2026-10-09 00:34	cache	164.07	✅
zreferences	2026-10-09 00:22	cache	1.0	✅

These lectures are built on `linux` instances through `github actions` that has access to a `gpu`. These lectures make use of the nvidia `T4` card.

These lectures are using the following python version

```
!python --version
```

```
Python 3.13.14
```

and the following package versions

```
!conda list
```

You can check the backend used by JAX using:

```
import jax
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")
```

```
JAX backend: gpu
```

and the hardware we are running on:

```
!nvidia-smi
```

```
Fri Oct  9 00:31:39 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version: 13.0     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                    0 |
| N/A   39C    P0             27W /   70W |     107MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            8526      C   ...da3/envs/quantecon/bin/python        104MiB |
+-----------------------------------------------------------------------------------------+
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.13/pty.py:95: RuntimeWarning:␣
↪os.fork() was called. os.fork() is incompatible with multithreaded code, and JAX␣
↪is multithreaded, so this will likely lead to a deadlock.
  pid, fd = os.forkpty()
```

BIBLIOGRAPHY

[Are08] Cristina Arellano. Default risk and income fluctuations in emerging economies. *The American Economic Review*, pages 690–712, 2008.

[Bia11] Javier Bianchi. Overborrowing and systemic externalities in the business cycle. *American Economic Review*, 101(7):3400–3426, December 2011. URL: https://www.aeaweb.org/articles?id=10.1257/aer.101.7.3400, doi:10.1257/aer.101.7.3400.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.

PROOF INDEX

main_loop_algo

`main_loop_algo` (*schelling_python*), 347

move_algo

`move_algo` (*schelling_python*), 346

INDEX

F

Fixed Point Theory, 98

K

Kesten processes
    heavy tails, 66

L

Linear State Space Models, 65
Lucas Model, 94
    Assets, 94
    Computation, 98
    Consumers, 95
    Dynamic Program, 95
    Equilibrium Constraints, 96
    Equilibrium Price Function, 96
    Pricing, 95
    Solving, 97

M

Markov process, inventory, 53
Models
    Lucas Asset Pricing, 93