Quantitative Economics with Python using JAX

Thomas J. Sargent & John Stachurski

Apr 01, 2024

CONTENTS

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

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 JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

This lecture provides a short introduction to Google JAX.

As mentioned above, the lecture was built using a GPU:

```
!nvidia-smi
```

```
Mon Apr  1 17:50:33 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  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.xla_extension.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)
```

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

```
EighResult(eigenvalues=Array([0.99999994, 0.99999994], dtype=float32),␣
↪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[20], line 1
----> 1 a[0] = 1

File /opt/conda/envs/quantecon/lib/python3.11/site-packages/jax/_src/numpy/array_
 ↪methods.py:278, in _unimplemented_setitem(self, i, x)
    273 def _unimplemented_setitem(self, i, x):
    274   msg = ("'{}' object does not support item assignment. JAX arrays are "
    275          "immutable. Instead of ``x[idx] = y``, use ``x = x.at[idx].
 ↪set(y)`` "
    276          "or another .at[] method: "
    277          "https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.
 ↪ndarray.at.html")
--> 278   raise TypeError(msg.format(type(self)))

TypeError: '<class 'jaxlib.xla_extension.ArrayImpl'>' object does not support item␣
 ↪assignment. JAX arrays are immutable. Instead of ``x[idx] = y``, use ``x = x.
 ↪at[idx].set(y)`` or another .at[] method: https://jax.readthedocs.io/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.

Note that, while mutation is discouraged, it is in fact possible with `at`, as in

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

```
133767904
```

```
a
```

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

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

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

We can check that the array is mutated by verifying its identity is unchanged:

```
id(a)
```

```
133767904
```

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.PRNGKey(1)
```

```
type(key)
```

```
jaxlib.xla_extension.ArrayImpl
```

```
print(key)
```

```
[0 1]
```

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

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

```
Array([[-1.35247421, -0.2712502 , -0.02920518],
       [ 0.34706456,  0.5464053 , -1.52325812],
       [ 0.41677264, -0.59710138, -0.5678208 ]], 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([[-1.35247421, -0.2712502 , -0.02920518],
       [ 0.34706456,  0.5464053 , -1.52325812],
       [ 0.41677264, -0.59710138, -0.5678208 ]], dtype=float64)
```

To produce a (quasi-) independent draw, best practice is to "split" the existing key:

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

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

```
Array([[ 1.85374374, -0.37683949, -0.61276867],
       [-1.91829718,  0.27219409,  0.54922246],
       [ 0.40451442, -0.58726839, -0.63967753]], dtype=float64)
```

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

```
Array([[-0.4300635 ,  0.22778552,  0.57241269],
       [-0.15969178,  0.46719192,  0.21165091],
       [ 0.84118631,  1.18671326, -0.16607783]], dtype=float64)
```

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

```
def gen_random_matrices(key, n, k):
    matrices = []
    for _ in range(k):
        key, subkey = random.split(key)
        matrices.append(random.uniform(subkey, (n, n)))
    return matrices
```

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

```
[[0.97440813 0.3838544 ]
 [0.9790686  0.99981046]]
[[0.3473302  0.17157842]
 [0.89346686 0.01403153]]
```

One point to remember is that JAX expects tuples to describe array shapes, even for flat arrays. Hence, to get a one-dimensional array of normal random draws we use `(len, )` for the shape, as in

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

```
Array([-0.64377279,  0.76961857, -0.29809604,  0.47858776, -2.00591299],      ␣
 ↪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 279 ms, sys: 274 µs, total: 280 ms
Wall time: 492 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 6.63 ms, sys: 179 µs, total: 6.81 ms
Wall time: 25.7 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 269 ms, sys: 0 ns, total: 269 ms
Wall time: 480 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 6.31 ms, sys: 742 µs, total: 7.05 ms
Wall time: 19 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_ready()
```

```
CPU times: user 5.45 ms, sys: 475 µs, total: 5.93 ms
Wall time: 25.3 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 1.29 ms, sys: 0 ns, total: 1.29 ms
Wall time: 114 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:

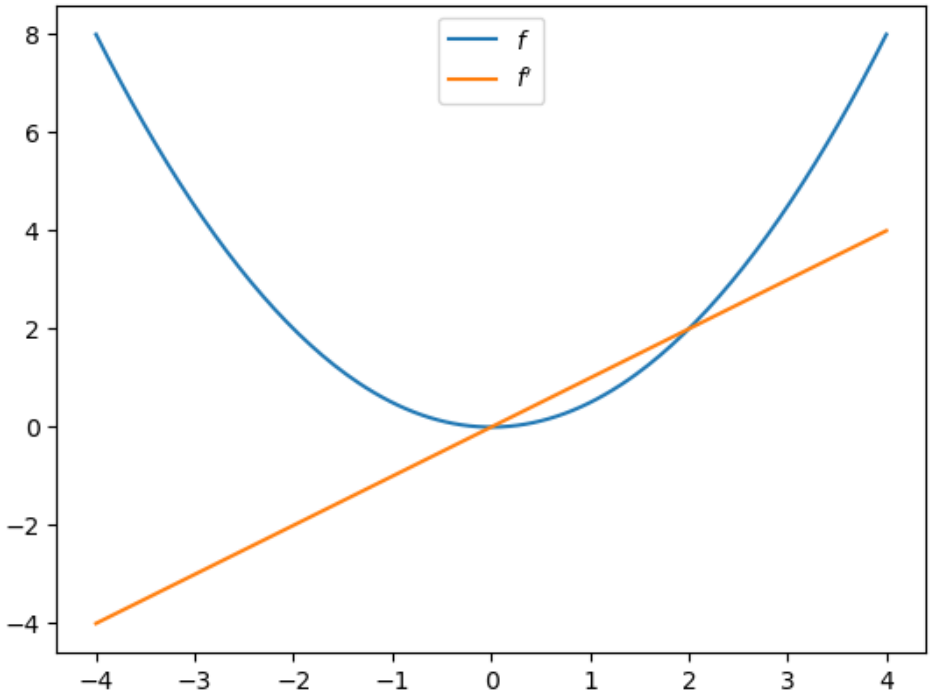
```
f_prime = jax.grad(f)
```

```
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()
```



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 12.8 s, sys: 2.42 s, total: 15.2 s
Wall time: 6.98 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)
```

```
CPU times: user 49 ms, sys: 0 ns, total: 49 ms
Wall time: 84.1 ms
```

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

```
CPU times: user 591 µs, sys: 0 ns, total: 591 µs
Wall time: 228 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)
```

```
CPU times: user 54 ms, sys: 0 ns, total: 54 ms
Wall time: 90.7 ms
```

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

```
CPU times: user 591 µs, sys: 0 ns, total: 591 µs
Wall time: 341 µs
```

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)
```

```
CPU times: user 58.7 ms, sys: 0 ns, total: 58.7 ms
Wall time: 94.7 ms
```

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

```
CPU times: user 2.23 ms, sys: 0 ns, total: 2.23 ms
Wall time: 1.74 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 and parallelization, 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 to Exercise 2.7.1

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.PRNGKey(1)):

    s = jnp.full(M, np.log(S0))
    h = jnp.full(M, h0)
    for t in range(n):
        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, :]
    expectation = jnp.mean(jnp.maximum(jnp.exp(s) - K, 0))

    return β**n * expectation
```

Let's run it once to compile it:

```
compute_call_price_jax()
```

```
Array(180876.48840921, dtype=float64)
```

And now let's time it:

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

```
CPU times: user 1.51 ms, sys: 0 ns, total: 1.51 ms
Wall time: 126 ms
```

```
Array(180876.48840921, dtype=float64)
```

CHAPTER

THREE

NEWTON'S METHOD VIA JAX

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

3.1 Overview

One of the key features of JAX is automatic differentiation.

While other software packages also offer this feature, the JAX version is particularly powerful because it integrates so closely with other core components of JAX, such as accelerated linear algebra, JIT compilation and parallelization.

The application of automatic differentiation we consider is 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 almost 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
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:52:19 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   28C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

3.2 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

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

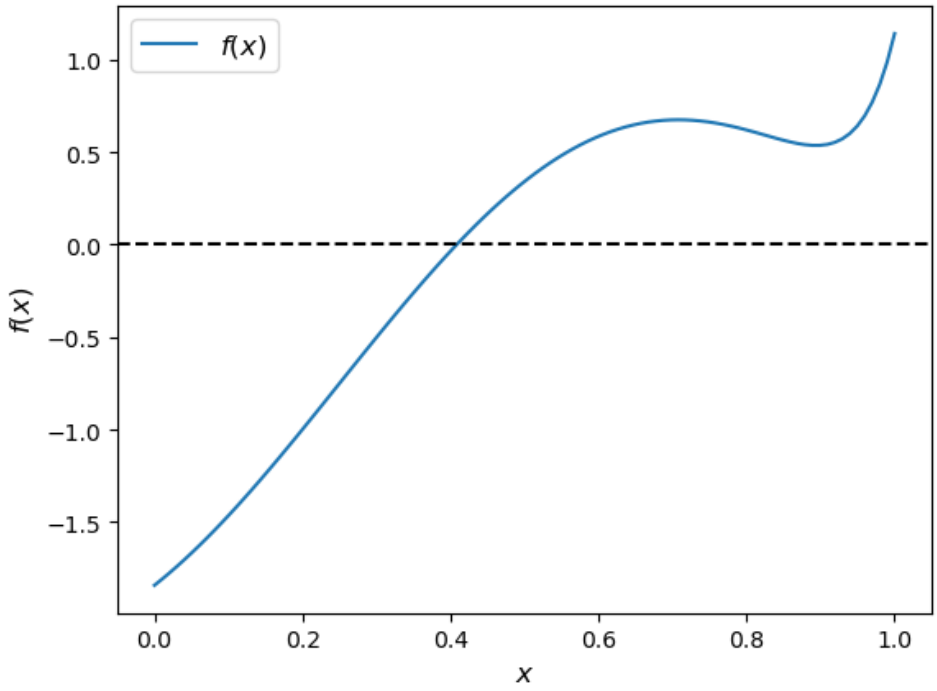
Here is a function called `newton` that takes a function f plus a scalar value x_0, iterates with q starting from x_0, and returns an approximate fixed point.

```
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)
        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.

3.3 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)
```

3.4 Computation

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

3.4.1 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

$$p_{n+1} = p_n - J_e(p_n)^{-1} e(p_n) \tag{3.1}$$

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
```

(continues on next page)

(continued from previous page)

```
    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
```

3.4.2 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.PRNGKey(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 = 29.97745704650879
iteration 2, error = 5.092828750610352
iteration 3, error = 0.10971635580062866
iteration 4, error = 5.19721070304513e-05
iteration 5, error = 1.2384003639454022e-05
```

```
iteration 6, error = 4.883217570750276e-06
CPU times: user 4.33 s, sys: 1.13 s, total: 5.47 s
Wall time: 3.84 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 26s, sys: 442 ms, total: 2min 26s
Wall time: 2min 25s
```

The result is also slightly less accurate:

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

```
Array(7.1525574e-07, dtype=float32)
```

3.5 Exercises

Exercise 3.5.1

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:

$$k1_0 = (1, 1, 1)$$
$$k2_0 = (3, 5, 5)$$
$$k3_0 = (50, 50, 50)$$

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 to Exercise 3.5.1

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
```

(continues on next page)

(continued from previous page)

```
CPU times: user 194 ms, sys: 16.4 ms, total: 210 ms
Wall time: 253 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 101 ms, sys: 8.32 ms, total: 109 ms
Wall time: 95.1 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 227 ms, sys: 12.5 ms, total: 240 ms
Wall time: 216 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
```

(continues on next page)

(continued from previous page)

```
CPU times: user 220 ms, sys: 4.62 ms, total: 224 ms
Wall time: 229 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 210 ms, sys: 12.4 ms, total: 222 ms
Wall time: 280 ms
```

We can see it steps towards a more accurate solution.

Exercise 3.5.2

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 to Exercise 3.5.2

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 112 ms, sys: 383 µs, total: 112 ms
Wall time: 134 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 92.5 ms, sys: 11.7 ms, total: 104 ms
Wall time: 85.7 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

FOUR

INVENTORY DYNAMICS

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

4.1 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 with a bulk order of S units (or, in some models, restocks up to level S).

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 collections import namedtuple
```

Here's a description of our GPU:

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:24:27 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

4.2 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.

```
Parameters = namedtuple('Parameters', ['s', 'S', 'μ', 'σ'])

# Create a default instance
params = Parameters(s=10, S=100, μ=1.0, σ=0.5)
```

4.3 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, X_vec, D):
    """
    Update by one period a cross-section of firms with inventory levels given by
    X_vec, given the vector of demand shocks in D.

       * 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
```

4.3.1 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

While Python loops are slow, this approach is reasonable here because efficiency of outer loops has far less influence on runtime than efficiency of inner loops.

(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 compute_cross_section(params, x_init, T, key, num_firms=50_000):
    # Set up initial distribution
    X_vec = jnp.full((num_firms, ), x_init)
    # Loop
    for i in range(T):
        Z = random.normal(key, shape=(num_firms, ))
```

(continues on next page)

(continued from previous page)

```
        D = jnp.exp(params.μ + params.σ * Z)

        X_vec = update_cross_section(params, X_vec, D)
        _, key = random.split(key)

    return X_vec
```

We'll use the following specification

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

Let's look at the timing.

```
%time X_vec = compute_cross_section(params, \
        x_init, T, key).block_until_ready()
```

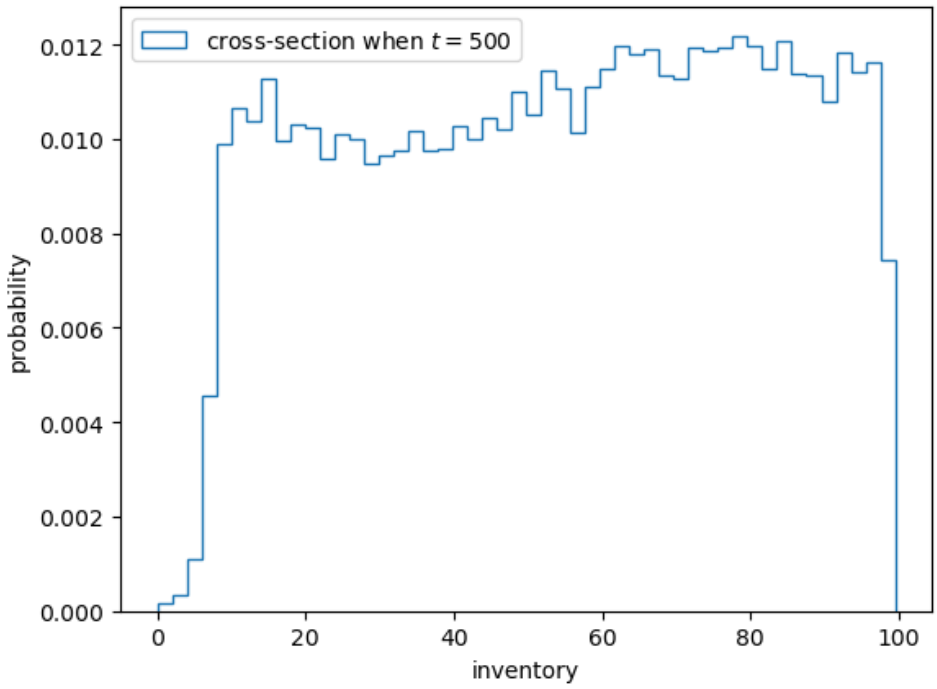
```
CPU times: user 1.35 s, sys: 489 ms, total: 1.84 s
Wall time: 978 ms
```

```
%time X_vec = compute_cross_section(params, \
        x_init, T, key).block_until_ready()
```

```
CPU times: user 999 ms, sys: 402 ms, total: 1.4 s
Wall time: 525 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()
```



4.3.2 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 compute_cross_section_fori(params, x_init, T, key, num_firms=50_000):

    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, inputs):
        # Unpack
        X, key = inputs
        # Draw shocks using key
        Z = random.normal(key, 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))
        # Refresh the key
        key, subkey = random.split(key)
        return X, subkey

    # Loop t from 0 to T, applying fori_update each time.
    # The initial condition for fori_update is (X, key).
    X, key = lax.fori_loop(0, T, fori_update, (X, key))

    return X
```

(continues on next page)

(continued from previous page)

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

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

```
%time X_vec = compute_cross_section_fori(params, \
    x_init, T, key).block_until_ready()
```

```
CPU times: user 296 ms, sys: 0 ns, total: 296 ms
Wall time: 208 ms
```

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

```
%time X_vec = compute_cross_section_fori(params, \
    x_init, T, key).block_until_ready()
```

```
CPU times: user 16.9 ms, sys: 0 ns, total: 16.9 ms
Wall time: 14.5 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 whole operation.

4.3.3 Further vectorization

For relatively small problems, we can make this code run even faster by generating all random variables at once.

This improves efficiency because we are taking more operations out of the loop.

```
def compute_cross_section_fori(params, x_init, T, key, num_firms=50_000):

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

    def update_cross_section(i, X):
        X = jnp.where(X <= s,
                  jnp.maximum(S - D[i, :], 0),
                  jnp.maximum(X - D[i, :], 0))
        return X

    X = lax.fori_loop(0, T, update_cross_section, X)

    return X

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

Let's test it with compile time included.

```
%time X_vec = compute_cross_section_fori(params, \
    x_init, T, key).block_until_ready()
```

```
CPU times: user 236 ms, sys: 0 ns, total: 236 ms
Wall time: 221 ms
```

Let's run again to eliminate compile time.

```
%time X_vec = compute_cross_section_fori(params, \
    x_init, T, key).block_until_ready()
```

```
CPU times: user 5.28 ms, sys: 0 ns, total: 5.28 ms
Wall time: 4.48 ms
```

On one hand, this version is faster than the previous one, where random variables were generated inside the loop.

On the other hand, this implementation consumes far more memory, as we need to store large arrays of random draws.

The high memory consumption becomes problematic for large problems.

4.4 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):

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

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

        X = update_cross_section(params, X, D)
        _, key = random.split(key)

        # 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
```

(continues on next page)

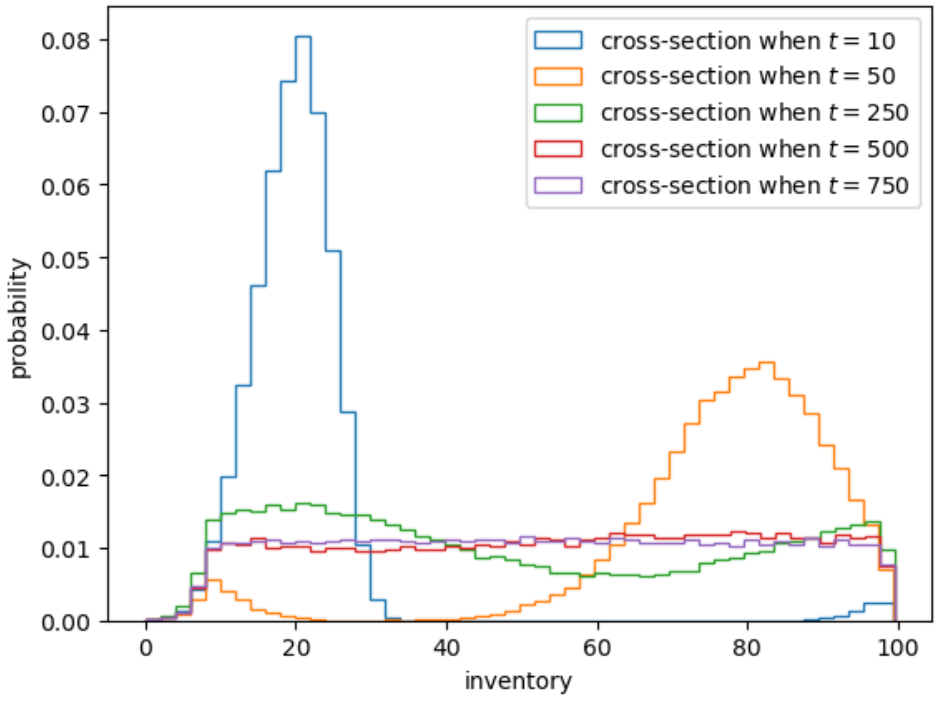
(continued from previous page)

```
key = random.PRNGKey(10)


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

```
CPU times: user 2.25 s, sys: 0 ns, total: 2.25 s
Wall time: 1.06 s
```

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.

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.

4.5 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.

4.5.1 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, key

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):
        Z = random.normal(key, shape=(num_firms, ))
        D = jnp.exp(params.μ + params.σ * Z)
        n_restock, X, key = update_stock(
            n_restock, X, params, D)
        key = random.fold_in(key, i)

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

```
key = random.PRNGKey(27)
%time freq = compute_freq(params, key).block_until_ready()
print(f"Frequency of at least two stock outs = {freq}")
```

```
CPU times: user 633 ms, sys: 0 ns, total: 633 ms
Wall time: 638 ms
Frequency of at least two stock outs = 0.4472379982471466
```

Exercise 4.5.1

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

Solution to Exercise 4.5.1

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

```
@jax.jit
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)
```

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

```
%time freq = compute_freq(params, key).block_until_ready()
%time freq = compute_freq(params, key).block_until_ready()

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

```
CPU times: user 338 ms, sys: 0 ns, total: 338 ms
Wall time: 294 ms
CPU times: user 2.47 ms, sys: 0 ns, total: 2.47 ms
Wall time: 4.91 ms
Frequency of at least two stock outs = 0.44674399495124817
```

CHAPTER

FIVE

KESTEN PROCESSES AND FIRM DYNAMICS

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

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

```
!pip install quantecon
```

5.1 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
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:51:07 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

5.2 Kesten processes

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

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{5.1}$$

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.

5.2.1 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

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{5.2}$$

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 (5.2) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{5.3}$$

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 (5.3) 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 (5.3) 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

$$s_{t+1} = e_{t+1} \mathbb{1}\{s_t < \bar{s}\} + (a_{t+1} s_t + b_{t+1}) \mathbb{1}\{s_t \geq \bar{s}\} \tag{5.4}$$

The motivation behind and interpretation of *(5.2.4)* can be found in our earlier Kesten process lecture.

What can we say about dynamics?

Although (5.4) 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 code to update a cross-section of firms according to the dynamics in *(5.2.4)*.

```
@jax.jit
def update_s(s, s_bar, a_random, b_random, e_random):
    exp_a = jnp.exp(a_random)
    exp_b = jnp.exp(b_random)
    exp_e = jnp.exp(e_random)

    s = jnp.where(s < s_bar,
                  exp_e,
                  exp_a * s + exp_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_draws(M=1_000_000,
                   μ_a=-0.5,
                   σ_a=0.1,
                   μ_b=0.0,
                   σ_b=0.5,
                   μ_e=0.0,
                   σ_e=0.5,
                   s_bar=1.0,
                   T=500,
                   s_init=1.0,
                   seed=123):

    key = random.PRNGKey(seed)

    # Initialize the array of s values with the initial value
    s = jnp.full((M, ), s_init)

    # Perform updates on s for time t
    for t in range(T):
        keys = random.split(key, 3)
        a_random = μ_a + σ_a * random.normal(keys[0], (M, ))
        b_random = μ_b + σ_b * random.normal(keys[1], (M, ))
        e_random = μ_e + σ_e * random.normal(keys[2], (M, ))

        s = update_s(s, s_bar, a_random, b_random, e_random)

        # Generate new key for the next iteration
        key = random.fold_in(key, t)

    return s

%time data = generate_draws().block_until_ready()
```

```
CPU times: user 4.43 s, sys: 2.26 s, total: 6.69 s
Wall time: 3.98 s
```

Running the above function again so we can see the speed with and without compile time.

```
%time data = generate_draws().block_until_ready()
```

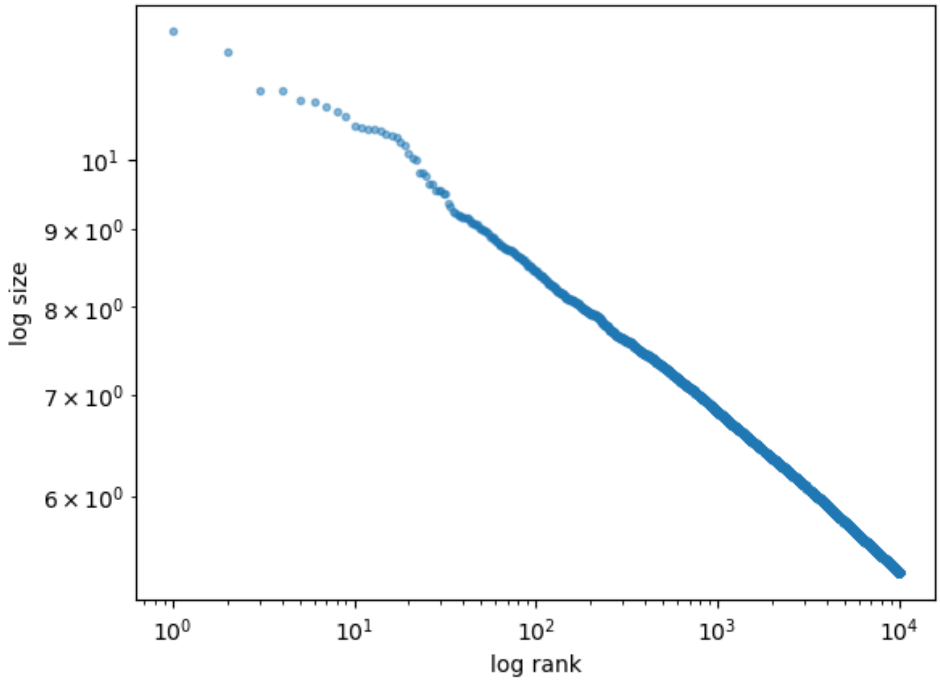
```
CPU times: user 3.82 s, sys: 744 ms, total: 4.56 s
Wall time: 1.94 s
```

Notice that we do not JIT-compile the `for` loops, since

1. acceleration of the outer loop makes little difference terms of compute time and
2. compiling the outer loop is often very slow.

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`

If the time horizon is not too large, we can try to further accelerate our code by replacing the `for` loop with `lax.fori_loop`.

Note, however, that

1. as mentioned above, there is not much speed gain in accelerating outer loops,
2. `lax.fori_loop` has a more complicated syntax, and, most importantly,
3. the `lax.fori_loop` implementation consumes far more memory, as we need to have to store large matrices of random draws

Hence the code below will fail due to out-of-memory errors when `T` and `M` are large.

Here is the `lax.fori_loop` version:

```
@jax.jit
def generate_draws_lax(μ_a=-0.5,
                       σ_a=0.1,
                       μ_b=0.0,
                       σ_b=0.5,
                       μ_e=0.0,
                       σ_e=0.5,
                       s_bar=1.0,
                       T=500,
                       M=500_000,
                       s_init=1.0,
                       seed=123):

    key = random.PRNGKey(seed)
    keys = random.split(key, 3)

    # Generate random draws and initial values
    a_random = μ_a + σ_a * random.normal(keys[0], (T, M))
    b_random = μ_b + σ_b * random.normal(keys[1], (T, M))
    e_random = μ_e + σ_e * random.normal(keys[2], (T, M))
    s = jnp.full((M, ), s_init)

    # Define the function for each update
    def update_s(i, s):
        a, b, e = a_random[i], b_random[i], e_random[i]
        s = jnp.where(s < s_bar,
                      jnp.exp(e),
                      jnp.exp(a) * s + jnp.exp(b))
        return s

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

%time data = generate_draws_lax().block_until_ready()
```

```
CPU times: user 415 ms, sys: 0 ns, total: 415 ms
Wall time: 408 ms
```

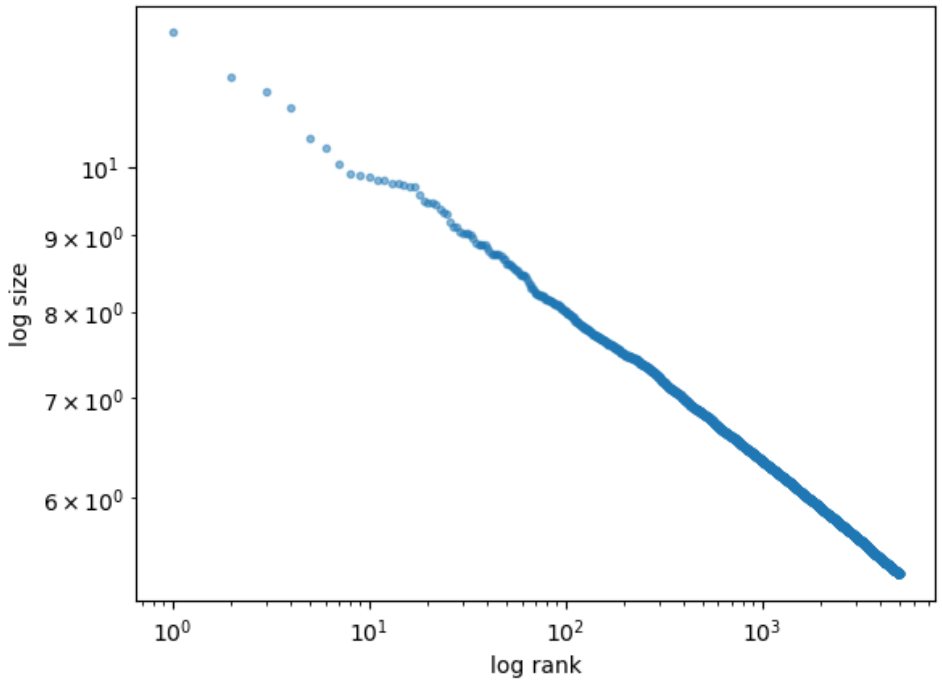
In this case, `M` is small enough for the code to run and we see some speed gain over the for loop implementation:

```
%time data = generate_draws_lax().block_until_ready()
```

```
CPU times: user 3.49 ms, sys: 0 ns, total: 3.49 ms
Wall time: 34.6 ms
```

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()
```



Let's rerun the `for` loop version on smaller `M` to compare the speed

```
%time generate_draws(M=500_000).block_until_ready()
```

```
CPU times: user 4.53 s, sys: 567 ms, total: 5.09 s
Wall time: 2.29 s
```

```
Array([2.389801 , 2.2558599, 3.3113828, ..., 2.7102313, 2.5520844,
       3.4196172], dtype=float32)
```

We see that the `lax.fori_loop` version is faster than the `for` loop version when memory is not an issue.

CHAPTER

SIX

WEALTH DISTRIBUTION DYNAMICS

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

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.

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
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 22:27:28 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   33C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

6.1 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

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{6.1}$$

where s_0 is a positive constant.

Thus,

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

6.2 Implementation

6.2.1 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)
    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
```

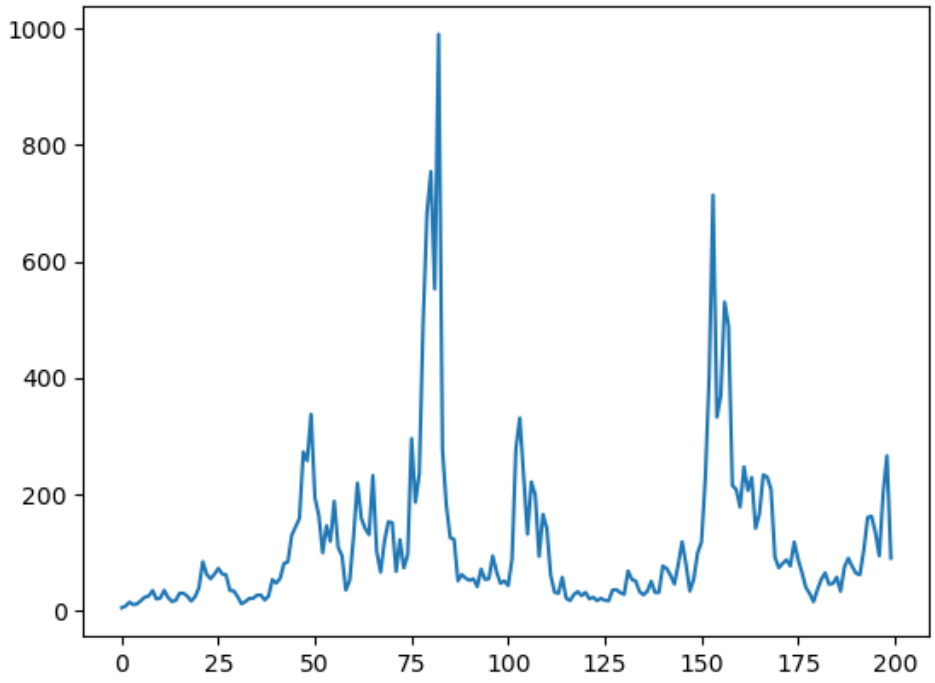
Here's a function to simulate the time series of wealth for an individual household

```
@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)
    w = np.empty(sim_length)
    w[0] = w_0
    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

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

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

    Returns the new distribution.

    """
    # Unpack
    household_params, aggregate_params = model

    num_households = len(w_distribution)
    new_distribution = np.empty_like(w_distribution)
```

(continues on next page)

(continued from previous page)

```
    z = z_sequence

    # Update each household
    for i in numba.prange(num_households):
        w = w_distribution[i]
        for t in range(sim_length):
            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 = time()
ψ_star = update_cross_section(model, ψ_0, z_sequence)
numba_time = time() - start_time
print(f"Generated cross-section in {numba_time} seconds.\n")
```

```
Generating cross-section using Numba
```

```
Generated cross-section in 49.96913409233093 seconds.
```

6.2.2 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 shock sequence z_sequence

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

    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
```

(continues on next page)

(continued from previous page)

```
    w = w_distribution
    n = len(w)

    # Update wealth
    for t, z in enumerate(z_sequence):
        U = jax.random.normal(key, (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)
        key, subkey = jax.random.split(key)

    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.PRNGKey(1234)
start_time = time()
ψ_star = update_cross_section_jax(model, ψ_0, z_sequence, key)
jax_time = time() - start_time
print(f"Generated cross-section in {jax_time} seconds.\n")
```

```
Generating cross-section using JAX
```

```
Generated cross-section in 1.881836175918579 seconds.
```

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

```
Repeating without compile time.
```

```
Generated cross-section in 0.7810535430908203 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):
```

(continues on next page)

(continued from previous page)

```
    """
    Shifts a cross-section of households forward in time

    Takes

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

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

    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
    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.PRNGKey(1234)
start_time = time()
ψ_star = update_cross_section_jax_compiled(
        model, ψ_0, num_households, z_sequence, key
)
jax_fori_time = time() - start_time
print(f"Generated cross-section in {jax_fori_time} seconds.\n")
```

```
Generating cross-section using JAX with compiled loop
```

```
Generated cross-section in 0.3298799991607666 seconds.
```

```
print("Repeating without compile time")
key = jax.random.PRNGKey(1234)
```

(continues on next page)

(continued from previous page)

```
start_time = time()
ψ_star = update_cross_section_jax_compiled(
        model, ψ_0, num_households, z_sequence, key
)
jax_fori_time = time() - start_time
print(f"Generated cross-section in {jax_fori_time} seconds")
```

```
Repeating without compile time
Generated cross-section in 0.17080473899841309 seconds
```

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

```
JAX is 292.5512 times faster.
```

6.2.3 Pareto tails

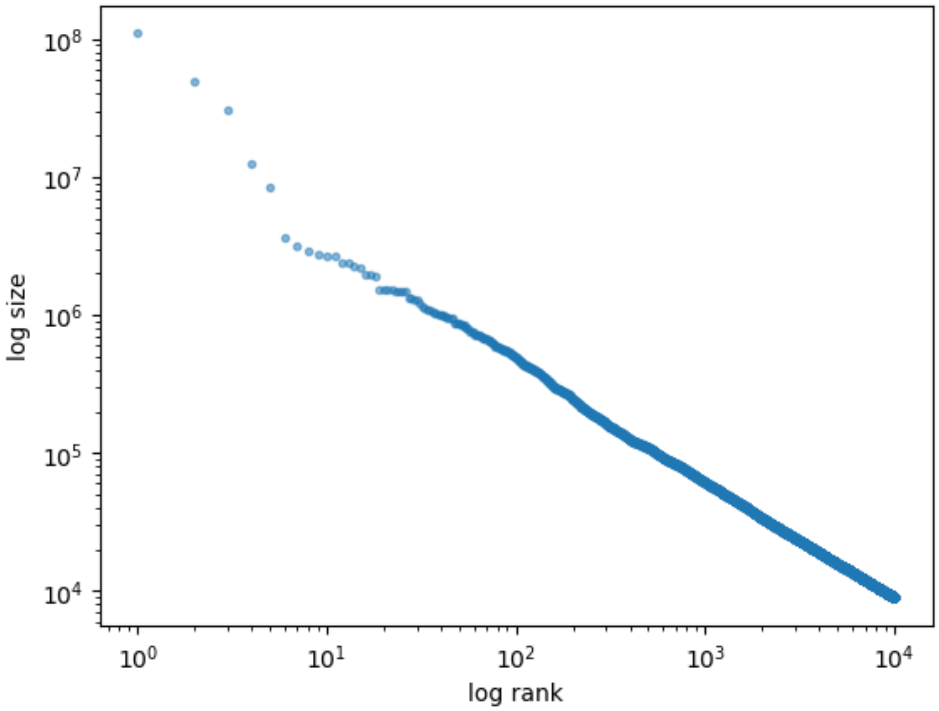
In most countries, the cross-sectional distribution of wealth exhibits a Pareto tail (power law).

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.PRNGKey(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()
```



6.2.4 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 $w_1, \ldots, w_n$, 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.PRNGKey(1234)
ψ_star = update_cross_section_jax_compiled(
        model, ψ_0, num_households, z_sequence, key
)
```

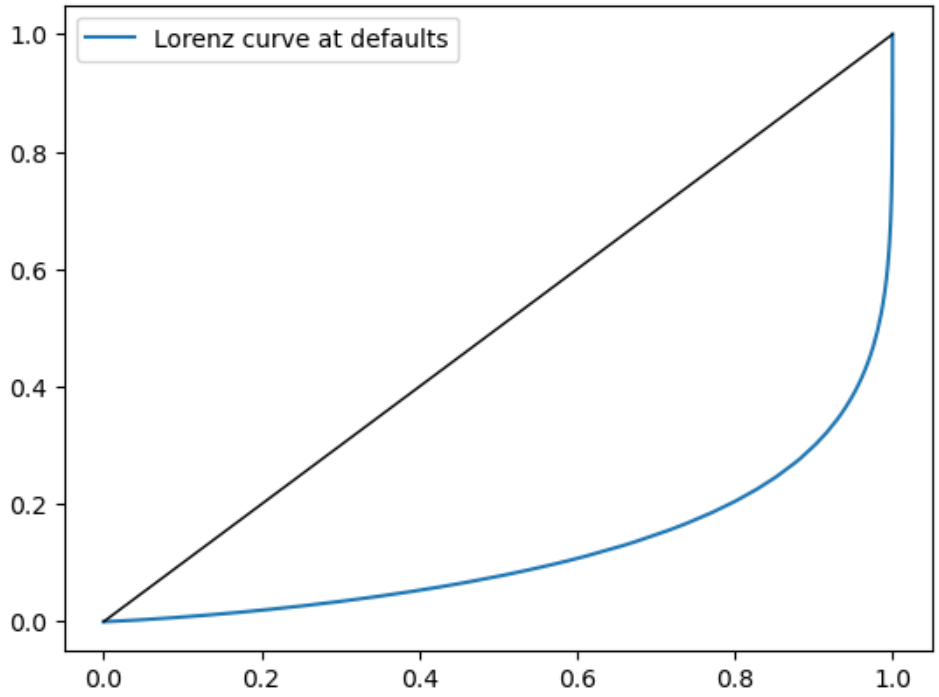
```
%time _ = lorenz_curve_jax(ψ_star, num_households)
```

```
CPU times: user 1.43 s, sys: 32.3 ms, total: 1.46 s
Wall time: 2.06 s
```

```
%time x, y = lorenz_curve_jax(ψ_star, num_households)
```

```
CPU times: user 823 µs, sys: 0 ns, total: 823 µs
Wall time: 354 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 $w_1, \ldots, w_n$, the Gini coefficient takes the form

$$G := \frac{\sum_{i=1}^n \sum_{j=1}^n |w_j - w_i|}{2n \sum_{i=1}^n w_i}. \tag{6.2}$$

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 110 ms, sys: 36 µs, total: 110 ms
Wall time: 3.08 s
```

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

```
CPU times: user 2.79 ms, sys: 11 µs, total: 2.8 ms
Wall time: 2.86 s
```

```
Array(0.758751, dtype=float32)
```

6.3 Exercises

Exercise 6.3.1

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 to Exercise 6.3.1

Here's one solution:

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

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

(continues on next page)

(continued from previous page)

```
    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()
gini
```

```
CPU times: user 108 ms, sys: 49 µs, total: 108 ms
Wall time: 2.97 s
```

```
Array(0.758751, dtype=float32)
```

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

```
CPU times: user 2.88 ms, sys: 4 µs, total: 2.88 ms
Wall time: 2.86 s
```

```
Array(0.758751, dtype=float32)
```

Exercise 6.3.2

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
```

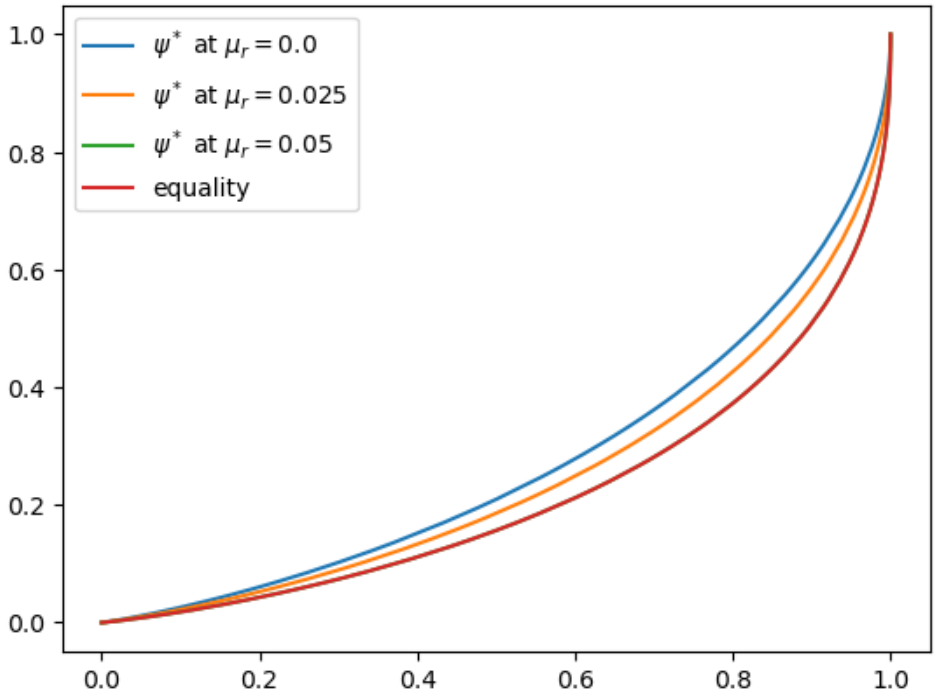
Once you have done that, plot the Gini coefficients as well.

Do the outcomes match your intuition?

Solution to Exercise 6.3.2

Here is one solution

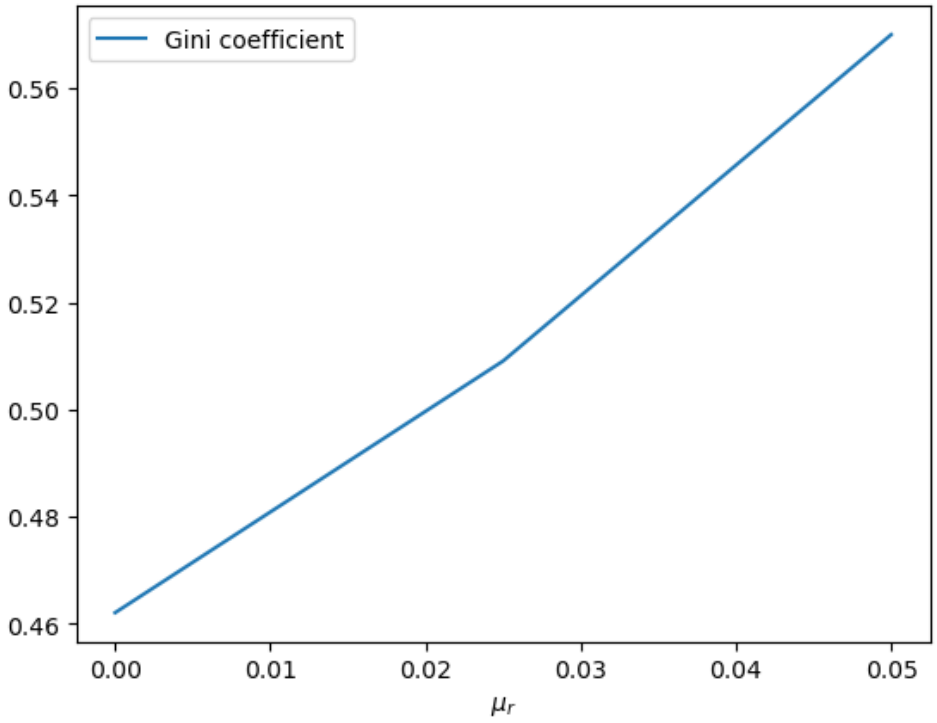
```
key = jax.random.PRNGKey(1234)
fig, ax = plt.subplots()
gini_vals = []
for μ_r in μ_r_vals:
    model = create_wealth_model(μ_r=μ_r)
    ψ_star = update_cross_section_jax_compiled(
            model, ψ_0, num_households, z_sequence, key
    )
    x, y = lorenz_curve_jax(ψ_star, num_households)
    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, y, 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 6.3.3

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 variance of the idiosyncratic term $\exp(\mu_r + \sigma_r \zeta)$ constant.)

Solution to Exercise 6.3.3

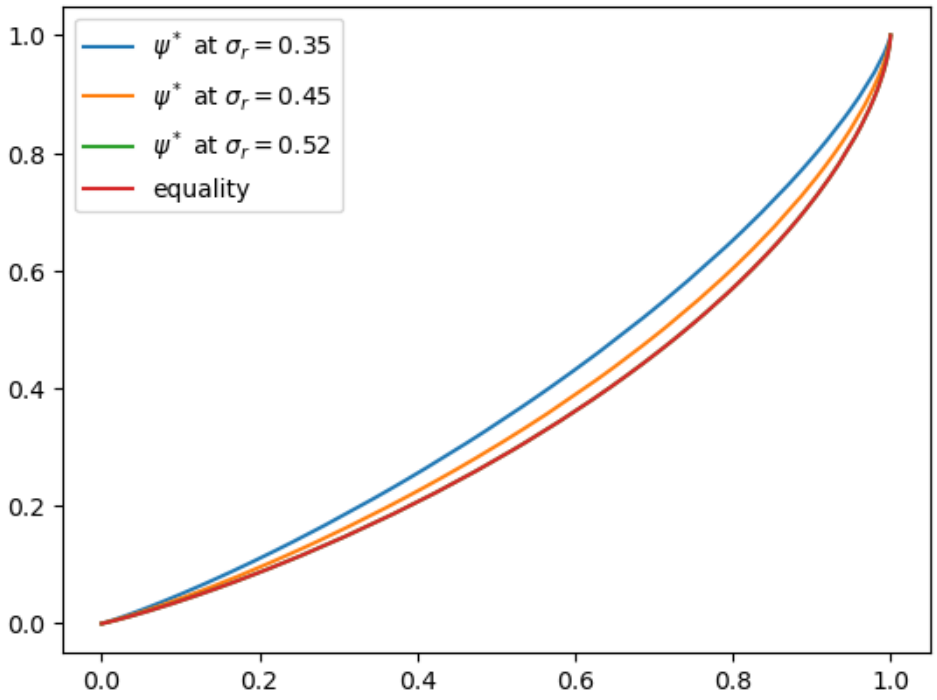
Here's one solution

```
key = jax.random.PRNGKey(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)
```

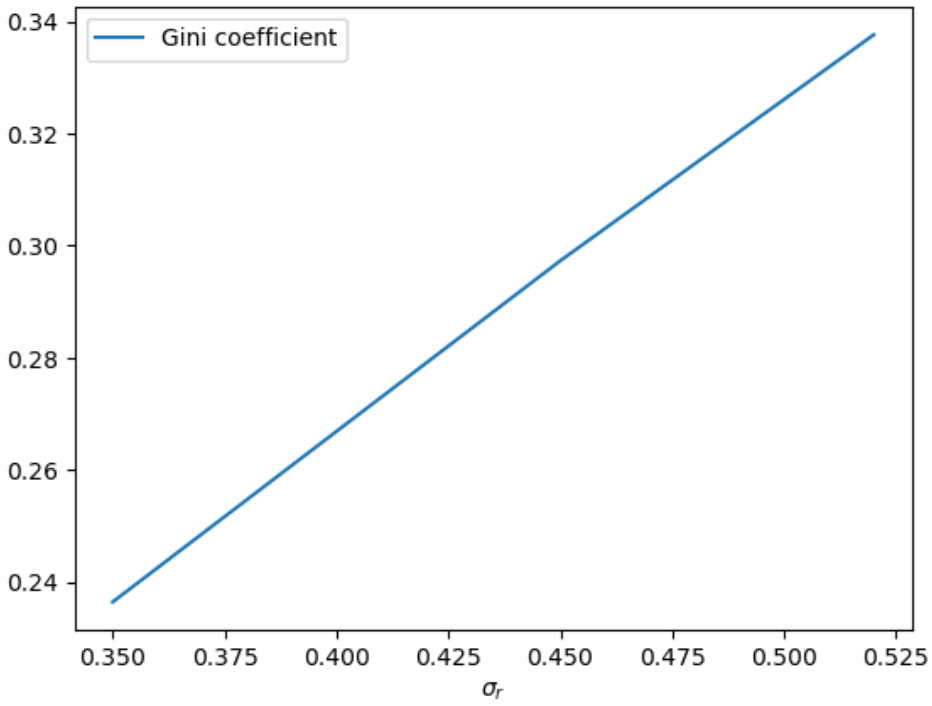
(continues on next page)

(continued from previous page)

```
    ax.plot(x, y, label=f'$\psi^*$ at $\sigma_r = {σ_r:0.2}$')
    gini_vals.append(g)
ax.plot(x, y, 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 6.3.4

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 to Exercise 6.3.4

Here's one solution.

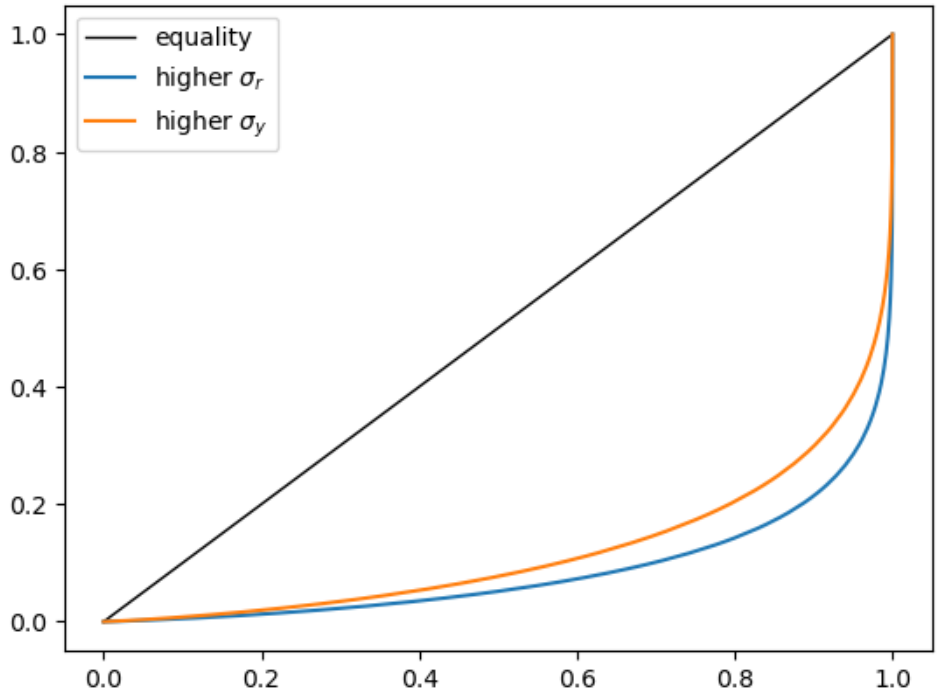
It shows that increasing volatility in financial income has a greater effect

```
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
)
```

(continues on next page)

(continued from previous page)

```
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

SEVEN

ASSET PRICING: THE LUCAS ASSET PRICING MODEL

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

Contents

7.1 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 [Luc78] 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 some imports:

```
import jax.numpy as jnp
import jax
import time
import numpy as np
import numba
from scipy.stats import lognorm
import matplotlib.pyplot as plt
```

7.2 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.

7.2.1 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{7.1}$$

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.

7.2.2 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 (7.1) 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

$$c + \pi' p(y) \leq \pi y + \pi p(y) \tag{7.2}$$

We can invoke the fact that utility is increasing to claim equality in (7.2) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\} \tag{7.3}$$

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 (7.3) 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 (7.3) 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

$$p(y) = \beta \int \frac{u'[G(y,z)]}{u'(y)} [G(y,z) + p(G(y,z))] \phi(dz) \tag{7.4}$$

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

$$p_t = \mathbb{E}_t \left[\beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \tag{7.5}$$

This is the famous consumption-based asset pricing equation.

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

7.2.3 Solving the Model

Equation (7.4) 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

$$f(y) := u'(y) p(y) \tag{7.6}$$

so that (7.4) becomes

$$f(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{7.7}$$

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

Equation (7.7) is a functional equation in f.

The plan is to solve out for f and convert back to p via (7.6).

To solve (7.7) 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

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{7.8}$$

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

The reason we do this is that a solution to (7.7) now corresponds to a function f^* satisfying $(Tf^*)(y) = f^*(y)$ for all y.

In other words, a solution is a *fixed point* of T.

This means that we can use fixed point theory to obtain and compute the solution.

A Little Fixed Point Theory

Let $cb\mathbb{R}_+$ be the set of continuous bounded functions $f\colon \mathbb{R}_+ \to \mathbb{R}_+$.

We now show that

1. T has exactly one fixed point f^* in $cb\mathbb{R}_+$.
2. For any $f \in cb\mathbb{R}_+$, the sequence $T^k f$ converges uniformly to f^*.

Note: If you find the mathematics heavy going you can take 1–2 as given and skip to the *next section*

Recall the Banach contraction mapping theorem.

It tells us that the previous statements will be true if we can find an $\alpha < 1$ such that

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \qquad \forall\, f, g \in cb\mathbb{R}_+ \tag{7.9}$$

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

To see that (7.9) is valid, pick any $f, g \in cb\mathbb{R}_+$ and any $y \in \mathbb{R}_+$.

Observe that, since integrals get larger when absolute values are moved to the inside,

$$\begin{aligned} |Tf(y) - Tg(y)| &= \left| \beta \int f[G(y,z)]\phi(dz) - \beta \int g[G(y,z)]\phi(dz) \right| \\ &\leq \beta \int |f[G(y,z)] - g[G(y,z)]|\, \phi(dz) \\ &\leq \beta \int \|f - g\| \phi(dz) \\ &= \beta \|f - g\| \end{aligned}$$

Since the right-hand side is an upper bound, taking the sup over all y on the left-hand side gives (7.9) with $\alpha := \beta$.

7.3 Computation

The preceding discussion tells that we can compute f^* by picking any arbitrary $f \in cb\mathbb{R}_+$ and then iterating with T.

The equilibrium price function p^* can then be recovered by $p^*(y) = f^*(y)/u'(y)$.

Let's try this when $\ln y_{t+1} = \alpha \ln y_t + \sigma \epsilon_{t+1}$ where $\{\epsilon_t\}$ is IID and standard normal.

Utility will take the isoelastic form $u(c) = c^{1-\gamma}/(1-\gamma)$, where $\gamma > 0$ is the coefficient of relative risk aversion.

We'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).

7.3.1 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)
        ϕ = lognorm(σ)
        draws = ϕ.rvs(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)

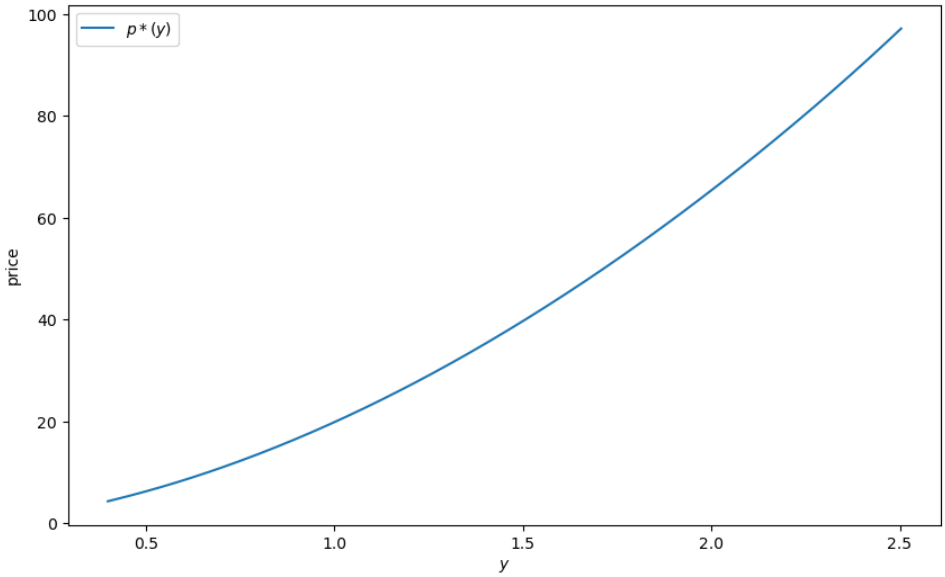
```
    """
    # Unpack
    γ, β, α, σ = params
    grid, draws, h = arrays
    # Set up and loop
    i = 0
    f = np.ones_like(grid)  # Initial guess of f
    error = tol + 1
    while error > tol and i < max_iter:
        Tf = T(params, arrays, f)
        error = np.max(np.abs(Tf - f))
        f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

# Now time execution without compile time
in_time = time.time()
price_vals = solve_model(params, arrays)
out_time = time.time()
numba_elapsed = out_time - in_time
print("Numba execution time = ", numba_elapsed)
```

```
Numba execution time =  4.230381965637207
```

```
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).

7.3.2 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):
    """
    Compute the equilibrium price function.

```

(continues on next page)

(continued from previous page)

```
    """
    # 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
```

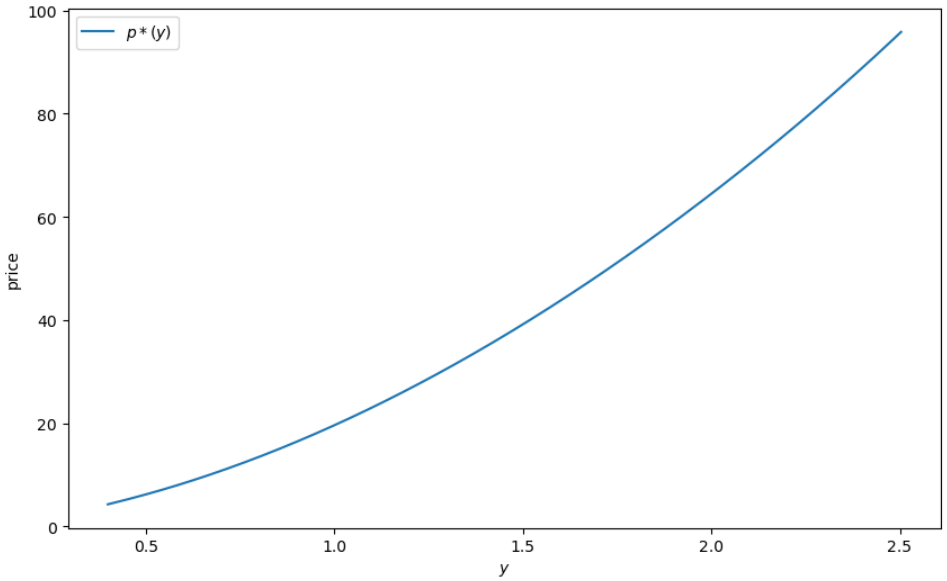
Now let's solve the model again and compare timing

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

# Solve once to compile
price_vals = solve_model(params, arrays)

# Now time execution without compile time
in_time = time.time()
price_vals = solve_model(params, arrays)
out_time = time.time()
jax_elapsed = out_time - in_time
print("JAX execution time = ", jax_elapsed)
print("Speedup factor = ", numba_elapsed / jax_elapsed)
```

```
JAX execution time =  0.4268157482147217
Speedup factor =  9.911494557855429
```

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()
```



7.4 Exercises

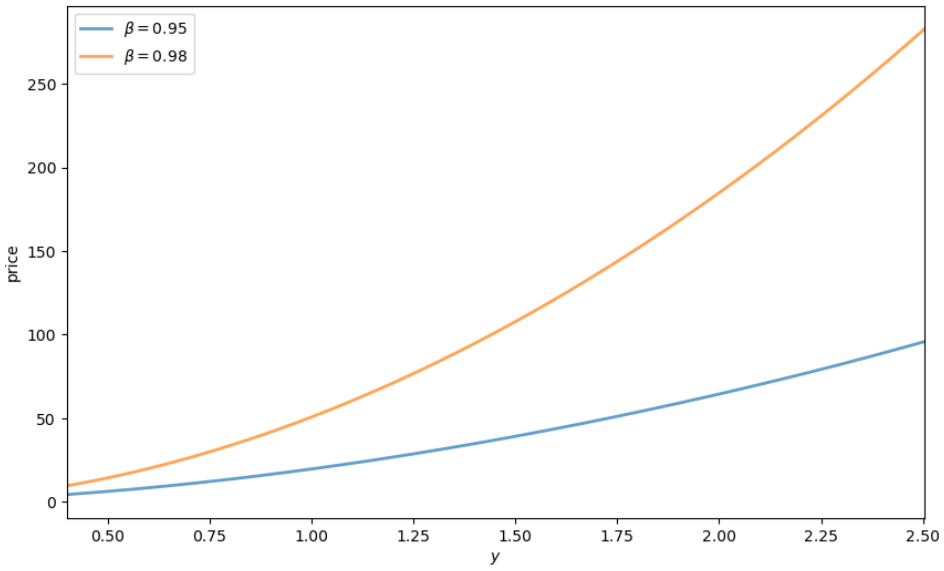
Exercise 7.4.1

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 to Exercise 7.4.1

```
fig, ax = plt.subplots(figsize=(10, 6))

for β in (.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
0
50
100
150
200
250
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50

CHAPTER

EIGHT

AN ASSET PRICING PROBLEM

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

8.1 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 *(8.5.5)*.

The code outputs below are generated by machine connected to the following GPU

```
!nvidia-smi
```

```
Mon Apr  1 17:51:43 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   28C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
```

(continues on next page)

(continued from previous page)

```
|                               |                      |                  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
```

We will use 64 bit floats with JAX in order to increase precision.

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

8.2 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

$$P_t = \beta \, \mathbb{E}_t D_{t+1} \tag{8.1}$$

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, *(8.2.1)* 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 *(8.2.1)* to

$$P_t = \mathbb{E}_t M_{t+1} D_{t+1} \tag{8.2}$$

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.

8.3 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 *(8.2.2)*, the price should be

$$P_t = \mathbb{E}_t M_{t+1}[D_{t+1} + P_{t+1}] \tag{8.3}$$

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 (8.3) by D_t to get

$$V_t = \mathbb{E}_t \left[M_{t+1} \frac{D_{t+1}}{D_t} (1 + V_{t+1}) \right] \tag{8.4}$$

We can also write this as

$$V_t = \mathbb{E}_t \left[M_{t+1} \exp(G^d_{t+1}) (1 + V_{t+1}) \right] \tag{8.5}$$

where

$$G^d_{t+1} = \ln \frac{D_{t+1}}{D_t}$$

is the growth rate of dividends.

Our aim is to solve *(8.3.3)* 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}

8.4 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

$$M_{t+1} = \beta \frac{u'(C_{t+1})}{u'(C_t)} \tag{8.6}$$

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

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \tag{8.7}$$

Inserting the CRRA specification into (8.6) and letting

$$G_{t+1}^c = \ln\left(\frac{C_{t+1}}{C_t}\right)$$

the growth rate rate of consumption, we obtain

$$M_{t+1} = \beta\left(\frac{C_{t+1}}{C_t}\right)^{-\gamma} = \beta\exp(G_{t+1}^c)^{-\gamma} = \beta\exp(-\gamma G_{t+1}^c) \tag{8.8}$$

8.5 Solving for the price-dividend ratio

Substituting *(8.4.3)* into (8.5) gives the price-dividend ratio formula

$$V_t = \beta\mathbb{E}_t\left[\exp(G_{t+1}^d - \gamma G_{t+1}^c)(1+V_{t+1})\right] \tag{8.9}$$

We assume there is a Markov chain $\{X_t\}$, which we call the **state process**, such that

$$G_{t+1}^c = \mu_c + X_t + \sigma_c\epsilon_{c,t+1}$$
$$G_{t+1}^d = \mu_d + X_t + \sigma_d\epsilon_{d,t+1}$$

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 *(8.5.1)*, the unknown function v satisfies the equation

$$v(X_t) = \beta\mathbb{E}_t\left\{\exp[a + (1-\gamma)X_t + \sigma_d\epsilon_{d,t+1} - \gamma\sigma_c\epsilon_{c,t+1}](1+v(X_{t+1}))\right\} \tag{8.10}$$

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

$$v(X_t) = \beta\mathbb{E}_t\left\{\exp\left[a + (1-\gamma)X_t + \frac{\sigma_d^2 + \gamma^2\sigma_c^2}{2}\right](1+v(X_{t+1}))\right\} \tag{8.11}$$

Conditioning on $X_t = x$, we can write this as

$$v(x) = \beta\sum_{y\in S}\left\{\exp\left[a + (1-\gamma)x + \frac{\sigma_d^2 + \gamma^2\sigma_c^2}{2}\right](1+v(y))\right\}P(x,y) \tag{8.12}$$

for all $x \in S$.

Suppose $S = \{x_1, \ldots, x_N\}$.

Then we can think of v as an N-vector and, using square brackets for indices on arrays, write

$$v[i] = \beta\sum_{j=1}^{N}\left\{\exp\left[a + (1-\gamma)x[i] + \frac{\sigma_d^2 + \gamma^2\sigma_c^2}{2}\right](1+v[j])\right\}P[i,j] \tag{8.13}$$

for $i = 1, \ldots, N$.

Equivalently, we can write

$$v[i] = \sum_{j=1}^{N} K[i,j](1 + v[j]) \tag{8.14}$$

where K is the matrix defined by

$$K[i,j] = \beta \left\{ \exp\left[a + (1-\gamma)x[i] + \frac{\sigma_d^2 + \gamma^2 \sigma_c^2}{2} \right] \right\} P[i,j] \tag{8.15}$$

Rewriting *(8.5.6)* in vector form yields

$$v = K(\mathbb{1} + v) \tag{8.16}$$

Notice that *(8.5.8)* 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

$$v = (I - K)^{-1} K\mathbb{1} \tag{8.17}$$

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 *(8.5.9)*.

8.6 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.PRNGKey(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
```

(continues on next page)

(continued from previous page)

```
        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
    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 *(8.5.7)*.

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))
```

(continues on next page)

(continued from previous page)

```
    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

    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)
```

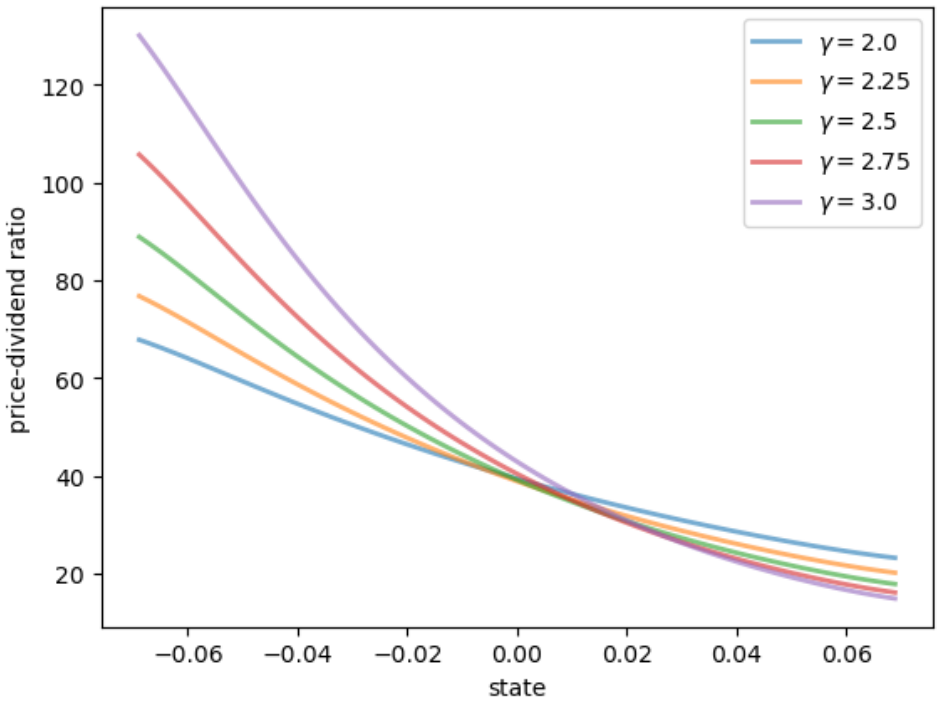
(continues on next page)

(continued from previous page)

```
    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 (8.8), higher growth decreases the discount factor, lowering the weight placed on future dividends.

8.7 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.

$$G_{t+1}^i = \mu_i + Z_t + \bar{\sigma} \exp(H_t^i)\epsilon_{i,t+1}, \qquad i \in \{c, d\}$$

where $\{Z_t\}$ is a finite Markov chain and $\{H_t^c\}$ and $\{H_t^d\}$ are volatility processes.

We assume that $\{H_t^c\}$ and $\{H_t^d\}$ are AR(1) processes of the form

$$H_{t+1}^i = \rho_i H_t^i + \sigma_i \eta_{i,t+1}, \qquad i \in \{c, d\}$$

Here $\{\eta_t^c\}$ and $\{\eta_t^d\}$ are IID and standard normal.

Let $X_t = (H_t^c, H_t^d, Z_t)$.

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 *(8.5.2)* to accommodate the new growth specifications, we find that v satisfies

$$\begin{aligned} v(X_t) =& \beta \times \\ & \mathbb{E}_t \left\{ \exp[a + (1-\gamma)Z_t + \bar{\sigma} \exp(H_t^d)\epsilon_{d,t+1} - \gamma \bar{\sigma} \exp(H_t^c)\epsilon_{c,t+1}](1 + v(X_{t+1})) \right\} \end{aligned} \tag{8.18}$$

where, as before, $a := \mu_d - \gamma \mu_c$

Conditioning on state $x = (h_c, h_d, z)$, this becomes

$$v(x) = \beta \mathbb{E}_t \exp[a + (1-\gamma)z + \bar{\sigma} \exp(h_d)\epsilon_{d,t+1} - \gamma \bar{\sigma} \exp(h_c)\epsilon_{c,t+1}](1 + v(X_{t+1})) \tag{8.19}$$

As before, we integrate out the independent shocks and use the rules for expectations of lognormals to obtain

$$v(x) = \beta \mathbb{E}_t \exp\left[a + (1-\gamma)z + \bar{\sigma}^2 \frac{\exp(2h_d) + \gamma^2 \exp(2h_c)}{2}\right](1 + v(X_{t+1})) \tag{8.20}$$

Let

$$\begin{aligned} A(h_c, h_d, z, h_c', h_d', z') := \\ \beta \exp\left[a + (1-\gamma)z + \bar{\sigma}^2 \frac{\exp(2h_d) + \gamma^2 \exp(2h_c)}{2}\right] P(h_c, h_c') Q(h_d, h_d') R(z, z') \end{aligned}$$

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 *(8.7.3)* more explicitly as

$$v(h_c, h_d, z) = \sum_{h_c', h_d', z'} (1 + v(h_c', h_d', z'))A(h_c, h_d, z, h_c', h_d', z') \tag{8.21}$$

Let's now write the state using indices, with (i, j, k) being the indices for (h_c, h_d, z).

Then (8.21) becomes

$$v[i, j, k] = \sum_{i', j', k'} A[i, j, k, i', j', k'](1 + v[i', j', k']) \tag{8.22}$$

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 *(8.7.5)* as

$$v = A(\mathbb{1} + v)$$

Provided that the spectral radius condition $r(A) < 1$ holds, the solution is given by

$$v = (I - A)^{-1} A \mathbb{1}$$

8.8 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.

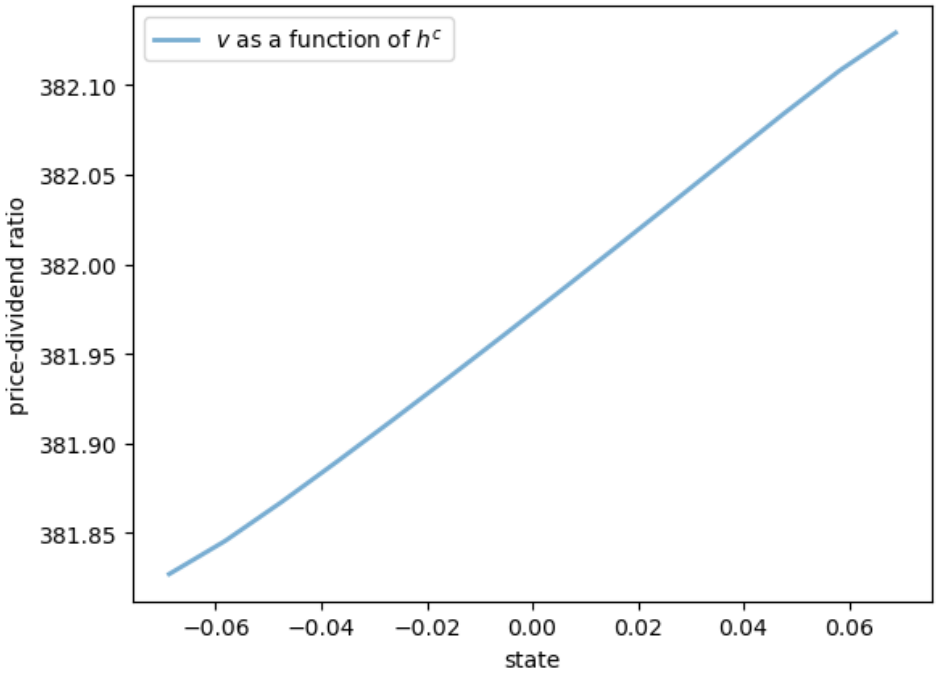
```
sv_model = create_sv_model()
P, hc_grid, Q, hd_grid, R, z_grid, β, γ, bar_σ, μ_c, μ_d = sv_model
```

```
qe.tic()
v = sv_pd_ratio(sv_model)
np_time = qe.toc()
```

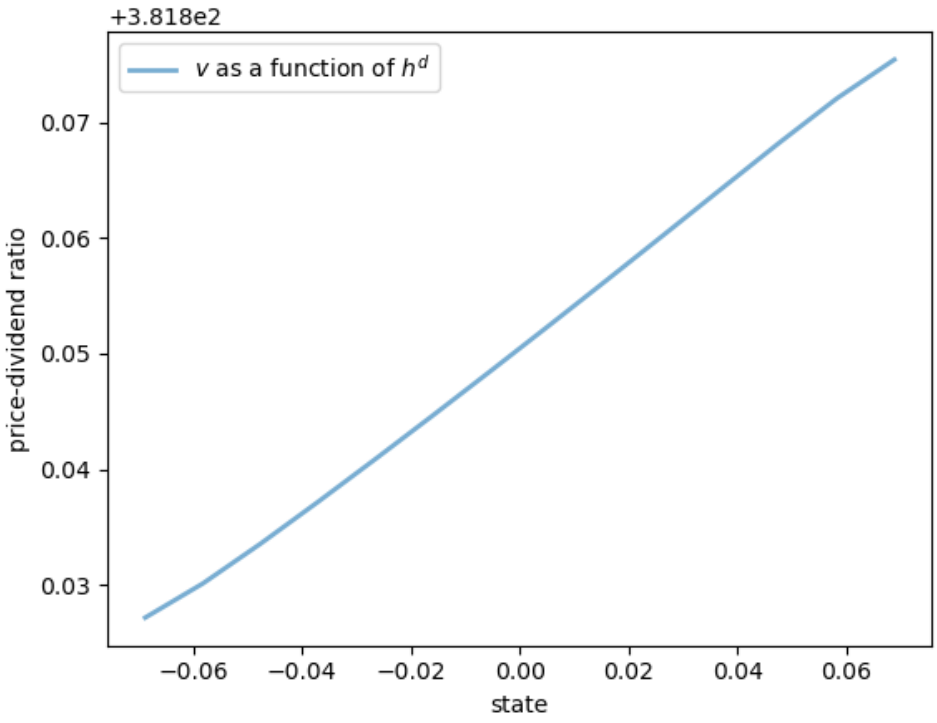
```
TOC: Elapsed: 0:00:0.99
```

Here are some plots of the solution v along the three dimensions.

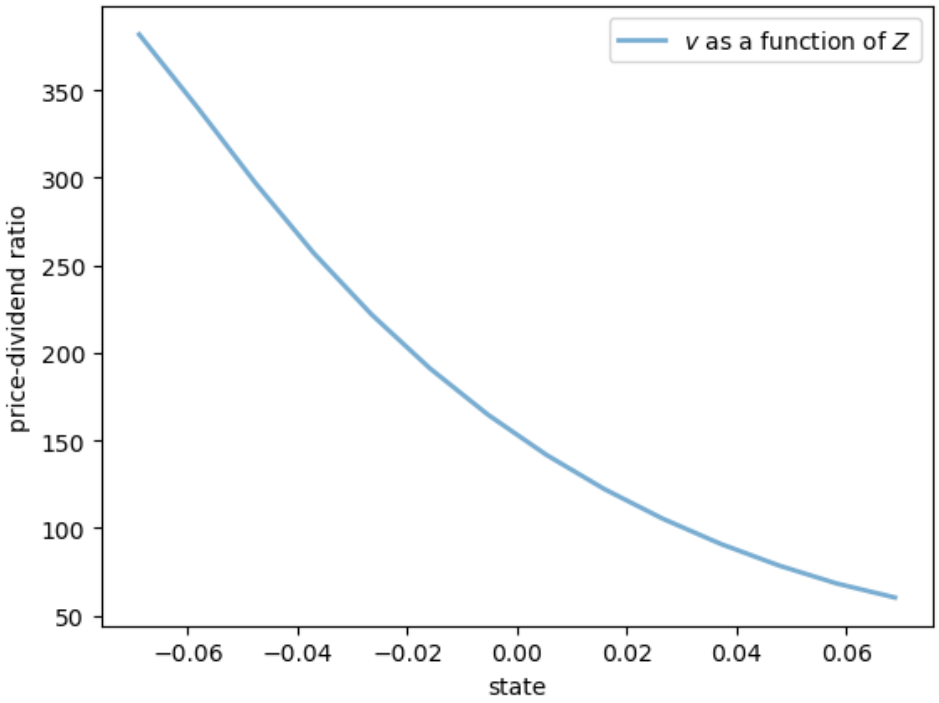
```
fig, ax = plt.subplots()
ax.plot(hc_grid, v[:, 0, 0], lw=2, alpha=0.6, label="$v$ as a function of $h^c$")
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend()
plt.show()
```




```
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()
```



8.9 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
```

(continues on next page)

(continued from previous page)

```
        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)
    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.

```
qe.tic()
v_jax = sv_pd_ratio_jax(sv_model_jax, shapes).block_until_ready()
jnp_time_0 = qe.toc()
```

```
TOC: Elapsed: 0:00:0.30
```

And now let's see without compile time.

```
qe.tic()
v_jax = sv_pd_ratio_jax(sv_model_jax, shapes).block_until_ready()
jnp_time = qe.toc()
```

```
TOC: Elapsed: 0:00:0.01
```

Here's the ratio of times:

```
jnp_time / np_time
```

```
0.017175145376472235
```

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
```

8.10 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.

```
qe.tic()
v_jax_linop = sv_pd_ratio_linop(sv_model, shapes).block_until_ready()
jnp_time_linop_0 = qe.toc()
```

```
TOC: Elapsed: 0:00:0.83
```

And now let's see without compile time.

```
qe.tic()
v_jax_linop = sv_pd_ratio_linop(sv_model, shapes).block_until_ready()
jnp_linop_time = qe.toc()
```

```
TOC: Elapsed: 0:00:0.00
```

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_time / jnp_time
```

```
0.152668011882742
```

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)

qe.tic()
_ = sv_pd_ratio_linop(sv_model, shapes).block_until_ready()
qe.toc()
```

```
TOC: Elapsed: 0:00:0.87
```

```
0.8738894462585449
```

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

NINE

OPTIMAL SAVINGS I: VALUE FUNCTION ITERATION

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

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
from collections import namedtuple
import matplotlib.pyplot as plt
import time
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:55:27 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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)
```

9.1 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.

9.2 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.)

9.2.1 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′).
    """

    # Unpack
    β, R, γ = params
    w_size, y_size = sizes
    w_grid, y_grid, Q = arrays
```

(continues on next page)

(continued from previous page)

```
    # 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)
```

9.2.2 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
params, sizes, arrays = model
β, R, γ = params
w_size, y_size = sizes
w_grid, y_grid, Q = arrays

print("Starting VFI.")
```

(continues on next page)

(continued from previous page)

```
start_time = time.time()
v_star, σ_star = value_function_iteration(model)
numpy_elapsed = time.time() - start_time
print(f"VFI completed in {numpy_elapsed} seconds.")
```

```
Starting VFI.
```

```
VFI completed in 22.65413236618042 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()
```

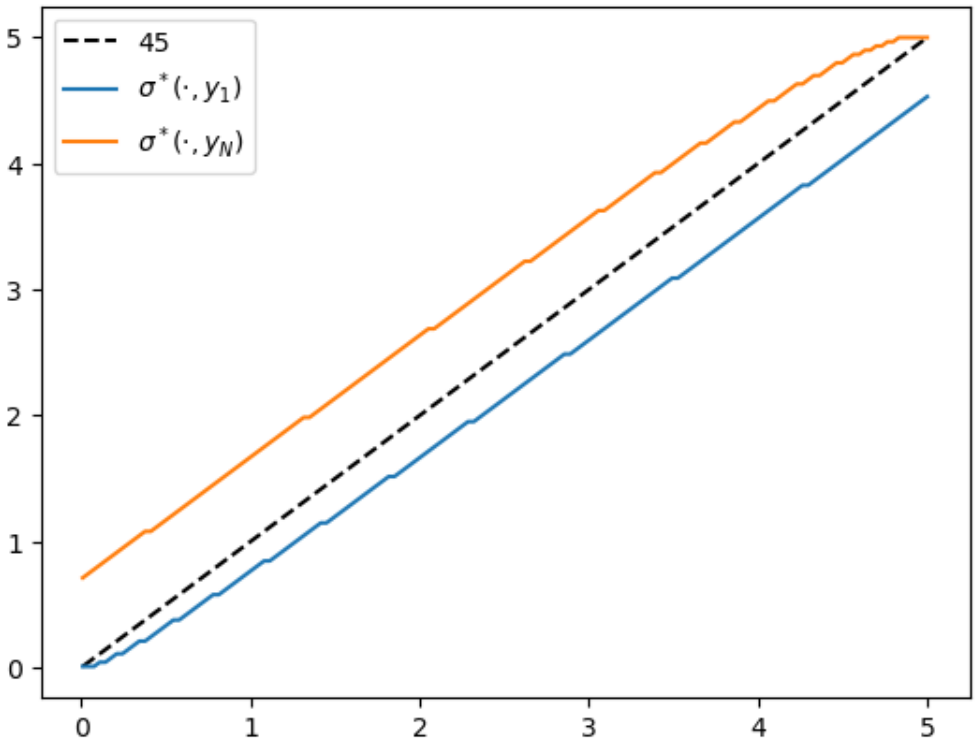


Fig. 9.1: Policy function

9.3 Switching to JAX

To switch over to JAX, we change `np` to `jnp` throughout and add some `jax.jit` requests.

9.3.1 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.
    """
    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]
    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,))
```

9.3.2 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
                          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,
```

(continues on next page)

(continued from previous page)

```
                                    (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)
```

9.3.3 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 = time.time()
v_star_jax, σ_star_jax = value_function_iteration(model)
jax_elapsed = time.time() - start_time
print(f"VFI completed in {jax_elapsed} seconds.")
```

```
Starting VFI using vectorization.
```

```
VFI completed in 0.49997925758361816 seconds.
```

The relative speed gain is

```
print(f"Relative speed gain = {numpy_elapsed / jax_elapsed}")
```

```
Relative speed gain = 45.31014441612444
```

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*.

9.4 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)
```

(continues on next page)

(continued from previous page)

```
    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 = time.time()
v_star_vmap, σ_star_vmap = value_iteration_vmap(model)
jax_vmap_elapsed = time.time() - start_time
print(f"VFI completed in {jax_vmap_elapsed} seconds.")
```

```
Starting VFI using vmap.
```

```
VFI completed in 0.3051910400390625 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_elapsed / jax_vmap_elapsed}")
```

```
Relative speed = 74.2293494700265
```

And here's the comparison with the first JAX implementation (which used direct vectorization).

```
print(f"Relative speed = {jax_elapsed / jax_vmap_elapsed}")
```

```
Relative speed = 1.6382501187440628
```

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

TEN

OPTIMAL SAVINGS II: ALTERNATIVE ALGORITHMS

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

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
from collections import namedtuple
import matplotlib.pyplot as plt
import time
```

Let's check the GPU we are running.

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 22:27:10 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   33C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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)
```

10.1 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
```

(continues on next page)

(continued from previous page)

```
                            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)

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

10.2 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
    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):
    """
    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
```

(continues on next page)

(continued from previous page)

```
    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,))
```

10.3 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

    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,
```

(continues on next page)

(continued from previous page)

```
                                    (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,))
```

10.4 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)

    def update(inputs):
        i, v, error = inputs
        last_v = v
        σ = get_greedy(v, params, sizes, arrays)
```

(continues on next page)

(continued from previous page)

```
        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
```

10.5 Plots

Create a model for consumption, perform policy iteration, and plot the resulting optimal policy function.

```
model = create_consumption_model()
# Unpack
params, sizes, arrays = model
β, R, γ = params
w_size, y_size = sizes
w_grid, y_grid, Q = arrays
```

```
σ_star = howard_policy_iteration(model)

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

(continues on next page)

(continued from previous page)

```
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()
```

```
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.
```

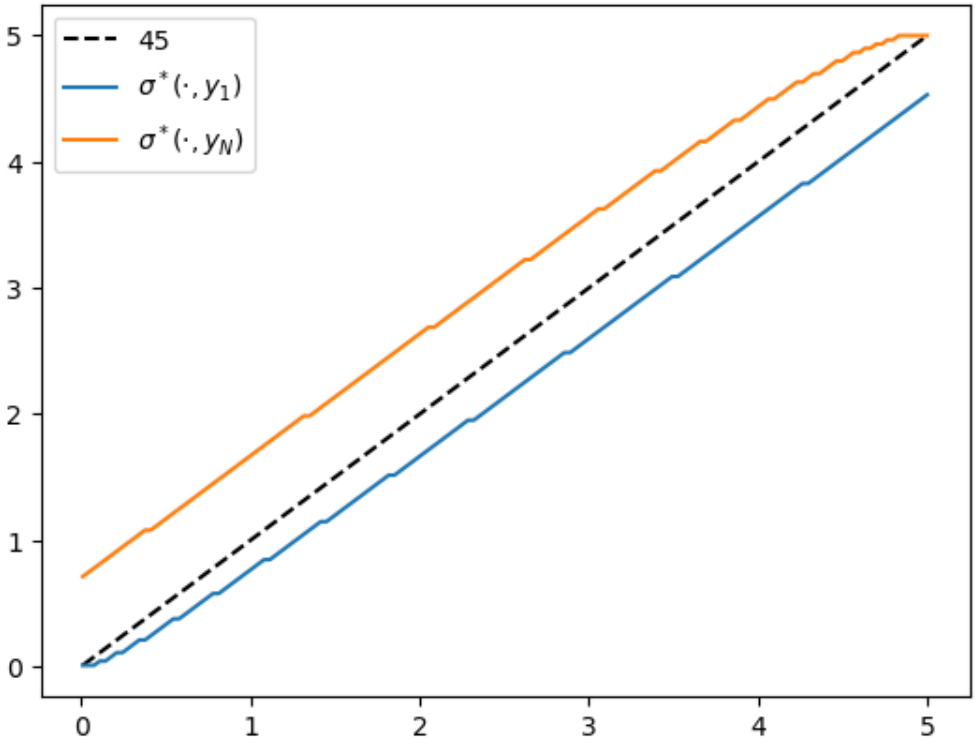


Fig. 10.1: Optimal policy function

10.6 Tests

Here's a quick test of the timing of each solver.

```
model = create_consumption_model()
```

```
print("Starting HPI.")
start_time = time.time()
out = howard_policy_iteration(model)
elapsed = time.time() - start_time
print(f"HPI completed in {elapsed} 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 0.035977840423583984 seconds.
```

```
print("Starting VFI.")
start_time = time.time()
out = value_function_iteration(model)
elapsed = time.time() - start_time
print(f"VFI completed in {elapsed} seconds.")
```

```
Starting VFI.
```

```
VFI completed in 0.2825751304626465 seconds.
```

```
print("Starting OPI.")
start_time = time.time()
out = optimistic_policy_iteration(model, m=100)
elapsed = time.time() - start_time
print(f"OPI completed in {elapsed} seconds.")
```

```
Starting OPI.
```

```
OPI completed in 0.5461685657501221 seconds.
```

```
def run_algorithm(algorithm, model, **kwargs):
    start_time = time.time()
    result = algorithm(model, **kwargs)
    end_time = time.time()
    elapsed_time = end_time - start_time
```

(continues on next page)

(continued from previous page)

```
    print(f"{algorithm.__name__} completed in {elapsed_time:.2f} seconds.")
    return result, elapsed_time
```

```
model = create_consumption_model()
σ_pi, pi_time = run_algorithm(howard_policy_iteration, model)
σ_vfi, vfi_time = run_algorithm(value_function_iteration, model, tol=1e-5)

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)
```

```
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.
howard_policy_iteration completed in 0.04 seconds.
```

```
value_function_iteration completed in 0.17 seconds.
optimistic_policy_iteration completed in 0.04 seconds.
optimistic_policy_iteration completed in 0.03 seconds.
optimistic_policy_iteration completed in 0.03 seconds.
optimistic_policy_iteration completed in 0.04 seconds.
optimistic_policy_iteration completed in 0.05 seconds.
```

```
optimistic_policy_iteration completed in 0.06 seconds.
optimistic_policy_iteration completed in 0.07 seconds.
optimistic_policy_iteration completed in 0.08 seconds.
```

```
optimistic_policy_iteration completed in 0.10 seconds.
optimistic_policy_iteration completed in 0.11 seconds.
```

```
optimistic_policy_iteration completed in 0.12 seconds.
optimistic_policy_iteration completed in 0.13 seconds.
```

```
optimistic_policy_iteration completed in 0.14 seconds.
optimistic_policy_iteration completed in 0.15 seconds.
```

```
optimistic_policy_iteration completed in 0.16 seconds.
```

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

(continues on next page)

(continued from previous page)

```
        jnp.full(len(m_vals), pi_time),
        lw=2, label="Howard policy iteration")
ax.plot(m_vals,
        jnp.full(len(m_vals), vfi_time),
        lw=2, label="value function iteration")
ax.plot(m_vals, opi_times,
        lw=2, label="optimistic policy iteration")
ax.legend(frameon=False)
ax.set_xlabel("$m$")
ax.set_ylabel("time")
plt.show()
```

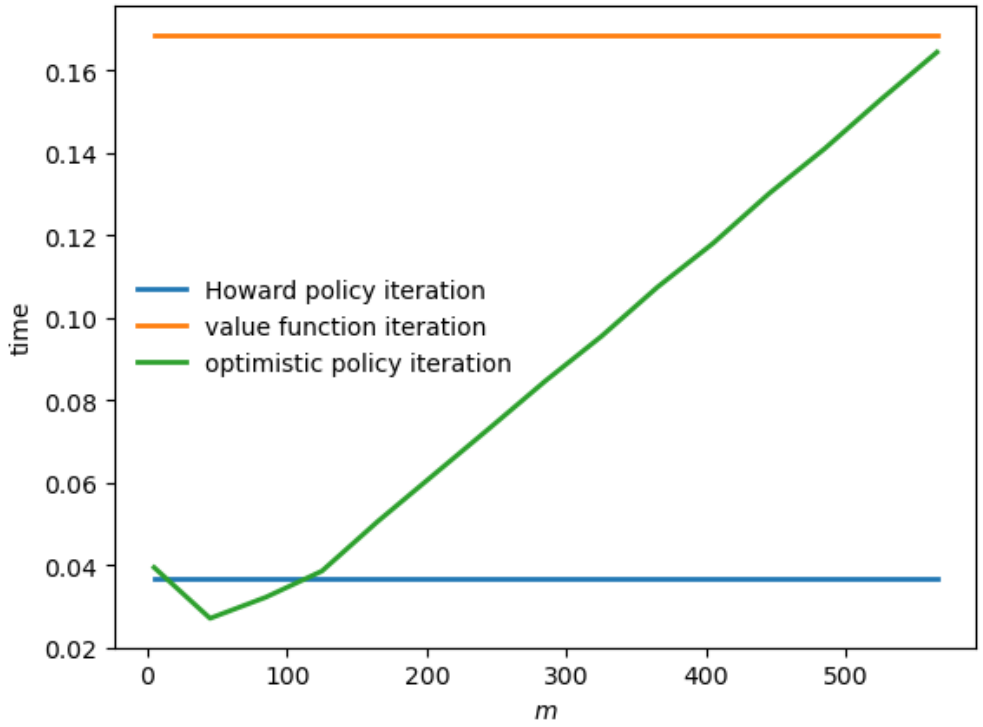


Fig. 10.2: Solver times

CHAPTER

ELEVEN

SHORTEST PATHS

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

11.1 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
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:56:17 2024
+---------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

11.2 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.

11.2.1 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

$$J_0(v) = 0 \text{ for all } v \tag{11.1}$$

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],
               [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 125 ms, sys: 19.6 ms, total: 145 ms
Wall time: 191 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]
```

```
CPU times: user 80.1 ms, sys: 6.15 ms, total: 86.2 ms
Wall time: 93.3 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]
```

```
CPU times: user 1.08 ms, sys: 1.1 ms, total: 2.18 ms
Wall time: 959 µs
```

```
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.

11.3 Exercises

Exercise 11.3.1

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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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,
```

```
Overwriting graph.txt
```

Solution to Exercise 11.3.1

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:])
```

(continues on next page)

(continued from previous page)

```
                    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 91.9 ms, sys: 12.7 ms, total: 105 ms
Wall time: 119 ms
```

```
%%time
J = compute_cost_to_go(Q).block_until_ready()
```

```
CPU times: user 1.39 ms, sys: 0 ns, total: 1.39 ms
Wall time: 850 µs
```

```
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

TWELVE

OPTIMAL INVESTMENT

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

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

$$P_t Y_t - c Y_t - \gamma (Y_{t+1} - Y_t)^2$$

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
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 22:26:52 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   33C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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)
```

(continues on next page)

(continued from previous page)

```
    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

# 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):
    """
```

(continues on next page)

(continued from previous page)

```
    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

    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]
```

(continues on next page)

(continued from previous page)

```
    # 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]
    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.
```

(continues on next page)

(continued from previous page)

```
    """
    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)
    i, error = 0, 1.0
    while error > 0 and i < maxiter:
```

(continues on next page)

(continued from previous page)

```
        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()
print("Starting HPI.")
qe.tic()
out = howard_policy_iteration(model)
elapsed = qe.toc()
print(out)
print(f"HPI completed in {elapsed} seconds.")
```

```
print("Starting VFI.")
qe.tic()
out = value_function_iteration(model)
elapsed = qe.toc()
print(out)
print(f"VFI completed in {elapsed} seconds.")
```

```
print("Starting OPI.")
qe.tic()
out = optimistic_policy_iteration(model, m=100)
elapsed = qe.toc()
print(out)
print(f"OPI completed in {elapsed} seconds.")
```

Here's the plot of the Howard policy, as a function of y at the highest and lowest values of z.

```
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
```

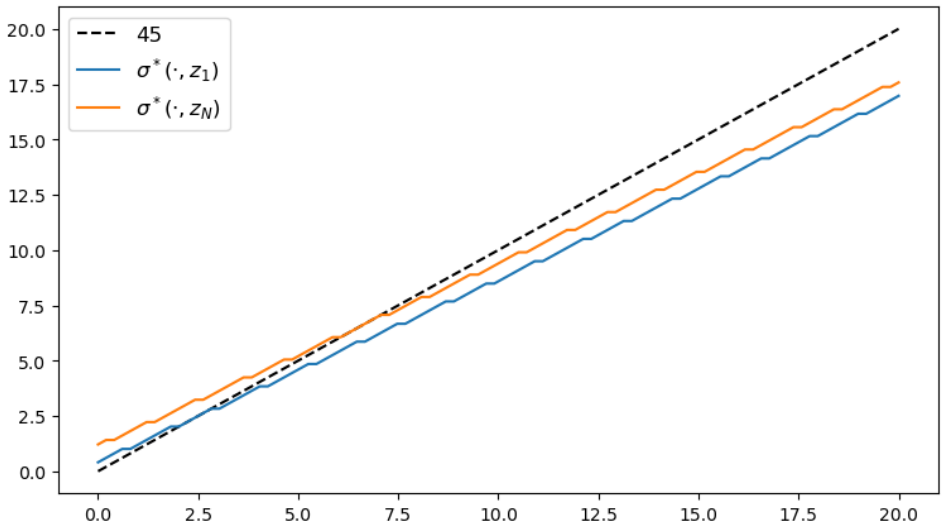
```
σ_star = howard_policy_iteration(model)

fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(y_grid, y_grid, "k--", label="45")
ax.plot(y_grid, y_grid[σ_star[:, 1]], label="$\\sigma^*(\cdot, z_1)$")
ax.plot(y_grid, y_grid[σ_star[:, -1]], label="$\\sigma^*(\cdot, z_N)$")
ax.legend(fontsize=12)
plt.show()
```

```
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.
```

(continues on next page)

(continued from previous page)

```
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.
```



Let's plot the time taken by each of the solvers and compare them.

```
m_vals = range(5, 600, 40)
```

```
model = create_investment_model()
print("Running Howard policy iteration.")
qe.tic()
σ_pi = howard_policy_iteration(model)
pi_time = qe.toc()
```

```
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.
TOC: Elapsed: 0:00:0.06
```

```
print(f"PI completed in {pi_time} seconds.")
print("Running value function iteration.")
```

(continues on next page)

(continued from previous page)

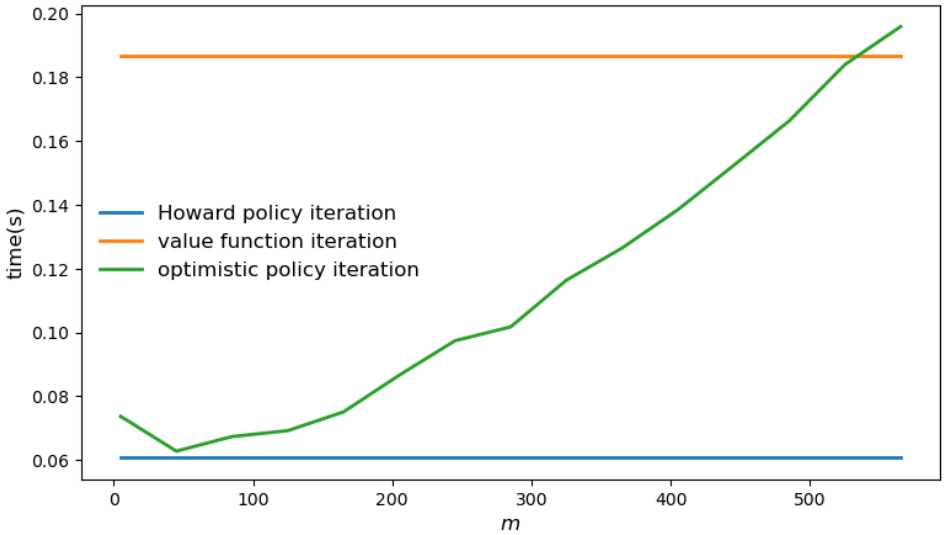
```
qe.tic()
σ_vfi = value_function_iteration(model, tol=1e-5)
vfi_time = qe.toc()
print(f"VFI completed in {vfi_time} seconds.")
```

```
PI completed in 0.060736656188964844 seconds.
Running value function iteration.
```

```
TOC: Elapsed: 0:00:0.18
VFI completed in 0.1866285800933838 seconds.
```

```
opi_times = []
for m in m_vals:
    print(f"Running optimistic policy iteration with m={m}.")
    qe.tic()
    σ_opi = optimistic_policy_iteration(model, m=m, tol=1e-5)
    opi_time = qe.toc()
    print(f"OPI with m={m} completed in {opi_time} seconds.")
    opi_times.append(opi_time)
```

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(m_vals, jnp.full(len(m_vals), pi_time),
        lw=2, label="Howard policy iteration")
ax.plot(m_vals, jnp.full(len(m_vals), vfi_time),
        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()
```



CHAPTER

THIRTEEN

INVENTORY MANAGEMENT MODEL

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

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
```

13.1 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

$$V_0 = \sum_{t \geq 0} \beta^t \pi_t \qquad \text{where} \quad \beta := \frac{1}{1+r}.$$

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

$$X_{t+1} = f(X_t, D_{t+1}, A_t) \qquad \text{where} \quad f(x, a, d) := (x - d) \vee 0 + a.$$

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

$$v(x) = \max_{a \in \Gamma(x)} \left\{ r(x, a) + \beta \sum_{d \geq 0} v(f(x, a, d))\phi(d) \right\} \tag{13.1}$$

13.2 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 (13.1) 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

$$B((y, z), a, v) = r(y, a) + \beta(z) \sum_{d,\, z'} v(f(y, a, d), z')\phi(d)Q(z, z'). \tag{13.2}$$

We set

$$R(y, a, y') := P\{f(y, a, d) = y'\} \quad \text{when} \quad D \sim \phi,$$

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 (13.2) 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 collections import namedtuple
import time
from numba import njit, prange
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:25:46 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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
Model = namedtuple("Model", ("c", "κ", "p", "z_vals", "Q"))
```

We need the following successive approximation function.

```
def successive_approx(T,                    # Operator (callable)
                      x_0,                  # Initial condition
```

(continues on next page)

(continued from previous page)

```
                    tolerance=1e-6,         # Error tolerance
                    max_iter=10_000,        # Max iteration bound
                    print_step=25,          # Print at multiples
                    verbose=False):
    x = x_0
    error = tolerance + 1
    k = 1
    while error > tolerance and k <= max_iter:
        x_new = T(x)
        error = jnp.max(jnp.abs(x_new - x))
        if verbose and k % print_step == 0:
            print(f"Completed iteration {k} with error {error}.")
        x = x_new
        k += 1
    if error > tolerance:
        print(f"Warning: Iteration hit upper bound {max_iter}.")
    elif verbose:
        print(f"Terminated successfully in {k} iterations.")
    return x
```

```
@jax.jit
def demand_pdf(p, d):
    return (1 - p)**d * p
```

```
K = 100
D_MAX = 101
```

Let's define a function to create an inventory model using the given parameters.

```
def create_sdd_inventory_model(
        ρ=0.98, ν=0.002, n_z=100, b=0.97,          # Z state parameters
        c=0.2, κ=0.8, p=0.6,                       # firm and demand parameters
        use_jax=True):
    mc = qe.tauchen(n_z, ρ, ν)
    z_vals, Q = mc.state_values + b, mc.P
    if use_jax:
        z_vals, Q = map(jnp.array, (z_vals, Q))
    return Model(c=c, κ=κ, p=p, z_vals=z_vals, Q=Q)
```

Here's the function `B` on the right-hand side of the Bellman equation.

```
@jax.jit
def B(x, i_z, a, v, model):
    """
    The function B(x, z, a, v) = r(x, a) + β(z) Σ_x' v(x') P(x, a, x').
    """
    c, κ, p, z_vals, Q = model
    z = z_vals[i_z]
    d_vals = jnp.arange(D_MAX)
    ϕ_vals = demand_pdf(p, d_vals)
    revenue = jnp.sum(jnp.minimum(x, d_vals)*ϕ_vals)
    profit = revenue - c * a - κ * (a > 0)
    v_R = jnp.sum(v[jnp.maximum(x - d_vals, 0) + a].T * ϕ_vals, axis=1)
    cv = jnp.sum(v_R*Q[i_z])
```

(continues on next page)

(continued from previous page)

```
    return profit + z * cv
```

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_vec_a = jax.vmap(B, in_axes=(None, None, 0, None, None))
```

```
@jax.jit
def B2(x, i_z, v, model):
    """
    The function B(x, z, a, v) = r(x, a) + β(z) Σ_x' v(x') P(x, a, x').
    """
    c, κ, p, z_vals, Q = model
    a_vals = jnp.arange(K)
    res = B_vec_a(x, i_z, a_vals, v, model)
    return jnp.where(a_vals < K - x + 1, res, -jnp.inf)
```

```
B2_vec_z = jax.vmap(B2, in_axes=(None, 0, None, None))
B2_vec_z_x = jax.vmap(B2_vec_z, in_axes=(0, None, None, None))
```

Next we define the Bellman operator.

```
@jax.jit
def T(v, model):
    """The Bellman operator."""
    c, κ, p, z_vals, Q = model
    i_z_range = jnp.arange(len(z_vals))
    x_range = jnp.arange(K + 1)
    res = B2_vec_z_x(x_range, i_z_range, 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."""
    c, κ, p, z_vals, Q  = model
    i_z_range = jnp.arange(len(z_vals))
    x_range = jnp.arange(K + 1)
    res = B2_vec_z_x(x_range, i_z_range, v, model)
    return jnp.argmax(res, axis=2)
```

Here's code to solve the model using value function iteration.

```
def solve_inventory_model(v_init, model):
    """Use successive_approx to get v_star and then compute greedy."""
    v_star = successive_approx(lambda v: T(v, model), v_init, verbose=True)
    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()
c, κ, p, z_vals, Q = model
n_z = len(z_vals)
v_init = jnp.zeros((K + 1, n_z), dtype=float)
```

```
in_time = time.time()
v_star, σ_star = solve_inventory_model(v_init, model)
jax_time = time.time() - in_time
```

```
Completed iteration 25 with error 0.5613828428334706.
Completed iteration 50 with error 0.3776464347688062.
Completed iteration 75 with error 0.2272706235969011.
Completed iteration 100 with error 0.12872204940708798.
Completed iteration 125 with error 0.06744149371262154.
```

```
Completed iteration 150 with error 0.030374639547666504.
Completed iteration 175 with error 0.01423099032950148.
Completed iteration 200 with error 0.007396776219316337.
Completed iteration 225 with error 0.003912238304586424.
Completed iteration 250 with error 0.0020680914166604225.
```

```
Completed iteration 275 with error 0.001092307533355097.
Completed iteration 300 with error 0.0005766427105982075.
Completed iteration 325 with error 0.0003043321707139057.
Completed iteration 350 with error 0.0001605907367547843.
Completed iteration 375 with error 8.473334524694565e-05.
```

```
Completed iteration 400 with error 4.4706045166265085e-05.
Completed iteration 425 with error 2.3586619946058818e-05.
Completed iteration 450 with error 1.2443945941242873e-05.
Completed iteration 475 with error 6.565178331641164e-06.
Completed iteration 500 with error 3.463639430378862e-06.
```

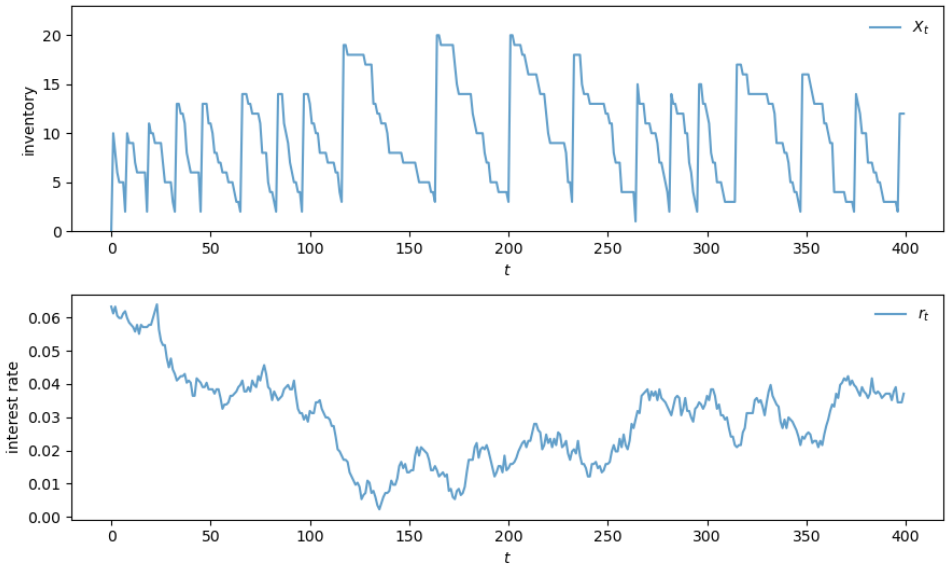
```
Completed iteration 525 with error 1.827332347659194e-06.
Terminated successfully in 550 iterations.
```

```
z_mc = qe.MarkovChain(Q, z_vals)
```

```
def sim_inventories(ts_length, X_init=0):
    """Simulate given the optimal policy."""
    global p, z_mc
    i_z = z_mc.simulate_indices(ts_length, init=1)
    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], i_z[t]]
    return X, z_vals[i_z]
```

```
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()
```

```
plot_ts()
```



13.3 Numba implementation

Let's try the same operations in Numba in order to compare the speed.

```
@njit
def demand_pdf_numba(p, d):
    return (1 - p)**d * p

@njit
def B_numba(x, i_z, a, v, model):
    """
    The function B(x, z, a, v) = r(x, a) + β(z) Σ_x' v(x') P(x, a, x').
    """
    c, κ, p, z_vals, Q = model
    z = z_vals[i_z]
    d_vals = np.arange(D_MAX)
    ϕ_vals = demand_pdf_numba(p, d_vals)
    revenue = np.sum(np.minimum(x, d_vals)*ϕ_vals)
    profit = revenue - c * a - κ * (a > 0)
    v_R = np.sum(v[np.maximum(x - d_vals, 0) + a].T * ϕ_vals, axis=1)
    cv = np.sum(v_R*Q[i_z])
    return profit + z * cv


@njit(parallel=True)
def T_numba(v, model):
    """The Bellman operator."""
    c, κ, p, z_vals, Q = model
    new_v = np.empty_like(v)
    for i_z in prange(len(z_vals)):
        for x in prange(K+1):
            v_1 = np.array([B_numba(x, i_z, a, v, model)
                            for a in range(K-x+1)])
            new_v[x, i_z] = np.max(v_1)
    return new_v


@njit(parallel=True)
def get_greedy_numba(v, model):
    """Get a v-greedy policy.  Returns a zero-based array."""
    c, κ, p, z_vals, Q = model
    n_z = len(z_vals)
    σ_star = np.zeros((K+1, n_z), dtype=np.int32)
    for i_z in prange(n_z):
        for x in range(K+1):
            v_1 = np.array([B_numba(x, i_z, a, v, model)
                            for a in range(K-x+1)])
            σ_star[x, i_z] = np.argmax(v_1)
    return σ_star



def solve_inventory_model_numba(v_init, model):
    """Use successive_approx to get v_star and then compute greedy."""
    v_star = successive_approx(lambda v: T_numba(v, model), v_init, verbose=True)
    σ_star = get_greedy_numba(v_star, model)
    return v_star, σ_star
```

```
model = create_sdd_inventory_model(use_jax=False)
c, κ, p, z_vals, Q  = model
n_z = len(z_vals)
v_init = np.zeros((K + 1, n_z), dtype=float)
```

```
in_time = time.time()
v_star_numba, σ_star_numba = solve_inventory_model_numba(v_init, model)
nb_time = time.time() - in_time
```

```
Completed iteration 25 with error 0.5613828428334706.
```

```
Completed iteration 50 with error 0.37764643476879556.
```

```
Completed iteration 75 with error 0.22727062359689398.
```

```
Completed iteration 100 with error 0.12872204940708798.
```

```
Completed iteration 125 with error 0.06744149371262864.
```

```
Completed iteration 150 with error 0.030374639547666504.
```

```
Completed iteration 175 with error 0.01423099032948727.
```

```
Completed iteration 200 with error 0.007396776219316337.
```

```
Completed iteration 225 with error 0.003912238304593529.
```

```
Completed iteration 250 with error 0.002068091416653317.
```

```
Completed iteration 275 with error 0.0010923075333622023.
```

```
Completed iteration 300 with error 0.0005766427105911021.
```

```
Completed iteration 325 with error 0.0003043321707281166.
```

```
Completed iteration 350 with error 0.00016059073676188973.
```

```
Completed iteration 375 with error 8.473334525405107e-05.
```

```
Completed iteration 400 with error 4.470604518047594e-05.
```

```
Completed iteration 425 with error 2.3586619960269672e-05.
```

```
Completed iteration 450 with error 1.2443945934137446e-05.
```

```
Completed iteration 475 with error 6.565178331641164e-06.
```

```
Completed iteration 500 with error 3.4636394445897167e-06.
```

```
Completed iteration 525 with error 1.827332347659194e-06.
```

```
Terminated successfully in 550 iterations.
```

Let's verify that the Numba and JAX implementations converge to the same solution.

```
np.allclose(v_star_numba, v_star)
```

```
True
```

Here's the speed comparison.

```
print(f"JAX vectorized implementation is {nb_time/jax_time} faster "
      "than Numba's parallel implementation")
```

```
JAX vectorized implementation is 900.5040249685053 faster than Numba's parallel␣
↪implementation
```

CHAPTER

FOURTEEN

ENDOGENOUS GRID METHOD

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

14.1 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
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:21:31 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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)
```

14.2 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

$$a_{t+1} \leq R(a_t - c_t) + Y_{t+1}, \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, ...$$

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
```

14.3 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.

14.3.1 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

$$u'(c) = \max \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) \left[R(a - c) + \hat{Y}, \, \hat{Y} \right] \,,\; u'(a) \right\} \tag{14.1}$$

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 *(14.3.1)* dominates for sufficiently small a.

Also, again using *(14.3.1)*, 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

$$c = \begin{cases} (u')^{-1} \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) [R(a-c) + \hat{Y}, \hat{Y}] \right\} & \text{if } a > \bar{a}(y) \\ a & \text{if } a \leq \bar{a}(y) \end{cases} \tag{14.2}$$

We begin with an *exogenous* grid of saving values $0 = s_0 < ... < s_{N-1}$

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

$$c_i = (u')^{-1} \left\{ \beta R \, \mathbb{E}_y (u' \circ \sigma) [R s_i + \hat{Y}, \hat{Y}] \right\} \quad \text{for all } i \tag{14.3}$$

and we set

$$a_i = s_i + c_i$$

We claim that each pair a_i, c_i obeys *(14.3.2)*.

Indeed, since $s_i > 0$, choosing c_i according to *(14.3.3)* 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 *(14.3.2)* holds.

We are now ready to iterate with K.

14.3.2 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
```

14.3.3 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
```

(continues on next page)

(continued from previous page)

```
    while i < max_iter and error > tol:
        a_new, σ_new = K_egm_nb(a_vec, σ_vec, constants, sizes, arrays)
        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
```

14.4 Solutions

Here we solve the IFP with JAX and Numba.

We will compare both the outputs and the execution time.

14.4.1 Outputs

```
model = ifp()
```

Here's a first run of the JAX code.

```
a_star_egm_jax, σ_star_egm_jax = successive_approx_jax(model,
                                                       print_skip=100)
```

```
Error at iteration 100 is 0.003274240577000098.
Error at iteration 200 is 0.0013133107388259013.
```

```
Error at iteration 300 is 0.0006550972250753961.
Error at iteration 400 is 0.0003800385932690719 7.
```

```
Error at iteration 500 is 0.00024736616926013255.
Error at iteration 600 is 0.00017446354504913053.
```

```
Error at iteration 700 is 0.000129892015863442.
Error at iteration 800 is 0.00010058769447773841.
```

```
Error at iteration 900 is 7.993256952376626e-05.
Error at iteration 1000 is 6.472028596182788e-05.
```

```
Error at iteration 1100 is 5.316228631624398e-05.
Error at iteration 1200 is 4.425450893941196e-05.
```

```
Error at iteration 1300 is 3.7260418253914906e-05.
Error at iteration 1400 is 3.1614060126861077e-05.
```

```
Error at iteration 1500 is 2.6984975752375462e-05.
Error at iteration 1600 is 2.3148392509719784e-05.
```

```
Error at iteration 1700 is 1.9940474091262317e-05.
Error at iteration 1800 is 1.723818132703947e-05.
```

```
Error at iteration 1900 is 1.4947303633494613e-05.
Error at iteration 2000 is 1.2994575430580468e-05.
```

```
Error at iteration 2100 is 1.132223596411741e-05.

Converged in 2192 iterations.
```

Next let's solve the same IFP with Numba.

```
qe.tic()
a_star_egm_nb, σ_star_egm_nb = successive_approx_numba(model,
                                                       print_skip=100)
qe.toc()
```

```
Error at iteration 100 is 0.0032742405770003202.
```

```
Error at iteration 200 is 0.0013133107388259013.
```

```
Error at iteration 300 is 0.0006550972250753961.
```

```
Error at iteration 400 is 0.0003800385932688499.
```

```
Error at iteration 500 is 0.00024736616926013255.
```

```
Error at iteration 600 is 0.00017446354504935258.
```

```
Error at iteration 700 is 0.000129892015863442.
```

```
Error at iteration 800 is 0.00010058769447773841.
```

```
Error at iteration 900 is 7.993256952354422e-05.
```

```
Error at iteration 1000 is 6.472028596182788e-05.
```

```
Error at iteration 1100 is 5.316228631624398e-05.
```

```
Error at iteration 1200 is 4.425450893941196e-05.
```

```
Error at iteration 1300 is 3.7260418253914906e-05.
```

```
Error at iteration 1400 is 3.1614060126861077e-05.
```

```
Error at iteration 1500 is 2.6984975752597506e-05.
```

```
Error at iteration 1600 is 2.3148392509719784e-05.
```

```
Error at iteration 1700 is 1.9940474091262317e-05.
```

```
Error at iteration 1800 is 1.7238181326817426e-05.
```

```
Error at iteration 1900 is 1.4947303633494613e-05.
```

```
Error at iteration 2000 is 1.2994575430802513e-05.
```

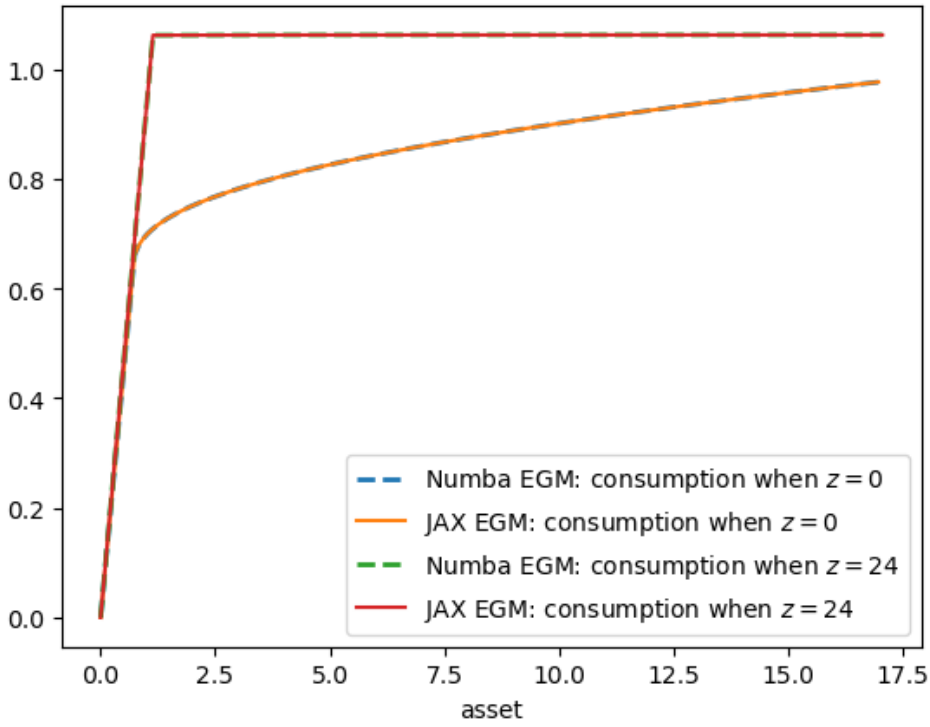
```
Error at iteration 2100 is 1.132223596411741e-05.
```

```
Converged in 2192 iterations.
TOC: Elapsed: 0:01:20.82
```

```
80.8220443725586
```

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()
```



14.4.2 Timing

Now let's compare execution time of the two methods

```
qe.tic()
a_star_egm_jax, σ_star_egm_jax = successive_approx_jax(model,
                                        print_skip=1000)
jax_time = qe.toc()
```

```
Error at iteration 1000 is 6.472028596182788e-05.
```

```
Error at iteration 2000 is 1.2994575430580468e-05.
```

```
Converged in 2192 iterations.
TOC: Elapsed: 0:00:3.47
```

```
qe.tic()
a_star_egm_nb, σ_star_egm_nb = successive_approx_numba(model,
                                        print_skip=1000)
numba_time = qe.toc()
```

```
Error at iteration 1000 is 6.472028596182788e-05.
```

```
Error at iteration 2000 is 1.2994575430802513e-05.
```

```
Converged in 2192 iterations.
TOC: Elapsed: 0:01:18.79
```

```
jax_time / numba_time
```

```
0.044070731287719704
```

The JAX code is significantly faster, as expected.

This difference will increase when more features (and state variables) are added to the model.

CHAPTER

FIFTEEN

DEFAULT RISK AND INCOME FLUCTUATIONS

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

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

```
!pip install quantecon
```

15.1 Overview

This lecture computes versions of Arellano's [Are08] model of sovereign default.

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

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

The foreign lenders must be compensated for default risk.

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

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

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

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

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

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

This can lead to

- spikes in interest rates
- temporary losses of access to international credit markets

- large drops in output, consumption, and welfare
- large capital outflows during recessions

Such dynamics are consistent with experiences of many countries.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import quantecon as qe
import random

import jax
import jax.numpy as jnp
from collections import namedtuple
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:20:43 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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)
```

15.2 Structure

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

15.2.1 Output, Consumption and Debt

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

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

The output good can be traded or consumed.

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

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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{15.1}$$

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 (15.1).

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.

15.2.2 Asset Markets

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

The bond market has the following features

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

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

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

$$c = y + B - q(B', y)B' \tag{15.2}$$

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

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

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

15.2.3 Financial Markets

Foreign creditors

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

$$q = \frac{1 - \delta}{1 + r} \tag{15.3}$$

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

15.2.4 Government's Decisions

At each point in time t, the government chooses between

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

Defaulting means declining to repay all of its current obligations.

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

But a sovereign default has two consequences:

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

15.2.5 Reentering International Credit Market

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

15.3 Equilibrium

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

1. Consumption and assets satisfy the national budget constraint.
2. The government maximizes household utility taking into account
- the resource constraint
- the effect of its choices on the price of bonds

- consequences of defaulting now for future net output and future borrowing and lending opportunities

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

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \tag{15.4}$$

Given zero profits for foreign creditors in equilibrium, we can combine (15.3) and (15.4) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{15.5}$$

15.3.1 Definition of Equilibrium

An *equilibrium* is

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

such that

- The three Bellman equations for $(v_c(B, y), v_d(y), v(B, y))$ are satisfied
- Given the price function $q(B', y)$, the default decision rule and the asset accumulation decision rule attain the optimal value function $v(B, y)$, and
- The price function $q(B', y)$ satisfies equation (15.5)

15.4 Computation

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

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

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

After that we'll perform some additional simulations.

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

- The appendix to [Are08] recommends value function iteration until convergence, updating the price, and then repeating.
- Instead, we update the bond price at every value function iteration step.

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

Here is a more detailed description of our algorithm:

1. Guess a 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 Numba.

We define a namedtuple to store parameters, grids and transition probabilities.

```
ArellanoEconomy = namedtuple('ArellanoEconomy',
    ('β',      # Time discount parameter
    'γ',       # Utility parameter
    'r',       # Lending rate
    'ρ',       # Persistence in the income process
    'η',       # Standard deviation of the income process
    'θ',       # Prob of re-entering financial markets
    'B_size', # Grid size for bonds
    'y_size', # Grid size for income
    'P',       # Markov matrix governing the income process
    'B_grid', # Bond unit grid
    'y_grid', # State values of the income process
    'def_y')) # 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, 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
```

(continues on next page)

(continued from previous page)

```
        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))
    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
```

(continues on next page)

(continued from previous page)

```
    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

    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()
```

```
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.
```

```
%%time
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.
CPU times: user 1.43 s, sys: 423 ms, total: 1.85 s
Wall time: 672 ms
```

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)

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

    # Perform simulation
    t = 0
    while t < T:

        # Update y_sim and B_sim
        y_sim = y_sim.at[t].set(y_grid[y_idx])
        B_sim = B_sim.at[t].set(B_grid[B_idx])
```

(continues on next page)

(continued from previous page)

```
        # if in default:
        if v_c[B_idx, y_idx] < v_d[y_idx] or in_default:
            # Update y_a_sim
            y_a_sim = y_a_sim.at[t].set(model.def_y[y_idx])
            d_sim = d_sim.at[t].set(1)
            Bp_idx = B0_idx
            # Re-enter financial markets next period with prob 9
            # in_default = False if jnp.random.rand() < model.9 else True
            in_default = False if random.uniform(key) < model.θ else True
            key, _ = random.split(key)  # Update the random key
        else:
            # Update y_a_sim
            y_a_sim = y_a_sim.at[t].set(y_sim[t])
            d_sim = d_sim.at[t].set(0)
            Bp_idx = B_star[B_idx, y_idx]

        q_sim = q_sim.at[t].set(q[Bp_idx, y_idx])

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

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

15.5 Results

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

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 [Are08], 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 Arrelano's figure.

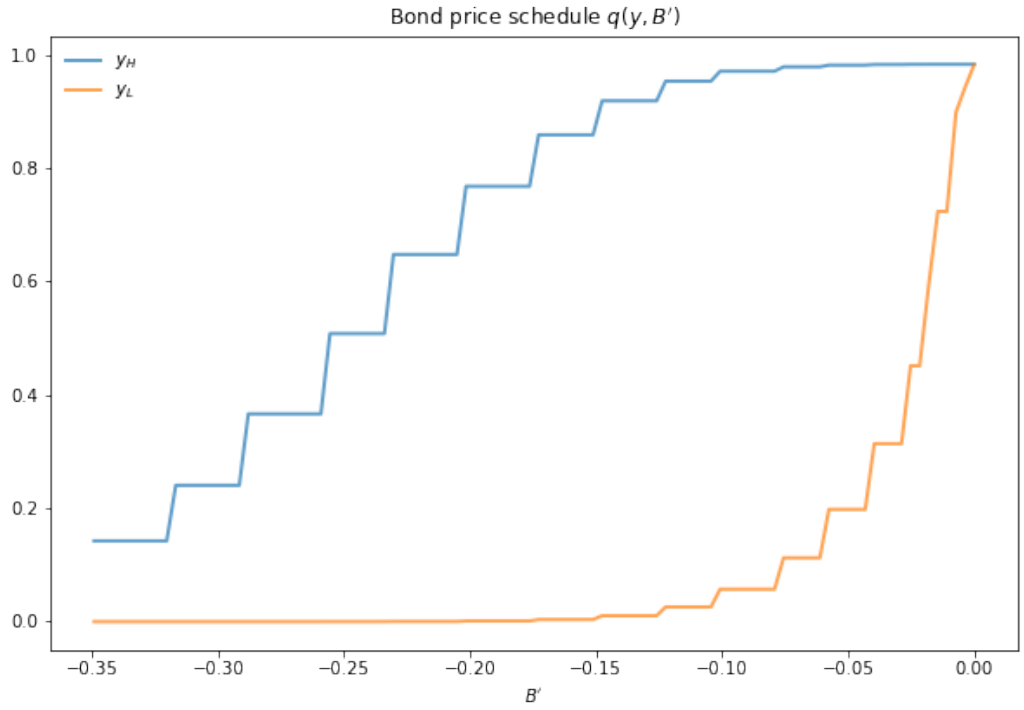
The figure shows that

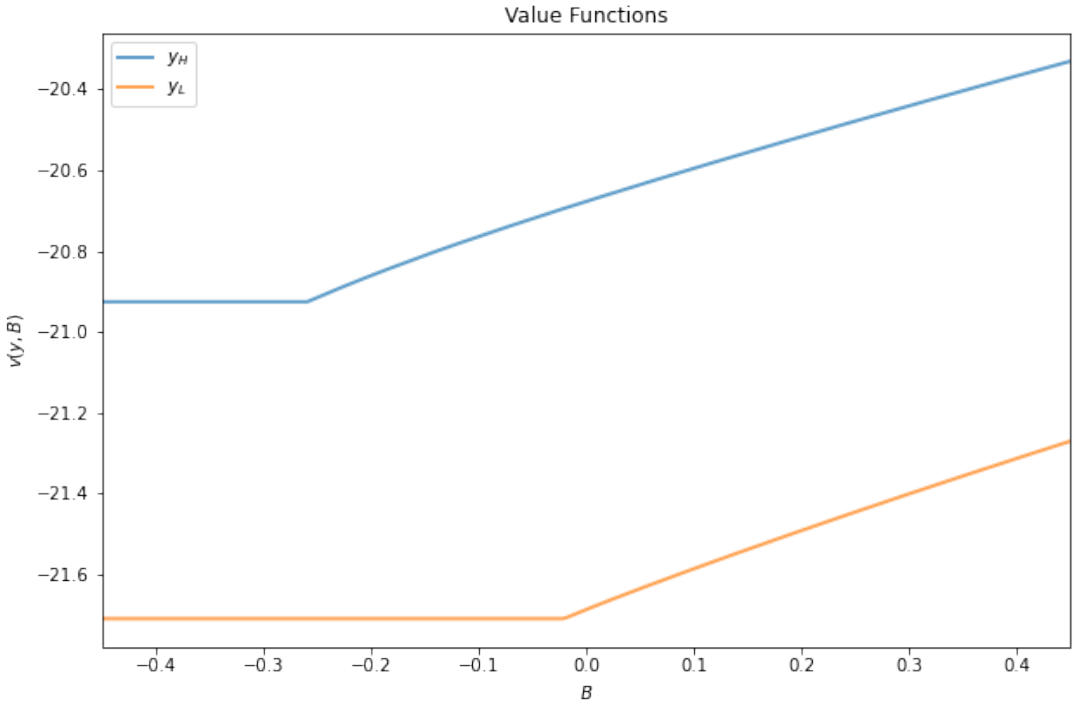
- Higher levels of debt (larger $-B'$) induce larger discounts on the face value, which correspond to higher interest rates.
- Lower income also causes more discounting, as foreign creditors anticipate greater likelihood of default.

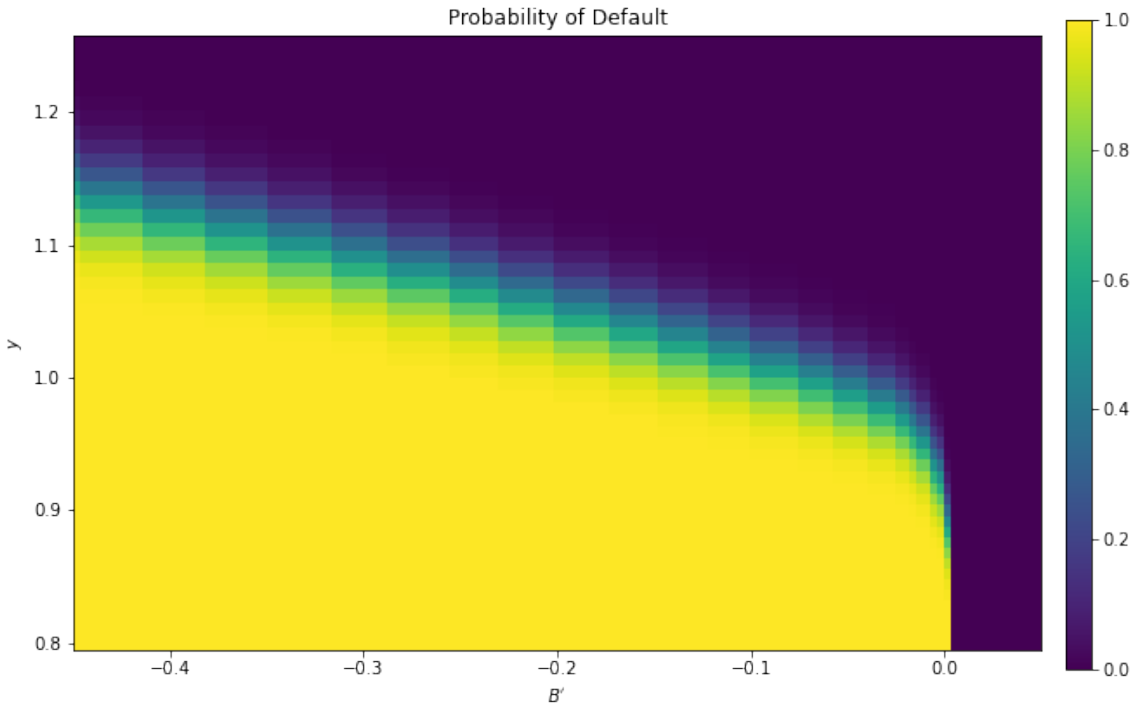
The next figure plots value functions and replicates the right hand panel of Figure 4 of [Are08].

We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (15.4).

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


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



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

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

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

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

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

15.6 Exercises

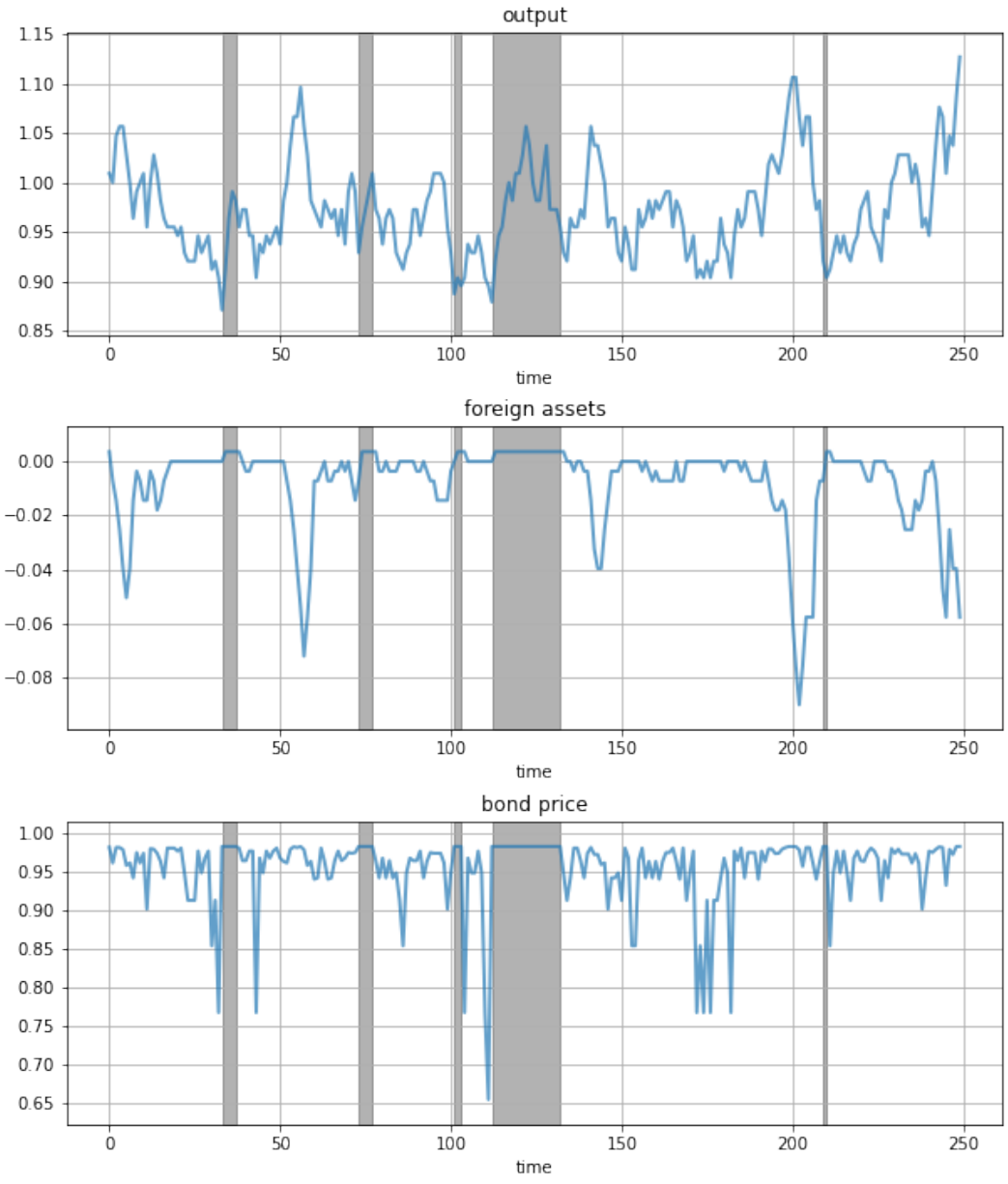
Exercise 15.6.1

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 to Exercise 15.6.1

Compute the value function, policy and equilibrium prices


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

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

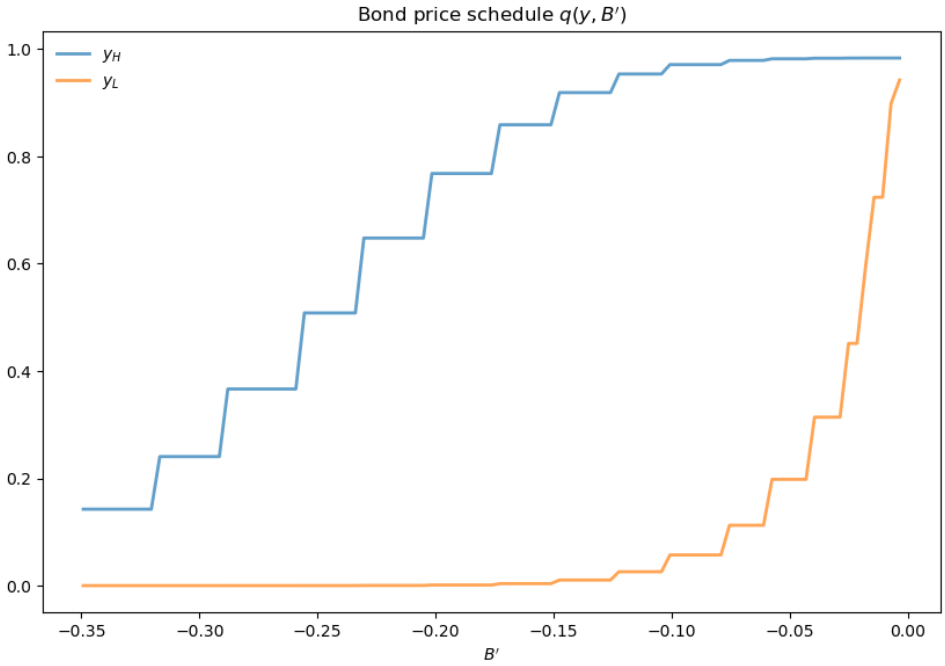
```
Entering iteration 0 with error 1.00000001.
Entering iteration 100 with error 0.017499341639204857.
```

```
Entering iteration 200 with error 0.00014189363558969603.
Entering iteration 300 with error 1.151467966309383e-06.
```

```
Terminating at iteration 399.
```

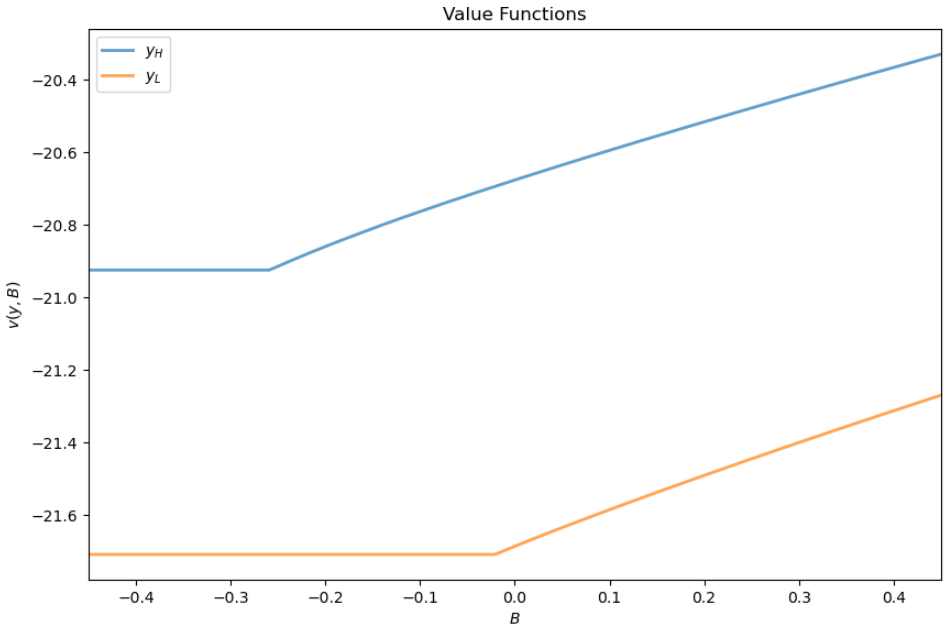
Compute the bond price schedule as seen in figure 3 of [Are08]

```
# 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) * .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()
```



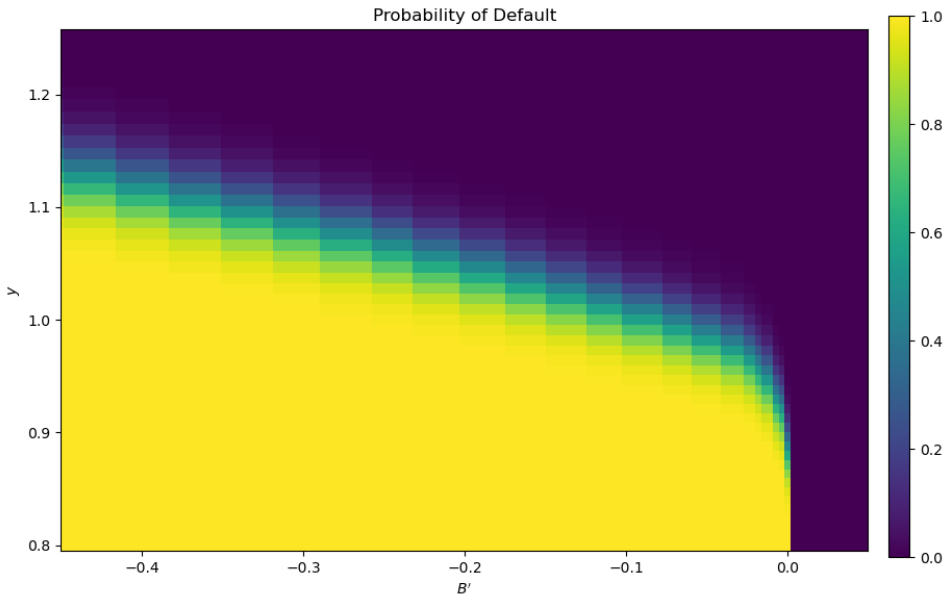
Draw a plot of the value functions

```
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([.92, .1, .02, .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

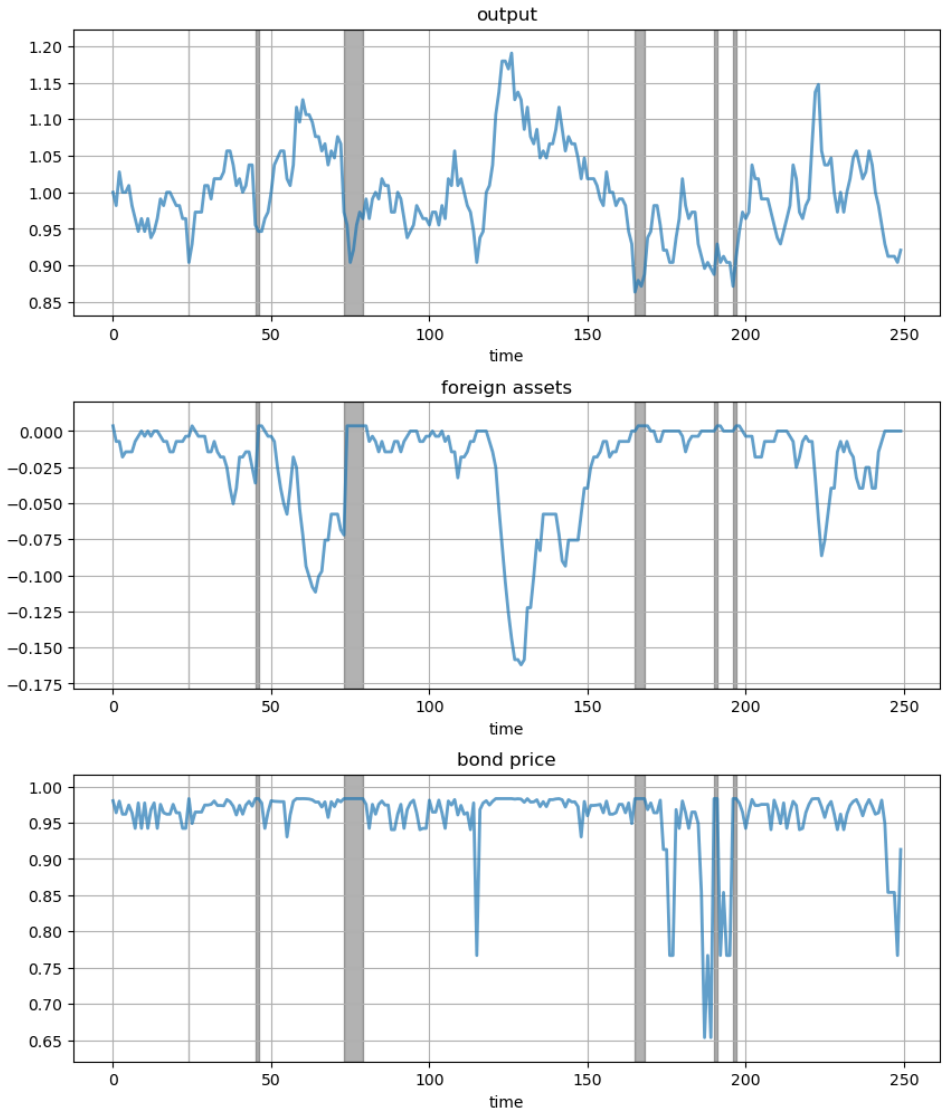
```
import jax.random as random
T = 250
key = random.PRNGKey(42)
y_sim, y_a_sim, B_sim, q_sim, d_sim = simulate(ae, T, v_c, v_d, q, B_star, key)

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    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

SIXTEEN

THE AIYAGARI MODEL

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

16.1 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]

16.1.1 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.

16.1.2 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
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:19:30 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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)
```

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.)

```
# Compute the stationary distribution of P by matrix inversion.

@jax.jit
def compute_stationary(P):
    n = P.shape[0]
```

(continues on next page)

(continued from previous page)

```
    I = jnp.identity(n)
    O = jnp.ones((n, n))
    A = I - jnp.transpose(P) + O
    return jnp.linalg.solve(A, jnp.ones(n))
```

16.2 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

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

where

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

The firm's problem is

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

The parameter δ is the depreciation rate.

These parameters are stored in the following namedtuple.

```
Firm = namedtuple('Firm', ('A', 'N', 'α', 'β', 'δ'))

def create_firm(A=1.0,
                N=1.0,
                α=0.33,
                β=0.96,
                δ=0.05):

    return Firm(A=A, N=N, α=α, β=β, δ=δ)
```

From the first-order condition with respect to capital,

the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K} \right)^{1-\alpha} - \delta \tag{16.1}$$

```
def r_given_k(K, firm):
    """
    Inverse demand curve for capital.  The interest rate associated with a
    given demand for capital K.
```

(continues on next page)

(continued from previous page)

```
    """
    A, N, α, β, δ = firm
    return A * α * (N / K)**(1 - α) - δ
```

Using (16.1) and the firm's first-order condition for labor,

we can pin down the equilibrium wage rate as a function of r as

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{16.2}$$

```
def r_to_w(r, f):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    A, N, α, β, δ = f
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))
```

16.3 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

$$a_{t+1} + c_t \leq w z_t + (1+r) a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate
- r is a net interest rate
- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with given stochastic matrix P.

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.

For now we assume that $u(c) = \log(c)$.

(CRRA utility is treated in the exercises.)

16.3.1 Primitives and Operators

This namedtuple stores the parameters that define a household asset accumulation problem and the grids used to solve it.

```
Household = namedtuple('Household', ('r', 'w', 'β', 'a_size', 'z_size', \
                                     'a_grid', 'z_grid', 'Π'))

def create_household(r=0.01,                       # Interest rate
                     w=1.0,                        # Wages
                     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):

    a_grid = jnp.linspace(a_min, a_max, a_size)
    z_grid, Π = map(jnp.array, (z_grid, Π))
    Π = jax.device_put(Π)
    z_grid = jax.device_put(z_grid)
    z_size = len(z_grid)
    a_grid, z_grid, Π = jax.device_put((a_grid, z_grid, Π))

    return Household(r=r, w=w, β=β, a_size=a_size, z_size=z_size, \
                     a_grid=a_grid, z_grid=z_grid, Π=Π)
```

```
def u(c):
    return jnp.log(c)
```

This is the 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').

```
def B(v, constants, sizes, arrays):
    # Unpack
    r, w, β = constants
    a_size, z_size = sizes
    a_grid, z_grid, Π = arrays

    # 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=3)                 # 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)

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

The next function computes greedy policies.

```
# Computes a v-greedy policy, returned as a set of indices
def get_greedy(v, constants, sizes, arrays):
    return jnp.argmax(B(v, constants, sizes, arrays), axis=2)

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

We need to know rewards at a given policy for policy iteration.

The following functions computes the array r_σ which gives current rewards given policy σ.

That is,

$$r_\sigma[i, j] = r[i, j, \sigma[i, j]]$$

```
def compute_r_σ(σ, constants, sizes, arrays):
    # Unpack
    r, w, β = constants
    a_size, z_size = sizes
    a_grid, z_grid, Π = arrays

    # 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

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

The value v_σ of a policy σ is defined as

$$v_\sigma = (I - \beta P_\sigma)^{-1} r_\sigma$$

Here we set up the linear map $v \to R_\sigma v$, where $R_\sigma := I - \beta P_\sigma$.

In the consumption problem, this map can be expressed as

$$(R_\sigma v)(a, z) = v(a, z) - \beta \sum_{z'} v(\sigma(a, z), z') \Pi(z, z')$$

Defining the map as above works in a more intuitive multi-index setting

(e.g. working with $v[i, j]$ rather than flattening v to a one-dimensional array)

and avoids instantiating the large matrix P_σ.

```
def R_σ(v, σ, constants, sizes, arrays):
    # Unpack
    r, w, β = constants
    a_size, z_size = sizes
    a_grid, z_grid, Π = arrays

    # Set up the array v[σ[i, j], jp]
    zp_idx = jnp.arange(z_size)
```

(continues on next page)

(continued from previous page)

```
    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=2)

R_σ = jax.jit(R_σ, static_argnums=(3,))
```

The next function computes the lifetime value of a given policy.

```
# Get the value v_σ of policy σ by inverting the linear map R_σ

def get_value(σ, constants, sizes, arrays):

    r_σ = compute_r_σ(σ, constants, sizes, arrays)
    # Reduce R_σ to a function in v
    partial_R_σ = lambda v: R_σ(v, σ, constants, sizes, arrays)
    # Compute inverse v_σ = (I - β P_σ)^{-1} r_σ
    return jax.scipy.sparse.linalg.bicgstab(partial_R_σ, r_σ)[0]

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

16.4 Solvers

We will solve the household problem using Howard policy iteration.

```
def policy_iteration(household, tol=1e-4, max_iter=10_000, verbose=False):
    """Howard policy iteration routine."""

    γ, w, β, a_size, z_size, a_grid, z_grid, Π = household

    constants = γ, w, β
    sizes = a_size, z_size
    arrays = a_grid, z_grid, Π

    σ = jnp.zeros(sizes, dtype=int)
    v_σ = get_value(σ, constants, sizes, arrays)
    i = 0
    error = tol + 1
    while error > tol and i < max_iter:
        σ_new = get_greedy(v_σ, constants, sizes, arrays)
        v_σ_new = get_value(σ_new, constants, sizes, arrays)
        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
```

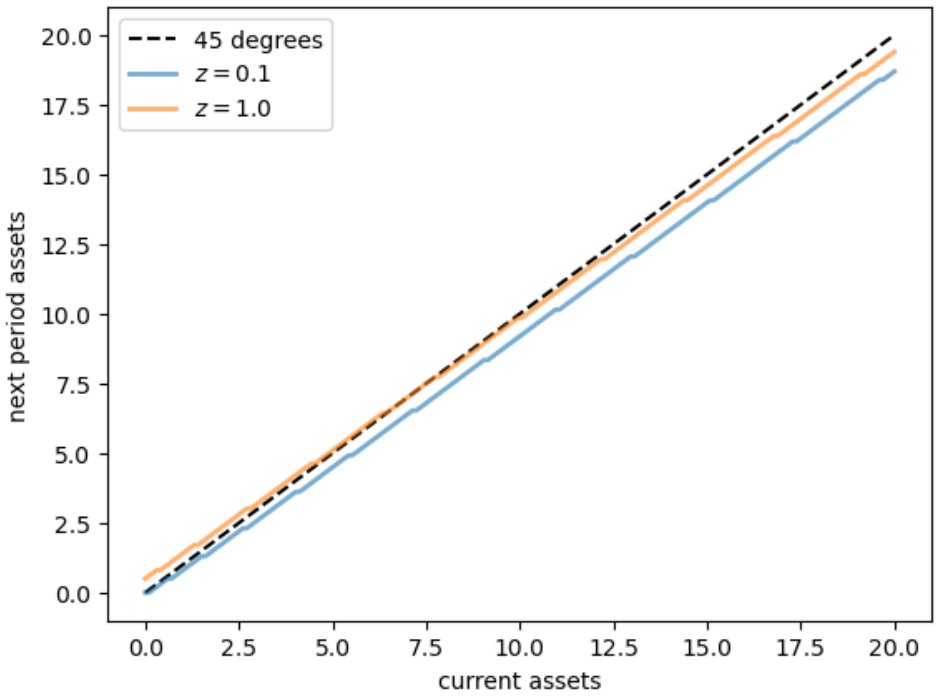
As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices.

```
# Create an instance of Housbehold
household = create_household()
```

```
%%time

σ_star = policy_iteration(household, verbose=True)


# The next plot shows asset accumulation policies at different values of the␣
 ↪exogenous state.
```

```
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 680 ms, sys: 44.1 ms, total: 724 ms
Wall time: 883 ms
```

```
γ, w, β, a_size, z_size, 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')
ax.legend(loc='upper left')
plt.show()
```



16.4.1 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 implements this calculation for a given policy σ.

First we compute the stationary distribution of P_σ, which is for the bivariate Markov chain of the state (a_t, z_t). Then we sum out z_t to get the marginal distribution for a_t.

```
def compute_asset_stationary(σ, constants, sizes, arrays):

    # Unpack
    r, w, β = constants
    a_size, z_size = sizes
    a_grid, z_grid, Π = arrays

    # 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

    # Reshape P_σ into a matrix
    n = a_size * z_size
    P_σ = jnp.reshape(P_σ, (n, n))

    # Get stationary distribution and reshape onto [i, j] grid
```

(continues on next page)

(continued from previous page)

```
    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

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

Let's give this a test run.

```
γ, w, β, a_size, z_size, a_grid, z_grid, Π = household
constants = γ, w, β
sizes = a_size, z_size
arrays = a_grid, z_grid, Π
ψ = compute_asset_stationary(σ_star, constants, sizes, arrays)
```

The distribution should sum to one:

```
ψ.sum()
```

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

Now we are ready to compute capital supply by households given wages and interest rates.

```
def capital_supply(household):
    """
    Map household decisions to the induced level of capital stock.
    """

    # Unpack
    γ, w, β, a_size, z_size, a_grid, z_grid, Π = household

    constants = γ, w, β
    sizes = a_size, z_size
    arrays = a_grid, z_grid, Π

    # Compute the optimal policy
    σ_star = policy_iteration(household)
    # Compute the stationary distribution
    ψ_a = compute_asset_stationary(σ_star, constants, sizes, arrays)

    # Return K
    return float(jnp.sum(ψ_a * a_grid))
```

16.5 Equilibrium

We construct a *stationary rational expectations equilibrium* (SREE).

In such an equilibrium

- prices induce behavior that generates aggregate quantities consistent with the prices
- aggregate quantities and prices are constant over time

In more detail, an SREE lists a set of prices, savings and production policies such that

- households want to choose the specified savings policies taking the prices as given
- firms maximize profits taking the same prices as given
- the resulting aggregate quantities are consistent with the prices; in particular, the demand for capital equals the supply
- aggregate quantities (defined as cross-sectional averages) are constant

In practice, once parameter values are set, we can check for an SREE by the following steps

1. pick a proposed quantity K for aggregate capital
2. determine corresponding prices, with interest rate r determined by (16.1) and a wage rate $w(r)$ as given in (16.2).
3. determine the common optimal savings policy of the households given these prices
4. compute aggregate capital as the mean of steady state capital given this savings policy

If this final quantity agrees with K then we have a SREE. Otherwise we adjust K.

These steps describe a fixed point problem which we solve below.

16.5.1 Visual inspection

Let's inspect visually as a first pass.

The following code draws aggregate supply and demand curves for capital.

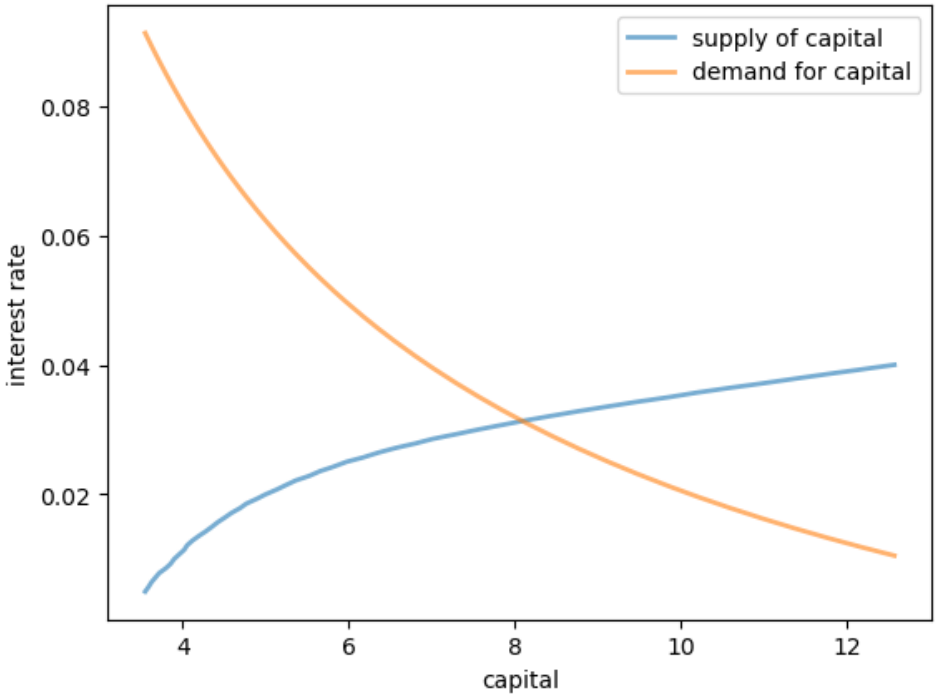
The intersection gives equilibrium interest rates and capital.

```
# Create default instances
household = create_household()
firm = create_firm()

# Create a grid of r values at which to compute demand and supply of capital
num_points = 50
r_vals = np.linspace(0.005, 0.04, num_points)
```

```
%%time

# Compute supply of capital
k_vals = np.empty(num_points)
for i, r in enumerate(r_vals):
    # _replace create a new nametuple with the updated parameters
    household = household._replace(r=r, w=r_to_w(r, firm))
    k_vals[i] = capital_supply(household)
```

```
CPU times: user 3.92 s, sys: 1.05 s, total: 4.96 s
Wall time: 2.87 s
```

```
# Plot against demand for capital by firms

fig, ax = plt.subplots()
ax.plot(k_vals, r_vals, lw=2, alpha=0.6, label='supply of capital')
ax.plot(k_vals, r_given_k(k_vals, firm), lw=2, alpha=0.6, label='demand for capital')
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```



Here's a plot of the excess demand function.

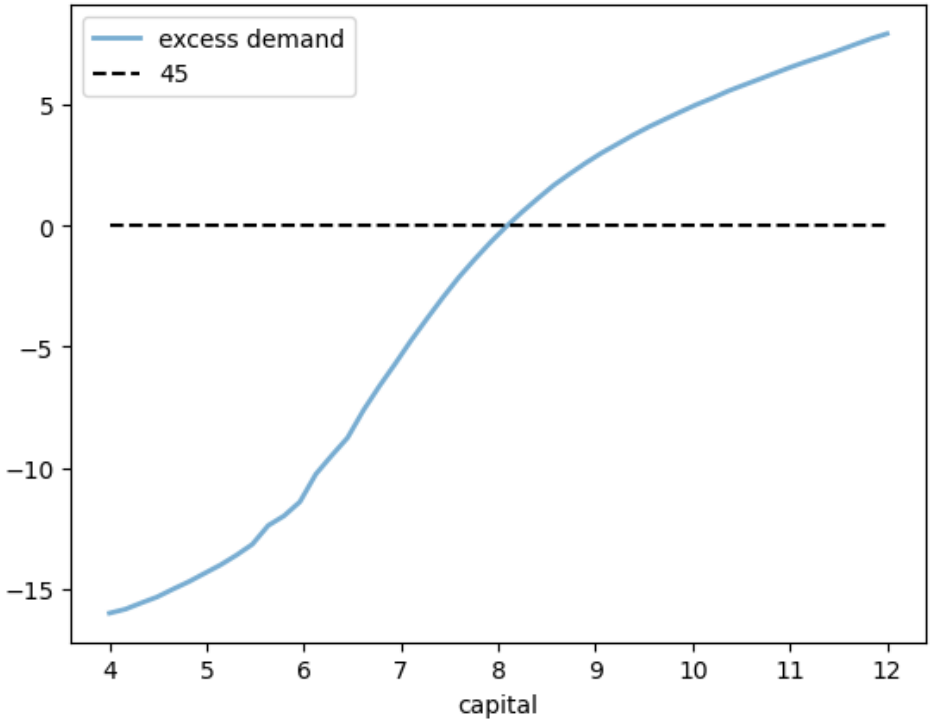
The equilibrium is the zero (root) of this function.

```
def excess_demand(K, firm, household):
    r = r_given_k(K, firm)
    w = r_to_w(r, firm)
    household = household._replace(r=r, w=w)
    return K - capital_supply(household)
```

```
%%time

num_points = 50
k_vals = np.linspace(4, 12, num_points)
out = [excess_demand(k, firm, household) for k in k_vals]
```

```
CPU times: user 3.55 s, sys: 1.06 s, total: 4.61 s
Wall time: 2.46 s
```

```
fig, ax = plt.subplots()
ax.plot(k_vals, out, lw=2, alpha=0.6, label='excess demand')
ax.plot(k_vals, np.zeros_like(k_vals), 'k--', label="45")
ax.set_xlabel('capital')
ax.legend()
plt.show()
```



16.5.2 Computing the equilibrium

Now let's compute the equilibrium

To do so, we use the bisection method, which is implemented in the next function.

```
def bisect(f, a, b, *args, tol=10e-2):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b
    count = 0
    while upper - lower > tol and count < 10000:
        middle = 0.5 * (upper + lower)
        if f(middle, *args) > 0:   # root is between lower and middle
            lower, upper = lower, middle
        else:                      # root is between middle and upper
```

(continues on next page)

(continued from previous page)

```
            lower, upper = middle, upper
        count += 1
    if count == 10000:
        print("Root might not be accurate")
    return 0.5 * (upper + lower), count
```

Now we call the bisection function on excess demand.

```
def compute_equilibrium(firm, household):
    print("\nComputing equilibrium capital stock")
    start = time.time()
    solution, count = bisect(excess_demand, 6.0, 10.0, firm, household)
    elapsed = time.time() - start
    print(f"Computed equilibrium in {count} iterations and {elapsed} seconds")
    return solution
```

```
%%time

household = create_household()
firm = create_firm()
compute_equilibrium(firm, household)
```

```
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.30586791038513184 seconds
CPU times: user 454 ms, sys: 105 ms, total: 559 ms
Wall time: 307 ms
```

```
8.09375
```

Notice how quickly we can compute the equilibrium capital stock using a simple method such as bisection.

16.6 Exercises

Exercise 16.6.1

Using the default household and firm model, produce a graph showing the behaviour of equilibrium capital stock with the increase in β.

Solution to Exercise 16.6.1

```
β_vals = np.linspace(0.9, 0.99, 40)
eq_vals = np.empty_like(β_vals)

for i, β in enumerate(β_vals):
    household = create_household(β=β)
    firm = create_firm(β=β)
    eq_vals[i] = compute_equilibrium(firm, household)
```

```
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.6480610370635986 seconds

Computing equilibrium capital stock
Computed equilibrium in 6 iterations and 0.1747879981994629 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.1788344383239746 seconds

Computing equilibrium capital stock
Computed equilibrium in 6 iterations and 0.1818697452545166 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.1953115463256836 seconds

Computing equilibrium capital stock
Computed equilibrium in 6 iterations and 0.18505477905273438 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.18501806259155273 seconds

Computing equilibrium capital stock
Computed equilibrium in 6 iterations and 0.19269037246704102 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.19585967063903809 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.20452356338500977 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.21703791618347168 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.21313738822937012 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.2120833396911621 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.22403168678283691 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.22535467147827148 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.23635244369506836 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.24231243133544922 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.2495734691619873 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.26811909675598145 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.264385461807251 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.26006364822387695 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.2814364433288574 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.27918124198913574 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.29192447662353516 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.305736780166626 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.31801581382751465 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.3143942356109619 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 34.1399991998672485 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.29398560523986816 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.27906274795532227 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.2763557434082031 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.29296875 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.28820300102233887 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.31885671615600586 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.3292419910430908 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.3484954833984375 seconds
Computing equilibrium capital stock
```

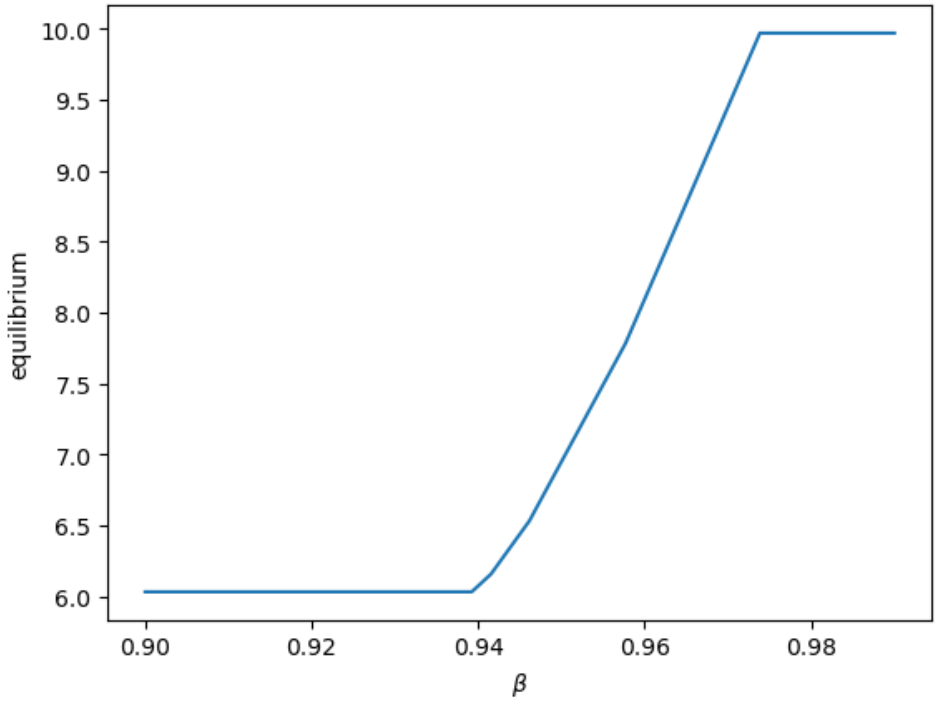
```
Computed equilibrium in 6 iterations and 0.36038827896118164 seconds
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.36896777153015137 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.37683749198913574 seconds

Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.38379454612731934 seconds
```

```
fig, ax = plt.subplots()
ax.plot(β_vals, eq_vals, ms=2)
ax.set_xlabel(r'$\beta$')
ax.set_ylabel('equilibrium')
plt.show()
```



Exercise 16.6.2

Switch to the CRRA utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

and re-do the plot of demand for capital by firms against the supply of captial.

Also, recompute the equilibrium.

Use the default parameters for households and firms.

Set $\gamma = 2$.

Solution to Exercise 16.6.2

Let's define the utility function

```
def u(c, γ=2):
    return c**(1 - γ) / (1 - γ)
```

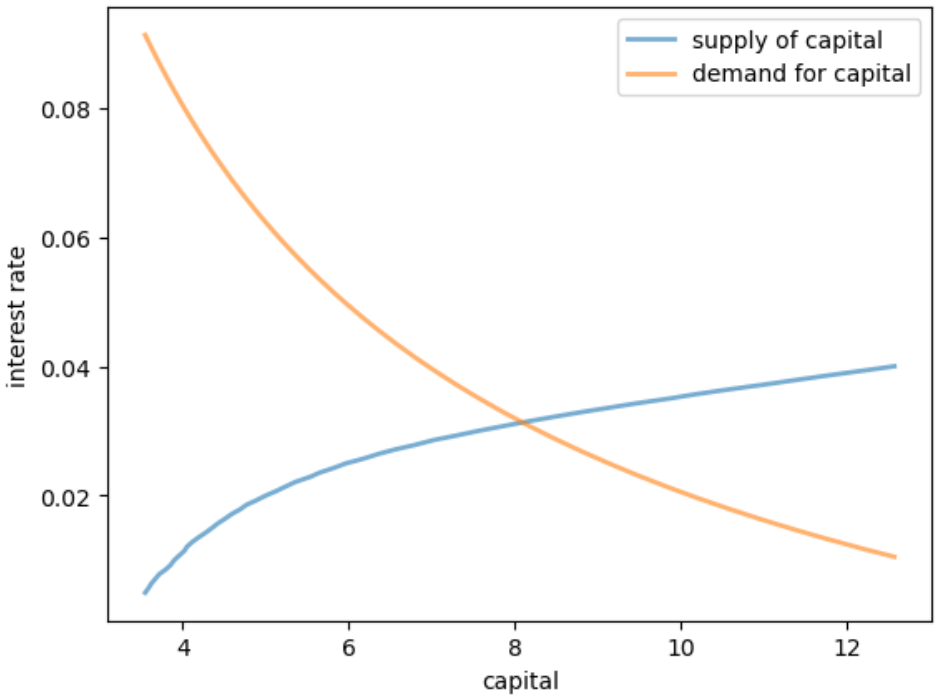
We need to re-compile all the jitted functions in order notice the change in the utility function.

```
B = jax.jit(B, static_argnums=(2,))
get_greedy = jax.jit(get_greedy, static_argnums=(2,))
compute_r_σ = jax.jit(compute_r_σ, static_argnums=(2,))
R_σ = jax.jit(R_σ, static_argnums=(3,))
get_value = jax.jit(get_value, static_argnums=(2,))
compute_asset_stationary = jax.jit(compute_asset_stationary,
                                   static_argnums=(2,))
```

Now, let's plot the the demand for capital by firms

```
# Create default instances
household = create_household()
firm = create_firm()

# Create a grid of r values at which to compute demand and supply of capital
num_points = 50
r_vals = np.linspace(0.005, 0.04, num_points)


# Compute supply of capital
k_vals = np.empty(num_points)
for i, r in enumerate(r_vals):
    household = household._replace(r=r, w=r_to_w(r, firm))
    k_vals[i] = capital_supply(household)
```

```
# Plot against demand for capital by firms

fig, ax = plt.subplots()
ax.plot(k_vals, r_vals, lw=2, alpha=0.6, label='supply of capital')
ax.plot(k_vals, r_given_k(k_vals, firm), lw=2, alpha=0.6, label='demand for capital')
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend()

plt.show()
```



Compute the equilibrium

```
%%time

household = create_household()
firm = create_firm()
compute_equilibrium(firm, household)
```

```
Computing equilibrium capital stock
```

```
Computed equilibrium in 6 iterations and 0.6728487014770508 seconds
CPU times: user 928 ms, sys: 65.7 ms, total: 994 ms
Wall time: 675 ms
```

```
8.09375
```

CHAPTER

SEVENTEEN

CAKE EATING: NUMERICAL METHODS

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

This lecture is the extended JAX implementation of this lecture.

Please refer that lecture for all background and notation.

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

```
!pip install quantecon
```

We will use the following imports.

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from collections import namedtuple
import time
```

Let's check the GPU we are running

```
!nvidia-smi
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:21:09 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16160MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

17.1 Reviewing the Model

Recall in particular that the Bellman equation is

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for all } x \geq 0. \tag{17.1}$$

where u is the CRRA utility function.

17.2 Implementation using JAX

The analytical solutions for the value function and optimal policy were found to be as follows.

```
@jax.jit
def c_star(x, β, γ):
    return (1 - β ** (1/γ)) * x

@jax.jit
def v_star(x, β, γ):
    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Let's define a model to represent the Cake Eating Problem.

```
CEM = namedtuple('CakeEatingModel',
                 ('β', 'γ', 'x_grid', 'c_grid'))
```

```
def create_cake_eating_model(β=0.96,           # discount factor
                             γ=1.5,            # degree of relative risk aversion
                             x_grid_min=1e-3,  # exclude zero for numerical stability
                             x_grid_max=2.5,   # size of cake
                             x_grid_size=200):
    x_grid = jnp.linspace(x_grid_min, x_grid_max, x_grid_size)

    # c_grid used for finding maximize function values using brute force
    c_grid = jnp.linspace(x_grid_min, x_grid_max, 100*x_grid_size)
    return CEM(β=β, γ=γ, x_grid=x_grid, c_grid=c_grid)
```

Now let's define the CRRA utility function.

```
# Utility function
@jax.jit
def u(c, cem):
    return (c ** (1 - cem.γ)) / (1 - cem.γ)
```

17.2.1 The Bellman Operator

We introduce the **Bellman operator** T that takes a function v as an argument and returns a new function Tv defined by

$$Tv(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

From v we get Tv, and applying T to this yields $T^2 v := T(Tv)$ and so on.

This is called **iterating with the Bellman operator** from initial guess v.

```
@jax.jit
def state_action_value(x, c, v_array, ce):
    """
    Right hand side of the Bellman equation given x and c.
    * x: scalar element `x`
    * c: c_grid, 1-D array
    * v_array: value function array guess, 1-D array
    * ce: Cake Eating Model instance
    """

    return jnp.where(c <= x,
                     u(c, ce) + ce.β * jnp.interp(x - c, ce.x_grid, v_array),
                     -jnp.inf)
```

In order to create a vectorized function using `state_action_value`, we use jax.vmap. This function returns a new vectorized version of the above function which is vectorized on the argument `x`.

```
state_action_value_vec = jax.vmap(state_action_value, (0, None, None, None))
```

```
@jax.jit
def T(v, ce):
    """
    The Bellman operator. Updates the guess of the value function.

    * ce: Cake Eating Model instance
    * v: value function array guess, 1-D array

    """
    return jnp.max(state_action_value_vec(ce.x_grid, ce.c_grid, v, ce), axis=1)
```

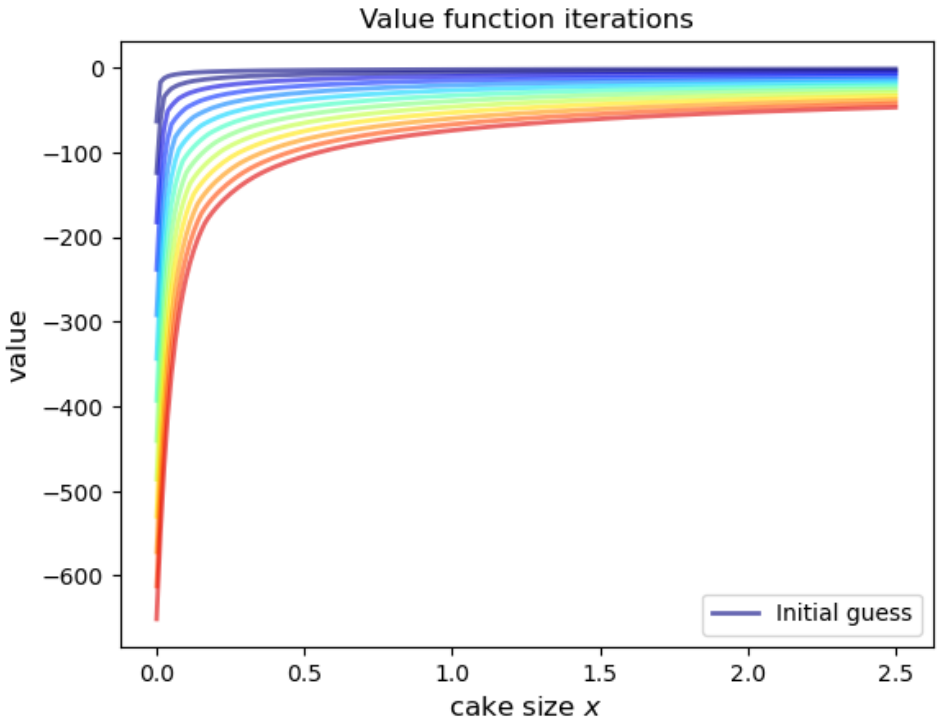
Let's start by creating a Cake Eating Model instance using the default parameterization.

```
ce = create_cake_eating_model()
```

Now let's see the iteration of the value function in action.

We start from guess v given by $v(x) = u(x)$ for every x grid point.

```
x_grid = ce.x_grid
v = u(x_grid, ce)       # Initial guess
n = 12                  # Number of iterations

fig, ax = plt.subplots()

ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial guess')

for i in range(n):
    v = T(v, ce)  # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

ax.legend()
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('cake size $x$', fontsize=12)
ax.set_title('Value function iterations')

plt.show()
```



Let's introduce a wrapper function called `compute_value_function` that iterates until some convergence conditions are satisfied.

```
def compute_value_function(ce,
                           tol=1e-4,
                           max_iter=1000,
                           verbose=True,
                           print_skip=25):

    # Set up loop
```

(continues on next page)

(continued from previous page)

```
    v = jnp.zeros(len(ce.x_grid)) # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, ce)

        error = jnp.max(jnp.abs(v - v_new))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

```
in_time = time.time()
v_jax = compute_value_function(ce)
jax_time = time.time() - in_time
```

```
Error at iteration 25 is 23.74322509765625.
Error at iteration 50 is 8.5570068359375.
Error at iteration 75 is 3.083984375.
```

```
Error at iteration 100 is 1.11151123046875.
Error at iteration 125 is 0.40069580078125.
Error at iteration 150 is 0.14447021484375.
```

```
Error at iteration 175 is 0.0521240234375.
Error at iteration 200 is 0.01885986328125.
Error at iteration 225 is 0.006866455078125.
```

```
Error at iteration 250 is 0.0025634765625.
Error at iteration 275 is 0.0009765625.
Error at iteration 300 is 0.00048828125.
```

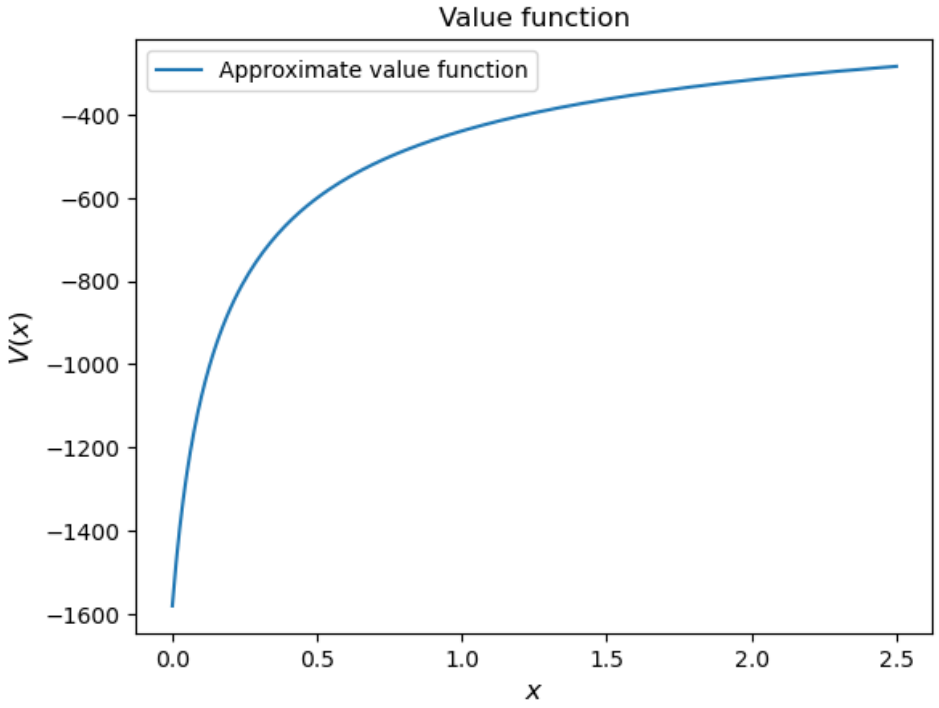
```
Error at iteration 325 is 0.000244140625.
Error at iteration 350 is 0.0001220703125.

Converged in 351 iterations.
```

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

ax.plot(x_grid, v_jax, label='Approximate value function')
ax.set_ylabel('$V(x)$', fontsize=12)
```

(continues on next page)

(continued from previous page)

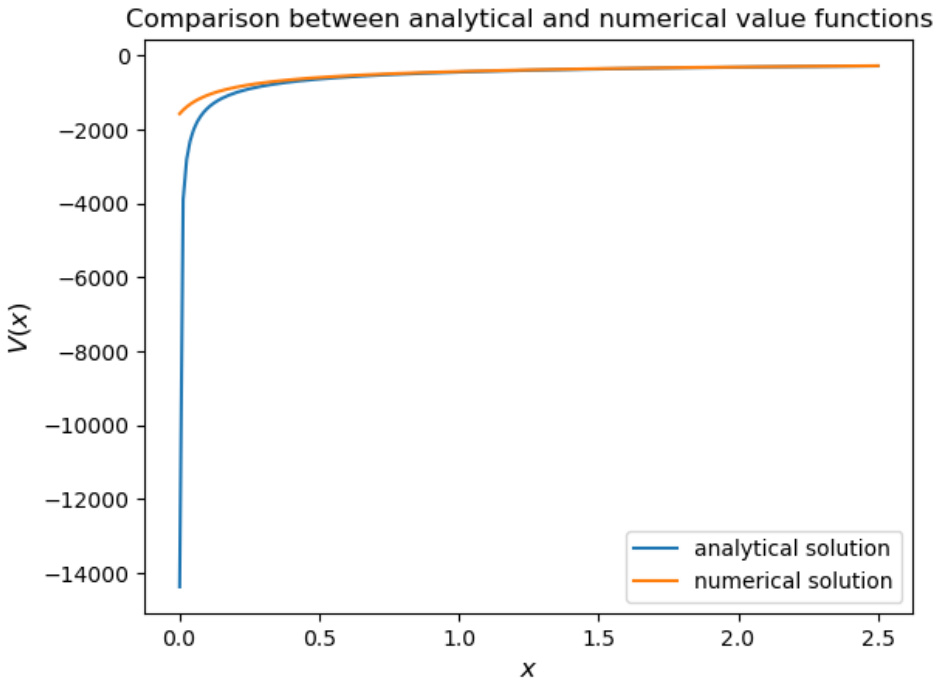
```
ax.set_xlabel('$x$', fontsize=12)
ax.set_title('Value function')
ax.legend()
plt.show()
```



Next let's compare it to the analytical solution.

```
v_analytical = v_star(ce.x_grid, ce.β, ce.γ)
```

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

ax.plot(x_grid, v_analytical, label='analytical solution')
ax.plot(x_grid, v_jax, label='numerical solution')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.legend()
ax.set_title('Comparison between analytical and numerical value functions')
plt.show()
```



17.2.2 Policy Function

Recall that the optimal consumption policy was shown to be

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x$$

Let's see if our numerical results lead to something similar.

Our numerical strategy will be to compute

$$\sigma(x) = \arg\max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

on a grid of x points and then interpolate.

For v we will use the approximation of the value function we obtained above.

Here's the function:

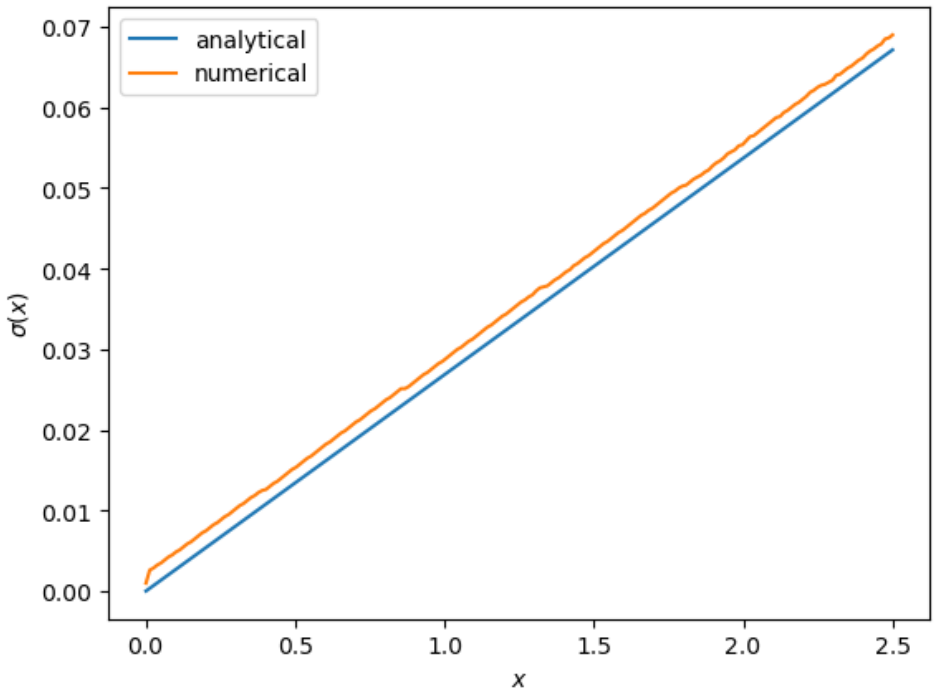
```
@jax.jit
def σ(ce, v):
    """
    The optimal policy function. Given the value function,
    it finds optimal consumption in each state.

    * ce: Cake Eating Model instance
    * v: value function array guess, 1-D array

    """
    i_cs =  jnp.argmax(state_action_value_vec(ce.x_grid, ce.c_grid, v, ce), axis=1)
    return ce.c_grid[i_cs]
```

Now let's pass the approximate value function and compute optimal consumption:

```
c = σ(ce, v_jax)
```

Let's plot this next to the true analytical solution

```
c_analytical = c_star(ce.x_grid, ce.β, ce.γ)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label='analytical')
ax.plot(ce.x_grid, c, label='numerical')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



17.3 Numba implementation

This section of the lecture is directly adapted from this lecture for the purpose of comparing the results of JAX implementation.

```
import numpy as np
from numba import prange, njit
from quantecon.optimize import brent_max
```

```
CEMN = namedtuple('CakeEatingModelNumba',
                  ('β', 'γ', 'x_grid'))
```

```
def create_cake_eating_model_numba(β=0.96,           # discount factor
                                   γ=1.5,            # degree of relative risk␣
 ↪aversion
                                   x_grid_min=1e-3,  # exclude zero for numerical␣
 ↪stability
                                   x_grid_max=2.5,   # size of cake
                                   x_grid_size=200):
    x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)
    return CEMN(β=β, γ=γ, x_grid=x_grid)
```

```
# Utility function
@njit
def u_numba(c, cem):
    return (c ** (1 - cem.γ)) / (1 - cem.γ)
```

```
@njit
def state_action_value_numba(c, x, v_array, cem):
    """
    Right hand side of the Bellman equation given x and c.
    * x: scalar element `x`
    * c: consumption
    * v_array: value function array guess, 1-D array
    * cem: Cake Eating Numba Model instance
    """
    return u_numba(c, cem) + cem.β * np.interp(x - c, cem.x_grid, v_array)
```

```
@njit
def T_numba(v, ce):
    """
    The Bellman operator.  Updates the guess of the value function.

    * ce is an instance of CakeEatingNumba Model
    * v is an array representing a guess of the value function

    """
    v_new = np.empty_like(v)

    for i in prange(len(ce.x_grid)):
        # Maximize RHS of Bellman equation at state x
        v_new[i] = brent_max(state_action_value_numba, 1e-10, ce.x_grid[i],
                             args=(ce.x_grid[i], v, ce))[1]
    return v_new
```

```
def compute_value_function_numba(ce,
                           tol=1e-4,
                           max_iter=1000,
                           verbose=True,
                           print_skip=25):

    # Set up loop
    v = np.zeros(len(ce.x_grid)) # Initial guess
    i = 0
    error = tol + 1
```

(continues on next page)

(continued from previous page)

```
    while i < max_iter and error > tol:
        v_new = T_numba(v, ce)

        error = np.max(np.abs(v - v_new))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

```
cen = create_cake_eating_model_numba()
```

```
in_time = time.time()
v_np = compute_value_function_numba(cen)
numba_time = time.time() - in_time
```

```
Error at iteration 25 is 23.8003755134813.
Error at iteration 50 is 8.577577195046615.
Error at iteration 75 is 3.091330659691039.
Error at iteration 100 is 1.1141054204751981.
Error at iteration 125 is 0.4015199357729671.
Error at iteration 150 is 0.14470646660583952.
Error at iteration 175 is 0.05215173547298946.
Error at iteration 200 is 0.018795314243106986.
```

```
Error at iteration 225 is 0.006773769545986852.
Error at iteration 250 is 0.002441244305884993.
Error at iteration 275 is 0.0008798164334962166.
Error at iteration 300 is 0.00031708295477983484.
Error at iteration 325 is 0.00011427565664234862.

Converged in 329 iterations.
```

```
ratio = numba_time/jax_time
print(f"JAX implementation is {ratio} times faster than NumPy.")
print(f"JAX time: {jax_time}")
print(f"Numba time: {numba_time}")
```

```
JAX implementation is 2.781856551302953 times faster than NumPy.
JAX time: 1.1836638450622559
Numba time: 3.29278302192688
```

Part V

Data and Empirics

CHAPTER

EIGHTEEN

MAXIMUM LIKELIHOOD ESTIMATION

GPU

This lecture was built using a machine with JAX installed and access to a GPU.

To run this lecture on Google Colab, click on the "play" icon top right, select Colab, and set the runtime environment to include a GPU.

To run this lecture on your own machine, you need to install Google JAX.

18.1 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
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:52:04 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
```

(continues on next page)

(continued from previous page)

```
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
```

```
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   28C    P0    37W / 300W |      0MiB / 16160MiB |      2%      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)
```

18.2 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.

18.2.1 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`.

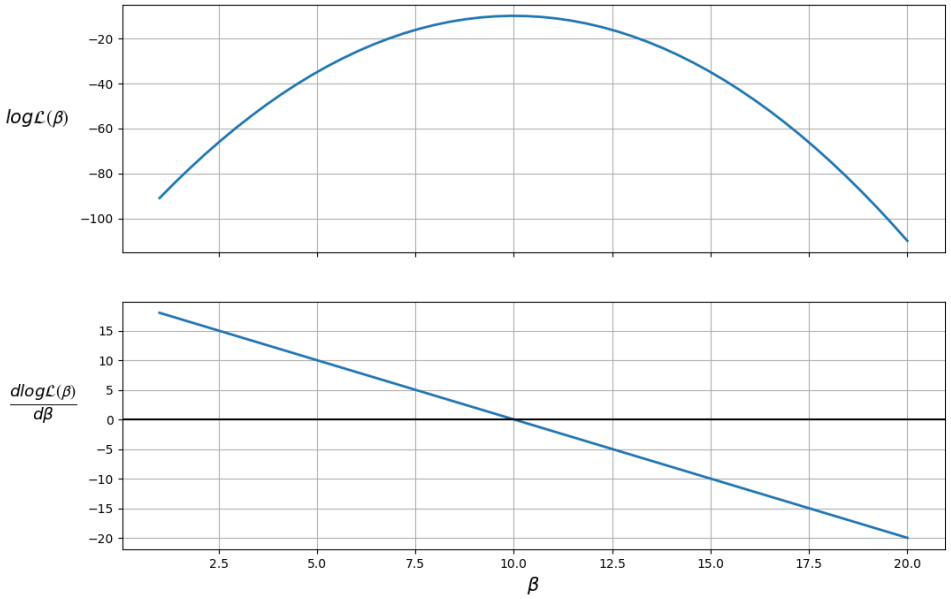
```
@jax.jit
def logL(β):
    return -(β - 10) ** 2 - 10
```

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.

18.2.2 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:

$$f(y_1, y_2, \ldots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

$$\text{where } \mu_i = \exp(\mathbf{x}_i'\beta) = \exp(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_k x_{ik})$$

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()}')

    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],
```

(continues on next page)

(continued from previous page)

```
                [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([[-2.56406008e-13],
       [-6.50202114e-13],
       [-5.05540054e-13]], dtype=float64)
```

18.3 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:                Mon, 01 Apr 2024   Pseudo R-squ.:                  0.2546
Time:                        17:52:08   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 18.3.1

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

$$\lambda_t = \exp(\beta_0 + \beta_1 x_t + \beta_2 x_t^2) \tag{18.1}$$

Simulate the values of x_t by sampling from a normal distribution and λ_t by using (18.1) 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 to Exercise 18.3.1

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.PRNGKey(seed)
x = jax.random.normal(key, shape)
```

We compute λ using (18.1)

```
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            -4.5444745e+07  ['-1.49', '0.312', '0.794']
1            -1.6303734e+07  ['-2.42', '0.311', '0.79']
2            -5689832.6      ['-3.22', '0.31', '0.78']
3            -1869457.7      ['-3.73', '0.306', '0.756']
4            -510284.64      ['-3.71', '0.297', '0.705']
5            -28066.381      ['-3.2', '0.283', '0.63']
6            127470.33       ['-2.77', '0.268', '0.563']
7            163044.64       ['-2.57', '0.257', '0.52']
8            166608.12       ['-2.51', '0.252', '0.503']
9            166666.4        ['-2.5', '0.251', '0.5']
10           166666.42       ['-2.5', '0.251', '0.5']
11           166666.42       ['-2.5', '0.251', '0.5']
Number of iterations: 12
β_hat = [-2.50016027  0.25079345  0.50008394]
```

The maximum likelihood estimates are similar to the true parameter values.

Part VI

Other

CHAPTER

NINETEEN

TROUBLESHOOTING

Contents

This page is for readers experiencing errors when running the code from the lectures.

19.1 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.

19.2 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

TWENTY

REFERENCES

CHAPTER

TWENTYONE

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
aiyagari_jax	2024-04-01 17:20	cache	65.97	✅
arellano	2024-04-01 17:21	cache	30.03	✅
cake_eating_numerical	2024-04-01 17:21	cache	21.03	✅
ifp_egm	2024-04-01 17:24	cache	180.97	✅
intro	2024-04-01 17:24	cache	1.16	✅
inventory_dynamics	2024-04-01 17:25	cache	72.43	✅
inventory_ssd	2024-04-01 17:50	cache	1494.46	✅
jax_intro	2024-04-01 17:50	cache	26.97	✅
kesten_processes	2024-04-01 17:51	cache	20.98	✅
lucas_model	2024-04-01 17:51	cache	20.78	✅
markov_asset	2024-04-01 17:51	cache	18.81	✅
mle	2024-04-01 17:52	cache	16.02	✅
newtons_method	2024-04-01 17:54	cache	160.49	✅
opt_invest	2024-04-01 22:27	cache	23.67	✅
opt_savings_1	2024-04-01 17:55	cache	38.21	✅
opt_savings_2	2024-04-01 22:27	cache	17.69	✅
short_path	2024-04-01 17:56	cache	6.89	✅
status	2024-04-01 17:56	cache	4.03	✅
troubleshooting	2024-04-01 17:24	cache	1.16	✅
wealth_dynamics	2024-04-01 22:29	cache	108.05	✅
zreferences	2024-04-01 17:24	cache	1.16	✅

These lectures are built on `linux` instances through `github actions` and `amazon web services (aws)` to enable access to a `gpu`. These lectures are built on a p3.2xlarge that has access to `8 vcpu's`, a `V100 NVIDIA Tesla GPU`, and `61 Gb` of memory.

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
```

```
/opt/conda/envs/quantecon/lib/python3.11/pty.py:89: 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()
```

```
Mon Apr  1 17:56:24 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   28C    P0    37W / 300W |    310MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

BIBLIOGRAPHY

[Are08] Cristina Arellano. Default risk and income fluctuations in emerging economies. *The American Economic Review*, pages 690–712, 2008.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.

INDEX

F

K

L

M