Quantitative Economics with Python using JAX

Thomas J. Sargent & John Stachurski

Aug 14, 2023
## CONTENTS

### I Introduction

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

2 An Introduction to JAX
   2.1 JAX as a NumPy Replacement ................................. 9
   2.2 Random Numbers .............................................. 13
   2.3 JIT Compilation ............................................... 14
   2.4 Functional Programming .................................... 15
   2.5 Gradients ....................................................... 17
   2.6 Writing vectorized code .................................... 18
   2.7 Exercises ....................................................... 21

3 Newton’s Method via JAX
   3.1 Overview ....................................................... 25
   3.2 The Equilibrium Problem .................................... 26
   3.3 Computation .................................................... 27
   3.4 Exercises ....................................................... 29

4 An Asset Pricing Problem
   4.1 Overview ....................................................... 35
   4.2 Pricing a single payoff ..................................... 35
   4.3 Pricing a cash flow .......................................... 36
   4.4 Choosing the stochastic discount factor ................. 37
   4.5 Solving for the price-dividend ratio .................... 37
   4.6 Code ............................................................... 38
   4.7 An Extended Example ....................................... 42
   4.8 Numpy Version .................................................. 43
   4.9 JAX Version ..................................................... 47
   4.10 A memory-efficient JAX version ......................... 49

### II Simulation

5 Inventory Dynamics
   5.1 Overview ....................................................... 55
   5.2 Sample paths .................................................. 56
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Example 1: marginal distributions</td>
<td></td>
<td>57</td>
</tr>
<tr>
<td>5.4</td>
<td>Example 2: restock frequency</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>Inventory Management Model</td>
<td>6.1 A model with constant discounting</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.2 Time varying discount rates</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.3 Numba implementation</td>
<td>71</td>
</tr>
<tr>
<td>7</td>
<td>Kesten Processes and Firm Dynamics</td>
<td>7.1 Overview</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2 Kesten processes</td>
<td>76</td>
</tr>
<tr>
<td>8</td>
<td>Wealth Distribution Dynamics</td>
<td>8.1 Lorenz Curves and the Gini Coefficient</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.2 A Model of Wealth Dynamics</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.3 Implementation using JAX</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.4 Applications</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.5 Exercises</td>
<td>96</td>
</tr>
<tr>
<td>III</td>
<td>Data and Empirics</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>9</td>
<td>Maximum Likelihood Estimation</td>
<td>9.1 Overview</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.2 MLE with numerical methods (JAX)</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.3 MLE with statsmodels</td>
<td>109</td>
</tr>
<tr>
<td>IV</td>
<td>Dynamic Programming</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>10</td>
<td>Shortest Paths</td>
<td>10.1 Overview</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10.2 Solving for Minimum Cost-to-Go</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10.3 Exercises</td>
<td>118</td>
</tr>
<tr>
<td>11</td>
<td>Optimal Investment</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>12</td>
<td>Optimal Savings</td>
<td>12.1 Overview</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.2 Model primitives</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.3 Operators</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.4 JIT compiled versions</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.5 Solvers</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.6 Plots</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.7 Tests</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>141</td>
</tr>
<tr>
<td>13</td>
<td>Endogenous Grid Method</td>
<td>13.1 Overview</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.2 Setup</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.3 Solution method</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.4 Solutions</td>
<td>153</td>
</tr>
<tr>
<td>14</td>
<td>Default Risk and Income Fluctuations</td>
<td>14.1 Overview</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14.2 Structure</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14.3 Equilibrium</td>
<td>162</td>
</tr>
</tbody>
</table>
This website presents a set of lectures on quantitative economic modeling using GPUs and Google JAX. The lectures are designed and written by Thomas J. Sargent and John Stachurski.

- **Introduction**
  - About
  - An Introduction to JAX
  - Newton's Method via JAX
  - An Asset Pricing Problem

- **Simulation**
  - Inventory Dynamics
  - Inventory Management Model
  - Kesten Processes and Firm Dynamics
  - Wealth Distribution Dynamics

- **Data and Empirics**
  - Maximum Likelihood Estimation

- **Dynamic Programming**
  - Shortest Paths
  - Optimal Investment
  - Optimal Savings
  - Endogenous Grid Method
  - Default Risk and Income Fluctuations
  - The Aiyagari Model
  - Cake Eating: Numerical Methods

- **Other**
  - Troubleshooting
  - References
  - Execution Statistics
Part I

Introduction
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
Quantitative Economics with Python using JAX

- 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.
GPU
This lecture was built using hardware that has 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.

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:

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

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

```python
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))
```
Quantitative Economics with Python using JAX

0.9

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

13.490001

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

```python
a
```

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

```python
type(a)
```

```
jaxlib.xla_extension.ArrayImpl
```

Even scalar-valued maps on arrays return JAX arrays.

```python
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:

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

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

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

```python
linalg.solve(B, A)
```

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

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

```
(Array([0.99999994, 0.99999994], dtype=float32),
 Array([[1. , 0. ],
        [0. , 1. ]], dtype=float32))
```
2.1.2 Differences

One difference between NumPy and JAX is that, when running on a GPU, JAX 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
da = np.linspace(0, 1, 3)
da
```

```
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
```

Cell In[19], line 1
----> 1 a[0] = 1

File /opt/conda/envs/quantecon/lib/python3.10/site-packages/jax/_src/numpy/array_-methods.py:263, in _unimplemented_setitem(self, i, x)
 258 def _unimplemented_setitem(self, i, x):
 259     msg = (""'/" object does not support item assignment. JAX arrays are "
 260     "immutable. Instead of `x[idx] = y`, use `x = x.at[idx].
 261   _set(y)`
```
In line with immutability, JAX does not support inplace operations:

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

```python
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

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

```python
id(a)
```

```
1148124832
```

```python
a
```

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

```python
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:

```python
id(a)
```

```
1148124832
```
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.

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

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

```python
key = random.PRNGKey(1)
type(key)
```

```
jaxlib.xla_extension.ArrayImpl
```

```python
print(key)
```

```
[0 1]
```

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

```python
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:

```python
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:

```python
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)
```

```python
random.normal(subkey, (3, 3))
```
The function below produces \( k \) (quasi-) independent random \( n \times n \) matrices using this procedure.

```python
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
```

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

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 \( (\text{len}, \; ) \) for the shape, as in

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

### 2.3 JIT Compilation

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

Consider the following pure Python function.

```python
def f(x, p=1000):
    return sum((k*x for k in range(p)))
```

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

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

How long does the function take to execute?

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

This code is not particularly fast.

While it is run on the GPU (since \( x \) is a JAX array), each vector \( k \times x \) has to be instantiated before the final sum is computed.

If we JIT-compile the function with JAX, then the operations are fused and no intermediate arrays are created.

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

Let's run once to compile it:

```python
f_jit(x)
```

And now let's time it.

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

Note the large speed gain.

## 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:

```python
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.

```python
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:

```python
x = np.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

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

Let’s take the derivative:

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

```python
f_prime(10.0)
```

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

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

```python
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.

```python
@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 17.3 s, sys: 2.23 s, total: 19.5 s
Wall time: 11.3 s
```

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

(Notice that we used a NumPy array for \( z_{\text{loops}} \) 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

\[ z_{\text{bad}} = f(x, y) \]

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

\[ z_{\text{bad}}.\text{shape} \]

\[(80,)\]

Here is what we actually wanted:
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 112 ms, sys: 0 ns, total: 112 ms
Wall time: 173 ms
```

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

```
CPU times: user 817 µs, sys: 0 ns, total: 817 µs
Wall time: 346 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 115 ms, sys: 0 ns, total: 115 ms
Wall time: 178 ms
```

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

```
CPU times: user 755 µs, sys: 0 ns, total: 755 µs
Wall time: 383 µs
```

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

### 2.6. Writing vectorized code
By comparison, the flat array $x$ is just

$$x.nbytes \quad \# \text{and } y \text{ is just a pointer to } x$$

48000

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

So let’s try a different approach using \texttt{jax.vmap}

First we vectorize $f$ in $y$.

$$f._\text{vec}_y = \text{jax.vmap}(f, \text{in_axes}=\text{(None, 0)})$$

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 $x$.

$$f._\text{vec} = \text{jax.vmap}(f._\text{vec}_y, \text{in_axes}=(0, \text{None}))$$

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

$$\text{%%time}$$
$$\text{z._vmap} = f._\text{vec}(x, y)$$

CPU times: user 104 ms, sys: 11.1 ms, total: 115 ms
Wall time: 176 ms

$$\text{%%time}$$
$$\text{z._vmap} = f._\text{vec}(x, y)$$

CPU times: user 2.4 ms, sys: 0 ns, total: 2.4 ms
Wall time: 2.04 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:

$$\text{jnp.allclose(z._vmap, z._mesh)}$$

Array(True, dtype=bool)
2.7 Exercises

Exercise 2.7.1

Recall that Newton’s method for solving for the root of \( f \) involves iterating on

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

Write a function called newton that takes a function \( f \) plus a guess \( x_0 \) and returns an approximate fixed point.

Your newton implementation should use automatic differentiation to calculate \( f' \).

Test your newton method on the function shown below.

\[
f(x) = \lambda x: \text{jnp.sin}(4 \times (x - 1/4)) + x + x^20 - 1
\]

\[
x = \text{jnp.linspace}(0, 1, 100)
\]

```python
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()
```

Solution to Exercise 2.7.1

Here’s a suitable function:
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

Let's try it:

newton(f, 0.2)

Array(0.4082935, dtype=float64, weak_type=True)

This number looks good, given the figure.

Exercise 2.7.2

In an earlier exercise on 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 2.7.2

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:

```python
compute_call_price_jax()
```

Array(180876.48840921, dtype=float64)

And now let's time it:

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

CPU times: user 1.9 ms, sys: 248 µs, total: 2.15 ms
Wall time: 111 ms

Array(180876.48840921, dtype=float64)
CHAPTER THREE

NEWTON’S METHOD VIA JAX

GPU

This lecture was built using hardware that has 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

In this lecture we highlight some of the capabilities of JAX, including JIT compilation and automatic differentiation.

The application is computing equilibria via Newton’s method, which we discussed in a more elementary QuantEcon lecture.

Here our focus is on how to apply JAX to this problem.

We use the following imports in this lecture

```python
import jax
import jax.numpy as jnp
from scipy.optimize import root
```

Let’s check the GPU we are running

```bash
!nvidia-smi
```
### 3.2 The Equilibrium Problem

In this section we describe the market equilibrium problem we will solve with JAX. We begin with a two good case, which is borrowed from an earlier lecture. Then we shift to higher dimensions.

#### 3.2.1 The Two Goods Market Equilibrium

Assume we have a market for two complementary goods where demand depends on the price of both components. We label them good 0 and good 1, with price vector \( p = (p_0, p_1) \).

Then the supply of good \( i \) at price \( p \) is,

\[
q_s^i(p) = b_i \sqrt{p_i}
\]

and the demand of good \( i \) at price \( p \) is,

\[
q_d^i(p) = \exp(-(a_{i0}p_0 + a_{i1}p_1)) + c_i
\]

Here \( a_{ij}, b_i \) and \( c_i \) are parameters for \( n \times n \) square matrix \( A \) and \( n \times 1 \) parameter vectors \( b \) and \( c \).

The excess demand function is,

\[
e_i(p) = q_d^i(p) - q_s^i(p), \quad i = 0, 1
\]

An equilibrium price vector \( p^* \) satisfies \( e_i(p^*) = 0 \).

We set

\[
A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \quad b = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \quad \text{and} \quad c = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}
\]

for this particular question.
3.2.2 A High-Dimensional Version

Let’s now shift to a linear algebra formulation, which allows us to handle arbitrarily many goods.

The supply function remains unchanged,

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

The demand function becomes

\[ q^d(p) = \exp(-A \cdot p) + c \]

Our new excess demand function is

\[ e(p) = \exp(-A \cdot p) + c - b \sqrt{p} \]

The function below calculates the excess demand for the given parameters

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

3.3 Computation

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

3.3.1 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 price vector is

\[ p_{n+1} = p_n - J_e(p_n)^{-1}e(p_n) \] (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 `jax.jacobian()`.

```python
def newton(f, x_0, tol=1e-5, max_iter=15):
    x = x_0
    f_jac = jax.jacobian(f)
    q = jax.jit(lambda x: 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)
        if jnp.any(jnp.isnan(y)):
            raise Exception('Solution not found with NaN generated')
        error = jnp.linalg.norm(x - y)
        x = y
    print(f'iteration {n}, error = {error}')
    print('\n' + f'Result = {x} \n')
    return x
```
3.3.2 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 $b$ and $c$ to 1.

```python
# 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 $p_0$

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

By leveraging the power of Newton’s method, JAX accelerated linear algebra, automatic differentiation, and a GPU, we obtain a relatively small error for this very large problem in just a few seconds:

```python
%%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 = 0.10971643775701523
iteration 4, error = 5.197196878725663e-05
iteration 5, error = 1.2309188605286181e-05
iteration 6, error = 4.6261807256087195e-06
Result = [1.4999796 1.503175 1.4918782 ... 1.4914232 1.4956646 1.4976945]

CPU times: user 4.9 s, sys: 1.36 s, total: 6.26 s
Wall time: 4.32 s
```

```python
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.

```python
%%time
```

(continues on next page)
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 25s, sys: 365 ms, total: 2min 25s
Wall time: 2min 25s

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

Array(7.1525574e-07, dtype=float32)

The result is also less accurate.

### 3.4 Exercises

#### Exercise 3.4.1

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

\[
k_{t+1} = g(k_t) \quad \text{where} \quad g(k) := sA\alpha + (1 - \delta)k
\]

However \(k_t\) is now a \(3 \times 1\) vector.

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

- \(k_{10} = (1, 1, 1)\)
- \(k_{20} = (3, 5, 5)\)
- \(k_{30} = (50, 50, 50)\)

**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 \(s = 0.3, \alpha = 0.3, \delta = 0.4\) and starting value:

\(k_0 = (1, 1, 1)\)
The result should converge to the analytical solution.

Solution to Exercise 3.4.1

Let's first define the parameters for this problem

```python
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(5.0, 3)]
```

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

```python
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

```python
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

Result = [3.8405814 3.870718 3.4109194]

CPU times: user 818 ms, sys: 29.1 ms, total: 847 ms
Wall time: 682 ms

Attempt 2: Starting value is [3. 5. 5.]
We find that the results are invariant to the starting values given the well-defined property of this question.

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

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

```python
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

```python
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

Result = [1.7846744 1.7846744 1.7846744]
```

(continues on next page)
The result is very close to the ground truth 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):

```python
# 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

Result = [1.78467418 1.78467418 1.78467418]
```

We can see it steps towards a more accurate solution.

---

**Exercise 3.4.2**

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:

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

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

\[ p_1^0 = (5, 5, 5) \]
\[ p_2^0 = (1, 1, 1) \]
\[ p_3^0 = (4.5, 0.1, 4) \]

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 3.4.2**

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(f'-*64')
    attempt += 1

Attempt 1: Starting value is [5. 5. 5.]

iteration 1, error = 9.243805733085065

Exception Traceback (most recent call last)
File <timed exec>:1
Cell In[4], line 13, in newton(f, x_0, tol, max_iter)
  11 y = q(x)
  12 if jnp.any(jnp.isnan(y)):
----> 13 raise Exception('Solution not found with NaN generated')
  14 error = jnp.linalg.norm(x - y)
  15 x = y

Exception: Solution not found with NaN generated

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 = 0.00520973031322213
iteration 6, error = 4.3632751705775364e-06
iteration 7, error = 3.0460818773540415e-12
iteration 8, error = 0.0

Result = [1.49744442 1.49744442 1.49744442]

(continues on next page)
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.
4.1 Overview

In this lecture we consider some asset pricing problems and use them to illustrate some foundations of JAX programming. 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 (4.5.5).

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

```python
!pip install quantecon
```

Below we use the following imports

```python
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.

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

4.2 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

\[
P_t = \beta \mathbb{E}_t D_{t+1}
\]  

(4.1)

Here \( \beta \) is a constant discount factor and \( \mathbb{E}_t D_{t+1} \) is the expectation of \( D_{t+1} \) at time \( t \).
Roughly speaking, (4.2.1) 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 (4.2.1) to

\[ P_t = \mathbb{E}_t M_{t+1} D_{t+1} \]  

(4.2)

In this expression, \( M_{t+1} \) replaces \( \beta \) 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.

### 4.3 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 (4.2.2), the price should be

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

(4.3)

Because prices generally grow over time, which complicates analysis, it will be easier for us to solve for the \textit{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 (4.3) by \( D_t \) to get

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

(4.4)

We can also write this as

\[ V_t = \mathbb{E}_t \left[ M_{t+1} \exp(G_{t+1}^d)(1 + V_{t+1}) \right] \]

(4.5)

\[ G_{t+1}^d = \ln \frac{D_{t+1}}{D_t} \]

is the growth rate of dividends. Our aim is to solve (4.3.3) but before that we need to specify

1. the stochastic discount factor \( M_{t+1} \) and
2. the growth rate of dividends \( G_{t+1}^d \)
4.4 Choosing the stochastic discount factor

We will adopt the stochastic discount factor described in [Luc78], which has the form

\[ M_{t+1} = \beta \frac{u'(C_{t+1})}{u'(C_t)} \]  

(4.6)

where \( u \) is a utility function and \( C_t \) is time \( t \) consumption of a representative consumer.

(An explanation of the ideas behind this expression is given in a later lecture and we omit further details and motivation.)

For utility, we’ll assume the constant relative risk aversion (CRRA) specification

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

(4.7)

Inserting the CRRA specification into (4.6) and letting

\[ G^c_{t+1} = \ln \left( \frac{C_{t+1}}{C_t} \right) \]

the growth rate rate of consumption, we obtain

\[ M_{t+1} = \beta \left( \frac{C_{t+1}}{C_t} \right)^{-\gamma} = \beta \exp(G^c_{t+1})^{-\gamma} = \beta \exp(-\gamma G^c_{t+1}) \]  

(4.8)

4.5 Solving for the price-dividend ratio

Substituting (4.4.3) into (4.5) gives the price-dividend ratio formula

\[ V_t = \beta E_t \left[ \exp(G^d_{t+1} - \gamma G^c_{t+1})(1 + V_{t+1}) \right] \]  

(4.9)

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

\[ G^c_{t+1} = \mu_c + X_t + \sigma_c \epsilon_{c,t+1} \]

\[ G^d_{t+1} = \mu_d + X_t + \sigma_d \epsilon_{d,t+1} \]

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 \( V_t = v(X_t) \) for some unknown function \( v \).

By (4.5.1), the unknown function \( v \) satisfies the equation

\[ v(X_t) = \beta E_t \left\{ \exp[a + (1-\gamma)X_t + \sigma_d \epsilon_{d,t+1} - \gamma \sigma_c \epsilon_{c,t+1}](1 + v(X_{t+1})) \right\} \]  

(4.10)

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 \( EY = \exp(c^2/2) \).
This yields

\[ v(X_t) = \beta \mathbb{E}_t \left\{ \exp \left[ a + (1 - \gamma)X_t + \frac{\sigma_a^2 + \gamma^2 \sigma_e^2}{2} \right] (1 + v(X_{t+1})) \right\} \]  

(4.11)

Conditioning on \( X_t = x \), we can write this as

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

(4.12)

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

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

(4.13)

for \( i = 1, \ldots, N \).

We can write (4.5.5) in vector form as

\[ v = K(\mathbb{1} + v) \]  

(4.14)

where \( K \) is the matrix defined by

\[ K[i, j] = \beta \left\{ \exp \left[ a + (1 - \gamma)x[i] + \frac{\sigma_a^2 + \gamma^2 \sigma_e^2}{2} \right] \right\} P[i, j] \]

Notice that (4.5.6) 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

\[ v = (I - K)^{-1} K\mathbb{1} \]  

(4.15)

whenever \( r(K) \), the spectral radius of \( K \), is strictly less than one.

Once we specify \( P \) and all the parameters, we can obtain \( K \) and then compute the solution (4.5.7).

### 4.6 Code

We will use the power iteration algorithm to check the spectral radius condition.

The function below computes the spectral radius of \( A \).

```python
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
```

(continues on next page)
b_k1 = jnp.dot(A, b_k)

# calculate the norm
b_k1_norm = jnp.linalg.norm(b_k1)

# Record the current estimate of the spectral radius
sr = jnp.sum(b_k1 * b_k) / jnp.sum(b_k * b_k)

# re-normalize the vector and continue
b_k = b_k1 / b_k1_norm

return sr

power_iteration_sr = jax.jit(power_iteration_sr, static_argnums=(1, 2))

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 \( \rho, \sigma \) 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,
    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 \( K \).
To exploit the parallelization capabilities of JAX, we use a vectorized (i.e., loop-free) implementation.

```python
def compute_K(model):
    # Set up
    P, S, β, γ, μ_c, μ_d, σ_c, σ_d = model
    N = len(S)
    # Reshape and multiply pointwise using broadcasting
    x = jnp.reshape(S, (N, 1))
    a = μ_d - γ * μ_c
    e = jnp.exp(a + (1 - γ) * x + (σ_d**2 + γ**2 * σ_c**2) / 2)
    K = β * e * P
    return K
```

Just to double check, let’s write a loop version and check we get the same matrix.

```python
def compute_K_loop(model):
    # Set up
    P, S, β, γ, μ_c, μ_d, σ_c, σ_d = model
    N = len(S)
    K = np.empty((N, N))
    a = μ_d - γ * μ_c
    for i, x in enumerate(S):
        for j, y in enumerate(S):
            e = jnp.exp(a + (1 - γ) * x + (σ_d**2 + γ**2 * σ_c**2) / 2)
            K[i, j] = β * e * P[i, j]
    return K
```

```python
model = create_model(N=10)
K1 = compute_K(model)
K2 = compute_K_loop(model)
jnp.allclose(K1, K2)
```

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

Now we can compute the price-dividend ratio:

```python
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:
    ```
Here's a plot of $v$ as a function of the state for several values of $\gamma$.

```python
model = create_model()
S = model.S
ys = jnp.linspace(2.0, 3.0, 5)

fig, ax = plt.subplots()

for $\gamma$ in ys:
    model = create_model($\gamma$)
    v = price_dividend_ratio(model)
    ax.plot(S, v, lw=2, alpha=0.6, label=rf'$\gamma = {\gamma}$')

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

### 4.7 An Extended Example

One problem with the last set is that volatility is constant through time (i.e., $\sigma_c$ and $\sigma_d$ 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

$$\begin{align*}
G_{t+1}^c &= \mu_c + Z_t + \sigma \exp(H_t^c)\epsilon_{c,t+1} \\
G_{t+1}^d &= \mu_d + Z_t + \bar{\sigma} \exp(H_t^d)\epsilon_{d,t+1}
\end{align*}$$

where $\{Z_t\}$ is a finite Markov chain and $\{H_t^c\}$ and $\{H_t^d\}$ are AR(1) processes of the form

$$\begin{align*}
H_{t+1}^c &= \rho_c H_t^c + \gamma \eta_{c,t+1} \\
H_{t+1}^d &= \rho_d H_t^d + \gamma \eta_{d,t+1}
\end{align*}$$

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 (4.5.2) to accommodate the new growth specifications, we find that $v$ satisfies

$$v(X_t) = \beta \mathbb{E}_t \{ \exp(a + (1 - \gamma)Z_t + \bar{\sigma} \exp(H_t^d)\epsilon_{d,t+1} - \gamma \bar{\sigma} \exp(H_t^c)\epsilon_{c,t+1})(1 + v(X_{t+1})) \}$$

(4.16)

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

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

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

(4.17)

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

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

(4.18)

Using the definition of the state and setting

$$\kappa(h_c, h_d, z) := \exp \left[ a + (1 - \gamma)z + \bar{\sigma}^2 \exp(2h_d) + \gamma^2 \exp(2h_c) \right]$$

we can write this more explicitly

$$v(h_c, h_d, z) = \beta \sum_{h_c', h_d', z'} \kappa(h_c, h_d, z)(1 + v(h_c', h_d', z')) P(h_c, h_c') Q(h_d, h_d') R(z, z')$$

(4.19)

Here $P, Q, R$ are the stochastic matrices for, respectively, discretized $\{H_t^c\}$, discretized $\{H_t^d\}$ and $\{Z_t\}$.

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

The last expression becomes

$$v[i, j, k] = \beta \sum_{i', j', k'} \kappa[i, j, k](1 + v[i', j', k'])(1 + v[i', j', k']) P[i, i'] Q[j, j'] R[k, k']$$

(4.20)
We define the multi-index array $H$ by

$$H[i, j, k, i', j', k'] = \beta \kappa[i, j, k] P[i, i'][j, j'][k, k']$$

then (4.20) becomes

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

(4.21)

One way to understand this is to reshape $v$ into an $N$-vector, where $N = I \times J \times K$, and $H$ into an $N \times N$ matrix. Then we can write (4.7.6) as

$$v = H(1 + v)$$

Provided that $H$ is invertible, the solution is given by

$$v = (I - H)^{-1} H 1$$

### 4.8 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

$$Z_{t+1} = \rho_z Z_t + \sigma_z \xi_{t+1}$$

```python
SVModel = namedtuple('SVModel',
    ('P', 'hc_grid',
     'Q', 'hd_grid',
     'R', 'z_grid',
     'beta', 'gamma', 'bar_sigma',
     'mu_c', 'mu_d'))

def create_sv_model(beta=0.98,  # discount factor
                    gamma=2.5,   # coefficient of risk aversion
                    I=14,        # size of state space for $h_c$
                    rho_c=0.9,   # persistence parameter for $h_c$
                    sigma_c=0.01,# volatility parameter for $h_c$
                    J=14,        # size of state space for $h_d$
                    rho_d=0.9,   # persistence parameter for $h_d$
                    sigma_d=0.01,# volatility parameter for $h_d$
                    K=14,        # size of state space for $z$
                    bar_sigma=0.01,# volatility scaling parameter
                    rho_z=0.9,   # persistence parameter for $z$
                    sigma_z=0.01,# persistence parameter for $z$
                    mu_c=0.001,  # mean growth of consumption
                    mu_d=0.005):  # mean growth of dividends

    mc = qe.tauchen(I, rho_c, sigma_c)
    hc_grid = mc.state_values
    P = mc.P
```

(continues on next page)
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 SVMModel(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 $H$.

```python
def compute_H(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 multiply pointwise using broadcasting
    hc = np.reshape(hc_grid, (I, 1, 1))
    hd = np.reshape(hd_grid, (1, J, 1))
    z = np.reshape(z_grid, (1, 1, K))
    P = np.reshape(P, (I, 1, 1, I, 1))
    Q = np.reshape(Q, (1, J, 1, J, 1))
    R = np.reshape(R, (1, 1, K, 1, J, 1))
    # Compute H 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
    x = np.exp(a + (1 - γ) * z + b)
    κ = np.exp(a + (1 - γ) * z + b)
    H = β * κ * P * Q * R
    H = np.reshape(H, (N, N))
return H
```

Here’s our function to compute the price-dividend ratio for the stochastic volatility model.

```python
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
    """
    # 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)
```

(continues on next page)
N = I * J * K

H = compute_H(sv_model)
    # Make sure that a unique solution exists
    if test_stable:
        test_stability(H)
    # Compute v
    ones_array = np.ones(N)
    Id = np.identity(N)
    v = scipy.linalg.solve(Id - H, H @ 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

g.e.tic()
v = sv_pd_ratio(sv_model)
np_time = g.e.toc()

TOC: Elapsed: 0:00:1.17

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_xlabel("price-dividend ratio")
ax.set_ylabel("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()

4.9 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.)

```python
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 SVMModel(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 $H$. 
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.

```python
def compute_H_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 multiply pointwise using broadcasting
    hc = jnp.reshape(hc_grid, (I, 1, 1))
    hd = jnp.reshape(hd_grid, (1, J, 1))
    z = jnp.reshape(z_grid, (1, 1, K))
    P = jnp.reshape(P, (I, 1, I, 1, 1))
    Q = jnp.reshape(Q, (1, J, 1, J, 1))
    R = jnp.reshape(R, (1, K, 1, K, 1))
    # Compute H 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
    x = jnp.exp(a + (1 - γ) * z + b)
    H = β * x * P * Q * R
    H = jnp.reshape(H, (N, N))
    return H
```

Here’s the function that computes the solution.

```python
def sv_pd_ratio_jax(sv_model, shapes):
    # Computes the price-dividend ratio of the asset for the stochastic volatility model.

    # Parameters
    sv_model: an instance of Model
                contains primitives

    # Returns
    v : array_like
        price-dividend ratio

    # 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)
    shapes = I, J, K
    N = I * J * K

    H = compute_H_jax(sv_model, shapes)

    # Compute v, reshape and return
    ones_array = jnp.ones(N)
    Id = jnp.identity(N)
    v = jax.scipy.linalg.solve(Id - H, H @ 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.
Quantitative Economics with Python using JAX

```python
compute_H_jax = jax.jit(compute_H_jax, static_argnums=(1,))
sv_pd_ratio_jax = jax.jit(sv_pd_ratio_jax, static_argnums=(1,))
```

```python
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.

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

TOC: Elapsed: 0:00:0.37

And now let's see without compile time.

```python
qe.tic()
v_jax = sv_pd_ratio_jax(sv_model_jax, shapes).block_until_ready()
jnp_time_1 = qe.toc()
```

TOC: Elapsed: 0:00:0.01

Here's the ratio of times:

```python
jnp_time_1 / np_time
```

0.01368523362365497

Let's check that the NumPy and JAX versions realize the same solution.

```python
v = jax.device_put(v)
print(jnp.allclose(v, v_jax))
```

True

### 4.10 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 \(H\) not as a matrix, but rather as the linear operator that transforms \(g\) into \(Hg\) via

\[
(Hg)[i,j,k] = \beta \sum_{i',j',k'} \kappa[i,j,k]g[i',j',k']P[i,i']Q[j,j']R[k,k']
\]
def H_operator(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 multiply pointwise using broadcasting
    hc = jnp.reshape(hc_grid, (I, 1, 1))
    hd = jnp.reshape(hd_grid, (1, J, 1))
    z = jnp.reshape(z_grid, (1, 1, K))
    P = jnp.reshape(P, (I, 1, I, 1, 1))
    Q = jnp.reshape(Q, (1, J, 1, J, 1))
    R = jnp.reshape(R, (1, 1, K, 1, 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)
    H = β * κ * P * Q * R
    Hg = jnp.sum(H * g, axis=(3, 4, 5))
    return Hg

The next function modifies our earlier `power_iteration_sr` function so that it can act on linear operators rather than matrices, also the spectral radius of the transition matrix less than one ensures the convergence of our calculations in the model.

def update_g(H_operator, sv_model, shapes, num_iterations=20):
    g_k = jnp.ones(shapes)
    for _ in range(num_iterations):
        g_k1 = H_operator(g_k, sv_model, shapes)
        sr = jnp.sum(g_k1 * g_k) / jnp.sum(g_k * g_k)
        g_k1_norm = jnp.linalg.norm(g_k1)
        g_k = g_k1 / g_k1_norm
    return sr

Let’s check the output

g.e.tic()
sr = update_g(H_operator, sv_model, shapes)
g.e.toc()
print(sr)

TOC: Elapsed: 0:00:1.77
0.9975973011724226

Now we write a version of the solution function for the price-dividend ratio that acts directly on the linear operator `H_operator`.

def sv_pd_ratio_jax_multi(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
    # Compute v
    ones_array = np.ones((I, J, K))
    # Set up the operator g -> g
    (continues on next page)
Let’s target these functions for JIT compilation.

```python
H_operator = jax.jit(H_operator, static_argnums=(2,))
sv_pd_ratio_jax_multi = jax.jit(sv_pd_ratio_jax_multi, static_argnums=(1,))
```

Let’s time the solution with compile time included.

```python
qe.tic()
v_jax_multi = sv_pd_ratio_jax_multi(sv_model, shapes).block_until_ready()
jnp_time_multi_0 = qe.toc()
```

TOC: Elapsed: 0:00:0.88

And now let’s see without compile time.

```python
qe.tic()
v_jax_multi = sv_pd_ratio_jax_multi(sv_model, shapes).block_until_ready()
jnp_time_multi_1 = qe.toc()
```

TOC: Elapsed: 0:00:0.00

Let’s verify the solution again:

```python
print(jnp.allclose(v, v_jax_multi))
```

True

Here’s the ratio of times between memory-efficient and direct version:

```python
jnp_time_multi_1 / jnp_time_1
```

0.43600999127980017

The speed is somewhat faster. In addition,

1. now we can work with much larger grids, and
2. the memory efficient version will be significantly faster with larger grids.

Here’s a moderately large example, where the state space has 15,625 elements.

```python
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)
```
Quantitative Economics with Python using JAX

```python
def sv_pd_ratio_jax_multi(sv_model, shapes):
    return sv_model.block_until_ready()
```

```python
qs.tic()
v_jax_multi = sv_pd_ratio_jax_multi(sv_model, shapes).block_until_ready()
njp_time_multi_2 = qs.toc()
```

TOC: Elapsed: 0:00:0.86

```python
jnp_time_multi_1 / jnp_time_1
```

0.4360999127980017

Chapter 4. An Asset Pricing Problem
Part II

Simulation
This lecture was built using hardware that has 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

- **Inventory Dynamics**
  - **Overview**
  - **Sample paths**
  - **Example 1: marginal distributions**
  - **Example 2: restock frequency**

### 5.1 Overview

This lecture explores JAX implementations of the exercises in the lecture on inventory dynamics.

We will use the following imports:

```python
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
```

Let's check the GPU we are running

```bash
!nvidia-smi
```
5.2 Sample paths

Consider a firm with inventory $X_t$.

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

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

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

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

(See our earlier lecture on inventory dynamics for background and motivation.)

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

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

where $\mu$ and $\sigma$ are parameters and $\{Z_t\}$ is IID and standard normal.

Here’s a namedtuple that stores parameters.

```python
Firm = namedtuple('Firm', ['s', 'S', 'mu', 'sigma'])
firm = Firm(s=10, S=100, mu=1.0, sigma=0.5)
```
### 5.3 Example 1: marginal distributions

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

We can approximate the distribution using a kernel density estimator. Kernel density estimators can be thought of as smoothed histograms.

We will use a kernel density estimator from scikit-learn.

Here is an example of using kernel density estimators and plotting the result.

```python
from sklearn.neighbors import KernelDensity

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

# Generate simulated data 
np.random.seed(42) 
sample_1 = np.random.normal(0, 2, size=10_000) 
sample_2 = np.random.gamma(2, 2, size=10_000) 

# Create a plot 
fig, ax = plt.subplots() 

# Plot the samples 
ax.hist(sample_1, alpha=0.2, density=True, bins=50) 
ax.hist(sample_2, alpha=0.2, density=True, bins=50) 

# Plot the KDE for each sample 
plot_kde(sample_1, ax, label=r'KDE over $X \sim N(0, 2)$') 
plot_kde(sample_2, ax, label=r'KDE over $X \sim Gamma(0, 2)$') 
ax.set_xlabel('value') 
ax.set_ylabel('density') 
ax.set_xlim([-5, 10]) 
ax.set_title('KDE of Simulated Normal and Gamma Data') 
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 investigate it using simulation. We can generate and plot the sequence \( \{ \psi_t \} \) at times \( t = 10, 50, 250, 500, 750 \) based on the kernel density estimator. We will see convergence, in the sense that differences between successive distributions are getting smaller. Here is one realization of the process in JAX using for loop

```python
# Define a jit-compiled function to update X and key @jax.jit
def update_X(X, firm, D):
    # Restock if the inventory is below the threshold
    res = jnp.where(X < firm.s,
                     jnp.maximum(firm.S - D, 0),
                     jnp.maximum(X - D, 0))
    return res

def shift_firms_forward(x_init, firm, 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(firm.mu + firm.sigma * Z)
        X = update_X(X, firm, D)
```
Quantitative Economics with Python using JAX

(continued from previous page)

```python
_, 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:]
```

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

fig, ax = plt.subplots()

# %time X = shift_firms_forward(x_init, firm, 
    # sample_dates, key).block_until_ready()

for i, date in enumerate(sample_dates):
    plot_kde(X[i, :], ax, label=f't = {date}')

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

Note that we did not JIT-compile the outer loop, since

5.3. Example 1: marginal distributions
1. *jit* compilation of the *for* loop can be very time consuming and
2. compiling outer loops only leads to minor speed gains.

### 5.3.1 Alternative implementation with *lax.scan*

An alternative to the *for* loop implementation is *lax.scan*.

Here is an example of the same function in *lax.scan*

```python
@jax.jit
def shift_firms_forward(x_init, firm, key, 
    num_firms=50_000, sim_length=750):
    s, S, mu, sigma = firm.s, firm.S, firm.mu, firm.sigma
    X = jnp.full((num_firms, ), x_init)
    Z = random.normal(key, shape=(sim_length, num_firms))
    D = jnp.exp(mu + sigma * Z)

    # Define the function for each update
    def update_X(X, D):
        res = jnp.where(X <= s,
                        jnp.maximum(S - D, 0),
                        jnp.maximum(X - D, 0))
        return res, res

    # Use lax.scan to perform the calculations on all states
    _, X_final = lax.scan(update_X, X, D)
    return X_final
```

The benefit of the *lax.scan* implementation is that we compile the whole operation.

The disadvantages are that
1. as mentioned above, there are only limited speed gains in accelerating outer loops,
2. *lax.scan* has a more complicated syntax, and, most importantly,
3. the *lax.scan* implementation consumes far more memory, as we need to have to store large matrices of random draws

Let's call the code to generate a cross-section that is in approximate equilibrium.

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

time X = shift_firms_forward(x_init, firm, key).block_until_ready()

for date in sample_dates:
    plot_kde(X[date, :], ax, label=f't = {date}')

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

CPU times: user 978 ms, sys: 0 ns, total: 978 ms
Wall time: 376 ms
Notice that by \( t = 500 \) or \( t = 750 \) the densities are barely changing.

We have reached a reasonable approximation of the stationary density.

You can test a few more initial conditions to show that they do not affect long-run outcomes.

For example, try rerunning the code above with all firms starting at \( X_0 = 20 \)

```python
x_init = 20.0

fig, ax = plt.subplots()

@time X = shift_firms_forward(x_init, firm, key).block_until_ready()

for date in sample_dates:
    plot_kde(X[date, :], ax, label=f't = {date}')

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

CPU times: user 895 ms, sys: 0 ns, total: 895 ms
Wall time: 310 ms

---

5.3. Example 1: marginal distributions
5.4 Example 2: restock frequency

Let’s go through another example where we calculate the probability of firms having restocks. Specifically we set the starting stock level to 70 ($X_0 = 70$), as we calculate the proportion of firms that need to order twice or more in the first 50 periods.

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

Again, we start with an easier for loop implementation

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

def compute_freq(firm, 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(firm.mu + firm.sigma * Z)
    n_restock, X, key = update_stock(n_restock, X, firm, D)
    key = random.fold_in(key, i)

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

cpu = random.PRNGKey(27)
freq = compute_freq(firm, cpu).block_until_ready()
print(f"Frequency of at least two stock outs = {freq}")

CPU times: user 992 ms, sys: 0 ns, total: 992 ms
Wall time: 1.25 s
Frequency of at least two stock outs = 0.4472379982471466

## 5.4.1 Alternative implementation with lax.scan

Now let’s write a lax.scan version that JIT compiles the whole function

```python
@jax.jit
def compute_freq(firm, key,
    x_init=70,
    sim_length=50,
    num_firms=1_000_000):

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

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

    # Define the function for each update
    def update_X(Xs, D):
        # Separate the inventory and restock counter
        X = Xs[0]
        restock_count = Xs[1]

        restock_count = jnp.where(X <= s,
            restock_count + 1,
            restock_count)
        X = jnp.where(X <= s,
            jnp.maximum(S - D, 0),
            ...)
```

(continues on next page)
\[
\begin{align*}
\text{jnp.maximum}(X - D, 0)) \\
Xs &= \text{jnp.vstack}((X, \text{restock\_count})) \\
\text{return} \ Xs, \ None \\
\end{align*}
\]

# Use lax.scan to perform the calculations on all states
\[
\begin{align*}
X_{\text{final}}, _ &= \text{lax.scan}(\text{update\_X}, Xs, D) \\
\text{return} \ \text{np.mean}(X_{\text{final}}[1] > 1) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{%time} \ freq &= \text{compute\_freq}(\text{firm}, \text{key}).\text{block\_until\_ready}() \\
\text{print}(\text{f"Frequency of at least two stock outs = {freq}"}) \\
\end{align*}
\]

CPU times: user 1.02 s, sys: 0 ns, total: 1.02 s
Wall time: 412 ms
Frequency of at least two stock outs = 0.44674399495124817
6.1 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 \( \pi_t \) be profits at time \( t \) and \( r > 0 \) be the interest rate, the value of the firm is

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

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

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

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

\[
\pi_t := X_t \land D_{t+1} - cA_t - \kappa 1\{A_t > 0\}.
\]
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 \( \kappa \) is a fixed cost of ordering inventory.

We can map our inventory problem into a dynamic program with state space \( X := \{0, \ldots, K \} \) and action space \( A := X \).

The feasible correspondence \( \Gamma \) is

\[ \Gamma(x) := \{0, \ldots, K - x\}, \]

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

\[ P(x, a, x') := P\{f(x, a, D) = x'\} \quad \text{when} \quad D \sim \phi. \]

When discounting is constant, the Bellman equation takes the form

\[ v(x) = \max_{a \in \Gamma(x)} \left\{ r(x, a) + \beta \sum_{d \geq 0} v(f(x, a, d)\phi(d)) \right\} \tag{6.1} \]

### 6.2 Time varying 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 \( \beta \) in (6.1) with a stochastic process \( (\beta_t) \) 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

\[ B((y, z), a, v) = r(y, a) + \beta(z) \sum_{a', z'} v(f(y, a, d), z')\phi(d)Q(z, z'). \tag{6.2} \]

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 (6.2) 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

---

**Chapter 6. Inventory Management Model**
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

Mon Aug 14 04:07:07 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03 Driver Version: 470.182.03 CUDA Version: 12.1 |
|-------------------------------+----------------------+----------------------+
| 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 35C P0 38W / 300W | 0MiB / 16160MiB | 2% Default |
| | | N/A |
| +-------------------------------+----------------------+----------------------+

Mon Aug 14 04:07:07 2023
+-----------------------------------------------------------------------------+
| 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", "x", "p", "z_vals", "Q"))

We need the following successive approximation function.

```python
def successive_approx(T,  # Operator (callable)
    x_0,  # Initial condition
    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:
        # (continues on next page)
```

6.2. Time varying discount rates 67
Quantitative Economics with Python using JAX

\[ x_{\text{new}} = T(x) \]
\[ \text{error} = \text{jnp}. \max(\text{jnp}. \abs(x_{\text{new}} - x)) \]

if \( \text{verbose} \) and \( k \% \text{print_step} == 0 \):
    print(f"Completed iteration \( k \) with error \( \text{error} \).")

\[ x = x_{\text{new}} \]
\[ k += 1 \]

if error > tolerance:
    print(f"Warning: Iteration hit upper bound \( \text{max_iter} \).")
elif verbose:
    print(f"Terminated successfully in \( k \) iterations.")
return x

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

\( \text{K} = 100 \)
\( \text{D_MAX} = 101 \)

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

def create_sdd_inventory_model:
    \( p=0.98, v=0.002, n_z=100, b=0.97 \), \( \# \text{Z state parameters} \)
    \( c=0.2, \kappa=0.8, p=0.6 \), \( \# \text{firm and demand parameters} \)
    use_jax=True:
    mc = qe.tauchen(n_z, \( \rho \), \( \nu \))
    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, \( \kappa=\kappa \), 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 \( B(x, z, a, v) = r(x, a) + \beta(z) \sum_x v(x') P(x, a, x') \).
    ""
    c, \( \kappa \), p, z_vals, Q = model
    z = z_vals[i_z]
    d_vals = jnp.arange(D_MAX)
    \( \phi \_\text{vals} = \text{demand}_\text{pdf}(p, d\_\text{vals}) \)
    revenue = jnp.sum(jnp.minimum(x, d_vals)*\( \phi \_\text{vals} \))
    profit = revenue - c * a - \( \kappa \) * (a > 0)
    v_R = jnp.sum(jnp.maximum(x - d_vals, 0) + a].T * \( \phi \_\text{vals} \), axis=1)
    cv = jnp.sum(v_R*Q[i_z])
    return profit + z * cv

We need to vectorize this function so that we can use it efficiently in JAX.

We apply a sequence of \texttt{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 $B(x, z, a, v) = r(x, a) + \beta(z) E_{x'} v(x') P(x, a, x')$.
    
    c, k, 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, k, 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, k, 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)
    o_star = get_greedy(v_star, model)
    return v_star, o_star

Now let’s create an instance and solve it.

model = create_sdd_inventory_model()
c, x, 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, o_star = solve_inventory_model(v_init, model)
jax_time = time.time() - in_time

6.2. Time varying discount rates 69
Completed iteration 25 with error 0.5613828428334706.
Completed iteration 50 with error 0.3776464347688062.
Completed iteration 75 with error 0.22720706235969011.
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 4.470604516625085e-05.
Completed iteration 425 with error 2.3586619946058818e-05.
Completed iteration 450 with error 1.2443945941242873e-05.
Completed iteration 475 with error 6.565178331641194e-06.
Completed iteration 500 with error 3.463639430378862e-06.
Completed iteration 525 with error 1.82732347659194e-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) + c_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

(continues on next page)
Quantitative Economics with Python using JAX

(continued from previous page)

```python
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()

6.3 Numba implementation

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

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

@njit
def B_numba(x, i_z, a, v, model):
    """
    The function $B(x, z, a, v) = r(x, a) + \beta(z) \Sigma_x' v(x') P(x, a, x')$.
    """
    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)
```

(continues on next page)
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 range(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 range(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.
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>0.22727062359689398</td>
</tr>
<tr>
<td>100</td>
<td>0.12872204940708798</td>
</tr>
<tr>
<td>125</td>
<td>0.06744149371262864</td>
</tr>
<tr>
<td>150</td>
<td>0.030374639547666504</td>
</tr>
<tr>
<td>175</td>
<td>0.01423099032948727</td>
</tr>
<tr>
<td>200</td>
<td>0.007396776219316337</td>
</tr>
<tr>
<td>225</td>
<td>0.0039122338304593529</td>
</tr>
<tr>
<td>250</td>
<td>0.002068091416653317</td>
</tr>
<tr>
<td>275</td>
<td>0.0010923075333622023</td>
</tr>
<tr>
<td>300</td>
<td>0.0005766427105911021</td>
</tr>
<tr>
<td>325</td>
<td>0.0003043321707281166</td>
</tr>
<tr>
<td>350</td>
<td>0.00016059073676188973</td>
</tr>
<tr>
<td>375</td>
<td>8.473334525405107e-05</td>
</tr>
<tr>
<td>400</td>
<td>4.470604518047594e-05</td>
</tr>
<tr>
<td>425</td>
<td>2.3586619960269672e-05</td>
</tr>
<tr>
<td>450</td>
<td>1.2443945934137446e-05</td>
</tr>
<tr>
<td>475</td>
<td>6.565178331641164e-06</td>
</tr>
<tr>
<td>500</td>
<td>3.4636394445897167e-06</td>
</tr>
<tr>
<td>525</td>
<td>1.827332347659194e-06</td>
</tr>
</tbody>
</table>

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.

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

JAX vectorized implementation is 770.0107410682273 faster than Numba's parallel implementation
CHAPTER
SEVEN

KESTEN PROCESSES AND FIRM DYNAMICS

Contents

• Kesten Processes and Firm Dynamics
  – Overview
  – Kesten processes

GPU

This lecture was built using hardware that has 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 what’s in Anaconda, this lecture will need the following libraries:

```bash
!pip install quantecon
```

7.1 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:

```python
import matplotlib.pyplot as plt
import quantecon as qe
import jax
import jax.numpy as jnp
from jax import random
```

Let's check the GPU we are running
7.2 Kesten processes

A Kesten process is a stochastic process of the form

\[ X_{t+1} = a_{t+1} X_t + \eta_{t+1} \]  

(7.1)

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.

7.2.1 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

$$\frac{s_{t+1}}{s_t} = a_{t+1} \quad (7.2)$$

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 (7.2) to

$$s_{t+1} = a_{t+1}s_t + b_{t+1} \quad (7.3)$$

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 (7.3) 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 (7.3) 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

$$s_{t+1} = e_{t+1}\mathbb{1}\{s_t < \bar{s}\} + (a_{t+1}s_t + b_{t+1})\mathbb{1}\{s_t \geq \bar{s}\} \quad (7.4)$$

The motivation behind and interpretation of (7.2.4) can be found in our earlier Kesten process lecture.

What can we say about dynamics?

Although (7.4) 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 (7.2.4).
@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 8.13 s, sys: 2.42 s, total: 10.6 s
Wall time: 8.34 s

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

```python
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.

7.2. Kesten processes
Alternative implementation with \texttt{lax.scan}

If the time horizon is not too large, we can try to further accelerate our code by replacing the \texttt{for} loop with \texttt{lax.scan}.

Note, however, that

1. as mentioned above, there is not much speed gain in accelerating outer loops,
2. \texttt{lax.scan} has a more complicated syntax, and, most importantly,
3. the \texttt{lax.scan} 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 \texttt{lax.scan} version:

```python
from jax import lax

@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(s, a_b_e_draws):
        a, b, e = a_b_e_draws
        s = jnp.where(s < s_bar,
                      jnp.exp(e),
                      jnp.exp(a) * s + jnp.exp(b))
        return s, None

    # Use lax.scan to perform the calculations on all states
    s_final, _ = lax.scan(update_s, s, (a_random, b_random, e_random))
    return s_final

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

In this case, $M$ is small enough for the code to run and we see some speed gain over the for loop implementation:
Quantitative Economics with Python using JAX

```python
#time data = generate_draws_lax().block_until_ready()
```

CPU times: user 3.53 ms, sys: 0 ns, total: 3.53 ms
Wall time: 34.9 ms

Here we produce the same rank-size plot:

```python
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

```python
#time generate_draws(M=500_000).block_until_ready()
```

CPU times: user 5.92 s, sys: 513 ms, total: 6.43 s
Wall time: 3.91 s

Array([2.389801 , 2.2558599, 3.3113828, ..., 2.7102313, 2.5520844,
6.4196172], dtype=float32)

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

7.2. Kesten processes
CHAPTER EIGHT

WEALTH DISTRIBUTION DYNAMICS

GPU
This lecture was built using hardware that has 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.

We will use the following imports.

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

Let’s check the GPU we are running

```bash
!nvidia-smi
```

(continues on next page)
8.1 Lorenz Curves and the Gini Coefficient

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

### 8.1.1 Lorenz Curves

One popular graphical measure of inequality is the **Lorenz curve**.

To illustrate, let us define a function `lorenz_curve_jax` that returns the cumulative share of people and the cumulative share of income earned.

```python
@jax.jit
def lorenz_curve_jax(y):
    n = y.shape[0]
    y = jnp.sort(y)
    s = jnp.concatenate((jnp.zeros(1), jnp.cumsum(y)))
    _cum_p = jnp.arange(1, n + 1) / n
    cum_income = s / s[n]
    cum_people = jnp.concatenate((jnp.zeros(1), _cum_p))
    return cum_people, cum_income
```

Let's suppose that

```python
n = 10_000  # Size of sample
rand_key = jax.random.PRNGKey(101)  # Set random key
w = jnp.exp(jax.random.normal(rand_key, shape=(n,)))  # Lognormal draws
```

is data representing the wealth of 10,000 households.

We can compute and plot the Lorenz curve as follows:

```python
%%time
f_vals, l_vals = lorenz_curve_jax(w)
```

CPU times: user 1.85 s, sys: 41.3 ms, total: 1.89 s
Wall time: 1.59 s

```python
%%time
# This will be much faster as it will use the jitted function
f_vals, l_vals = lorenz_curve_jax(w)
```

CPU times: user 749 µs, sys: 257 µs, total: 1.01 ms
Wall time: 475 µs
Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

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

```python
a_vals = (1, 2, 5)  # Pareto tail index
n = 10_000  # size of each sample

fig, ax = plt.subplots()
for a in a_vals:
    rand_key = jax.random.PRNGKey(a*100)
    u = jax.random.uniform(rand_key, shape=(n,))
    y = u**(-1/a)  # distributed as Pareto with tail index a
    f_vals, l_vals = lorenz_curve_jax(y)
    ax.plot(f_vals, l_vals, label=f'$a = {a}$')
ax.plot(f_vals, f_vals, label='equality')
ax.legend()
plt.show()
```
You can see that, as the tail parameter of the Pareto distribution increases, inequality decreases. This is to be expected, because a higher tail index implies less weight in the tail of the Pareto distribution.

### 8.1.2 The Gini Coefficient

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

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

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

Let's define a function to compute the Gini coefficient.

```python
@jax.jit
def gini_jax(y):
    n = y.shape[0]
    g_sum = 0

    # Define the function for each update
def sum_y_gini(g_sum, i):
        g_sum += jnp.sum(jnp.abs(y[i] - y))
    return g_sum, g_sum

    g_sum, _ = jax.lax.scan(sum_y_gini, 0, jnp.arange(n))
    return g_sum / (2 * n * jnp.sum(y))
```

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

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

n = 100

for a in a_vals:
    rand_key = jax.random.PRNGKey(a)
    y = jax.random.weibull_min(rand_key, 1, a, shape=(n,))
    ginis.append(gini_jax(y))
    ginis_theoretical.append(1 - 2**(-1/a))

fig, ax = plt.subplots()
ax.plot(a_vals, ginis, label='estimated gini coefficient')
ax.plot(a_vals, ginis_theoretical, label='theoretical gini coefficient')
ax.legend()
ax.set_xlabel("Weibull parameter $a$")
ax.set_ylabel("Gini coefficient")
plt.show()
## 8.2 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics. The model we will study is

\[ w_{t+1} = (1 + r_{t+1}) s(w_t) + y_{t+1} \]  

(8.1)

where

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

### 8.3 Implementation using JAX

Let’s define a model to represent the wealth dynamics.

```python
# NamedTuple Model
Model = namedtuple("Model", ("w_hat", "s_0", "c_y", "μ_y", "c_r", "μ_r", "σ_r", "a", "b", "σ_z", "z_mean", "z_var", "y_mean"))
```

Here’s a function to create the Model with the given parameters

```python
def create_wealth_model(w_hat=1.0, s_0=0.75, c_y=1.0, μ_y=1.0, σ_y=0.2, c_r=0.05, μ_r=0.1, σ_r=0.5, a=0.5, b=0.0, σ_z=0.1):
    
    """
    Create a wealth model with given parameters and return an instance of NamedTuple Model.
    """
    z_mean = b / (1 - a)
    z_var = σ_z**2 / (1 - a**2)
    exp_z_mean = jnp.exp(z_mean + z_var / 2)
    R_mean = c_r * exp_z_mean + jnp.exp(μ_r + c_r**2 / 2)
    y_mean = c_y * exp_z_mean + jnp.exp(μ_y + c_y**2 / 2)
    
    # Test a stability condition that ensures wealth does not diverge to infinity.
    a = R_mean * s_0
    if a >= 1:
        raise ValueError("Stability condition failed.")
    return Model(w_hat=w_hat, s_0=s_0, c_y=c_y, μ_y=μ_y, σ_r=σ_r, a=a, b=b, σ_z=σ_z)
```

(continues on next page)
\[
\begin{align*}
\sigma_y = \sigma_y, \ \ c_r = c_r, \ \ \mu_r = \mu_r, \ \ \sigma_r = \sigma_r, \ \ \ a = a, \\
b = b, \ \ \sigma_z = \sigma_z, \ \ \text{z_mean} = \text{z_mean}, \ \ \text{z_var} = \text{z_var}, \ \ \text{y_mean} = \text{y_mean}
\end{align*}
\]

The following function updates one period with the given current wealth and persistent state.

```python
def update_states_jax(arrays, wdy, size, rand_key):
    """Update one period, given current wealth w and persistent state z. They are stored in the form of tuples under the arrays argument."
    # Unpack w and z
    w, z = arrays
    rand_key, *subkey = jax.random.split(rand_key, 3)
    zp = wdy.a * z + wdy.b + wdy.c_z * jax.random.normal(rand_key, shape=size)

    # Update wealth
    y = wdy.c_y * jnp.exp(zp) + jnp.exp(wdy.p_y + wdy.c_y * jax.random.normal(subkey[0], shape=size))
    wp = y
    R = wdy.c_r * jnp.exp(zp) + jnp.exp(wdy.p_r + wdy.c_r * jax.random.normal(subkey[1], shape=size))
    wp += (w >= wdy.w_hat) * R * wdy.s_0 * w
    return wp, zp
```

Here's function to simulate the time series of wealth for individual households using a `for` loop and JAX.

```python
# Using JAX and for loop

def wealth_time_series_for_loop_jax(w_0, n, wdy, size, rand_seed=1):
    """Generate a single time series of length n for wealth given initial value w_0.
    * This implementation uses a `for` loop.
    * The initial persistent state z_0 for each household is drawn from the stationary distribution of the AR(1) process.
    """
    rand_key = jax.random.PRNGKey(rand_seed)
    rand_key, *subkey = jax.random.split(rand_key, n)
    w_0 = jax.device_put(w_0).reshape(size)
    z = wdy.z_mean + jnp.sqrt(wdy.z_var) * jax.random.normal(rand_key, shape=size)
    w = [w_0]
    for t in range(n-1):
        w_, z = update_states_jax((w[t], z), wdy, size, subkey[t])
        w.append(w_)
```

(continues on next page)
Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution using the above function.

```python
wdy = create_wealth_model()  # default model
ts_length = 200
size = (1,)

%%time
w_jax_result = wealth_time_series_for_loop_jax(wdy.y_mean,
                                              ts_length, wdy, size).block_until_ready()

CPU times: user 3.28 s, sys: 370 ms, total: 3.65 s
Wall time: 3.1 s
```

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

We can further try to optimize and speed up the compile time of the above function by replacing `for` loop with `jax.lax.scan`.

```python
def wealth_time_series_jax(w_0, n, wdy, size, rand_seed=1):
    ""
```

Generate a single time series of length n for wealth given initial value w_0.

* This implementation uses `jax.lax.scan`.

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

```
rand_key = jax.random.PRNGKey(rand_seed)
rand_key, subkey = jax.random.split(rand_key, n)
w_0 = jax.device_put(w_0).reshape(size)
z_init = wdy.z_mean + jnp.sqrt(wdy.z_var) * jax.random.normal(rand_key,
shape=size)
arrays = w_0, z_init
rand_sub_keys = jnp.array(subkey)
w_final = jnp.array([w_0])

# Define the function for each update
def update_w_z(arrays, rand_sub_key):
    wp, zp = update_states_jax(arrays, wdy, size, rand_sub_key)
    return (wp, zp), wp

arrays_last, w_values = jax.lax.scan(update_w_z, arrays, rand_sub_keys)
return jnp.concatenate((w_final, w_values))

# Create the jit function
wealth_time_series_jax = jax.jit(wealth_time_series_jax, static_argnums=(1,3,))
```

Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution and also observe the difference in time between `wealth_time_series_jax` and `wealth_time_series_for_loop_jax`.

```
wdy = create_wealth_model()  # default model
ts_length = 200
size = (1,)

%%time
w_jax_result = wealth_time_series_jax(wdy.y_mean, ts_length, wdy, size).block_until_ready()
```

```
CPU times: user 1.31 s, sys: 12.9 ms, total: 1.33 s
Wall time: 736 ms
```

Running the above function again will be even faster because of JAX's JIT.

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

```python
def update_cross_section_jax(w_distribution, shift_length, wdy, size, rand_seed=2):
    
    """
    Shifts a cross-section of household forward in time
    *
    * wdy: NamedTuple Model
    * w_distribution: array_like, represents current cross-section
    Takes a current distribution of wealth values as w_distribution and updates each w_t in w_distribution to w_{t+j}, where j = shift_length.
    Returns the new distribution.
    """
    new_dist = wealth_time_series_jax(w_distribution, shift_length, wdy, size, rand_seed)
```

(continues on next page)
new_distribution = new_dist[-1, :]
return new_distribution

# Create the jit function
update_cross_section_jax = jax.jit(update_cross_section_jax, static_argnums=(1,3,))

8.4 Applications

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

8.4.1 Inequality Measures

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

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

```python
def generate_lorenz_and_gini_jax(wdy, num_households=100_000, T=500):
    """
    Generate the Lorenz curve data and gini coefficient corresponding to a
    WealthDynamics mode by simulating num_households forward to time T.
    """
    size = (num_households,)
    ψ_0 = jnp.full(size, wdy.y_mean)
    ψ_star = update_cross_section_jax(ψ_0, T, wdy, size)
    return gini_jax(ψ_star), lorenz_curve_jax(ψ_star)

# Create the jit function
generate_lorenz_and_gini_jax = jax.jit(generate_lorenz_and_gini_jax,
                                       static_argnums=(1,2,))
```

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

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

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

for μ_r in μ_r_vals:
    wdy = create_wealth_model(μ_r=μ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini_jax(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\psi_r = {μ_r:.2f}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, 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.

```python
def fig, ax = plt.subplots()
ax.plot(μ_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel("μ_r")
ax.legend()
plt.show()
```
Once again, we see that inequality increases as returns on financial income rise.

Let’s finish this section by investigating what happens when we change the volatility term $\sigma_r$ in financial returns.

```python
%%time
fig, ax = plt.subplots()
σ_r_vals = (0.35, 0.45, 0.52)
gini_vals = []
for σ_r in σ_r_vals:
    wdy = create_wealth_model(σ_r=σ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini_jax(wdy)
    ax.plot(f_vals, l_vals, label=f'\psi^* at $\sigma_r = {σ_r:.2f}$')
    gini_vals.append(gv)
ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```
We see that greater volatility has the effect of increasing inequality in this model.

## 8.5 Exercises

### Exercise 8.5.1

For a wealth or income distribution with Pareto tail, a higher tail index suggests lower inequality. Indeed, it is possible to prove that the Gini coefficient of the Pareto distribution with tail index $a$ is $1/(2a - 1)$. To the extent that you can, confirm this by simulation. In particular, generate a plot of the Gini coefficient against the tail index using both the theoretical value just given and the value computed from a sample via `gini_jax`. For the values of the tail index, use `a_vals = jnp.linspace(1, 10, 25)`. Use sample of size 1,000 for each $a$ and the sampling method for generating Pareto draws employed in the discussion of Lorenz curves for the Pareto distribution. To the extent that you can, interpret the monotone relationship between the Gini index and $a$.

### Solution to Exercise 8.5.1

Here is one solution, which produces a good match between theory and simulation.
In general, for a Pareto distribution, a higher tail index implies less weight in the right hand tail. This means less extreme values for wealth and hence more equality. More equality translates to a lower Gini index.

**Exercise 8.5.2**

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

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

The theorem does not directly apply here, since savings is not always constant and since the multiplicative and additive terms in (8.1) are not IID.

At the same time, given the similarities, perhaps Pareto tails will arise.
Quantitative Economics with Python using JAX

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

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

For sample size and initial conditions, use

```python
num_households = 250_000
T = 500  # Shift forward T periods
ψ_0 = jnp.full((num_households, ), wdy.y_mean)  # Initial distribution
```

**Solution to Exercise 8.5.2**

First let's generate the distribution:

```python
num_households = 250_000
T = 500  # how far to shift forward in time
size = (num_households, )
wdy = create_wealth_model()
ψ_0 = jnp.full(size, wdy.y_mean)
ψ_star = update_cross_section_jax(ψ_0, T, wdy, size)
```

Let's define a function to get the rank data

```python
def rank_size(data, c=1):
    w = -jnp.sort(-data)  # Reverse sort
    w = w[:int(len(w) * c)]  # extract top (c * 100)%
    rank_data = jnp.arange(len(w)) + 1
    size_data = w
    return rank_data, size_data
```

Now let's see the rank-size plot:

```python
fig, ax = plt.subplots()
rank_data, size_data = rank_size(ψ_star, c=0.001)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")
plt.show()
```
Part III

Data and Empirics
CHAPTER
NINE

MAXIMUM LIKELIHOOD ESTIMATION

Contents

- Maximum Likelihood Estimation
  - Overview
  - MLE with numerical methods (JAX)
  - MLE with statsmodels

GPU

This lecture was built using hardware that has 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.

9.1 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:

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

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

### 9.2 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.

#### 9.2.1 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 $\log L$.

```
@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.

\[
dlogL = jax.vmap(jax.grad(logL))
\]

\[
\beta = jnp.linspace(1, 20)
\]

```
fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(\beta, logL(\beta), lw=2)
ax2.plot(\beta, dlogL(\beta), lw=2)

ax1.set_ylabel(r'$\log \mathcal{L}(\beta)$', rotation=0, labelpad=35, fontsize=15)
ax2.set_ylabel(r'$\frac{d\log \mathcal{L}(\beta)}{d\beta}$', 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, \( \beta_0 \) (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.

9.2.2 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:

\[
 f(y_1, y_2, \ldots, y_n \mid x_1, x_2, \ldots, x_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i},
\]

where \( \mu_i = \exp(x_i' \beta) = \exp(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_k x_{ik}) \)

We create a `namedtuple` to store the observed values

```python
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

\[
 \max_{\beta} \left( \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i! \right)
\]

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

\[
y_i! = e^{\log(\Gamma(n+1))}
\]

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

```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
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`.

```python
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 $\beta_0$.

The algorithm will update the parameter vector according to the updating rule, and recalculate the gradient and Hessian matrices at the new parameter estimates.

```python
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}"\theta":<60}''
        print(header)
        print("-" * len(header))

    # While loop runs while any value in error is greater
    # than the tolerance until max iterations are reached
    while jnp.any(error > tol) and i < max_iter:
        H, G = jnp.squeeze(H_poisson_logL(β, model)), G_poisson_logL(β, model)
        β_new = β - (jnp.dot(jnp.linalg.inv(H), G))
        error = jnp.abs(β_new - β)
        β = β_new

        if display:
            β_list = [f'{t:.3}'' for t in list(β.flatten())]
            update = f'{i:<13}"poisson_logL(β, model):<16.8}''β_list}''
            print(update)

        i += 1

    print(f'Number of iterations: {i}''
    print(f'β_hat = (β.flatten()))''
```

(continues on next page)
Let's try out our algorithm with a small dataset of 5 observations and 3 variables in $X$.

\[
X = \text{jnp.array([[1, 2, 5], [1, 1, 3], [1, 4, 2], [1, 5, 2], [1, 3, 1]])}
\]

\[
y = \text{jnp.array([1, 0, 1, 1, 0])}
\]

# Take a guess at initial $\beta$s
init_\beta = \text{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
\hat{\beta} = \text{newton_raphson(poi, init_\beta, display=True)}

<table>
<thead>
<tr>
<th>Iteration_k</th>
<th>Log-likelihood</th>
<th>θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-4.3447622</td>
<td>['-1.49', '0.265', '0.244']</td>
</tr>
<tr>
<td>1</td>
<td>-3.5742413</td>
<td>['-3.38', '0.528', '0.474']</td>
</tr>
<tr>
<td>2</td>
<td>-3.3999526</td>
<td>['-5.06', '0.782', '0.702']</td>
</tr>
<tr>
<td>3</td>
<td>-3.3788646</td>
<td>['-5.92', '0.909', '0.82']</td>
</tr>
<tr>
<td>4</td>
<td>-3.3783559</td>
<td>['-6.07', '0.933', '0.843']</td>
</tr>
<tr>
<td>5</td>
<td>-3.3783555</td>
<td>['-6.08', '0.933', '0.843']</td>
</tr>
<tr>
<td>6</td>
<td>-3.3783555</td>
<td>['-6.08', '0.933', '0.843']</td>
</tr>
</tbody>
</table>

Number of iterations: 7
\hat{\beta} = [-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 $\hat{\beta}$.

\[
\text{G_poisson_logL(\hat{\beta}, poi)}
\]

\[
\text{Array([[-2.55892529e-13], [-6.50313137e-13], [-5.01695907e-13]], dtype=\text{float64})}
\]
9.3 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.

```python
X_numpy = X.__array__()
y_numpy = y.__array__()
stats_poisson = Poisson(y_numpy, X_numpy).fit()
print(stats_poisson.summary())
```

```markdown
Optimization terminated successfully.
Current function value: 0.675671
Iterations 7

Poisson Regression Results

| 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 9.3.1**

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

$$\lambda_t = \exp(\beta_0 + \beta_1 x_t + \beta_2 x_t^2) \tag{9.1}$$

Simulate the values of $x_t$ by sampling from a normal distribution and $\lambda_t$ by using (9.1) and the following constants:

$$\beta_0 = -2.5, \quad \beta_1 = 0.25, \quad \beta_2 = 0.5$$
Quantitative Economics with Python using JAX

Try to obtain the approximate values of $\beta_0, \beta_1, \beta_2$, by simulating a Poission 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 these parameters.

---

**Solution to Exercise 9.3.1**

Let's start by defining “true” parameter values.

$$\beta_0 = -2.5$$
$$\beta_1 = 0.25$$
$$\beta_2 = 0.5$$

To simulate the model, we sample 500,000 values of $x_t$ from the standard normal distribution.

```python
seed = 32
shape = (500_000, 1)
key = jax.random.PRNGKey(seed)
x = jax.random.normal(key, shape)
```

We compute $\lambda$ using (9.1)

$$\lambda = \text{jnp.exp}(\beta_0 + \beta_1 \cdot x + \beta_2 \cdot x^2)$$

Let's define $y_t$ by sampling from a Poisson distribution with mean as $\lambda_t$.

```python
y = jax.random.poisson(key, lam, shape)
```

Now let's try to recover the true parameter values using the Newton-Raphson method described above.

```python
X = jnp.hstack((jnp.ones(shape), x, x**2))

# Take a guess at initial \beta s
init_\beta = 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
\beta_hat = newton_raphson(poi, init_\beta, tol=1e-5, display=True)
```

```
<table>
<thead>
<tr>
<th>Iteration_k</th>
<th>Log-likelihood θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-4.5444745e+07</td>
</tr>
<tr>
<td>1</td>
<td>-1.6303734e+07</td>
</tr>
<tr>
<td>2</td>
<td>-568932.6</td>
</tr>
<tr>
<td>3</td>
<td>-1869457.7</td>
</tr>
</tbody>
</table>
```

(continues on next page)
The maximum likelihood estimates are similar to the true parameter values.
Part IV

Dynamic Programming
10.1 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.

```python
import numpy as np
import jax.numpy as jnp
import jax
```

Let's check the GPU we are running

```bash
!nvidia-smi
```

```
Mon Aug 14 04:43:14 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03  Driver Version: 470.182.03  CUDA Version: 12.1 |
|-------------------------------+----------------------+----------------------+
| 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 | 0000000:00:1E.0 Off | 0 |
| N/A 35C PO 38W / 300W | 0MiB / 16160MiB | 2% Default |
| | | N/A |
```

(continues on next page)
10.2 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.

10.2.1 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

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

(10.1)

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 $Q$.

```python
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, 4, inf],
                [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:
```python
# 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
```

```python
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.

```python
%%time
jax.lax.while_loop(cond_fun, body_fun, init_val=(0, J, False))
```

CPU times: user 124 ms, sys: 17.2 ms, total: 141 ms
Wall time: 164 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.

```python
max_iter = 500
num_nodes = Q.shape[0]
J = jnp.zeros(num_nodes)
```
Now, this runs faster once we have the JIT compiled JAX version of the functions.

```python
%%time
jax.lax.while_loop(cond_fun, body_fun, init_val=(0, J, False))[1]
```

CPU times: user 1.78 ms, sys: 1.05 ms, total: 2.83 ms
Wall time: 1.49 ms

```
Array([ 8., 10., 3., 5., 4., 1., 0.], dtype=float32)
```

**Note:** Large speed gains while using `jax.lax.while_loop` won’t be realized unless the shortest path problem is relatively large.

### 10.3 Exercises

#### Exercise 10.3.1

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.

```bash
%%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)
| 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, node71 | 1705.38, node42 51.52, node48 129.45 |
| node33, node81 | 1854.73, node41 3.23, node47 111.54 |
| node34, node73 | 1075.38, node41 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 | 701.60, 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 |

(continued on next page)
Solution to Exercise 10.3.1

First let's write a function that reads in the graph data above and builds a distance matrix.

```python
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 = int(elements[0])
            if node != destination_node:
                for element in elements:
                    destination, cost = element.split()
                    Q[node, int(destination)] = cost
```

(continues on next page)
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_condition = 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()
```
Quantitative Economics with Python using JAX

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

```
CPU times: user 1.18 ms, sys: 160 µs, total: 1.34 ms
Wall time: 964 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
```
This lecture was built using hardware that has 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.

We require the following library to be installed.

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

A monopolist 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.

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 \( y_{\text{grid}} \).

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 asqe
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

Let's check the GPU we are running

!nvidia-smi

Mon Aug 14 04:39:10 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03 Driver Version: 470.182.03 CUDA Version: 12.1 |
|-------------------------------+----------------------+----------------------+
| 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 38W / 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 need the following successive approximation function.

```python
def successive_approx(T, x_0, tolerance=1e-6, max_iter=1e+000, print_step=25, 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:
        (continues on next page)```
Let's define a function to create an investment model using the given parameters.

```python
def create_investment_model(
    r=0.01,  # Interest rate
    a_0=10.0, a_1=1.0,  # Demand parameters
    γ=25.0, c=1.0,  # Adjustment and unit cost
    y_min=0.0, y_max=20.0, y_size=100,  # Grid for output
    ρ=0.9, ν=1.0,  # AR(1) parameters
    z_size=150):  # Grid size for shock
    """
    A function that takes in parameters and returns an instance of Model that
    contains data for the investment problem.
    """
    β = 1 / (1 + r)
    y_grid = jnp.linspace(y_min, y_max, y_size)
    mc = qe.tauchen(z_size, ρ, ν)
    z_grid, Q = mc.state_values, mc.P

    # Break up parameters into static and nonstatic components
    constants = β, a_0, a_1, γ, c
    sizes = y_size, z_size
    arrays = y_grid, z_grid, Q

    # Shift arrays to the device (e.g., GPU)
    arrays = tuple(map(jax.device_put, arrays))
    return constants, sizes, arrays
```

Let's re-write the vectorized version of the right-hand side of the Bellman equation (before maximization), which is a 3D array representing:

\[
B(y, z, y') = r(y, z, y') + \beta \sum_{z'} v(y', z') Q(z, z')
\]

for all \((y, z, y')\).

```python
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))  # \(y[i] \rightarrow y[i, j, ip]\)
    z = jnp.reshape(z_grid, (1, z_size, 1))  # \(z[j] \rightarrow z[i, j, ip]\)
    yp = jnp.reshape(y_grid, (1, y_size))  # \(yp[ip] \rightarrow yp[i, j, ip]\)
    r = (a_0 - a_1 * y + z - c) * y - y * (yp - y)**2
```

(continues on next page)
# Calculate continuation rewards at all combinations of (y, z, yp)

\[
v = \text{jnp.} \text{reshape}(v, (1, 1, \text{y_size}, \text{z_size})) \quad \# \ \text{v[ip, jp]} \rightarrow \text{v[i, j, ip, jp]}
\]

\[
Q = \text{jnp.} \text{reshape}(Q, (1, \text{z_size}, 1, \text{z_size})) \quad \# \ \text{Q[j, jp]} \rightarrow \text{Q[i, j, ip, jp]}
\]

\[
EV = \text{jnp.} \text{sum}(v \ast Q, \text{axis=3}) \quad \# \ \text{sum over last index jp}
\]

# Compute the right-hand side of the Bellman equation

\[
\text{return} \ r + \beta \ast EV
\]

# Create a jitted function

\[
B = \text{jax.jit}(B, \text{static_argnums}={2,})
\]

Define a function to compute the current rewards given policy \(\sigma\).

\[
\text{def compute_r_\sigma(\sigma, constants, sizes, arrays):}

\quad \text{"""
Compute the array } r_\sigma[i, j] = r[i, j, \sigma[i, j]], \text{ which gives current}
\quad \text{rewards given policy } \sigma.\quad \text{"""}

\quad \text{# Unpack model}
\quad \beta, a_0, a_1, y, c = \text{constants}
\quad \text{y_size, z_size = sizes}
\quad \text{y_grid, z_grid, Q = arrays}

\quad \text{# Compute } r_\sigma[i, j]
\quad y = \text{jnp.} \text{reshape}(y_grid, (\text{y_size}, 1))
\quad z = \text{jnp.} \text{reshape}(z_grid, (1, \text{z_size}))
\quad yp = y_grid[\sigma]
\quad r_\sigma = (a_0 - a_1 \ast y + z - c) \ast y - y \ast (yp - y)**2

\quad \text{return } r_\sigma
\]

# Create the jitted function

\[
\text{compute_r_\sigma = jax.jit(compute_r_\sigma, static_argnums}={2,})
\]

Define the Bellman operator.

\[
\text{def T(v, constants, sizes, arrays):}

\quad \text{"""
\quad \text{"""The Bellman operator."""}

\quad \text{return} \ \text{jnp.max}(B(v, \text{constants, sizes, arrays}), \text{axis}=2)
\]

\[
T = \text{jax.jit}(T, \text{static_argnums}={2,})
\]

The following function computes a v-greedy policy.

\[
\text{def get_greedy(v, constants, sizes, arrays):}

\quad \text{"""
\quad \text{"""Computes a v-greedy policy, returned as a set of indices."""}

\quad \text{return} \ \text{jnp.argmax}(B(v, \text{constants, sizes, arrays}), \text{axis}=2)
\]

\[
\text{get_greedy = jax.jit(get_greedy, static_argnums}={2,})
\]

Define the \(\sigma\)-policy operator.
def T_σ(v, σ, constants, sizes, arrays):
    # 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]

    # 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 compute the lifetime value of following policy σ.
The basic problem is to solve the linear system
\[ v(y, z) = r(y, z, σ(y, z)) + β \sum_{z'} v(σ(y, z), z')Q(z, z) \]
for v.

It turns out to be helpful to rewrite this as
\[ v(y, z) = r(y, z, σ(y, z)) + β \sum_{y', z'} v(y', z')P_σ(y, z, y', z') \]
where \( P_σ(y, z, y', z') = 1\{y' = σ(y, z)\}Q(z, z') \).

We want to write this as \( v = r_σ + βP_σv \) and then solve for v.

Note, however, that v is a multi-index array, rather than a vector.
The value \( v_σ \) of a policy σ is defined as
\[ v_σ = (I - βP_σ)^{-1}r_σ \]

Here we set up the linear map \( v \mapsto R_σv \),
where \( R_σ := I - βP_σ \)

In the investment problem, this map can be expressed as
\[ (R_σv)(y, z) = v(y, z) - β \sum_{z'} v(σ(y, z), z')Q(z, z') \]

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 \( P_σ \).

Let’s define the function \( R_σ \).
```python
def R_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)

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

Define a function to get the value 𝑣_σ of policy 𝜎 by inverting the linear map 𝑅_σ.

```python
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 R_σ to a function in v
    partial_R_σ = lambda v: R_σ(v, σ, constants, sizes, arrays)

    return jax.scipy.sparse.linalg.bicgstab(partial_R_σ, r_σ)[0]

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

Now we define the solvers, which implement VFI, HPI and OPI.

```python
# Implements VFI-Value Function iteration

def value_iteration(model, tol=1e-5):
    constants, sizes, arrays = model
    _T = lambda v: T(v, constants, sizes, arrays)
    vz = jnp.zeros(sizes)

    v_star = successive_approx(_T, vz, tolerance=tol)
    return get_greedy(v_star, constants, sizes, arrays)
```

```python
# Implements HPI-Howard policy iteration routine

def policy_iteration(model, maxiter=250):
    constants, sizes, arrays = model
    σ = jnp.zeros(sizes, dtype=int)
```

(continues on next page)
i, error = 0, 1.0
while error > 0 and i < maxiter:
    v_σ = get_value(σ, constants, sizes, arrays)
    σ_new = get_greedy(v_σ, constants, 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

# Implements the OPI-Optimal policy Iteration routine

def optimistic_policy_iteration(model, tol=1e-5, m=10):
    constants, sizes, arrays = model
    v = jnp.zeros(sizes)
    error = tol + 1
    while error > tol:
        last_v = v
        σ = get_greedy(v, constants, sizes, arrays)
        for _ in range(m):
            v = T_σ(v, σ, constants, sizes, arrays)
        error = jnp.max(jnp.abs(v - last_v))
    return get_greedy(v, constants, sizes, arrays)

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

print("Starting VFI.")
qe.tic()
out = value_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, y, c = constants
y_size, z_size = sizes
y_grid, z_grid, Q = arrays
Quantitative Economics with Python using JAX

\[ \sigma_{\text{star}} = \text{policy\_iteration}(\text{model}) \]

\[
\text{fig, ax = plt.subplots(figsize=(9, 5))}
\]
\[
\text{ax.plot(y\_grid, y\_grid, "k--", label="45")}
\]
\[
\text{ax.plot(y\_grid, y\_grid[\sigma_{\text{star}}[:, 1]], label="\(\sigma^* (\cdot, z_1)\")\}}
\]
\[
\text{ax.plot(y\_grid, y\_grid[\sigma_{\text{star}}[:, -1]], label="\(\sigma^* (\cdot, z_N)\")\}}
\]
\[
\text{ax.legend(fontsize=12)}
\]
\[
\text{plt.show()}
\]

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.
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.

---

45

\[\sigma^*(\cdot, z_1)\]

\[\sigma^*(\cdot, z_N)\]

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

\[
\text{m\_vals = range(5, 3000, 100)}
\]

\[
\text{model = create\_investment\_model()}
\]
\[
\text{print("Running Howard policy iteration.")}
\]
\[
\text{qe.tic()}
\]
\[
\sigma_{\text{pi}} = \text{policy\_iteration(model)}
\]
\[
\text{pi\_time = qe.toc()}
\]

Running Howard policy iteration.
Concluded loop 1 with error 50.
Concluded loop 2 with error 26.

(continues on next page)
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: 0:00:0.05

print(f"PI completed in {pi_time} seconds.")
print("Running value function iteration.")
qe.tic()
σ_vfi = value_iteration(model, tol=1e-5)
vfi_time = qe.toc()
print(f"VFI completed in {vfi_time} seconds.")

PI completed in 0.05300116539001465 seconds.
Running value function iteration.

TOC: Elapsed: 0:00:1.40
VFI completed in 1.4035112857818604 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()
This lecture was built using hardware that has 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 what’s in Anaconda, this lecture will need the following libraries:

```python
!pip install quantecon
```

We will use the following imports:

```python
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

```bash
!nvidia-smi
```

```
Mon Aug 14 04:42:17 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03 Driver Version: 470.182.03 CUDA Version: 12.1 |
+-----------------------------------------------------------------------------+
| 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 34C P0 38W / 300W | 0MiB / 16160MiB | 2% Default |
+-----------------------------------------------------------------------------+
| Processes: |
```

(continues on next page)
Use 64 bit floats with JAX in order to match NumPy code

- By default, JAX uses 32-bit datatypes.
- By default, NumPy uses 64-bit datatypes.

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

### 12.1 Overview

We consider an optimal savings problem with CRRA utility and budget constraint

\[ W_{t+1} + C_t \leq R W_t + Y_t \]

We assume that labor income \( (Y_t) \) is a discretized AR(1) process.

The right-hand side of the Bellman equation is

\[ B((w, y), w', v) = u(R w + y - w') + \beta \sum y' v(w', y') Q(y, y'). \]

where

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

We use successive approximation for VFI.

```python
def successive_approx(T,
    x_0,  # Initial condition
    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
```
12.2 Model primitives

Here's a namedtuple definition for storing parameters and grids.

```python
Model = namedtuple('Model',
    ('β', 'R', 'γ', 'w_grid', 'y_grid', 'Q'))
```

def create_consumption_model(R=1.01, β=0.98, γ=2.5, w_min=0.01, w_max=5.0, w_size=150, ρ=0.9, ν=0.1, y_size=100):
    # Income parameters
    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
    return Model(β=β, R=R, γ=γ, w_grid=w_grid, y_grid=y_grid, Q=Q)

def create_consumption_model_jax(R=1.01, β=0.98, γ=2.5, w_min=0.01, w_max=5.0, w_size=150, ρ=0.9, ν=0.1, y_size=100):
    # Income parameters
    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
    β, R, γ = jax.device_put([β, R, γ])
    w_grid, y_grid, Q = tuple(map(jax.device_put, [w_grid, y_grid, Q]))
    sizes = w_size, y_size
    return (β, R, γ), sizes, (w_grid, y_grid, Q)
```

Here's the right hand side of the Bellman equation:

def B(v, constants, sizes, arrays):
    # Unpack
Quantitative Economics with Python using JAX

β, R, γ = constants
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)

12.3 Operators

Now we define the policy operator \( T_\sigma \)

```python
def compute_r_σ(σ, constants, sizes, arrays):
    
    # Unpack model
    β, R, γ = constants
    w_size, y_size = sizes
    w_grid, y_grid, Q = arrays

    # Compute r_σ[i, j]
    w = jnp.reshape(w_grid, (w_size, 1))
y = jnp.reshape(y_grid, (1, y_size))
wp = w_grid[σ]
c = R * w + y - wp
r_σ = c***(1-γ)/(1-γ)

    return r_σ

def T_σ(v, σ, constants, sizes, arrays):
    
    # Unpack model
    β, R, γ = constants
    w_size, y_size = sizes
    w_grid, y_grid, Q = arrays

    r_σ = compute_r_σ(σ, constants, sizes, arrays)
```

(continues on next page)
# Compute the array $v[\sigma[i, j], jp$

```
yp_idx = jnp.arange(y_size)
yp_idx = jnp.reshape(yp_idx, (1, 1, y_size))
\sigma = jnp.reshape(\sigma, (w_size, y_size, 1))
V = v[\sigma, yp_idx]
```

# Convert $Q[j, jp]$ to $Q[i, j, jp]$

```
Q = jnp.reshape(Q, (1, y_size, y_size))
```

# Calculate the expected sum $\sum_jp v[\sigma[i, j], jp] * Q[i, j, jp]$

```
Ev = jnp.sum(V * Q, axis=2)
```

```
return r_\sigma + \beta * Ev
```

and the Bellman operator $T$

```
def T(v, constants, sizes, arrays):
    """The Bellman operator."
    return jnp.max(B(v, constants, sizes, arrays), axis=2)
```

The next function computes a $v$-greedy policy given $v$

```
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)
```

The function below computes the value $v_\sigma$ of following policy $\sigma$.

The basic problem is to solve the linear system

$$v(w, y) = u(Rw + y - \sigma(w, y)) + \beta \sum_{y'} v(\sigma(w, y'), y')Q(y, y)$$

for $v$.

It turns out to be helpful to rewrite this as

$$v(w, y) = r(w, y, \sigma(w, y)) + \beta \sum_{w', y'} v(w', y')P_\sigma(w, y, w', y')$$

where $P_\sigma(w, y, w', y') = 1\{w' = \sigma(w, y)\}Q(y, y')$.

We want to write this as $v = r_\sigma + P_\sigma v$ and then solve for $v$

Note, however,

- $v$ is a 2 index array, rather than a single vector.
- $P_\sigma$ has four indices rather than 2

The code below

1. reshapes $v$ and $r_\sigma$ to 1D arrays and $P_\sigma$ to a matrix
2. solves the linear system
3. converts back to multi-index arrays.

12.3. Operators 137
def \( R_\sigma(v, \sigma, \text{constants}, \text{sizes}, \text{arrays}) \):
    
    """
    The value \( v_\sigma \) of a policy \( \sigma \) is defined as
    \[
    v_\sigma = (I - \beta P_\sigma)^{-1} r_\sigma
    \]
    
    Here we set up the linear map \( v \rightarrow R_\sigma v \), where \( R_\sigma := I - \beta P_\sigma \).
    
    In the consumption problem, this map can be expressed as
    \[
    (R_\sigma v)(w, y) = v(w, y) - \beta \sum_{y'} v(\sigma(w, y'), y') Q(y, y')
    \]
    
    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 \( P_\sigma \).
    """
    
    \( \beta, R, \gamma \) = \text{constants}
    \( w_size, y_size \) = \text{sizes}
    \( w_grid, y_grid, Q \) = \text{arrays}
    
    # Set up the array \( v[\sigma[i, j], jp] \)
    zp_idx = jnp.arange(y_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, y_size))
    \( \sigma = jnp.reshape(\sigma, (w_size, y_size, 1)) \)
    \( V = v[\sigma, zp_idx] \)
    
    # Expand \( Q[j, jp] \) to \( Q[i, j, jp] \)
    \( Q = jnp.reshape(Q, (1, 1, y_size, y_size)) \)
    
    # Compute and return \( v[i, j] - \beta \sum_{jp} v[\sigma[i, j], jp] \times Q[j, jp] \)
    return v - \beta \times jnp.sum(V \times Q, axis=2)

def \( \text{get\_value}(\sigma, \text{constants}, \text{sizes}, \text{arrays}) \):
    """Get the value \( v_\sigma \) of policy \( \sigma \) by inverting the linear map \( R_\sigma \)."
    """
    \( \beta, R, \gamma \) = \text{constants}
    \( w_size, y_size \) = \text{sizes}
    \( w_grid, y_grid, Q \) = \text{arrays}
    
    \( r_\sigma = \text{compute\_r\_sigma}(\sigma, \text{constants}, \text{sizes}, \text{arrays}) \)
    
    # Reduce \( R_\sigma \) to a function in \( v \)
    partial_R_\sigma = \text{lambda} v: R_\sigma(v, \sigma, \text{constants}, \text{sizes}, \text{arrays})
    
    return jax.scipy.sparse.linalg.bicgstab(partial_R_\sigma, r_\sigma)[0]
12.4 JIT compiled versions

B = jax.jit(B, static_argnums=(2,))
compute_r_σ = jax.jit(compute_r_σ, static_argnums=(2,))
T = jax.jit(T, static_argnums=(2,))
get_greedy = jax.jit(get_greedy, static_argnums=(2,))
get_value = jax.jit(get_value, static_argnums=(2,))
T_σ = jax.jit(T_σ, static_argnums=(3,))
R_σ = jax.jit(R_σ, static_argnums=(3,))

12.5 Solvers

Now we define the solvers, which implement VFI, HPI and OPI.

# Implements VFI-Value Function iteration

def value_iteration(model, tol=1e-5):
    constants, sizes, arrays = model
    _T = lambda v: T(v, constants, sizes, arrays)
    vz = jnp.zeros(sizes)
    v_star = successive_approx(_T, vz, tolerance=tol)
    return get_greedy(v_star, constants, sizes, arrays)

# Implements HPI-Howard policy iteration routine

def policy_iteration(model, maxiter=250):
    constants, 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, constants, sizes, arrays)
        σ_new = get_greedy(v_σ, constants, 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

# Implements the OPI-Optimal policy Iteration routine

def optimistic_policy_iteration(model, tol=1e-5, m=10):
    constants, sizes, arrays = model
    v = jnp.zeros(sizes)
    error = tol + 1
    while error > tol:
        last_v = v
        σ = get_greedy(v, constants, sizes, arrays)
        for _ in range(m):
            v = T_σ(v, σ, constants, sizes, arrays)
            error = jnp.max(jnp.abs(v - last_v))
        return get_greedy(v, constants, sizes, arrays)
12.6 Plots

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

```python
fontsize = 12
model = create_consumption_model_jax()
# Unpack
constants, sizes, arrays = model
β, R, γ = constants
w_size, y_size = sizes
w_grid, y_grid, Q = arrays

σ_star = policy_iteration(model)

fig, ax = plt.subplots(figsize=(9, 5.2))
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(fontsize=fontsize)
plt.show()
```

Concluded loop 1 with error 77.
Concluded loop 2 with error 55.
Concluded loop 3 with error 28.
Concluded loop 4 with error 17.
Concluded loop 5 with error 7.
Concluded loop 6 with error 3.
Concluded loop 7 with error 1.
Concluded loop 8 with error 1.
Concluded loop 9 with error 0.
12.7 Tests

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

```python
model = create_consumption_model_jax()

print("Starting HPI.")
start_time = time.time()
out = 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 55.
Concluded loop 3 with error 28.
Concluded loop 4 with error 17.
Concluded loop 5 with error 7.
Concluded loop 6 with error 3.
Concluded loop 7 with error 1.
Concluded loop 8 with error 1.
Concluded loop 9 with error 0.
HPI completed in 0.03301715850830078 seconds.

print("Starting VFI.")
start_time = time.time()
out = value_iteration(model)
elapsed = time.time() - start_time
print(f"VFI(jax not in succ) completed in {elapsed} seconds."

Starting VFI.
VFI(jax not in succ) completed in 1.047234296798706 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 0.291840076465332 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
    print(f"{algorithm.__name__} completed in {elapsed_time:.2f} seconds."
    return result, elapsed_time
```

12.7. Tests
model = create_consumption_model_jax()
pi, pi_time = run_algorithm(policy_iteration, model)
\(\sigma_{\text{pi}}, \pi_{\text{time}} = \text{run\_algorithm}(\text{policy\_iteration, model})\)
\(\sigma_{\text{vfi}}, \text{vfi\_time} = \text{run\_algorithm}(\text{value\_iteration, model, tol=1e-5})\)

m_vals = range(5, 3000, 100)
opi_times = []
for m in m_vals:
  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 55.
Concluded loop 3 with error 28.
Concluded loop 4 with error 17.
Concluded loop 5 with error 7.
Concluded loop 6 with error 3.
Concluded loop 7 with error 1.
Concluded loop 8 with error 1.
Concluded loop 9 with error 0.
policy_iteration completed in 0.03 seconds.

value_iteration completed in 0.34 seconds.
opiiteration completed in 0.17 seconds.

optimistic_policy_iteration completed in 0.10 seconds.
opiiteration completed in 0.18 seconds.

optimistic_policy_iteration completed in 0.33 seconds.
opiiteration completed in 0.52 seconds.

optimistic_policy_iteration completed in 0.60 seconds.
opiiteration completed in 0.53 seconds.

optimistic_policy_iteration completed in 0.71 seconds.
opiiteration completed in 0.80 seconds.

optimistic_policy_iteration completed in 0.90 seconds.
opiiteration completed in 0.99 seconds.

optimistic_policy_iteration completed in 1.16 seconds.
opiiteration completed in 1.10 seconds.
optimistic_policy_iteration completed in 1.06 seconds.

optimistic_policy_iteration completed in 1.25 seconds.

optimistic_policy_iteration completed in 1.36 seconds.

optimistic_policy_iteration completed in 1.17 seconds.

optimistic_policy_iteration completed in 1.44 seconds.

optimistic_policy_iteration completed in 1.87 seconds.

optimistic_policy_iteration completed in 1.95 seconds.

optimistic_policy_iteration completed in 2.03 seconds.

optimistic_policy_iteration completed in 1.40 seconds.

optimistic_policy_iteration completed in 2.06 seconds.

optimistic_policy_iteration completed in 2.23 seconds.

optimistic_policy_iteration completed in 2.47 seconds.

optimistic_policy_iteration completed in 2.50 seconds.

optimistic_policy_iteration completed in 2.57 seconds.

optimistic_policy_iteration completed in 2.71 seconds.

optimistic_policy_iteration completed in 2.72 seconds.

optimistic_policy_iteration completed in 2.86 seconds.

```python
fig, ax = plt.subplots(figsize=(9, 5.2))
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=fontsize, frameon=False)
ax.set_xlabel("$m$", fontsize=fontsize)
ax.set_ylabel("time", fontsize=fontsize)
plt.show()
```

12.7. Tests
Howard policy iteration
value function iteration
optimistic policy iteration
13.1 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.

We will use the following libraries and imports.

```python
!pip install --upgrade quantecon interpolation

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

from interpolation import interp
from numba import njit, float64
from numba.experimental import jitclass
```

Let’s check the GPU we are running

```bash
!nvidia-smi
```

Mon Aug 14 03:59:39 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03 Driver Version: 470.182.03 CUDA Version: 12.1          |
|-------------------------------+----------------------+----------------------+
(continues on next page)
We use 64 bit floating point numbers for extra precision.

\[
\text{jax.config.update("jax_enable_x64", True)}
\]

### 13.2 Setup

We consider a household that chooses a state-contingent consumption plan \( \{c_t\}_{t \geq 0} \) to maximize

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

subject to

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

Here \( R = 1 + r \) where \( r \) is the interest rate.

The income process \( \{Y_t\} \) is a Markov chain generated by stochastic matrix \( P \).

The matrix \( P \) and the grid of values taken by \( Y_t \) are obtained by discretizing the AR(1) process

\[
Y_{t+1} = \rho Y_t + \nu \epsilon_{t+1}
\]

where \( \{\epsilon_t\} \) is IID and standard normal.

Utility has the CRRA specification

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

The following function stores default parameter values for the income fluctuation problem and creates suitable arrays.
13.3 Solution method

Let \( S = \mathbb{R}_+ \times Y \) be the set of possible values for the state \((a_t, Y_t)\).

We aim to compute an optimal consumption policy \( \sigma^* : S \to \mathbb{R} \), 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.

13.3.1 Euler equation

The Euler equation for the optimization problem is

\[
    u'(c_t) = \max \{ \beta R \mathbb{E}_t u'(c_{t+1}), u'(a_t) \}
\]

An explanation for this expression can be found here.

We rewrite the Euler equation in functional form

\[
    (u' \circ \sigma)(a, y) = \max \{ \beta R \mathbb{E}_y (u' \circ \sigma)[R(a - \sigma(a, y)) + \tilde{Y}, \tilde{Y}], u'(a) \}
\]

where \((u' \circ \sigma)(a, y) := u'(\sigma(a, y))\) and \(\sigma\) is a consumption policy.

For given consumption policy \(\sigma\), we define \((K\sigma)(a, y)\) as the unique \(c \in [0, a]\) that solves

\[
    u'(c) = \max \{ \beta R \mathbb{E}_y (u' \circ \sigma)[R(a - c) + \tilde{Y}, \tilde{Y}], u'(a) \}
\]

It can be shown that

1. iterating with \(K\) computes an optimal policy and

```python
def ifp(R=1.01,
    β=0.99,   # gross interest rate
    γ=1.5,    # discount factor
    γ=1.5,    # CRRA preference parameter
    s_max=16, # savings grid max
    s_size=200, # savings grid size
    p=0.99,   # income persistence
    ν=0.02,   # income volatility
    y_size=25):
    # Create income Markov chain
    mc = qe.tauuchen(y_size, p, 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)
    sizes = s_size, y_size
    s_grid, y_grid, P = jax.device_put((s_grid, y_grid, P))
    # require R \(\beta < 1\) for convergence
    assert R * β < 1, "Stability condition violated."
    return (β, R, γ), sizes, (s_grid, y_grid, P)
```
2. if $\sigma$ is increasing in its first argument, then so is $K \sigma$

Hence below we always assume that $\sigma$ is increasing in its first argument.

The EGM is a technique for computing the update $K \sigma$ given $\sigma$ along a grid of asset values.

Notice that, since $u'(a) \to \infty$ as $a \downarrow 0$, the second term in the max in (13.3.1) dominates for sufficiently small $a$.

Also, again using (13.3.1), 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 $\hat{a}(y)$ given by

$$\hat{a}(y) = (u')^{-1} \left\{ \beta R \mathbb{E}_y(u' \circ \sigma)[R(s + \hat{Y}, \hat{Y})] \right\}$$

We can now write

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

Equivalently, we can state that the $c$ satisfying $c = (K \sigma)(a, y)$ obeys

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

(13.2)

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

Using the exogenous savings grid, and a fixed value of $y$, we create an endogenous asset grid $a_0, \ldots, a_{N-1}$ and a consumption grid $c_0, \ldots, c_{N-1}$ as follows.

First we set $a_0 = c_0 = 0$, since zero consumption is an optimal (in fact the only) choice when $a = 0$.

Then, for $i > 0$, we compute

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

(13.3)

and we set

$$a_i = s_i + c_i$$

We claim that each pair $a_i, c_i$ obeys (13.3.2).

Indeed, since $s_i > 0$, choosing $c_i$ according to (13.3.3) gives

$$c_i = (u')^{-1} \left\{ \beta R \mathbb{E}_y(u' \circ \sigma)[R(s_i + \hat{Y}, \hat{Y})] \right\} \geq \hat{a}(y)$$

where the inequality uses the fact that $\sigma$ is increasing in its first argument.

If we now take $a_i = s_i + c_i$ we get $a_i > \hat{a}(y)$, so the pair $(a_i, c_i)$ satisfies

$$c_i = (u')^{-1} \left\{ \beta R \mathbb{E}_y(u' \circ \sigma)[R(a_i - c_i) + \hat{Y}, \hat{Y}] \right\} \quad \text{and } a_i > \hat{a}(y)$$

Hence (13.3.2) holds.

We are now ready to iterate with $K$. 

148 Chapter 13. Endogenous Grid Method
13.3.2 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)

```python
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 * c ** (-γ)
    def u_prime_inv(u_prime):
        return u_prime * c ** (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
    c_out = c * e_0
    # Compute a_out by a = s + c
    a_out = np.reshape(s_grid, (s_size, 1)) + c_out
    return a_out, c_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.

```python
K_egm_jax = jax.jit(K_egm, static_argnums=(3,))
```

Next we define a successive approximator that repeatedly applies $K$.

```python
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
    o_init = jnp.repeat(s_grid, y_size)
    o_init = jnp.reshape(o_init, (s_size, y_size))
    a_init = jnp.copy(o_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.\n")
    return a_new, σ_new
```

### 13.3.3 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.

```python
ifp_data = [
    ('R', float64),
    ('β', float64),
    ('γ', float64),
    ('P', float64[:,:]),
```

(continues on next page)
('y_grid', float64[:]),
('s_grid', float64[:])
]

# Use the JAX IFP data as our defaults for the Numba version
model = ifp()
constants, sizes, arrays = model
R, γ = constants
s_size, y_size = sizes
s_grid, y_grid, P = (np.array(a) for a in arrays)

@jitclass(ifp_data)
class IFP:

def __init__(self, 
    R=R, 
    β=β, 
    γ=γ, 
    P=np.array(P),
    y_grid=np.array(y_grid),
    s_grid=s_grid):
    
    self.R, self.β, self.γ = R, β, γ
    self.P, self.y_grid = P, y_grid
    self.s_grid = s_grid

    # Recall that we need R β < 1 for convergence.
    assert self.R * self.β < 1, "Stability condition violated."

def u_prime(self, c):
    return c**(-self.γ)

def u_prime_inv(self, u_prime):
    return u_prime**(-1/self.γ)

@njit
def K_egm_nb(a_in, σ_in, ifp):
    # Simplify names
    s_grid, u_prime = ifp.s_grid, ifp.u_prime
    u_prime_inv = ifp.u_prime_inv
    n = len(y_grid)

    # Linear interpolation of policy using endogenous grid
    def σ(a, z):
        return interp(a_in[:, z], σ_in[:, z], a)

    # Allocate memory for new consumption array
    σ_out = np.zeros_like(σ_in)
    a_out = np.zeros_like(a_in)

(continues on next page)
for i, s in enumerate(s_grid[1:]):
    i += 1
    for z in range(n):
        expect = 0.0
        for z_hat in range(n):
            expect += u_prime(c(R * s + y_grid[z_hat], z_hat)) * P[z, z_hat]
        c = u_prime_inv(b * R * expect)
        o_out[i, z] = c
        a_out[i, z] = s + c

return a_out, o_out

def successive_approx_numba(model,
                        # Class with model information
tol=1e-5,
max_iter=100_000,
verbose=True,
print_skip=25):
    # Unpack
    P, s_grid = model.P, model.s_grid
    n = len(P)

do_init = np.repeat(s_grid, y_size)
    o_init = np.reshape(o_init, (s_size, y_size))
a_init = np.copy(o_init)
a_vec, o_vec = a_init, o_init

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        a_new, o_new = K_egm_nb(a_vec, o_vec, model)
        error = np.max(np.abs(o_vec - o_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        a_vec, o_vec = np.copy(a_new), np.copy(o_new)

    if error > tol:
        print("Failed to converge!")
    else:
        print(f"\nConverged in {i} iterations."

    return a_new, o_new
13.4 Solutions

Here we solve the IFP with JAX and Numba.
We will compare both the outputs and the execution time.

13.4.1 Outputs

```python
ifp_jax = ifp()

ifp_numba = IFP()
```

Here’s a first run of the JAX code.

```python
a_star_egm_jax, c_star_egm_jax = successive_approx_jax(ifp_jax,
                      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 0.00038003859326907197.

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 2.69849752375462e-05.
Error at iteration 1600 is 2.3148392509791784e-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.
Next let's solve the same IFP with Numba.

```python
qe.tic()
a_star_egm_nb, σ_star_egm_nb = successive_approx_numba(ifp_numba,
    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 0.00038003859326907197.

Error at iteration 500 is 0.00024736616926013255.

Error at iteration 600 is 0.00017446354504935258.

Error at iteration 700 is 0.0001299892015863442.

Error at iteration 800 is 0.0001005876947773841.

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 4.42545089341196e-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.
Quantitative Economics with Python using JAX

Error at iteration 1700 is 1.9940474091262317e-05.

Error at iteration 1800 is 1.7238181326817426e-05.

Error at iteration 1900 is 1.494730363494613e-05.

Error at iteration 2000 is 1.2994575430802513e-05.

Error at iteration 2100 is 1.132223596411741e-05.

Converged in 2192 iterations.
TOC: Elapsed: 0:00:24.26

24.267033576965332

Now let's check the outputs in a plot to make sure they are the same.

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

n = len(ifp_numba.P)
for z in (0, y_size-1):
    ax.plot(a_star_egm_nb[:, z],
             c_star_egm_nb[:, z],
             '--', lw=2,
             label=f"Numba EGM: consumption when $z={z}$")
ax.plot(a_star_egm_jax[:, z],
             c_star_egm_jax[:, z],
             label=f"JAX EGM: consumption when $z={z}$")

ax.set_xlabel('asset')
plt.legend()
plt.show()
```
13.4.2 Timing

Now let’s compare execution time of the two methods

```python
qe.tic()
a_star_egm_jax, σ_star_egm_jax = successive_approx_jax(ifp_jax,
            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: 0:00:5.00
```

```python
qe.tic()
a_star_egm_nb, σ_star_egm_nb = successive_approx_numba(ifp_numba,
            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: 0:00:22.48

0.22270170405468626

The JAX code is significantly faster, as expected.
This difference will increase when more features (and state variables) are added to the model.
CHAPTER FOURTEEN

DEFAULT RISK AND INCOME FLUCTUATIONS

GPU

This lecture was built using hardware that has 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 what’s in Anaconda, this lecture will need the following libraries:

```python
!pip install quantecon
```

14.1 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:

```python
import matplotlib.pyplot as plt
import quantecon as qe
import random
import jax
import jax.numpy as jnp
```

Let's check the GPU we are running

```bash
!nvidia-smi
```

```
Mon Aug 14 03:58:40 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03 Driver Version: 470.182.03 CUDA Version: 12.1          |
|-------------------------------+----------------------+----------------------+
<p>| GPU Name Persistence-M| Bus-Id Disp.A | Volatile Uncorr. ECC |</p>
<table>
<thead>
<tr>
<th>Fan Temp Perf Pwr:Usage/Cap</th>
<th>Memory-Usage</th>
<th>GPU-Util Compute M.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------------------------------+------------------------+---------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Tesla V100-SXM2... Off</td>
<td>00000000:00:1E.0 Off</td>
<td>0</td>
</tr>
<tr>
<td>N/A 35C PO 38W / 300W</td>
<td>0MiB / 16160MiB</td>
<td>2% Default</td>
</tr>
<tr>
<td>+-------------------------------+------------------------+---------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processes:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU GI CI PID Type Process name GPU Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID ID Usage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-------------------------------+------------------------+---------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No running processes found</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

### 14.2 Structure

In this section we describe the main features of the model.
14.2.1 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

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]  

(14.1)

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 (14.1).

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.

14.2.2 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

\[
c = y + B - q(B', y)B'
\]  

(14.2)

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.
14.2.3 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 \( \delta \), 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

\[
q = \frac{1 - \delta}{1 + r}
\]  
(14.3)

Next we turn to how the government in effect chooses the default probability \( \delta \).

14.2.4 Government’s Decisions

At each point in time \( t \), the government chooses between

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

Defaulting means declining to repay all of its current obligations.

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

But a sovereign default has two consequences:

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

14.2.5 Reentering International Credit Market

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

14.3 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

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

The value of paying is

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

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

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') d'y' \quad (14.4)$$

Given zero profits for foreign creditors in equilibrium, we can combine (14.3) and (14.4) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \quad (14.5)$$

14.3. Equilibrium
### 14.3.1 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 (14.5)

### 14.4 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 class that will store parameters, grids and transition probabilities.
class Arellano_Economy:
" Stores data and creates primitives for the Arellano economy."

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

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

    # Set up grids
    self.y_grid_size = y_grid_size
    self.B_grid_size = B_grid_size
    B_grid = jnp.linspace(B_grid_min, B_grid_max, B_grid_size)
    mc = qe.markov.tauchen(y_grid_size, ρ, η)
    y_grid, P = jnp.exp(mc.state_values), mc.P

    # Put grids on the device
    self.B_grid = jax.device_put(B_grid)
    self.y_grid = jax.device_put(y_grid)
    self.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(self.y_grid), self.y_grid)

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

    def sizes(self):
        return self.B_grid_size, self.y_grid_size

    def arrays(self):
        return self.P, self.B_grid, self.y_grid, self.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)
\[ \delta(B, y) := \sum_{y'} \I{v_c(B, y') < v_d(y')} P(y, y') \, dy' \]

""
# Unpack
\( \beta, \gamma, r, \rho, \eta, \theta \) = params
\( B, y \) size = sizes
\( P, B_{grid}, y_{grid}, \text{def}_y \) = 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 \( \delta[i_B, i_y] \)
default_states = v_c < v_d
delta = jnp.sum(default_states \times P, axis=2)
q = (1 - delta) / (1 + r)
return q

Next we introduce Bellman operators that updated \( v_d \) and \( v_c \).

```python
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
    \( \beta, \gamma, r, \rho, \eta, \theta \) = params
    \( B, y \) size = sizes
    \( P, B_{grid}, y_{grid}, \text{def}_y \) = arrays

    # Index at which B is near zero
    B0_idx = jnp.searchsorted(B_grid, 1e-10)

    current_utility = u(def_y, gamma)
v = jnp.maximum(v_c[B0_idx, :], v_d)
w = \theta \times v + (1 - \theta) \times v_d
A = jnp.reshape(w, (1, y_size))
cont_value = jnp.sum(A \times P, axis=1)
return current_utility + \beta \times cont_value
```

```python
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,

    \[
    \text{bellman}(B, y) = u(y - q(B', y) B' + B) + \beta \sum_{y'} v(B', y') P(y, y')
    \]

    If consumption is not positive then returns -np.inf
    ""
    # Unpack
    \( \beta, \gamma, r, \rho, \eta, \theta \) = params

    (continues on next page)
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))

# 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, 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 the Arellano_Economy class 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 Arellano_Economy, which is hard for the JIT compiler to handle, and strip it down to more basic objects, which are then passed out to jitted functions.
def solve(model, tol=1e-8, max_iter=10_000):
    
    """Given an instance of Arellano_Economy, this function computes the optimal policy and value functions."
    """
    # Unpack
    params = model.params()
    sizes = model.sizes()
    arrays = model.arrays()
    B_size, y_size = sizes
    
    # 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 = Arellano_Economy()

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

```python
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 `Arellano_Economy` and `T` is the length of
    the simulation. Endogenous objects `v_c`, `v_d`, `q` and `B_star` are
    assumed to come from a solution to `model`.
    '''

    # Unpack elements of the model
    B_size, y_size = model.sizes()
    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])

        # 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 θ
            in_default = False if jnp.random.rand() < model.θ else True
            in_default = False if random.uniform(key) < model.θ else True
```

(continues on next page)
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

14.5 Results

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

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

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

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

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

The first figure shows the bond price schedule and replicates Figure 3 of Arellano, where \( y_L \) and \( Y_H \) are particular below average and above average values of output \( y \).
• $y_L$ is 5% below the mean of the $y$ grid values
• $y_H$ is 5% above the mean of the $y$ grid values

The grid used to compute this figure was relatively fine ($y_{\text{grid\_size}}, B_{\text{grid\_size}} = 51, 251$), which explains the minor differences between this and Arrelano’s figure.

The figure shows that

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

The next figure plots value functions and replicates the right hand panel of Figure 4 of [Are08].
We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (14.4).

The next plot shows these default probabilities over $(B', y)$ as a heat map.
As anticipated, the probability that the government chooses to default in the following period increases with indebtedness and falls with income.

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

The grey vertical bars correspond to periods when the economy is excluded from financial markets because of a past default.
One notable feature of the simulated data is the nonlinear response of interest rates. Periods of relative stability are followed by sharp spikes in the discount rate on government debt.
14.6 Exercises

Exercise 14.6.1

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

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

Solution to Exercise 14.6.1

Solution to this exercise.

Compute the value function, policy and equilibrium prices

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

Entering iteration 0 with error 1.000000001.
Entering iteration 100 with error 0.017499341639204857.

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

Terminating at iteration 399.

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

```python
# Unpack some useful names
B_grid, y_grid, P = ae.B_grid, ae.y_grid, ae.P
B_size, y_size = ae.sizes()
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
        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("$B'$")
```

(continues on next page)
Draw a plot of the value functions

```python
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

```python
# 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()
```

14.6. Exercises
Plot a time series of major variables simulated from the model

```python
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)
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()
This lecture was built using hardware that has 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.

15.1 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]

15.1.1 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.
15.1.2 Preliminaries

We use the following imports:

```python
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
```

Let's check the GPU we are running:

```
!nvidia-smi
```

Mon Aug 14 03:57:34 2023
```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03 Driver Version: 470.182.03 CUDA Version: 12.1 |
|-------------------------------+----------------------+----------------------+
| 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 35C P0 38W / 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.)

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

@jax.jit
def compute_stationary(P):
    n = P.shape[0]
    I = jnp.identity(n)
    O = jnp.ones((n, n))
    A = I - jnp.transpose(P) + O
    return jnp.linalg.solve(A, jnp.ones(n))
```
15.2 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

\[ Y_t = A K_t^\alpha N^{1-\alpha} \]

where

- \( A \) and \( \alpha \) 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

\[
\max_{K, N} \{ AK_t^\alpha N^{1-\alpha} - (r + \delta)K - wN \}
\]

The parameter \( \delta \) is the depreciation rate.

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

\[
r = A \alpha \left( \frac{N}{K} \right)^{1-\alpha} - \delta \tag{15.1}
\]

Using this expression and the firm’s first-order condition for labor, we can pin down the equilibrium wage rate as a function of \( r \) as

\[
w(r) = A (1 - \alpha) \left( A \alpha / (r + \delta) \right)^{\alpha/(1-\alpha)} \tag{15.2}
\]

These parameters and equations are stored in the following class.

```python
class Firm:
    def __init__(self,
                 A=1.0,
                 N=1.0,
                 alpha=0.33,
                 beta=0.96,
                 delta=0.05):

    def rd(self, K):
        """
        Inverse demand curve for capital. The interest rate associated with
        a given demand for capital K.
        """
        return A * alpha * (N / K)**(1 - alpha) - delta
```

(continues on next page)
15.3 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of ex-ante identical households face a common borrowing constraint.

The savings problem faced by a typical household is

\[
\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

subject to

\[
a_{t+1} + c_t \leq wz_t + (1 + r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B
\]

where

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

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

In this simple version of the model, households supply labor inelastically because they do not value leisure.

Below we provide code to solve the household problem, taking \(r\) and \(w\) as fixed.

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

(CRRA utility is treated in the exercises.)

15.3.1 Primitives and Operators

This class stores the parameters that define a household asset accumulation problem and the grids used to solve it.
\[ \beta = 0.96, \quad \text{# Discount factor} \]
\[ \Pi = \begin{bmatrix} 0.9, & 0.1 \\ 0.1, & 0.9 \end{bmatrix}, \quad \text{# Markov chain} \]
\[ z_{\text{grid}} = [0.1, 1.0], \quad \text{# Exogenous states} \]
\[ a_{\text{min}} = 1e-10, \quad a_{\text{max}} = 20, \quad \text{# Asset grid} \]
\[ a_{\text{size}} = 200 \):

# Store values, set up grids over \( a \) and \( z \)
self.r, self.w, self.\beta = r, w, \beta
self.a_size = a_size
self.a_grid = jnp.linspace(a_min, a_max, a_size)
z_grid, \Pi = map(jnp.array, (z_grid, \Pi))
self.\Pi = jax.device_put(\Pi)
self.z_grid = jax.device_put(z_grid)
self.z_size = len(z_grid)

def constants(self):
    return self.r, self.w, self.\beta

def sizes(self):
    return self.a_size, self.z_size

def arrays(self):
    return self.a_grid, self.z_grid, self.\Pi

@jax.jit
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, \beta = constants
    a_size, z_size = sizes
    a_grid, z_grid, \Pi = 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 = wz + (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]
\Pi = jnp.reshape(\Pi, (1, z_size, 1, z_size))   # \Pi[jp] -> \Pi[i, j, ip, jp]
EV = jnp.sum(v * \Pi, axis=3)   # sum over last index \( jp \)

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, u(c) + \beta * EV, -jnp.inf)
The next function computes greedy policies.

```python
# 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 $r_{\sigma}$ which gives current rewards given policy $\sigma$.

That is,

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

```python
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 $r_\sigma[i, j]$
    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 $v_\sigma$ of a policy $\sigma$ is defined as

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

Here we set up the linear map $v \rightarrow R_\sigma v$, 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 $P_\sigma$.

The following linear operator is also needed for policy iteration.

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

(continues on next page)
Quantitative Economics with Python using JAX

a_grid, z_grid, Π = 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, (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,))

The following function is used for optimistic policy iteration.

def T_σ(v, σ, constants, sizes, arrays):
    "The σ-policy operator."

    # Unpack model
    γ, w, β = constants
    a_size, z_size = sizes
    a_grid, z_grid, Π = 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, (a_size, z_size, 1))
V = v[σ, zp_idx]

    # Convert Q[j, jp] to Q[i, j, jp]
Π = jnp.reshape(Π, (1, z_size, z_size))

    # Calculate the expected sum Σ_jp v[σ[i, j], jp] * Q[i, j, jp]
Ev = jnp.sum(V * Π, axis=2)

    return r_σ + β * jnp.sum(V * Π, axis=2)

(continues on next page)
Quantitative Economics with Python using JAX

T_σ = jax.jit(T_σ, static_argnums=(3,))

15.4 Solvers

We will solve the household problem using Howard policy iteration.

```python
def policy_iteration(household, verbose=True):
    """Howard policy iteration routine.""
    constants = household.constants()
    sizes = household.sizes()
    arrays = household.arrays()
    
vz = jnp.zeros(sizes)
σ = jnp.zeros(sizes, dtype=int)
i, error = 0, 1.0
while error > 0:
    v_σ = get_value(σ, constants, sizes, arrays)
    σ_new = get_greedy(v_σ, constants, sizes, arrays)
    error = jnp.max(jnp.abs(σ_new - σ))
    σ = σ_new
    i = i + 1
    if verbose:
        print(f"Concluded loop \{i\} with error \{error\}.")
return σ
```

We can also solve the problem using optimistic policy iteration.

```python
def optimistic_policy_iteration(household, tol=1e-5, m=10):
    constants = household.constants()
    sizes = household.sizes()
    arrays = household.arrays()
    
v = jnp.zeros(sizes)
error = tol + 1
while error > tol:
    last_v = v
    σ = get_greedy(v, constants, sizes, arrays)
    for _ in range(m):
        v = T_σ(v, σ, constants, sizes, arrays)
    error = jnp.max(jnp.abs(v - last_v))
return get_greedy(v, constants, sizes, arrays)
```

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

```python
# Example prices
r = 0.03
w = 0.956

# Create an instance of Household
household = Household(r=r, w=w)
```
%%%time
σ_star_hpi = policy_iteration(household)

Concluded loop 1 with error 101.
Concluded loop 2 with error 76.
Concluded loop 3 with error 36.
Concluded loop 4 with error 17.
Concluded loop 5 with error 12.
Concluded loop 6 with error 6.
Concluded loop 7 with error 3.
Concluded loop 8 with error 2.
Concluded loop 9 with error 1.
Concluded loop 10 with error 1.
Concluded loop 11 with error 1.
Concluded loop 12 with error 1.
Concluded loop 13 with error 1.
Concluded loop 14 with error 1.
Concluded loop 15 with error 0.
CPU times: user 1.89 s, sys: 132 ms, total: 2.02 s
Wall time: 1.61 s

%%%time
σ_star = optimistic_policy_iteration(household)

CPU times: user 572 ms, sys: 108 ms, total: 680 ms
Wall time: 635 ms

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

a_size, z_size = household.sizes()
a_grid, z_grid, P = household.arrays()

fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(a_grid, a_grid, 'k--') # 45 degrees
for j in range(z_size):
    lb = f'$z = \{z_grid[j]:.2}\$'
    ax.plot(a_grid, a_grid[σ_star[:, j]], 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()
15.4.1 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 $\sigma$.

First we compute the stationary distribution of $P_\sigma$, 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 $a_t$.

```python
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 $P_\sigma$ as an array of the form $P_\sigma[i, j, ip, jp]$
    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)
```

\[ \Pi = jnp.\text{reshape}(\Pi, (1, z_{\text{size}}, 1, z_{\text{size}})) \]
\[ P_\sigma = A \ast \Pi \]

# Reshape P_\sigma into a matrix
\[ n = a_{\text{size}} \ast z_{\text{size}} \]
\[ P_\sigma = jnp.\text{reshape}(P_\sigma, (n, n)) \]

# Get stationary distribution and reshape onto [i, j] grid
\[ \psi = \text{compute_stationary}(P_\sigma) \]
\[ \psi = jnp.\text{reshape}(\psi, (a_{\text{size}}, z_{\text{size}})) \]

# Sum along the rows to get the marginal distribution of assets
\[ \psi_a = jnp.\text{sum}(\psi, \text{axis}=1) \]
\[ \text{return } \psi_a \]

compute_asset_stationary = jax.jit(compute_asset_stationary,

Let’s give this a test run.

constants = household.constants()
sizes = household.sizes()
arrays = household.arrays()
\[ \psi = \text{compute_asset_stationary}(\sigma_{\text{star}}, \text{constants}, \text{sizes}, \text{arrays}) \]

The distribution should sum to one:

\[ \psi.\text{sum()} \]

Array(1., dtype=float64)

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

```python
def capital_supply(household):
    """
    Map household decisions to the induced level of capital stock.
    """
    constants = household.constants()
sizes = household.sizes()
arrays = household.arrays()

    # Compute the optimal policy
\[ \sigma_{\text{star}} = \text{optimistic_policy_iteration}(\text{household}) \]

    # Compute the stationary distribution
\[ \psi_a = \text{compute_asset_stationary}(\sigma_{\text{star}}, \text{constants}, \text{sizes}, \text{arrays}) \]

    # Return K
\[ \text{return float}(\text{jnp.sum}(\psi_a \ast \text{household.a_grid})) \]
```
15.5 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 (15.1) and a wage rate $w(r)$ as given in (15.2).
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.

15.5.1 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.

```python
# Create default instances
household = Household()
firm = 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):
    household.r = r
    household.w = firm.r_to_w(r)
    k_vals[i] = capital_supply(household)
```

Chapter 15. The Aiyagari Model
# Plot against demand for capital by firms

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

Here's a plot of the excess demand function.

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

```python
%%time
num_points = 50
k_vals = np.linspace(4, 12, num_points)
out = [excess_demand(k, firm, household) for k in k_vals]
```
15.5.2 Computing the equilibrium

Now let's compute the equilibrium.

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

```python
def bisect(f, a, b, *args, tol=10e-2):
    
    # Implement 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
            upper, lower = upper, middle
        count += 1
```
Now we call the bisection function on excess demand.

```python
def compute_equilibrium(household, firm):
    solution = bisect(excess_demand, 6.0, 10.0, firm, household)
    return solution
```

```bash
%%time

household = Household()
firm = Firm()
compute_equilibrium(household, firm)
```

CPU times: user 1.51 s, sys: 498 ms, total: 2.01 s
Wall time: 650 ms

8.09375

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

### 15.6 Exercises

**Exercise 15.6.1**

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

**Solution to Exercise 15.6.1**

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

for i, β in enumerate(β_vals):
    household = Household(β=β)
    firm = Firm(β=β)
    eq_vals[i] = compute_equilibrium(household, firm)

fig, ax = plt.subplots()
ax.plot(β_vals, eq_vals, ms=2)
ax.set_xlabel(r'$\beta$')
ax.set_ylabel('equilibrium')
plt.show()
```
Exercise 15.6.2
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 capital.

Also, recompute the equilibrium.

Use the default parameters for households and firms.

Set \( \gamma = 2 \).

Solution to Exercise 15.6.2
Let's define the utility function

```python
@jax.jit
def u(c, \gamma=2):
    return c**(1 - \gamma) / (1 - \gamma)
```

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

```python
B = jax.jit(B, static_argnums=(2,))
get_greedy = jax.jit(get_greedy, static_argnums=(2,))
compute_r_\sigma = jax.jit(compute_r_\sigma, static_argnums=(2,))
R_\sigma = jax.jit(R_\sigma, static_argnums=(3,))
get_value = jax.jit(get_value, static_argnums=(2,))
```

(continues on next page)
Now, let's plot the demand for capital by firms.

```python
# Create default instances
household = Household()
firm = 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.r = r
    household.w = firm.r_to_w(r)
    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, firm.rd(k_vals), lw=2, alpha=0.6, label='demand for capital')
ax.grid()
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend()
plt.show()
```
Compute the equilibrium

```python
%%time
household = Household()
firm = Firm()
compute_equilibrium(household, firm)
```

CPU times: user 2.29 s, sys: 196 ms, total: 2.48 s
Wall time: 1.01 s

8.09375
CHAPTER
SIXTEEN

CAKE EATING: NUMERICAL METHODS

GPU

This lecture was built using hardware that has 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 what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation quantecon
```

We will use the following imports.

```python
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
```

```
Mon Aug 14 03:59:14 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03 Driver Version: 470.182.03 CUDA Version: 12.1 |
|-------------------------------+----------------------+----------------------+
| 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 35C P0 39W / 300W | 0MiB / 16160MiB | 2% Default |
|=========================================================================
```

(continues on next page)
16.1 Reviewing the Model

Recall in particular that the Bellman equation is

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

(16.1)

where \( u \) is the CRRA utility function.

16.2 Implementation using JAX

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

\[
\begin{align*}
@\text{jax.jit} \\
def \text{c\_star}(x, \beta, \gamma): \\
& \quad \text{return } (1 - \beta ** (1/\gamma)) * x \\
@\text{jax.jit} \\
def \text{v\_star}(x, \beta, \gamma): \\
& \quad \text{return } (1 - \beta**((1 / \gamma)**(-\gamma)) * (x**((1-\gamma) / (1-\gamma)))) \\
\end{align*}
\]

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

```python
CEM = namedtuple('CakeEatingModel', ('\beta', '\gamma', 'x\_grid', 'c\_grid'))
```

```python
@\text{jax.jit} 
\text{def create\_cake\_eating\_model}(\beta=0.96, \gamma=1.5, \text{x\_grid\_min}=1e-3, \text{x\_grid\_max}=2.5, \text{x\_grid\_size}=200): 
\text{x\_grid} = \text{jnp.linspace(x\_grid\_min, x\_grid\_max, x\_grid\_size)}
\text{# c\_grid used for finding maximize function values using brute force} 
\text{c\_grid} = \text{jnp.linspace(x\_grid\_min, x\_grid\_max, 100*\text{x\_grid\_size})} 
\text{return} \text{CEM(\beta, \gamma, x\_grid=x\_grid, c\_grid=c\_grid)}
```

Now let’s define the CRRA utility function.
# Utility function

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

## 16.2.1 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 $T^2v := T(Tv)$ and so on.

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

```python
@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 $x$.

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

```python
@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.

```python
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.

```python
def compute_value_function(ce,
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
```
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"Converged 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)
ax.set_xlabel('$x$', fontsize=12)
ax.set_title('Value function')

(continues on next page)
Next let's compare it to the analytical solution.

```python
v_analytical = v_star(ce.x_grid, ce.β, ce.y)

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()
```
16.2.2 Policy Function

Recall that the optimal consumption policy was shown to be

$$\sigma^*(x) = (1 - \beta^{1/\gamma}) x$$

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:

```python
@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:
Quantitative Economics with Python using JAX

\[ c = \sigma(ce, v_{jax}) \]

Let's plot this next to the true analytical solution

\[ c_{\text{analytical}} = c_{\text{star}}(ce.x_{\text{grid}}, ce.\beta, ce.y) \]

```python
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()
```

16.3 Numba implementation

This section of the lecture is directly adapted from this lecture for the purpose of comparing the results of JAX implementation.

```python
import numpy as np
from numba import prange, njit
from interpolation import interp
from quantecon.optimize import brent_max

CEMN = namedtuple('CakeEatingModelNumba', ('\beta', '\gamma', 'x_grid'))
```
def create_cake_eating_model_numba(β=0.96, # discount factor
γ=1.5, # degree of relative risk...
# relative aversion
x_grid_min=1e-3, # exclude zero for numerical...
# relative 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 * interp(cem.x_grid, v_array, x - c)

@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)
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.\n")

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 0.018795314243220673.
Error at iteration 225 is 0.006773769546100559.
Error at iteration 250 is 0.002441244305714463.
Error at iteration 275 is 0.0008798164340646508.
Error at iteration 300 is 0.0003170829550072085.
Error at iteration 325 is 0.00011427565630128811.

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 1.907669255009183 times faster than NumPy.
JAX time: 1.2125990390777588
Numba time: 2.3132379055023193
Part V

Other
This page is for readers experiencing errors when running the code from the lectures.

17.1 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.

17.2 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
REFERENCES
CHAPTER
NINETEEN

EXECUTION STATISTICS

This table contains the latest execution statistics.

<table>
<thead>
<tr>
<th>Document</th>
<th>Modified</th>
<th>Method</th>
<th>Run Time (s)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>aiyagari_jax</td>
<td>2023-08-14 03:58</td>
<td>cache</td>
<td>60.76</td>
<td>✅</td>
</tr>
<tr>
<td>arellano</td>
<td>2023-08-14 03:59</td>
<td>cache</td>
<td>35.94</td>
<td>✅</td>
</tr>
<tr>
<td>cake_eating_numerical</td>
<td>2023-08-14 03:59</td>
<td>cache</td>
<td>24.03</td>
<td>✅</td>
</tr>
<tr>
<td>ifp_egm</td>
<td>2023-08-14 04:00</td>
<td>cache</td>
<td>75.06</td>
<td>✅</td>
</tr>
<tr>
<td>intro</td>
<td>2023-08-14 04:00</td>
<td>cache</td>
<td>4.02</td>
<td>✅</td>
</tr>
<tr>
<td>inventory_dynamics</td>
<td>2023-08-14 04:06</td>
<td>cache</td>
<td>369.16</td>
<td>✅</td>
</tr>
<tr>
<td>inventory_ssd</td>
<td>2023-08-14 04:34</td>
<td>cache</td>
<td>1630.75</td>
<td>✅</td>
</tr>
<tr>
<td>jax_intro</td>
<td>2023-08-14 04:34</td>
<td>cache</td>
<td>50.69</td>
<td>✅</td>
</tr>
<tr>
<td>kesten_processes</td>
<td>2023-08-14 04:35</td>
<td>cache</td>
<td>30.03</td>
<td>✅</td>
</tr>
<tr>
<td>markov_asset</td>
<td>2023-08-14 04:35</td>
<td>cache</td>
<td>25.03</td>
<td>✅</td>
</tr>
<tr>
<td>mle</td>
<td>2023-08-14 04:36</td>
<td>cache</td>
<td>20.03</td>
<td>✅</td>
</tr>
<tr>
<td>newtons_method</td>
<td>2023-08-14 04:39</td>
<td>cache</td>
<td>164.65</td>
<td>✅</td>
</tr>
<tr>
<td>opt_invest</td>
<td>2023-08-14 04:42</td>
<td>cache</td>
<td>187.75</td>
<td>✅</td>
</tr>
<tr>
<td>opt_savings</td>
<td>2023-08-14 04:43</td>
<td>cache</td>
<td>63.04</td>
<td>✅</td>
</tr>
<tr>
<td>short_path</td>
<td>2023-08-14 04:43</td>
<td>cache</td>
<td>10.0</td>
<td>✅</td>
</tr>
<tr>
<td>status</td>
<td>2023-08-14 04:43</td>
<td>cache</td>
<td>4.01</td>
<td>✅</td>
</tr>
<tr>
<td>troubleshooting</td>
<td>2023-08-14 04:00</td>
<td>cache</td>
<td>4.02</td>
<td>✅</td>
</tr>
<tr>
<td>wealth_dynamics</td>
<td>2023-08-14 04:44</td>
<td>cache</td>
<td>34.01</td>
<td>✅</td>
</tr>
<tr>
<td>zreferences</td>
<td>2023-08-14 04:44</td>
<td>cache</td>
<td>4.02</td>
<td>✅</td>
</tr>
</tbody>
</table>

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:

```python
import jax
print(f"JAX backend: {jax.devices()[0].platform}")
```

JAX backend: gpu

and the hardware we are running on:

```bash
!nvidia-smi
```
### Execution Statistics

<table>
<thead>
<tr>
<th>GPU Name</th>
<th>Persistence-M</th>
<th>Bus-Id</th>
<th>Disp.A</th>
<th>Volatile Uncorr. ECC</th>
<th>Fan Temp</th>
<th>Perf</th>
<th>Pwr:Usage/Cap</th>
<th>Memory-Usage</th>
<th>GPU-Util Compute M.</th>
<th>MIG M.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Tesla V100-SXM2...</td>
<td>Off</td>
<td>00000000:00:1E.0</td>
<td>Off</td>
<td>34C</td>
<td>P0</td>
<td>38W / 300W</td>
<td>310MiB / 16160MiB</td>
<td>0%</td>
<td>Default</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Processes:

<table>
<thead>
<tr>
<th>GPU</th>
<th>GI</th>
<th>CI</th>
<th>PID</th>
<th>Type</th>
<th>Process name</th>
<th>GPU Memory</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td>Usage</td>
<td>Usage</td>
</tr>
</tbody>
</table>

INDEX

K
Kesten processes
  heavy tails, 76

L
Linear State Space Models, 75

M
Markov process, inventory, 55