Quantitative Economics with Python using JAX

Thomas J. Sargent & John Stachurski

May 08, 2023

CONTENTS

This website presents a set of lectures on quantitative economic modeling, designed and written by Thomas J. Sargent and John Stachurski specifically using JAX and accelerating using the GPU.

For an overview of the series, see this page

CHAPTER

ONE

ABOUT THESE LECTURES

1.1 About

This lecture series introduces quantitative economics using Google JAX.

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

1.2 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

CHAPTER

TWO

INVENTORY DYNAMICS

Contents

2.1 Overview

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

We will use the following imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import jax
import jax.numpy as jnp
from jax import random, lax
from collections import namedtuple
```

Let's check the backend used by JAX and the devices available

```
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")

# Check the devices available for JAX
print(jax.devices())
```

```
JAX backend: gpu
[StreamExecutorGpuDevice(id=0, process_index=0, slice_index=0)]
```

2.2 Sample paths

Consider a firm with inventory X_t.

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

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

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

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

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

Here's a `namedtuple` that stores parameters.

```
Firm = namedtuple('Firm', ['s', 'S', 'mu', 'sigma'])

firm = Firm(s=10, S=100, mu=1.0, sigma=0.5)
```

2.3 Example 1: marginal distributions

Now let's look at the marginal distribution ψ_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.

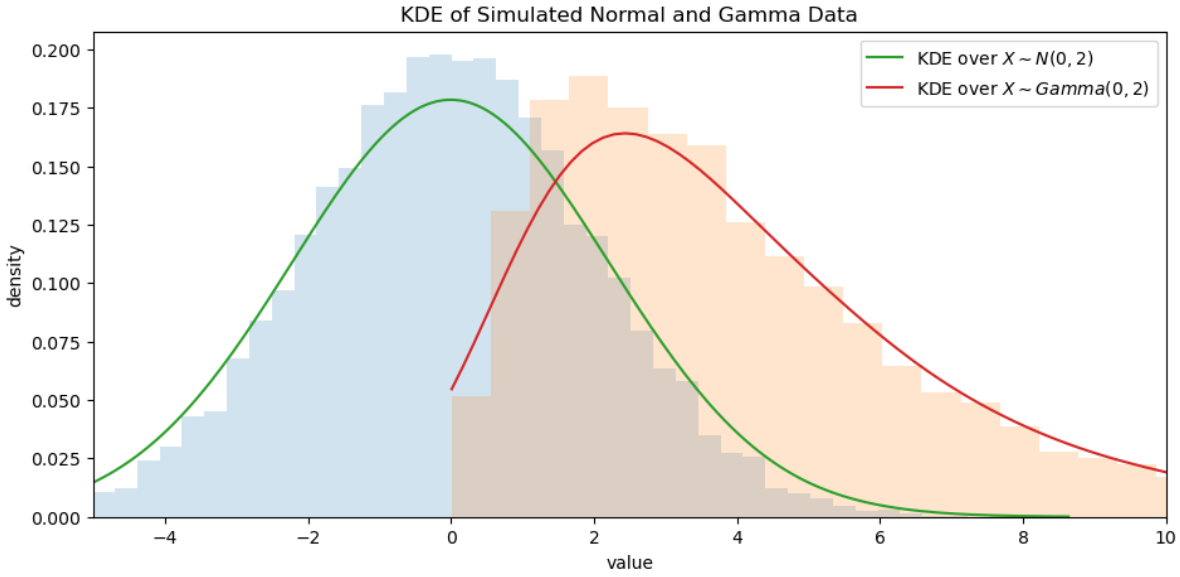
Here is an example of using kernel density estimators and plotting the result

```
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()
```

(continues on next page)

(continued from previous page)

```
# 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

```
# 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
```

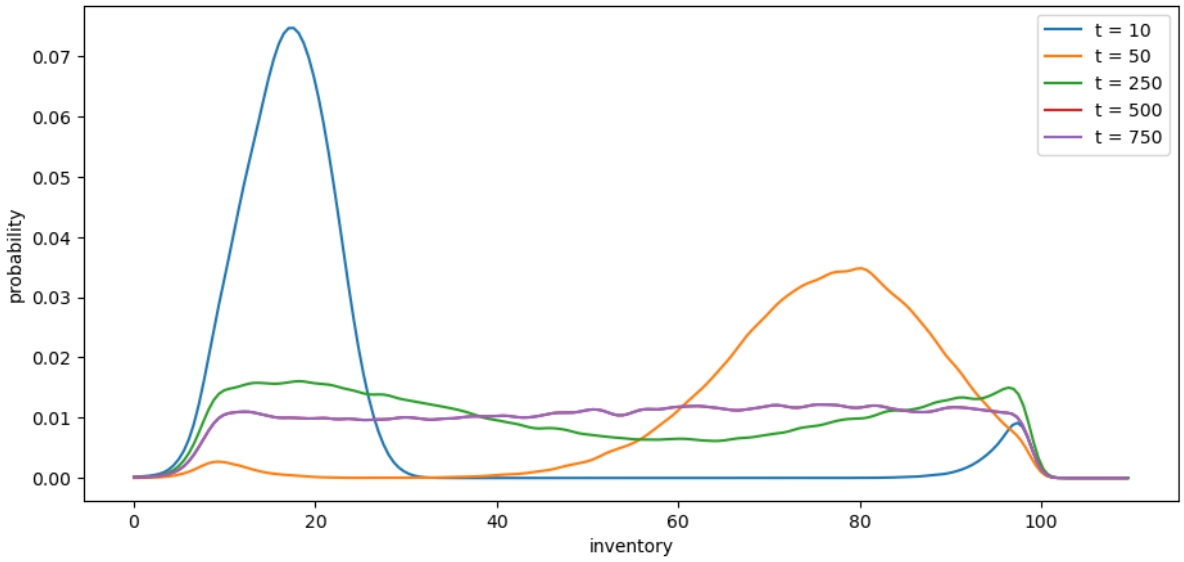
(continues on next page)

(continued from previous page)

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

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

    return res[1:]
```

```
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()
```

```
CPU times: user 2.55 s, sys: 500 ms, total: 3.05 s
Wall time: 2.49 s
```



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

1. `jit` compilation of the `for` loop can be very time consuming and
2. compiling outer loops only leads to minor speed gains.

2.3.1 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`

```
@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

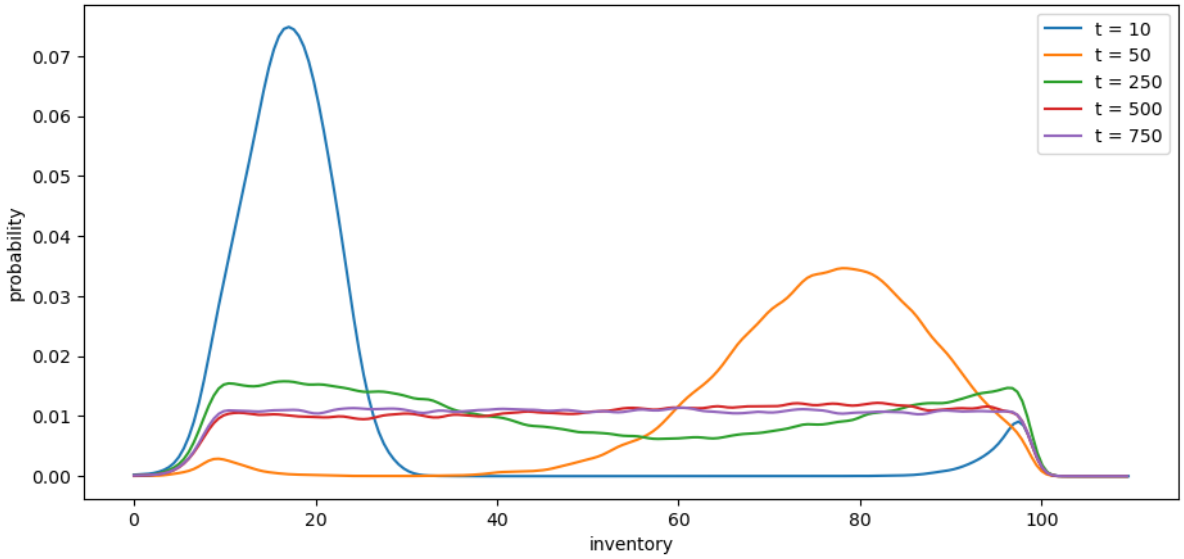
1. as mentioned above, there are only limited speed gains in accelerating outer loops,

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.

```
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 990 ms, sys: 0 ns, total: 990 ms
Wall time: 379 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 $X_0 = 20$

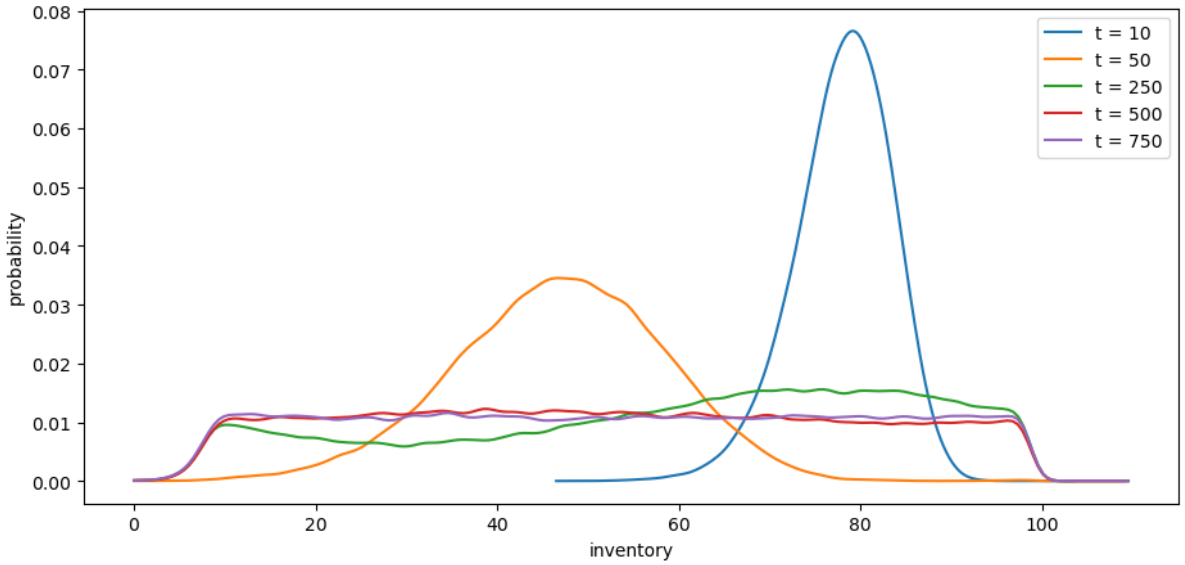
```
x_init = 20.0

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}')
```

(continues on next page)

(continued from previous page)

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

```
CPU times: user 882 ms, sys: 0 ns, total: 882 ms
Wall time: 303 ms
```



2.4 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

```
# 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,
```

(continues on next page)

(continued from previous page)

```
                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)
```

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

```
CPU times: user 962 ms, sys: 0 ns, total: 962 ms
Wall time: 1.22 s
Frequency of at least two stock outs = 0.4472379982471466
```

2.4.1 Alternative implementation with `lax.scan`

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

```
@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]
```

(continues on next page)

(continued from previous page)

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

        Xs = jnp.vstack((X, restock_count))
        return Xs, None

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

    return np.mean(X_final[1] > 1)
```

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

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

```
CPU times: user 1.01 s, sys: 0 ns, total: 1.01 s
Wall time: 391 ms
Frequency of at least two stock outs = 0.44674399495124817
```

CHAPTER

THREE

THE AIYAGARI MODEL

3.1 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]

3.1.1 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.

3.1.2 Preliminaries

We use the following imports

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

Let's check the backend used by JAX and the devices available.

```
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")

# Check the devices available for JAX
print(jax.devices())
```

```
JAX backend: gpu
[StreamExecutorGpuDevice(id=0, process_index=0, slice_index=0)]
```

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

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

We will use the following function to compute stationary distributions of stochastic matrices. (For a reference to the algorithm, see p. 88 of Economic Dynamics.)

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

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

3.2 Firms

Firms produce output by hiring capital and labor.

Firms act competitively and face constant returns to scale.

Since returns to scale are constant the number of firms does not matter.

Hence we can consider a single (but nonetheless competitive) representative firm.

The firm's output is

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

where

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

The firm's problem is

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

The parameter δ is the depreciation rate.

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

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta \tag{3.1}$$

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

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{3.2}$$

These parameters and equations are stored in the following class.

```
class Firm:

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

        self.A, self.N, self.α, self.β, self.δ = A, N, α, β, δ

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


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

3.3 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of *ex-ante* identical households face a common borrowing constraint.

The savings problem faced by a typical household is

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

subject to

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

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate

- r is a net interest rate
- B is the maximum amount that the agent is allowed to borrow

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

In this simple version of the model, households supply labor inelastically because they do not value leisure.

Below we provide code to solve the household problem, taking r and w as fixed.

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

(CRRA utility is treated in the exercises.)

3.3.1 Primitives and Operators

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

```
class Household:

    def __init__(self,
                 r=0.01,                       # Interest rate
                 w=1.0,                        # Wages
                 β=0.96,                       # Discount factor
                 Π=[[0.9, 0.1], [0.1, 0.9]],   # Markov chain
                 z_grid=[0.1, 1.0],            # Exogenous states
                 a_min=1e-10, a_max=20,        # Asset grid
                 a_size=200):

        # Store values, set up grids over a and z
        self.r, self.w, self.β = r, w, β
        self.a_size = a_size
        self.a_grid = jnp.linspace(a_min, a_max, a_size)
        z_grid, Π = map(jnp.array, (z_grid, Π))
        self.Π = jax.device_put(Π)
        self.z_grid = jax.device_put(z_grid)
        self.z_size = len(z_grid)

    def constants(self):
        return self.r, self.w, self.β

    def sizes(self):
        return self.a_size, self.z_size

    def arrays(self):
        return self.a_grid, self.z_grid, self.Π
```

```
@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, β = constants
    a_size, z_size = sizes
    a_grid, z_grid, Π = arrays

    # Compute current consumption as array c[i, j, ip]
    a  = jnp.reshape(a_grid, (a_size, 1, 1))    # a[i]   ->  a[i, j, ip]
    z  = jnp.reshape(z_grid, (1, z_size, 1))    # z[j]   ->  z[i, j, ip]
    ap = jnp.reshape(a_grid, (1, 1, a_size))    # ap[ip] -> ap[i, j, ip]
    c = w*z + (1 + r)*a - ap

    # Calculate continuation rewards at all combinations of (a, z, ap)
    v = jnp.reshape(v, (1, 1, a_size, z_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Π = jnp.reshape(Π, (1, z_size, 1, z_size))  # Π[j, jp]  -> Π[i, j, ip, jp]
    EV = jnp.sum(v * Π, axis=3)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, u(c) + β * EV, -jnp.inf)

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

The next function computes greedy policies.

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

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

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

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

That is,

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

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

    # Compute r_σ[i, j]
    a = jnp.reshape(a_grid, (a_size, 1))
    z = jnp.reshape(z_grid, (1, z_size))
    ap = a_grid[σ]
    c = (1 + r)*a + w*z - ap
    r_σ = u(c)

    return r_σ

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

The value v_σ of a policy σ is defined as

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

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

In the consumption problem, this map can be expressed as

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

Defining the map as above works in a more intuitive multi-index setting (e.g. working with $v[i, j]$ rather than flattening v to a one-dimensional array) and avoids instantiating the large matrix P_σ.

The following linear operator is also needed for policy iteration.

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

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

(continues on next page)

(continued from previous page)

```
    # 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)

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

3.4 Solvers

We will solve the household problem using Howard policy iteration.

```
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.

```
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)
```

(continues on next page)

(continued from previous page)

```
        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.

```
# Example prices
r = 0.03
w = 0.956

# Create an instance of Housbehold
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 1.84 s, sys: 163 ms, total: 2 s
Wall time: 1.53 s
```

```
%%time

σ_star = optimistic_policy_iteration(household)
```

```
CPU times: user 512 ms, sys: 107 ms, total: 619 ms
Wall time: 570 ms
```

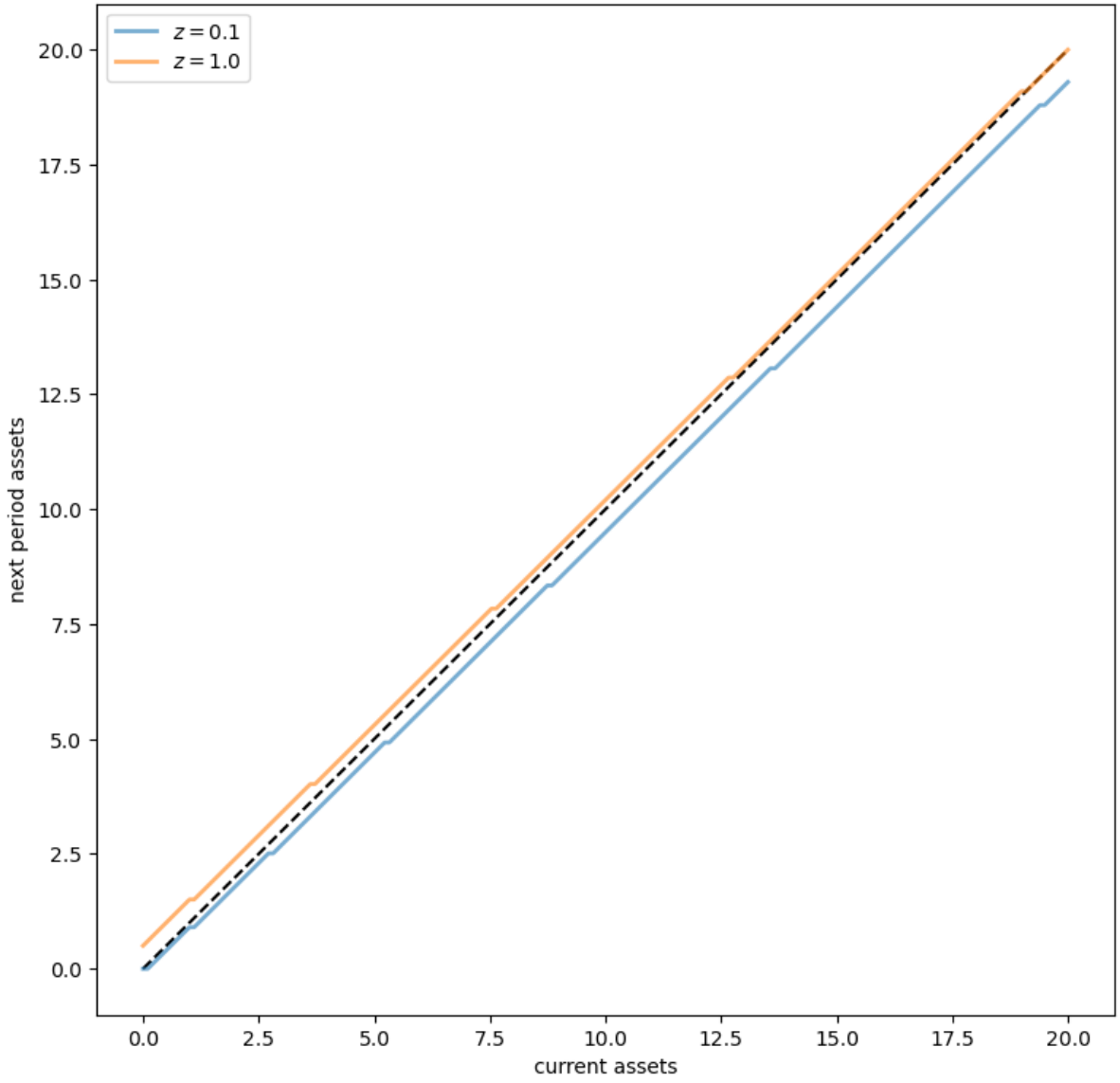
The next plot shows asset accumulation policies at different values of the exogenous state.

```
a_size, z_size = household.sizes()
a_grid, z_grid, Π = 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')
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylabel('next period assets')
ax.legend(loc='upper left')

plt.show()
```



3.4.1 Capital Supply

To start thinking about equilibrium, we need to know how much capital households supply at a given interest rate r.

This quantity can be calculated by taking the stationary distribution of assets under the optimal policy and computing the mean.

The next function implements this calculation for a given policy σ.

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

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

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

    # Construct P_σ as an array of the form P_σ[i, j, ip, jp]
    ap_idx = jnp.arange(a_size)
    ap_idx = jnp.reshape(ap_idx, (1, 1, a_size, 1))
    σ = jnp.reshape(σ, (a_size, z_size, 1, 1))
    A = jnp.where(σ == ap_idx, 1, 0)
    Π = jnp.reshape(Π, (1, z_size, 1, z_size))
    P_σ = A * Π

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

    # Get stationary distribution and reshape onto [i, j] grid
    ψ = compute_stationary(P_σ)
    ψ = jnp.reshape(ψ, (a_size, z_size))

    # Sum along the rows to get the marginal distribution of assets
    ψ_a = jnp.sum(ψ, axis=1)
    return ψ_a

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

Let's give this a test run.

```
constants = household.constants()
sizes = household.sizes()
arrays = household.arrays()
ψ = compute_asset_stationary(σ_star, constants, sizes, arrays)
```

The distribution should sum to one:

```
ψ.sum()
```

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

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

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

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

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

3.5 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 (3.1) and a wage rate $w(r)$ as given in (3.2).
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.

3.5.1 Visual inspection

Let's inspect visually as a first pass.

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

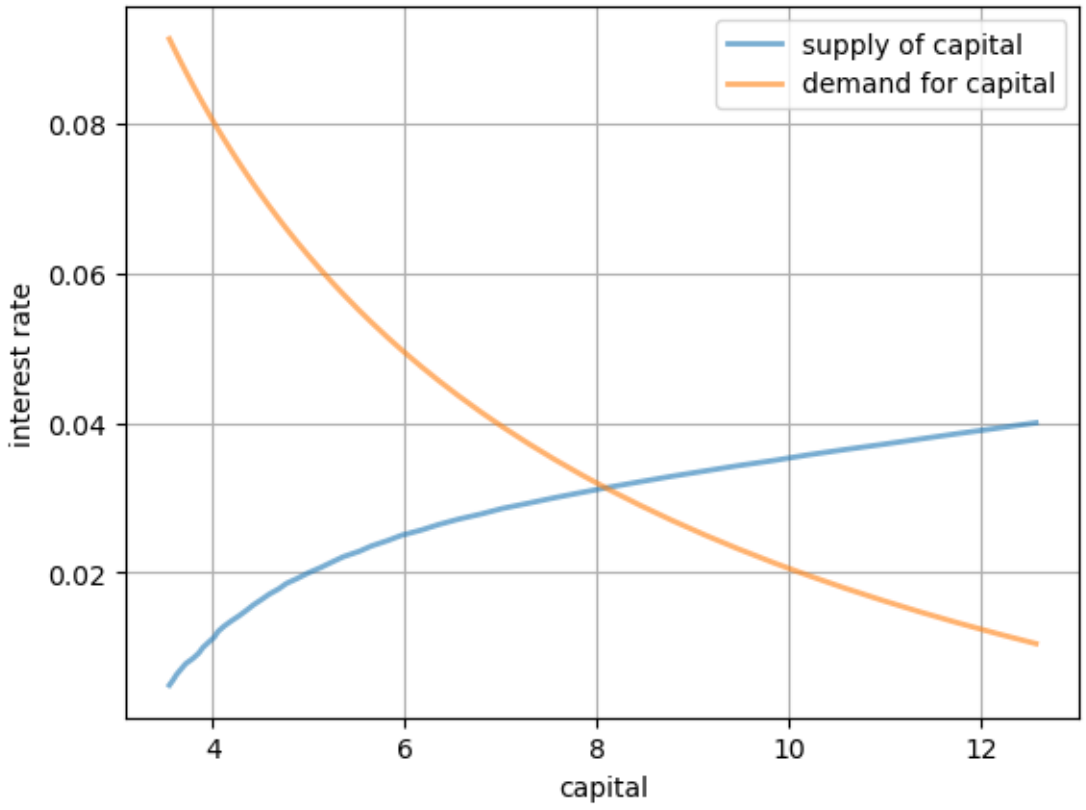
The intersection gives equilibrium interest rates and capital.

```
# Create default instances
household = 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)
```

```
CPU times: user 11.8 s, sys: 3.46 s, total: 15.2 s
Wall time: 5.08 s
```

```
# Plot against demand for capital by firms

fig, ax = plt.subplots()
ax.plot(k_vals, r_vals, lw=2, alpha=0.6, label='supply of capital')
ax.plot(k_vals, 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(loc='upper right')

plt.show()
```



Here's a plot of the excess demand function.

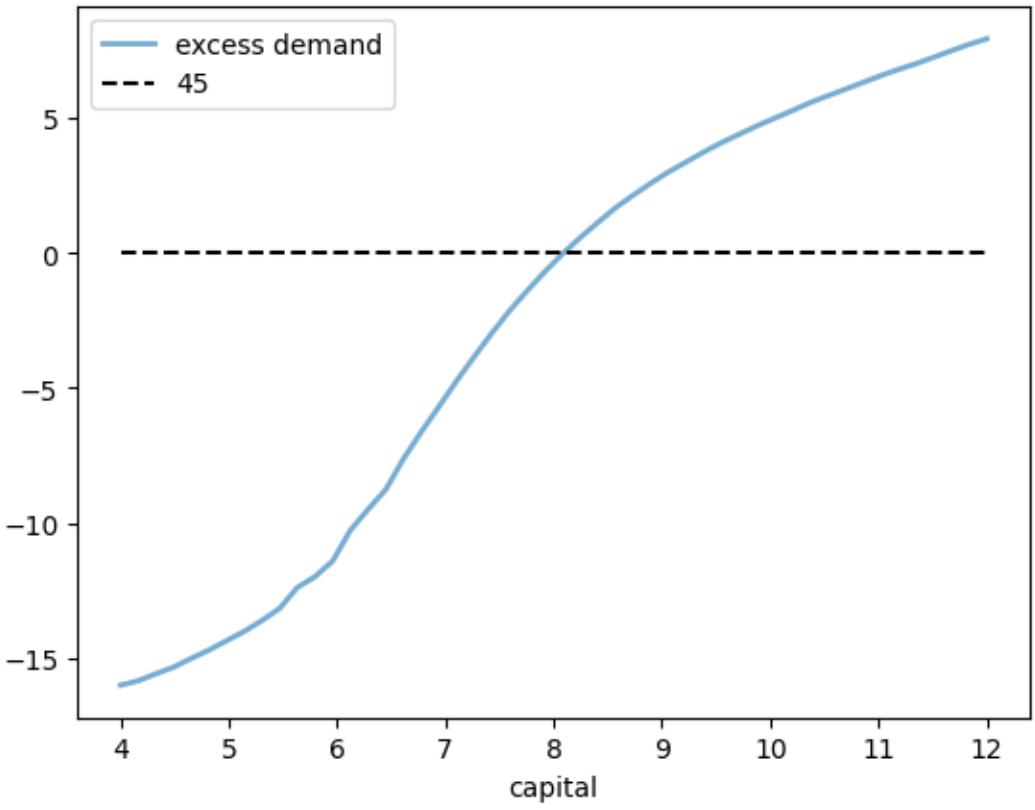
The equilibrium is the zero (root) of this function.

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

```
%%time

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

```
CPU times: user 11.8 s, sys: 4.24 s, total: 16 s
Wall time: 4.89 s
```

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



3.5.2 Computing the equilibrium

Now let's compute the equilibrium

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

```
def bisect(f, a, b, *args, tol=10e-2):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b
    count = 0
    while upper - lower > tol and count < 10000:
        middle = 0.5 * (upper + lower)
        if f(middle, *args) > 0:   # root is between lower and middle
            lower, upper = lower, middle
        else:                      # root is between middle and upper
            lower, upper = middle, upper
        count += 1
    if count == 10000:
        print("Root might not be accurate")
    return 0.5 * (upper + lower)
```

Now we call the bisection function on excess demand.

```
def compute_equilibrium(household, firm):
    solution = bisect(excess_demand, 6.0, 10.0, firm, household)
    return solution
```

```
%%time

household = Household()
firm = Firm()
compute_equilibrium(household, firm)
```

```
CPU times: user 1.45 s, sys: 381 ms, total: 1.83 s
Wall time: 584 ms
```

```
8.09375
```

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

3.6 Exercises

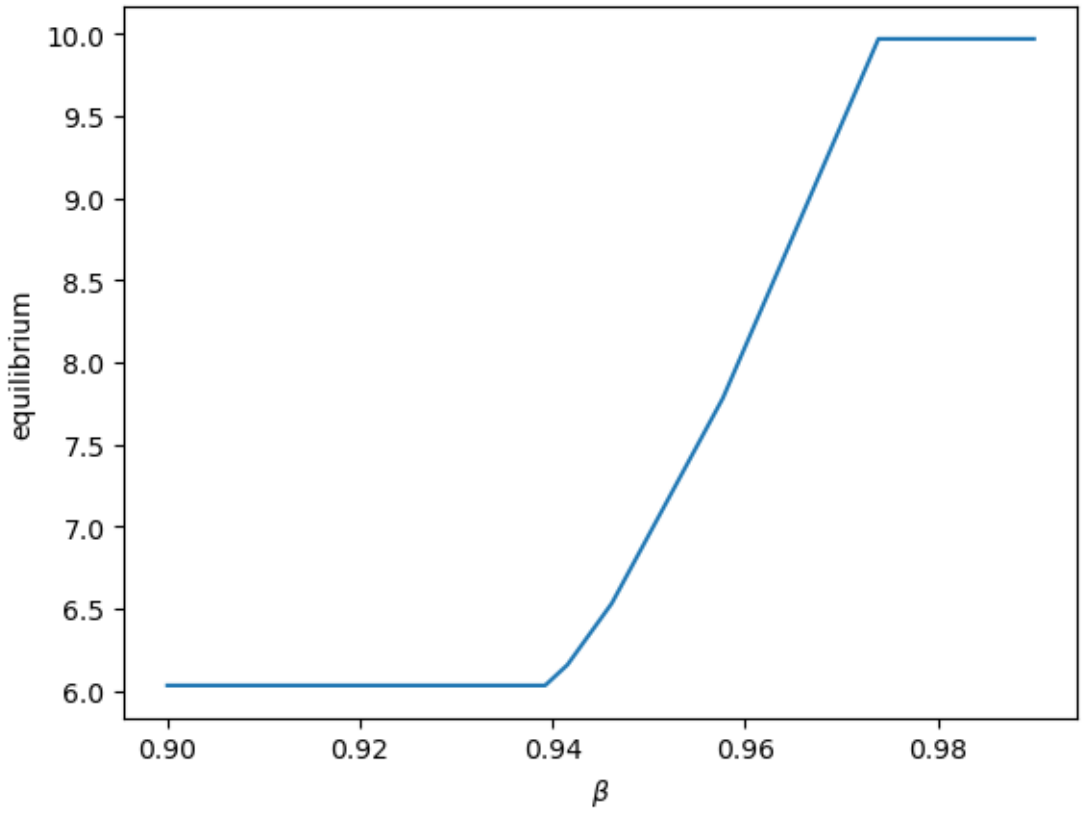
Exercise 3.1

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

Solution to Exercise 3.1

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

for i, β in enumerate(β_vals):
    household = 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 3.2

Switch to the CRRA utility function

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

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

Also, recompute the equilibrium.

Use the default parameters for households and firms.

Set $\gamma = 2$.

Solution to Exercise 3.2

Let's define the utility function

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

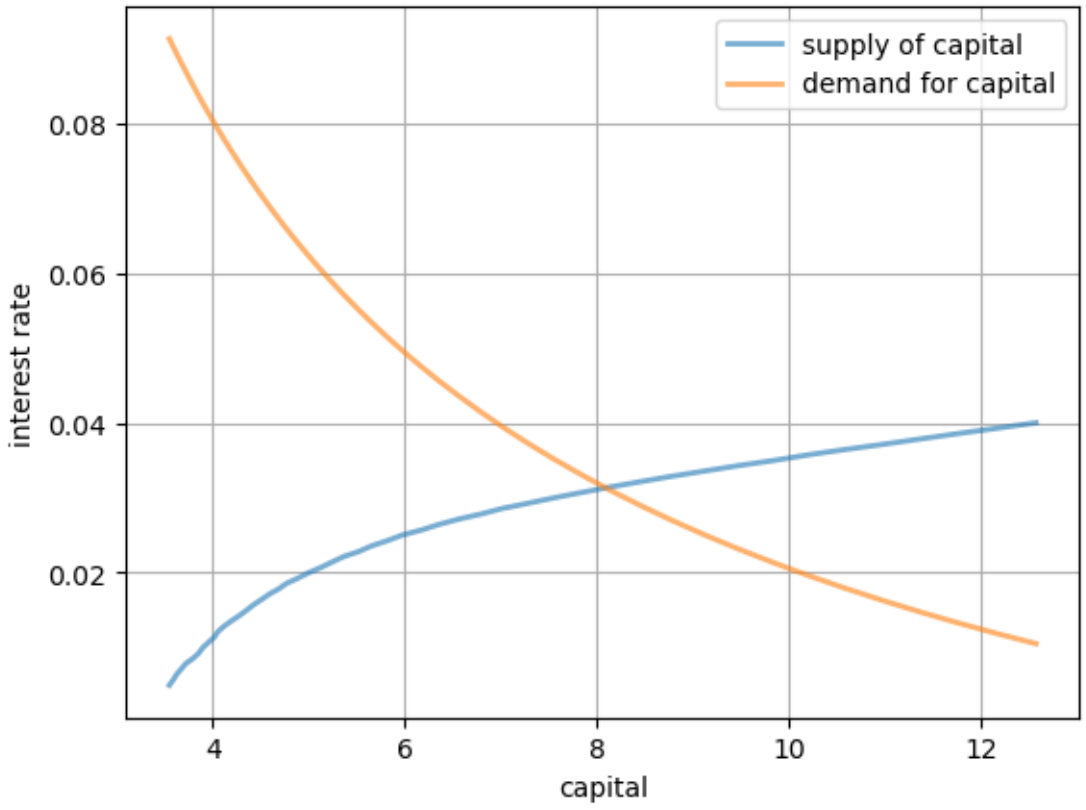
We need to re-compile all the jitted functions in order notice the change in the utility function.

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

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

```
# Create default instances
household = 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

```
%%time

household = Household()
firm = Firm()
compute_equilibrium(household, firm)
```

```
CPU times: user 2.01 s, sys: 362 ms, total: 2.37 s
Wall time: 957 ms
```

```
8.09375
```

CHAPTER

FOUR

USING NEWTON'S METHOD TO SOLVE ECONOMIC MODELS

4.1 Overview

Continuing from the Newton's Method lecture, we are going to solve the multidimensional problem with `JAX`.

More information about JAX can be found here.

We use the following imports in this lecture

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

Let's check the backend used by JAX and the devices available.

```
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")

# Check the devices available for JAX
print(jax.devices())
```

```
JAX backend: gpu
[StreamExecutorGpuDevice(id=0, process_index=0, slice_index=0)]
```

4.2 The Two Goods Market Equilibrium

Let's have a quick recap of this problem – a more detailed explanation and derivation can be found at A 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,

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

and the demand of good i at price p is,

$$q_i^d(p) = \exp(-(a_{i0} p_0 + a_{i1} p_1)) + c_i$$

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,

$$e_i(p) = q_i^d(p) - q_i^s(p), \quad i = 0, 1$$

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

We set

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

for this particular question.

4.2.1 The Multivariable Market Equilibrium

We can now easily get the multivariable version of the problem above.

The supply function remains unchanged,

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

The demand function is,

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

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

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

4.3 Using Newton's Method

Now let's use the multivariate version of Newton's method to compute the equilibrium price

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

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

The iteration starts from some initial guess of the price vector p_0.

Here, instead of coding Jacobian by hand, We use the `jax.jacobian()` function to auto-differentiate and calculate the Jacobian.

With only slight modification, we can generalize our previous attempt to multi-dimensional problems

```
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
```

(continues on next page)

(continued from previous page)

```
    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
```

4.3.1 A High-Dimensional Problem

We now apply the multivariate Newton's Method 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.

```
dim = 5_000
seed = 32

# Create a random matrix A and normalize the rows to sum to one
key = jax.random.PRNGKey(seed)

A = jax.random.uniform(key, [dim, dim])

s = jnp.sum(A, axis=0)
A = A / s

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

Here's our initial condition p_0

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

By 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:

```
%%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 4.79 s, sys: 1.46 s, total: 6.25 s
Wall time: 4.3 s
```

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

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

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

```
%%time

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

```
CPU times: user 2min 24s, sys: 355 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.

4.4 Exercises

Exercise 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

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

However k_t is now a 3×1 vector.

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

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

Hint:

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

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

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

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

The result should converge to the analytical solution.

Solution to Exercise 4.1

Let's first define the parameters for this problem

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

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

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

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

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

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

```
iteration 1, error = 50.496315002441406
iteration 2, error = 41.1093864440918
iteration 3, error = 4.294127464294434
iteration 4, error = 0.3854290544986725
iteration 5, error = 0.0054382034577429295
iteration 6, error = 8.92080606718082e-07
```

(continues on next page)

(continued from previous page)

```
Result = [3.8405814 3.870718  3.4109194]

CPU times: user 784 ms, sys: 48.8 ms, total: 833 ms
Wall time: 662 ms
------------------------------------------------------------
Attempt 2: Starting value is [3. 5. 5.]
```

```
iteration 1, error = 2.0701100826263428
iteration 2, error = 0.12642373144626617
iteration 3, error = 0.0006017307168804109
iteration 4, error = 3.3717478231665154e-07

Result = [3.8405814 3.8707182 3.4109197]

CPU times: user 580 ms, sys: 10.1 ms, total: 590 ms
Wall time: 227 ms
------------------------------------------------------------
Attempt 3: Starting value is [50. 50. 50.]
```

```
iteration 1, error = 73.00942993164062
```

```
iteration 2, error = 6.493789196014404
iteration 3, error = 0.6806989312171936
iteration 4, error = 0.016202213242650032
iteration 5, error = 1.0600916539260652e-05
iteration 6, error = 9.830249609876773e-07

Result = [3.840581  3.8707182 3.41092  ]

CPU times: user 1.21 s, sys: 16.4 ms, total: 1.23 s
Wall time: 505 ms
------------------------------------------------------------
```

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

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

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

Note the error is very small.

We can also test our results on the known solution

```
A = jnp.array([[2.0, 0.0, 0.0],
               [0.0, 2.0, 0.0],
               [0.0, 0.0, 2.0]])
s = 0.3
α = 0.3
```

(continues on next page)

(continued from previous page)

```
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]

CPU times: user 1.17 s, sys: 8.33 ms, total: 1.18 s
Wall time: 481 ms
```

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)

```
# 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]

CPU times: user 854 ms, sys: 4.53 ms, total: 858 ms
Wall time: 681 ms
```

We can see it steps towards a more accurate solution.

Exercise 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:

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

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

$$p1_0 = (5, 5, 5)$$
$$p2_0 = (1, 1, 1)$$
$$p3_0 = (4.5, 0.1, 4)$$

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 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('-'*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 = 0.005209730313222213
iteration 6, error = 4.3632751705775364e-06
iteration 7, error = 3.0460818773540415e-12
iteration 8, error = 0.0

Result = [1.49744442 1.49744442 1.49744442]

CPU times: user 840 ms, sys: 8.13 ms, total: 848 ms
Wall time: 237 ms
------------------------------------------------------------
```

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

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

Substitute the result back to the formula to check our result

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

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

We can see the result is very accurate.

CHAPTER

FIVE

KESTEN PROCESSES AND FIRM DYNAMICS

Contents

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

```
!pip install quantecon
```

5.1 Overview

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

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

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

This lecture focuses on implementing the same computations in JAX.

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import quantecon as qe
import jax
import jax.numpy as jnp
from jax import random
```

Let's check the backend used by JAX and the devices available

```
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")

# Check the devices available for JAX
print(jax.devices())
```

```
JAX backend: gpu
[StreamExecutorGpuDevice(id=0, process_index=0, slice_index=0)]
```

5.2 Kesten processes

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

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

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

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

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

In particular, we will assume that

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

5.2.1 Application: firm dynamics

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

Gibrat's law

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

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

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

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

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

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

However, we can get close to the data by modifying (5.2) to

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

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

We now study the implications of this specification.

Heavy tails

If the conditions of the Kesten–Goldie Theorem are satisfied, then (5.3) implies that the firm size distribution will have Pareto tails.

This matches empirical findings across many data sets.

But there is another unrealistic aspect of the firm dynamics specified in (5.3) that we need to address: it ignores entry and exit.

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

In this setting, firm dynamics can be expressed as

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

The motivation behind and interpretation of *(5.4)* can be found in our earlier Kesten process lecture.

What can we say about dynamics?

Although (5.4) is not a Kesten process, it does update in the same way as a Kesten process when s_t is large.

So perhaps its stationary distribution still has Pareto tails?

We can investigate this question via simulation and rank-size plots.

The approach will be to

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

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

In the simulation, we assume that each of a_t, b_t and e_t is lognormal.

Here's code to update a cross-section of firms according to the dynamics in *(5.4)*.

```
@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,
```

(continues on next page)

(continued from previous page)

```
                    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()
```

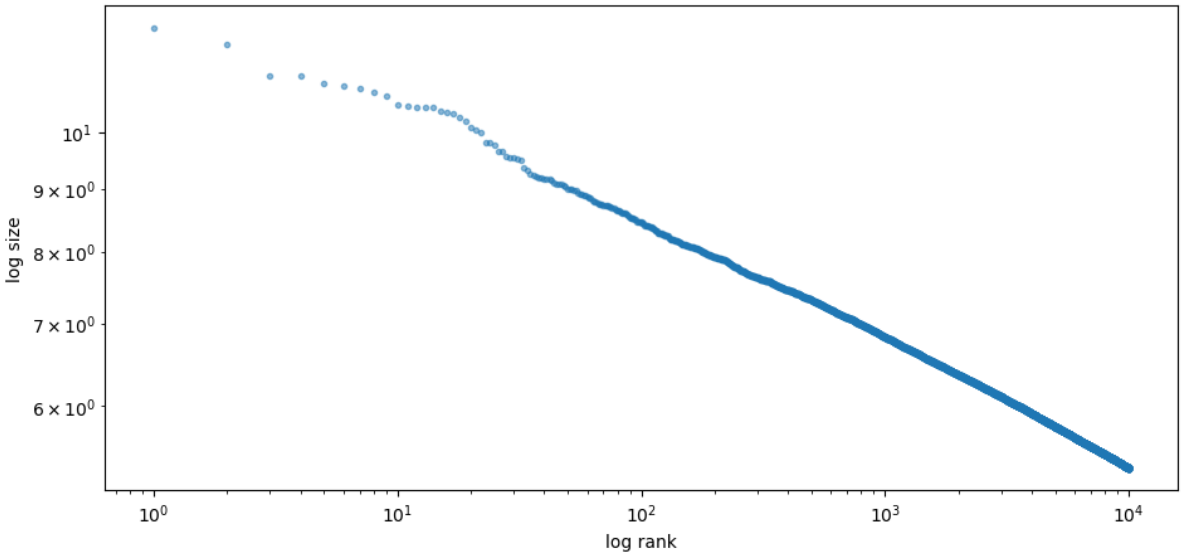
```
CPU times: user 7.56 s, sys: 1.47 s, total: 9.03 s
Wall time: 6.91 s
```

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

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

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

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

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

plt.show()
```



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

Alternative implementation with `lax.scan`

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

Note, however, that

1. as mentioned above, there is not much speed gain 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

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

Here is the `lax.scan` version:

```
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=1_000_000,
                       s_init=1.0,
                       seed=123):

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

    # Generate random draws and initial values
```

(continues on next page)

(continued from previous page)

```
    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()
```

```
CPU times: user 1.09 s, sys: 50.4 ms, total: 1.14 s
Wall time: 546 ms
```

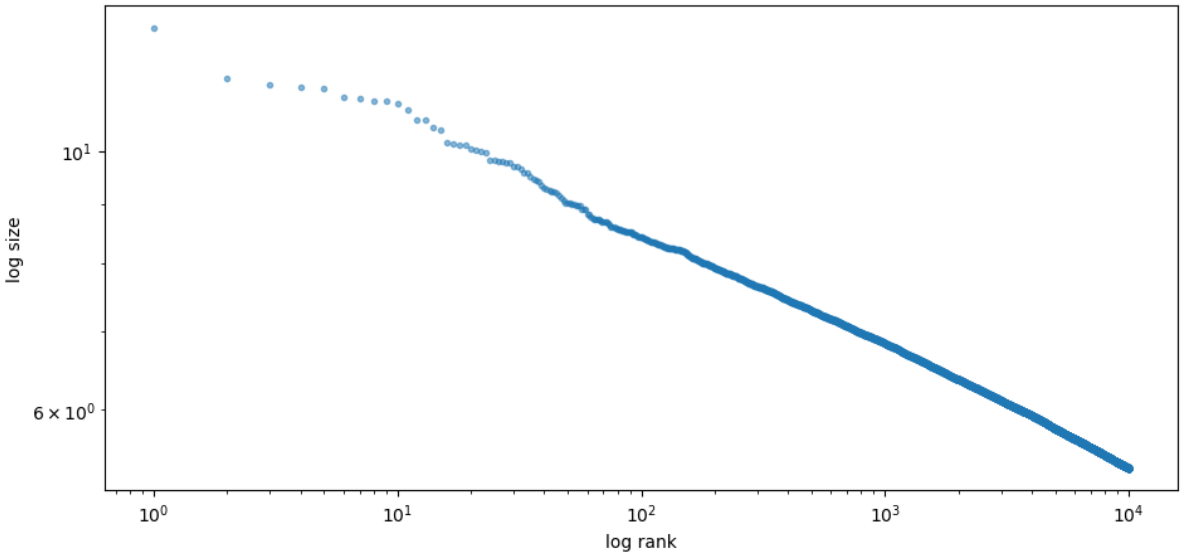
In this case, `T` and `M` are small enough for the code to run and we see some speed gain over the for loop implementation:

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

```
CPU times: user 2.88 ms, sys: 540 µs, total: 3.42 ms
Wall time: 68.1 ms
```

Here we produce the same rank-size plot:

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

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

plt.show()
```


log size
10^1
6×10^0
10^0
10^1
10^2
10^3
10^4
log rank

CHAPTER

SIX

WEALTH DISTRIBUTION DYNAMICS

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.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import jax
import jax.numpy as jnp
from collections import namedtuple
%matplotlib inline
```

Let's check the backend used by JAX and the devices available

```
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")

# Check the devices available for JAX
print(jax.devices())
```

```
JAX backend: gpu
[StreamExecutorGpuDevice(id=0, process_index=0, slice_index=0)]
```

6.1 Lorenz Curves and the Gini Coefficient

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

6.1.1 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.

```
@jax.jit
def lorenz_curve_jax(y):
    n = y.shape[0]
    y = jnp.sort(y)
    s = jnp.concatenate((jnp.zeros(1), jnp.cumsum(y)))
```

(continues on next page)

(continued from previous page)

```
    _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

```
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:

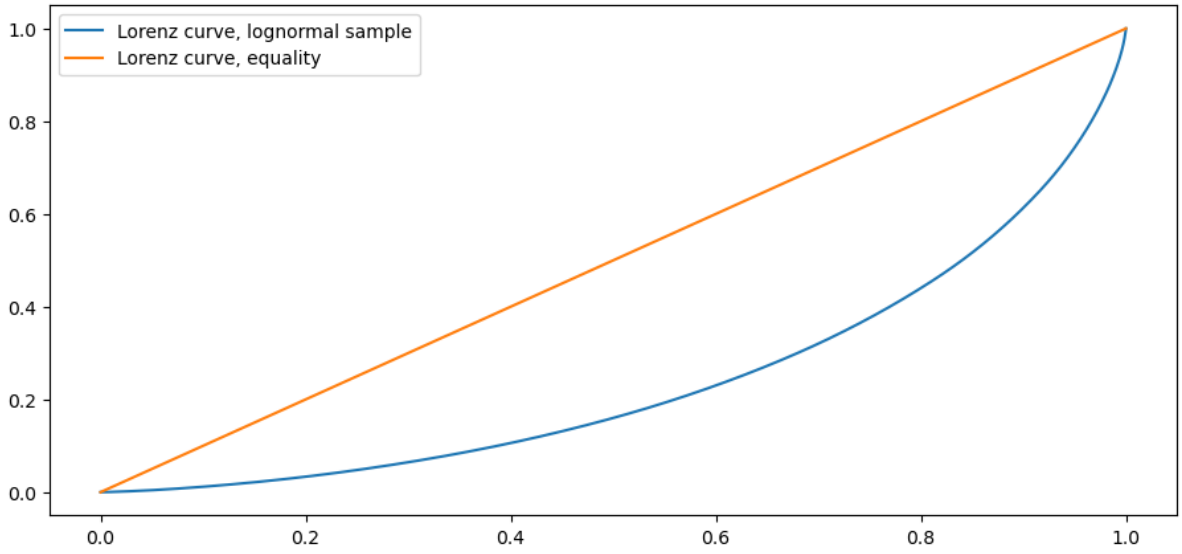
```
%%time

f_vals, l_vals = lorenz_curve_jax(w)
```

```
CPU times: user 1.84 s, sys: 19.1 ms, total: 1.85 s
Wall time: 1.57 s
```

```
%%time

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

```
CPU times: user 898 µs, sys: 271 µs, total: 1.17 ms
Wall time: 463 µs
```

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

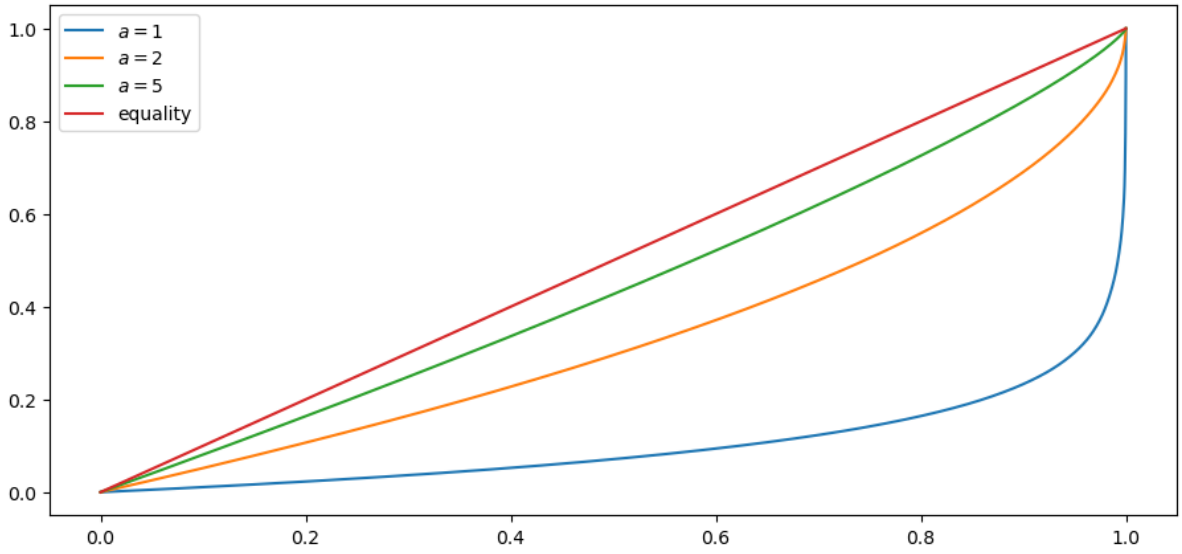


Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

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

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

```
fig, ax = plt.subplots()
for a in a_vals:
    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.

6.1.2 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.

```
@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.

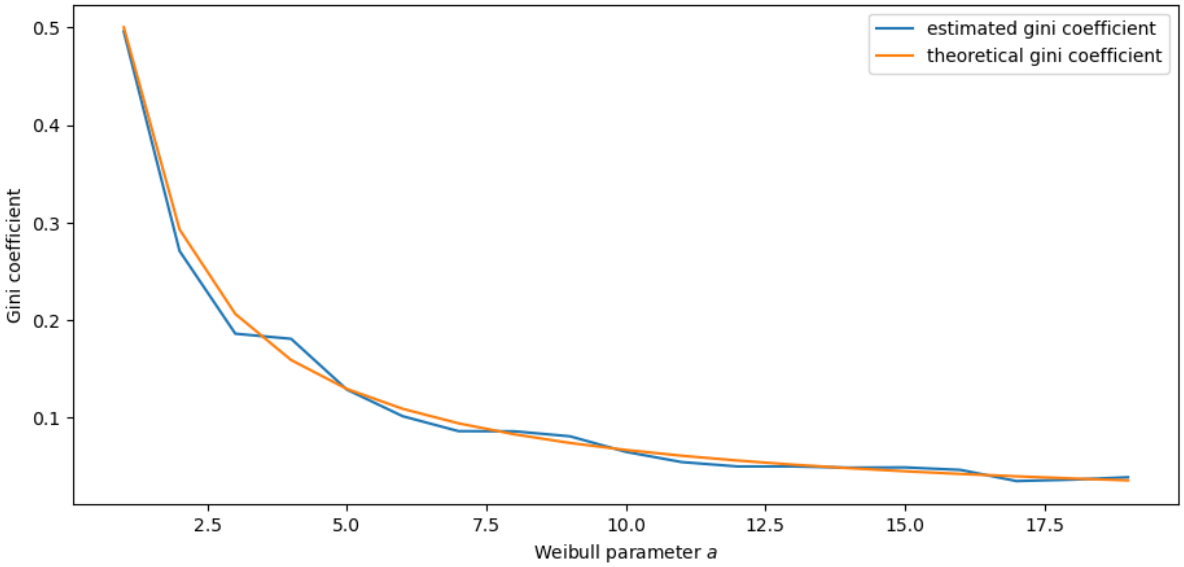
```
a_vals = range(1, 20)
ginis = []
ginis_theoretical = []
n = 100

for a in a_vals:
    rand_key = jax.random.PRNGKey(a)
```

(continues on next page)

(continued from previous page)

```
    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()
```



The simulation shows that the fit is good.

6.2 A Model of Wealth Dynamics

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

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \tag{6.1}$$

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

6.3 Implementation using JAX

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

```
# NamedTuple Model

Model = namedtuple("Model", ("w_hat", "s_0", "c_y", "μ_y",
                             "σ_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

```
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
    and 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 + σ_r**2 / 2)
    y_mean = c_y * exp_z_mean + jnp.exp(μ_y + σ_y**2 / 2)
    # Test a stability condition that ensures wealth does not diverge
    # to infinity.
    α = R_mean * s_0
    if α >= 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_y=σ_y, c_r=c_r, μ_r=μ_r, σ_r=σ_r, a=a,
                 b=b, σ_z=σ_z, z_mean=z_mean, z_var=z_var, y_mean=y_mean)
```

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

```
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.σ_z * jax.random.normal(rand_key, shape=size)

    # Update wealth
    y = wdy.c_y * jnp.exp(zp) + jnp.exp(
```

(continues on next page)

(continued from previous page)

```
                        wdy.μ_y + wdy.σ_y * jax.random.normal(subkey[0], shape=size))
    wp = y

    R = wdy.c_r * jnp.exp(zp) + jnp.exp(
                        wdy.μ_r + wdy.σ_r * jax.random.normal(subkey[1], shape=size))
    wp += (w >= wdy.w_hat) * R * wdy.s_0 * w
    return wp, zp


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

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

```
# 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.

        * wdy: NamedTuple Model
        * w_0: scalar/vector
        * n: int
        * size: size/shape of the w_0
        * rand_seed: int (Used to generate PRNG key)
    """
    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_)
    return jnp.array(w)


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

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

```
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()
```

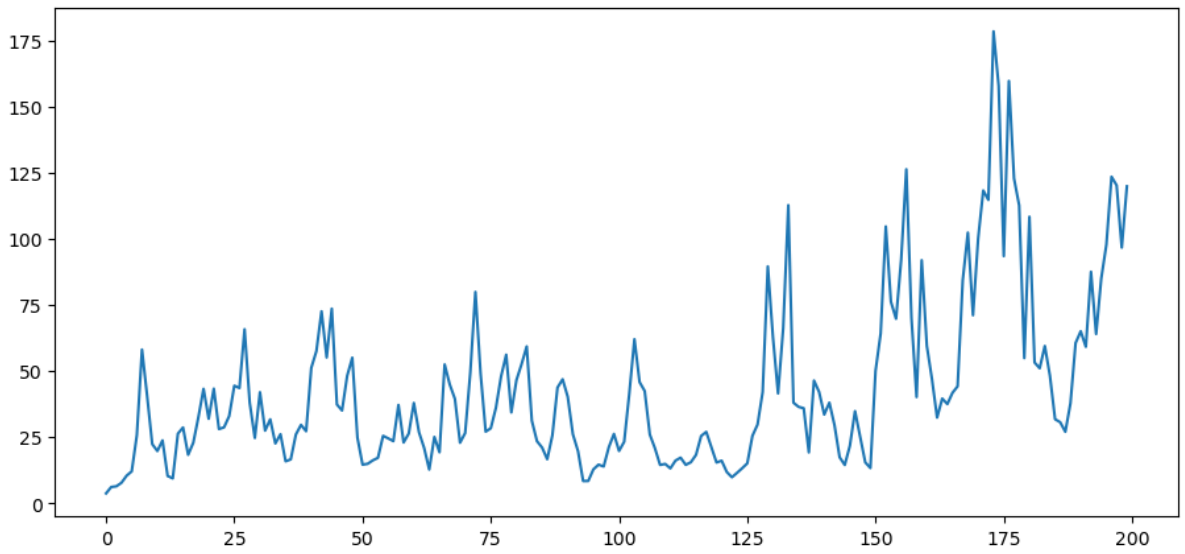
```
CPU times: user 56.7 s, sys: 895 ms, total: 57.6 s
Wall time: 54.5 s
```

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

```
%%time

# 2nd time is expected to be very fast because of JIT
w_jax_result = wealth_time_series_for_loop_jax(wdy.y_mean,
                                               ts_length, wdy, size).block_until_
 ↪ready()
```

```
CPU times: user 12.2 ms, sys: 345 µs, total: 12.5 ms
Wall time: 8.81 ms
```

```
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`.

```
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`.
```

(continues on next page)

(continued from previous page)

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

        * wdy: NamedTuple Model
        * w_0: scalar/vector
        * n: int
        * size: size/shape of the w_0
        * rand_seed: int (Used to generate PRNG key)
    """
    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: 257 µs, total: 1.31 s
Wall time: 742 ms
```

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

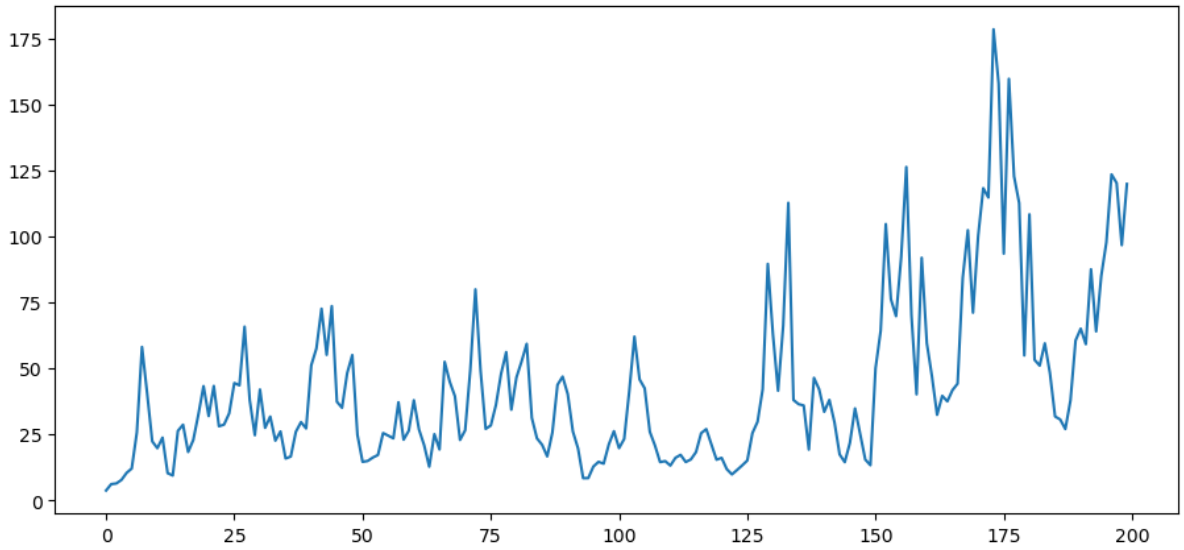
```
%%time

# 2nd time is expected to be very fast because of JIT
w_jax_result = wealth_time_series_jax(wdy.y_mean, ts_length, wdy, size).block_until_
↪ready()
```

(continues on next page)

(continued from previous page)

```
CPU times: user 11.4 ms, sys: 115 µs, total: 11.5 ms
Wall time: 8.1 ms
```

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



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

```
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)
    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,))
```

6.4 Applications

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

6.4.1 Inequality Measures

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

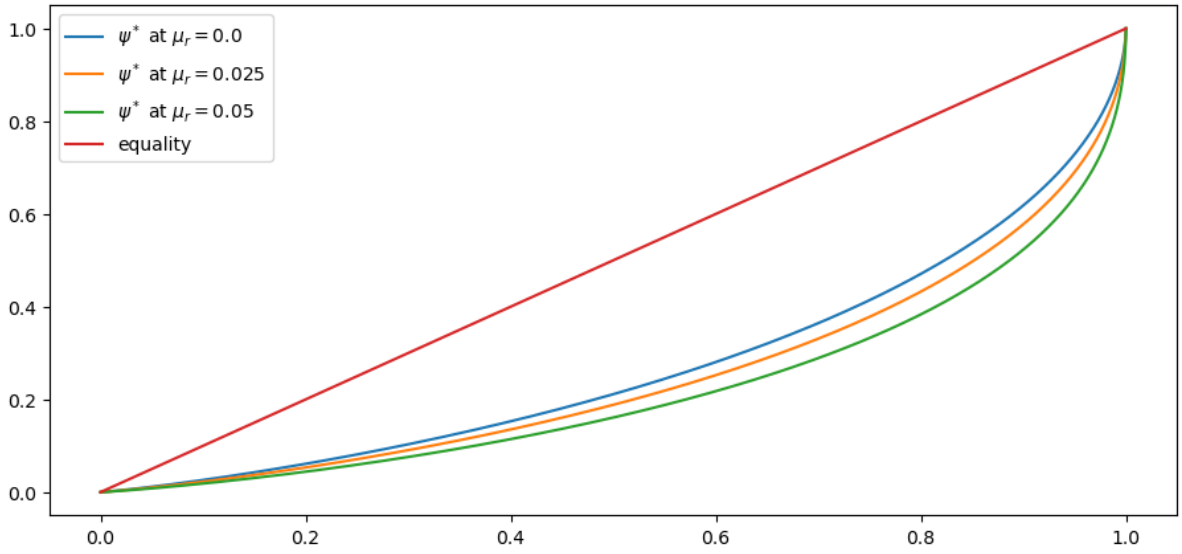
The next function generates a cross section and then computes the Lorenz curve and Gini coefficient.

```
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.

```
%%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 $\mu_r = {μ_r:0.2}$')
    gini_vals.append(gv)

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

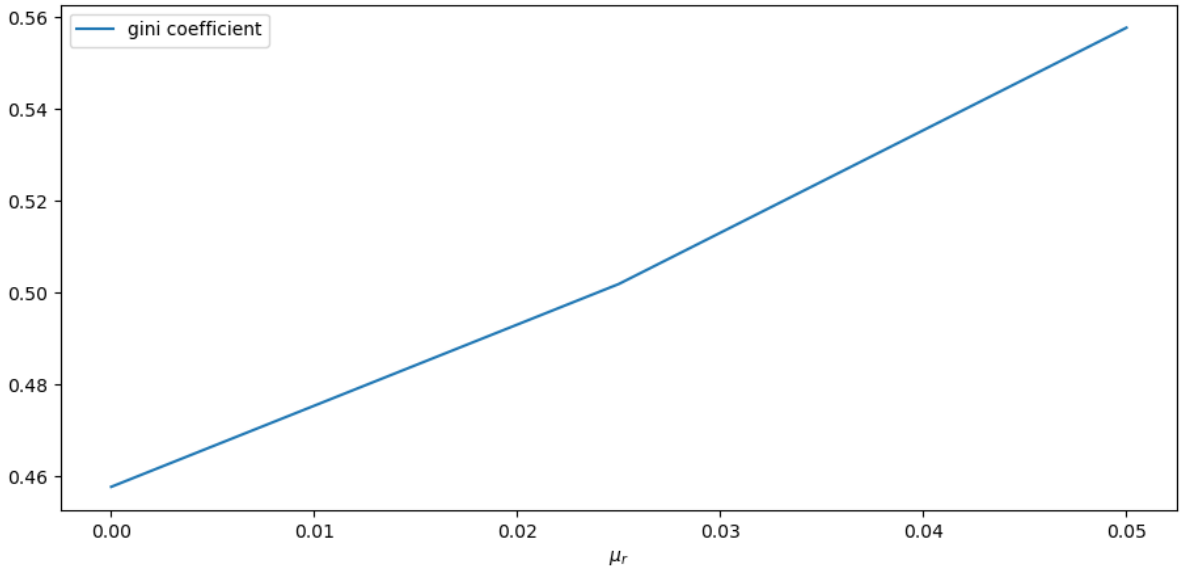



```
CPU times: user 7.76 s, sys: 21.7 ms, total: 7.78 s
Wall time: 6.42 s
```

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

Now let's check the Gini coefficient.

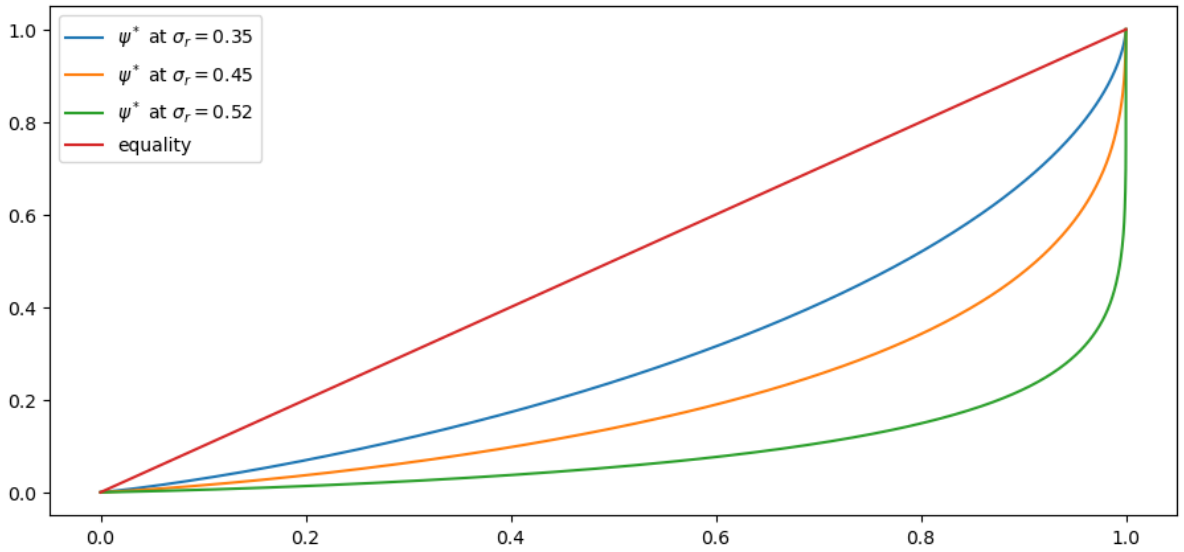
```
fig, ax = plt.subplots()
ax.plot(μ_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel("$\mu_r$")
ax.legend()
plt.show()
```



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

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

```
%%time

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

for σ_r in σ_r_vals:
    wdy = 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 $\sigma_r = {σ_r:0.2}$')
    gini_vals.append(gv)

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




```
CPU times: user 3.14 s, sys: 9.04 ms, total: 3.15 s
Wall time: 3.09 s
```

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

6.5 Exercises

Exercise 6.1

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

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

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

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

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

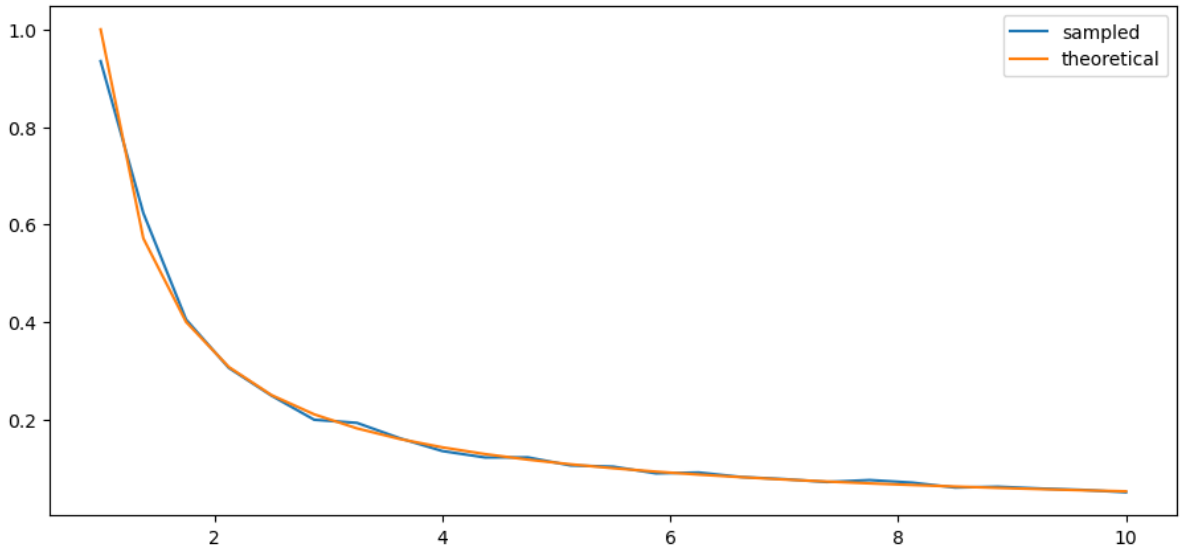
Use sample of size 1,000 for each a and the sampling method for generating Pareto draws employed in the discussion of Lorenz curves for the Pareto distribution.

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

Solution to Exercise 6.1

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

```
a_vals = jnp.linspace(1, 10, 25)  # Pareto tail index
ginis = []

n = 1000                          # size of each sample
fig, ax = plt.subplots()
for i, a in enumerate(a_vals):
    rand_key = jax.random.PRNGKey(i*10)
    u = jax.random.uniform(rand_key, shape=(n,))
    y = u**(-1/a)
    ginis.append(gini_jax(y))
ax.plot(a_vals, ginis, label='sampled')
ax.plot(a_vals, 1/(2*a_vals - 1), label='theoretical')
ax.legend()
plt.show()
```



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

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

More equality translates to a lower Gini index.

Exercise 6.2

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 (6.1) are not IID.

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

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

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

For sample size and initial conditions, use

Solution to Exercise 6.2

First let's generate the distribution:

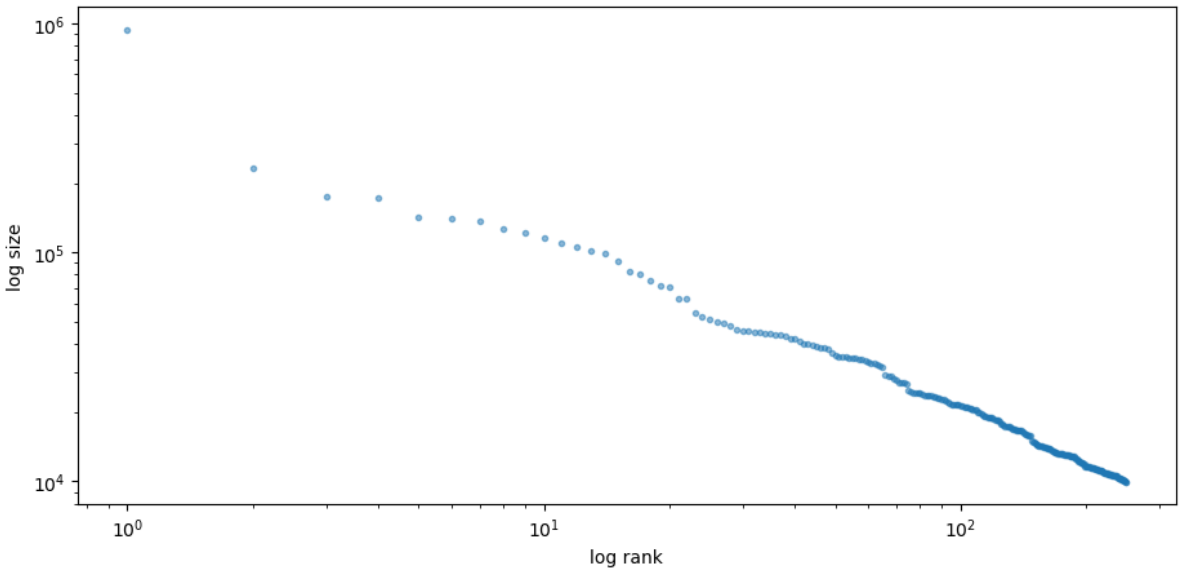
```
num_households = 250_000
T = 500  # how far to shift forward in time
size = (num_households, )

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

```
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:

```
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()
```


10^6
10^5
10^4
log size
10^0
10^1
10^2
log rank

CHAPTER

SEVEN

SHORTEST PATHS

7.1 Overview

This lecture is the extended version of the shortest path lecture using JAX. Please see that lecture for all background and notation.

Let's start by importing the libraries.

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

Let's check the backend used by JAX and the devices available.

```
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")

# Check the devices available for JAX
print(jax.devices())
```

```
JAX backend: gpu
[StreamExecutorGpuDevice(id=0, process_index=0, slice_index=0)]
```

7.2 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.

7.2.1 The Algorithm

The standard algorithm for finding J is to start an initial guess and then iterate.

This is a standard approach to solving nonlinear equations, often called the method of **successive approximations**.

Our initial guess will be

$$J_0(v) = 0 \text{ for all } v \tag{7.1}$$

Now

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v}\{c(v,w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n, go to 2

This sequence converges to J.

Let's start by defining the **distance matrix** Q.

```
inf = jnp.inf
Q = jnp.array([[inf, 1,   5,   3,   inf, inf, inf],
               [inf, inf, inf, 9,   6,   inf, inf],
               [inf, inf, inf, inf, inf, 2,   inf],
               [inf, inf, inf, inf, inf, 4,   8],
               [inf, inf, inf, inf, inf, inf, 4],
               [inf, inf, inf, inf, inf, inf, 1],
               [inf, inf, inf, inf, inf, inf, 0]])
```

Notice that the cost of staying still (on the principle diagonal) is set to

- `jnp.inf` for non-destination nodes — moving on is required.
- `0` for the destination node — here is where we stop.

Let's try with this example using python `while` loop and some `jax` vectorized code:

```
%%time

num_nodes = Q.shape[0]
J = jnp.zeros(num_nodes)

max_iter = 500
i = 0

while i < max_iter:
    next_J = jnp.min(Q + J, axis=1)
    if jnp.allclose(next_J, J):
        break
    else:
        J = next_J.copy()
        i += 1

print("The cost-to-go function is", J)
```

```
The cost-to-go function is [ 8. 10.  3.  5.  4.  1.  0.]
CPU times: user 1.83 s, sys: 541 ms, total: 2.37 s
Wall time: 2.53 s
```

We can further optimize the above code by using jax.lax.while_loop. The extra acceleration is due to the fact that the entire operation can be optimized by the JAX compiler and launched as a single kernel on the GPU.

```
max_iter = 500
num_nodes = Q.shape[0]
J = jnp.zeros(num_nodes)
```

```
def body_fun(values):
    # Define the body function of while loop
```

(continues on next page)

(continued from previous page)

```
    i, J, break_cond = values

    # Update J and break condition
    next_J = jnp.min(Q + J, axis=1)
    break_condition = jnp.allclose(next_J, J)

    # Return next iteration values
    return i + 1, next_J, break_condition
```

```
def cond_fun(values):
    i, J, break_condition = values
    return ~break_condition & (i < max_iter)
```

Let's see the timing for JIT compilation of the functions and runtime results.

```
%%time

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

```
CPU times: user 136 ms, sys: 7.71 ms, total: 144 ms
Wall time: 167 ms
```

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

Now, this runs faster once we have the JIT compiled JAX version of the functions.

```
%%time

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

```
CPU times: user 1.61 ms, sys: 884 µs, total: 2.49 ms
Wall time: 1.27 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.

7.3 Exercises

Exercise 7.1

The text below describes a weighted directed graph.

The line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04

- node8 at cost 11.11
- node14 at cost 72.21

No other nodes can be reached directly from node0.

Other lines have a similar interpretation.

Your task is to use the algorithm given above to find the optimal path and its cost.

```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
```

(continues on next page)

(continued from previous page)

```
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Overwriting graph.txt
```

Solution to Exercise 7.1

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

```
num_nodes = 100
destination_node = 99
def map_graph_to_distance_matrix(in_file):

    # First let's set of the distance matrix Q with inf everywhere
    Q = np.full((num_nodes, num_nodes), np.inf)

    # Now we read in the data and modify Q
    with open(in_file) as infile:
        for line in infile:
            elements = line.split(',')
            node = elements.pop(0)
            node = int(node[4:])    # convert node description to integer
            if node != destination_node:
                for element in elements:
                    destination, cost = element.split()
                    destination = int(destination[4:])
                    Q[node, destination] = float(cost)
            Q[destination_node, destination_node] = 0
    return jnp.array(Q)
```

Let's write a function `compute_cost_to_go` that returns J given any valid Q.

```
@jax.jit
def compute_cost_to_go(Q):
    num_nodes = Q.shape[0]
    J = jnp.zeros(num_nodes)       # Initial guess
    max_iter = 500
    i = 0

    def body_fun(values):
        # Define the body function of while loop
        i, J, break_cond = values

        # Update J and break condition
        next_J = jnp.min(Q + J, axis=1)
        break_condition = jnp.allclose(next_J, J)

        # Return next iteration values
        return i + 1, next_J, break_condition

    def cond_fun(values):
        i, J, break_condition = values
        return ~break_condition & (i < max_iter)

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

Finally, here's a function that uses the `cost-to-go` function to obtain the optimal path (and its cost).

```
def print_best_path(J, Q):
    sum_costs = 0
```

(continues on next page)

(continued from previous page)

```
    current_node = 0
    while current_node != destination_node:
        print(current_node)
        # Move to the next node and increment costs
        next_node = jnp.argmin(Q[current_node, :] + J)
        sum_costs += Q[current_node, next_node]
        current_node = next_node
    print(destination_node)
    print('Cost: ', sum_costs)
```

Okay, now we have the necessary functions, let's call them to do the job we were assigned.

```
Q = map_graph_to_distance_matrix('graph.txt')
```

Let's see the timings for jitting the function and runtime results.

```
%%time

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

```
CPU times: user 156 ms, sys: 12.8 ms, total: 168 ms
Wall time: 180 ms
```

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

```
CPU times: user 876 µs, sys: 456 µs, total: 1.33 ms
Wall time: 918 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
```

(continues on next page)

(continued from previous page)

```
94
96
97
98
99
Cost:  160.55
```

The total cost of the path should agree with $J[0]$ so let's check this.

```
J[0].item()
```

```
160.5500030517578
```

CHAPTER

EIGHT

OPTIMAL INVESTMENT

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

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

Combining with the demand curve and writing y, y' for Y_t, Y_{t+1}, this becomes

$$r(y, z, y') := (a_0 - a_1 y + z - c)y - \gamma(y' - y)^2$$

The firm maximizes present value of expected discounted profits. The Bellman equation is

$$v(y, z) = \max_{y'} \left\{ r(y, z, y') + \beta \sum_{z'} v(y', z') Q(z, z') \right\}.$$

We discretize y to a finite grid `y_grid`.

In essence, the firm tries to choose output close to the monopolist profit maximizer, given Z_t, but is constrained by adjustment costs.

Let's begin with the following imports

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

Let's check the backend used by JAX and the devices available

```
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")

# Check the devices available for JAX
print(jax.devices())
```

```
JAX backend: gpu
[StreamExecutorGpuDevice(id=0, process_index=0, slice_index=0)]
```

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.

```
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:
        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
```

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

for all (y, z, y').

```
def B(v, constants, sizes, arrays):
    """
    A vectorized version of the right-hand side of the Bellman equation
    (before maximization)
    """

    # Unpack
    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
    y_grid, z_grid, Q = arrays

    # Compute current rewards r(y, z, yp) as array r[i, j, ip]
    y  = jnp.reshape(y_grid, (y_size, 1, 1))    # y[i]   ->  y[i, j, ip]
    z  = jnp.reshape(z_grid, (1, z_size, 1))    # z[j]   ->  z[i, j, ip]
    yp = jnp.reshape(y_grid, (1, 1, y_size))    # yp[ip] -> yp[i, j, ip]
    r = (a_0 - a_1 * y + z - c) * y - γ * (yp - y)**2

    # Calculate continuation rewards at all combinations of (y, z, yp)
    v = jnp.reshape(v, (1, 1, y_size, z_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Q = jnp.reshape(Q, (1, z_size, 1, z_size))  # Q[j, jp]  -> Q[i, j, ip, jp]
    EV = jnp.sum(v * Q, axis=3)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return r + β * EV

# Create a jitted function
B = jax.jit(B, static_argnums=(2,))
```

Define a function to compute the current rewards given policy σ.

```
def compute_r_σ(σ, constants, sizes, arrays):
    """
    Compute the array r_σ[i, j] = r[i, j, σ[i, j]], which gives current
    rewards given policy σ.
    """

    # Unpack model
    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
```

(continues on next page)

(continued from previous page)

```
    y_grid, z_grid, Q = arrays

    # Compute r_σ[i, j]
    y = jnp.reshape(y_grid, (y_size, 1))
    z = jnp.reshape(z_grid, (1, z_size))
    yp = y_grid[σ]
    r_σ = (a_0 - a_1 * y + z - c) * y - γ * (yp - y)**2

    return r_σ


# Create the jitted function
compute_r_σ = jax.jit(compute_r_σ, static_argnums=(2,))
```

Define the Bellman operator.

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

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

The following function computes a v-greedy policy.

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

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

Define the σ-policy operator.

```
def T_σ(v, σ, constants, sizes, arrays):
    """The σ-policy operator."""

    # Unpack model
    β, a_0, a_1, γ, c = constants
    y_size, z_size = sizes
    y_grid, z_grid, Q = arrays

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

    # Compute the array v[σ[i, j], jp]
    zp_idx = jnp.arange(z_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, z_size))
    σ = jnp.reshape(σ, (y_size, z_size, 1))
    V = v[σ, zp_idx]

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

(continues on next page)

(continued from previous page)

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

Next, we want to computes the lifetime value of following policy σ.

The basic problem is to solve the linear system

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

for v.

It turns out to be helpful to rewrite this as

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

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

We want to write this as $v = r_\sigma + \beta P_\sigma v$ 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

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

Here we set up the linear map $v \mapsto R_\sigma v$,

where $R_\sigma := I - \beta P_\sigma$

In the investment problem, this map can be expressed as

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

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.

```
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.

```
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.

```
# 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
    vz = jnp.zeros(sizes)
    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):
```

(continues on next page)

(continued from previous page)

```
            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, γ, c = constants
y_size, z_size = sizes
y_grid, z_grid, Q = arrays
```

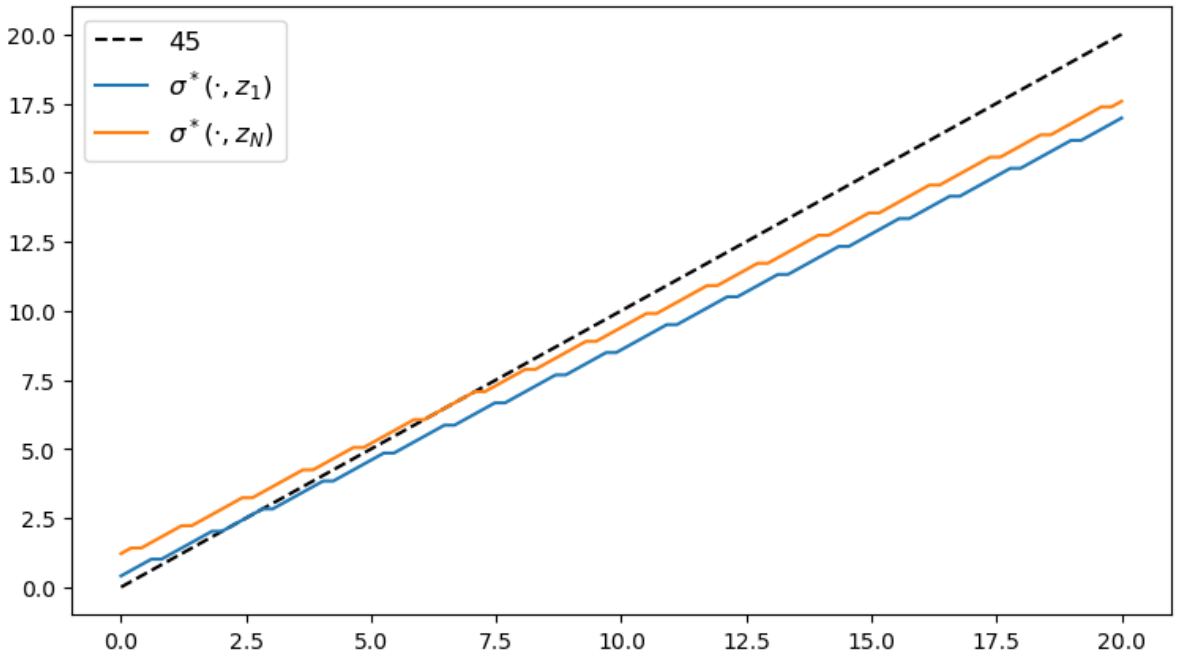
```
σ_star = policy_iteration(model)

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

```
Concluded loop 1 with error 50.
Concluded loop 2 with error 26.
Concluded loop 3 with error 17.
Concluded loop 4 with error 10.
Concluded loop 5 with error 7.
Concluded loop 6 with error 4.
Concluded loop 7 with error 3.
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
```

(continues on next page)

(continued from previous page)

```
Concluded loop 10 with error 1.
Concluded loop 11 with error 1.
Concluded loop 12 with error 0.
```



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

```
m_vals = range(5, 3000, 100)
```

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

```
Running Howard policy iteration.
Concluded loop 1 with error 50.
Concluded loop 2 with error 26.
Concluded loop 3 with error 17.
Concluded loop 4 with error 10.
Concluded loop 5 with error 7.
Concluded loop 6 with error 4.
Concluded loop 7 with error 3.
Concluded loop 8 with error 1.
Concluded loop 9 with error 1.
Concluded loop 10 with error 1.
Concluded loop 11 with error 1.
Concluded loop 12 with error 0.
TOC: Elapsed: 0:00:0.05
```

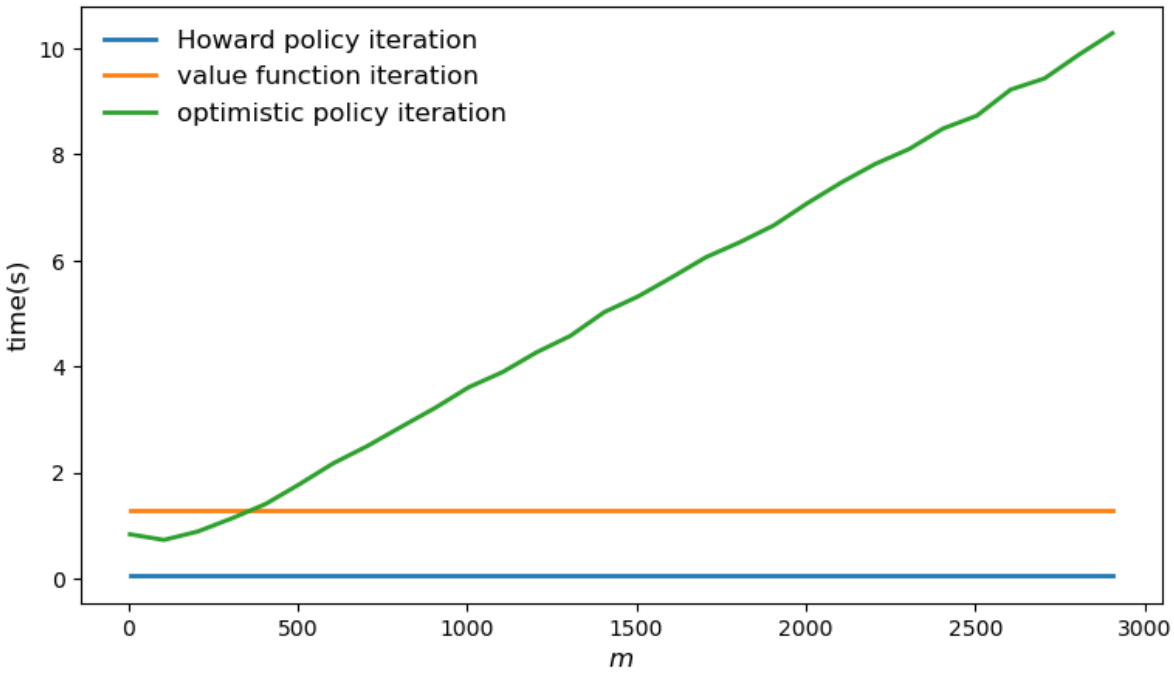
```
print(f"PI completed in {pi_time} seconds.")
print("Running value function iteration.")
qe.tic()
σ_vfi = value_iteration(model, tol=1e-5)
vfi_time = qe.toc()
print(f"VFI completed in {vfi_time} seconds.")
```

```
PI completed in 0.052335262298583984 seconds.
Running value function iteration.
```

```
TOC: Elapsed: 0:00:1.29
VFI completed in 1.2941081523895264 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()
```


Howard policy iteration
value function iteration
optimistic policy iteration
time(s)
m
0
2
4
6
8
10
0
500
1000
1500
2000
2500
3000

CHAPTER

NINE

OPTIMAL SAVINGS

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

```
!pip install quantecon
```

```
Requirement already satisfied: quantecon in /opt/conda/envs/quantecon/lib/python3.
↪10/site-packages (0.7.0)
Requirement already satisfied: requests in /opt/conda/envs/quantecon/lib/python3.
↪10/site-packages (from quantecon) (2.28.1)
Requirement already satisfied: scipy>=1.5.0 in /opt/conda/envs/quantecon/lib/
↪python3.10/site-packages (from quantecon) (1.10.0)
Requirement already satisfied: numba>=0.49.0 in /opt/conda/envs/quantecon/lib/
↪python3.10/site-packages (from quantecon) (0.56.4)
Requirement already satisfied: sympy in /opt/conda/envs/quantecon/lib/python3.10/
↪site-packages (from quantecon) (1.11.1)
Requirement already satisfied: numpy>=1.17.0 in /opt/conda/envs/quantecon/lib/
↪python3.10/site-packages (from quantecon) (1.23.5)
Requirement already satisfied: setuptools in /opt/conda/envs/quantecon/lib/python3.
↪10/site-packages (from numba>=0.49.0->quantecon) (65.6.3)
Requirement already satisfied: llvmlite<0.40,>=0.39.0dev0 in /opt/conda/envs/
↪quantecon/lib/python3.10/site-packages (from numba>=0.49.0->quantecon) (0.39.1)
```

```
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /opt/conda/envs/quantecon/
↪lib/python3.10/site-packages (from requests->quantecon) (1.26.14)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/envs/quantecon/lib/
↪python3.10/site-packages (from requests->quantecon) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/envs/quantecon/lib/
↪python3.10/site-packages (from requests->quantecon) (2022.12.7)
Requirement already satisfied: charset-normalizer<3,>=2 in /opt/conda/envs/
↪quantecon/lib/python3.10/site-packages (from requests->quantecon) (2.0.4)
Requirement already satisfied: mpmath>=0.19 in /opt/conda/envs/quantecon/lib/
↪python3.10/site-packages/mpmath-1.2.1-py3.10.egg (from sympy->quantecon) (1.2.1)
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and␣
↪conflicting behaviour with the system package manager. It is recommended to use␣
↪a virtual environment instead: https://pip.pypa.io/warnings/venv
```

We will use the following imports:

```
import quantecon as qe
import jax
```

(continues on next page)

(continued from previous page)

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

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.

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

9.1 Overview

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

$$W_{t+1} + C_t \leq RW_t + Y_t$$

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

The right-hand side of the Bellman equation is

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

where

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

We use successive approximation for VFI.

```
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:
        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
```

9.2 Model primitives

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

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

```
def create_consumption_model(R=1.01,                    # Gross interest rate
                             β=0.98,                    # Discount factor
                             γ=2.5,                     # CRRA parameter
                             w_min=0.01,                # Min wealth
                             w_max=5.0,                 # Max wealth
                             w_size=150,                # Grid side
                             ρ=0.9, ν=0.1, y_size=100): # Income parameters
    """
    A function that takes in parameters and returns an instance of Model that
    contains data for the optimal savings problem.
    """
    w_grid = jnp.linspace(w_min, w_max, w_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), 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,                    # Gross interest rate
                             β=0.98,                    # Discount factor
                             γ=2.5,                     # CRRA parameter
                             w_min=0.01,                # Min wealth
                             w_max=5.0,                 # Max wealth
                             w_size=150,                # Grid side
                             ρ=0.9, ν=0.1, y_size=100): # Income parameters
    """
    A function that takes in parameters and returns a JAX-compatible version of Model␣
↪that
    contains data for the optimal savings problem.
    """
    w_grid = jnp.linspace(w_min, w_max, w_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), 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):
    """
    A vectorized version of the right-hand side of the Bellman equation
    (before maximization), which is a 3D array representing

        B(w, y, w') = u(Rw + y - w') + β Σ_y' v(w', y') Q(y, y')

    for all (w, y, w').
    """
```

(continues on next page)

(continued from previous page)

```
    # Unpack
    β, R, γ = 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)
```

9.3 Operators

Now we define the policy operator T_σ

```
def compute_r_σ(σ, constants, sizes, arrays):
    """
    Compute the array r_σ[i, j] = r[i, j, σ[i, j]], which gives current
    rewards given policy σ.
    """

    # Unpack model
    β, 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):
    "The σ-policy operator."

    # 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)

(continued from previous page)

```
    # Compute the array v[σ[i, j], jp]
    yp_idx = jnp.arange(y_size)
    yp_idx = jnp.reshape(yp_idx, (1, 1, y_size))
    σ = jnp.reshape(σ, (w_size, y_size, 1))
    V = v[σ, yp_idx]

    # Convert Q[j, jp] to Q[i, j, jp]
    Q = jnp.reshape(Q, (1, y_size, y_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 * jnp.sum(V * Q, axis=2)
```

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_σ of following policy σ.

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_σ has four indices rather than 2

The code below

1. reshapes v and r_σ to 1D arrays and P_σ to a matrix
2. solves the linear system
3. converts back to multi-index arrays.

```
def get_value(σ, constants, sizes, arrays):
    "Get the value v_σ of policy σ by inverting the linear map R_σ."

    # Unpack
    β, R, γ = constants
    w_size, y_size = sizes
    w_grid, y_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]
```

```
def R_σ(v, σ, constants, sizes, arrays):
    """
    The value v_σ of a policy σ is defined as

        v_σ = (I - β P_σ)^{-1} r_σ

    Here we set up the linear map v -> R_σ v, where R_σ := I - β P_σ.

    In the consumption problem, this map can be expressed as

        (R_σ v)(w, y) = v(w, y) - β Σ_y' v(σ(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_σ.

    """

    β, R, γ = constants
    w_size, y_size = sizes
    w_grid, y_grid, Q = arrays

    # Set up the array v[σ[i, j], jp]
    zp_idx = jnp.arange(y_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, y_size))
    σ = jnp.reshape(σ, (w_size, y_size, 1))
    V = v[σ, zp_idx]

    # Expand Q[j, jp] to Q[i, j, jp]
    Q = jnp.reshape(Q, (1, y_size, y_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)
```

9.4 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,))
```

9.5 Solvers

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

```
def value_iteration(model, tol=1e-5):
    "Implements VFI."

    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)
```

```
def policy_iteration(model):
    "Howard policy iteration routine."
    constants, sizes, arrays = model
    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
        print(f"Concluded loop {i} with error {error}.")
    return σ
```

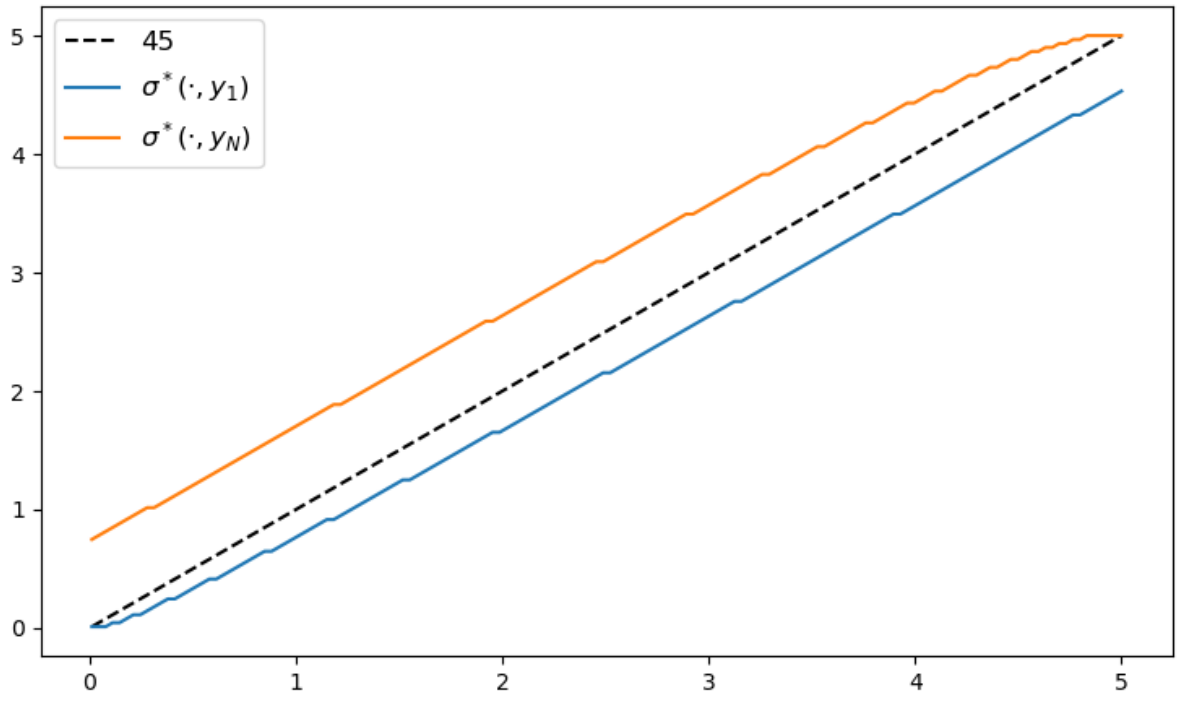
```
def optimistic_policy_iteration(model, tol=1e-5, m=10):
    "Implements the OPI routine."
    constants, sizes, arrays = model
    v = jnp.zeros(sizes)
    error = tol + 1
    while error > tol:
        last_v = v
        σ = get_greedy(v, constants, sizes, arrays)
        for _ in range(m):
   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

9.6 Plots

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

```
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.
```



9.7 Tests

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

```
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 0.03335976600646973 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 0.8902058601379395 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 0.3079864978790283 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.")
```

(continues on next page)

(continued from previous page)

```
    return result, elapsed_time

model = create_consumption_model_jax()
σ_pi, pi_time = run_algorithm(policy_iteration, model)
σ_vfi, vfi_time = run_algorithm(value_iteration, model, tol=1e-5)

m_vals = range(5, 3000, 100)
opi_times = []
for m in m_vals:
    σ_opi, opi_time = run_algorithm(optimistic_policy_iteration, model, m=m, tol=1e-5)
    opi_times.append(opi_time)

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()
```

```
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 0.31 seconds.
optimistic_policy_iteration completed in 0.12 seconds.
```

```
optimistic_policy_iteration completed in 0.09 seconds.
optimistic_policy_iteration completed in 0.15 seconds.
```

```
optimistic_policy_iteration completed in 0.30 seconds.
```

```
optimistic_policy_iteration completed in 0.40 seconds.
```

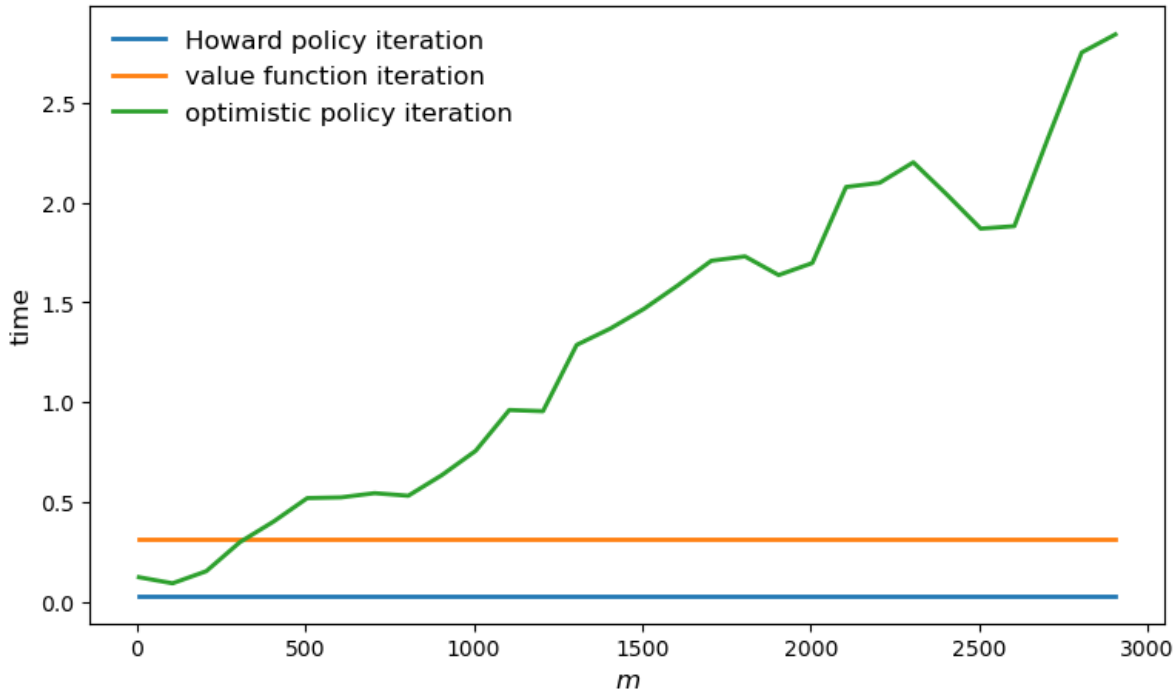
```
optimistic_policy_iteration completed in 0.52 seconds.
```

```
optimistic_policy_iteration completed in 0.52 seconds.
```

```
optimistic_policy_iteration completed in 0.54 seconds.
```

```
optimistic_policy_iteration completed in 0.53 seconds.
optimistic_policy_iteration completed in 0.63 seconds.
optimistic_policy_iteration completed in 0.76 seconds.
optimistic_policy_iteration completed in 0.96 seconds.
optimistic_policy_iteration completed in 0.95 seconds.
optimistic_policy_iteration completed in 1.29 seconds.
optimistic_policy_iteration completed in 1.37 seconds.
optimistic_policy_iteration completed in 1.47 seconds.
optimistic_policy_iteration completed in 1.59 seconds.
optimistic_policy_iteration completed in 1.71 seconds.
optimistic_policy_iteration completed in 1.73 seconds.
optimistic_policy_iteration completed in 1.64 seconds.
optimistic_policy_iteration completed in 1.70 seconds.
optimistic_policy_iteration completed in 2.08 seconds.
optimistic_policy_iteration completed in 2.10 seconds.
optimistic_policy_iteration completed in 2.20 seconds.
optimistic_policy_iteration completed in 2.04 seconds.
optimistic_policy_iteration completed in 1.87 seconds.
optimistic_policy_iteration completed in 1.88 seconds.
optimistic_policy_iteration completed in 2.33 seconds.
optimistic_policy_iteration completed in 2.75 seconds.
```

```
optimistic_policy_iteration completed in 2.84 seconds.
```



CHAPTER

TEN

TROUBLESHOOTING

Contents

This page is for readers experiencing errors when running the code from the lectures.

10.1 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.

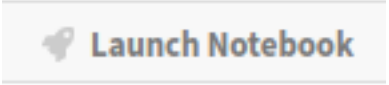
You also need to keep the external code libraries, such as QuantEcon.py up to date.

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.

10.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

ELEVEN

REFERENCES

CHAPTER

TWELVE

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
aiyagari_jax	2023-05-08 03:42	cache	56.72	✅
intro	2023-05-08 03:42	cache	1.38	✅
inventory_dynamics	2023-05-08 03:48	cache	354.08	✅
kesten_processes	2023-05-08 03:49	cache	23.03	✅
newtons_method	2023-05-08 03:51	cache	161.91	✅
opt_invest	2023-05-08 03:54	cache	175.75	✅
opt_savings	2023-05-08 03:55	cache	58.04	✅
short_path	2023-05-08 03:55	cache	7.91	✅
status	2023-05-08 03:42	cache	1.38	✅
troubleshooting	2023-05-08 03:42	cache	1.38	✅
wealth_dynamics	2023-05-08 03:57	cache	84.72	✅
zreferences	2023-05-08 03:42	cache	1.38	✅

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.

INDEX

K

L

M