Part V

Consumption, Savings and Growth

CHAPTER

THIRTYTHREE

CAKE EATING I: INTRODUCTION TO OPTIMAL SAVING

Contents

33.1 Overview

In this lecture we introduce a simple "cake eating" problem.

The intertemporal problem is: how much to enjoy today and how much to leave for the future?

Although the topic sounds trivial, this kind of trade-off between current and future utility is at the heart of many savings and consumption problems.

Once we master the ideas in this simple environment, we will apply them to progressively more challenging—and useful—problems.

The main tool we will use to solve the cake eating problem is dynamic programming.

Readers might find it helpful to review the following lectures before reading this one:

- The *shortest paths lecture*
- The *basic McCall model*
- The *McCall model with separation*
- The *McCall model with separation and a continuous wage distribution*

In what follows, we require the following imports:

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

33.2 The Model

We consider an infinite time horizon $t = 0, 1, 2, 3..$

At $t = 0$ the agent is given a complete cake with size $\bar{x}$.

Let x_t denote the size of the cake at the beginning of each period, so that, in particular, $x_0 = \bar{x}$.

We choose how much of the cake to eat in any given period t.

After choosing to consume c_t of the cake in period t there is

$$x_{t+1} = x_t - c_t$$

left in period $t + 1$.

Consuming quantity c of the cake gives current utility $u(c)$.

We adopt the CRRA utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \qquad (\gamma > 0,\ \gamma \neq 1) \tag{1}$$

In Python this is

```
def u(c, γ):

    return c**(1 - γ) / (1 - γ)
```

Future cake consumption utility is discounted according to $\beta \in (0, 1)$.

In particular, consumption of c units t periods hence has present value $\beta^t u(c)$

The agent's problem can be written as

$$\max_{\{c_t\}} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{2}$$

subject to

$$x_{t+1} = x_t - c_t \quad \text{and} \quad 0 \leq c_t \leq x_t \tag{3}$$

for all t.

A consumption path $\{c_t\}$ satisfying (3) where $x_0 = \bar{x}$ is called **feasible**.

In this problem, the following terminology is standard:

- x_t is called the **state variable**
- c_t is called the **control variable** or the **action**
- β and γ are **parameters**

33.2.1 Trade-Off

The key trade-off in the cake-eating problem is this:

- Delaying consumption is costly because of the discount factor.
- But delaying some consumption is also attractive because u is concave.

The concavity of u implies that the consumer gains value from *consumption smoothing*, which means spreading consumption out over time.

This is because concavity implies diminishing marginal utility—a progressively smaller gain in utility for each additional spoonful of cake consumed within one period.

33.2.2 Intuition

The reasoning given above suggests that the discount factor β and the curvature parameter γ will play a key role in determining the rate of consumption.

Here's an educated guess as to what impact these parameters will have.

First, higher β implies less discounting, and hence the agent is more patient, which should reduce the rate of consumption.

Second, higher γ implies that marginal utility $u'(c) = c^{-\gamma}$ falls faster with c.

This suggests more smoothing, and hence a lower rate of consumption.

In summary, we expect the rate of consumption to be *decreasing in both parameters*.

Let's see if this is true.

33.3 The Value Function

The first step of our dynamic programming treatment is to obtain the Bellman equation.

The next step is to use it to calculate the solution.

33.3.1 The Bellman Equation

To this end, we let $v(x)$ be maximum lifetime utility attainable from the current time when x units of cake are left.

That is,

$$v(x) = \max \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{4}$$

where the maximization is over all paths $\{c_t\}$ that are feasible from $x_0 = x$.

At this point, we do not have an expression for v, but we can still make inferences about it.

For example, as was the case with the *McCall model*, the value function will satisfy a version of the *Bellman equation*.

In the present case, this equation states that v satisfies

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for any given } x \geq 0. \tag{5}$$

The intuition here is essentially the same it was for the McCall model.

Choosing c optimally means trading off current vs future rewards.

Current rewards from choice c are just $u(c)$.

Future rewards given current cake size x, measured from next period and assuming optimal behavior, are $v(x - c)$.

These are the two terms on the right hand side of (5), after suitable discounting.

If c is chosen optimally using this trade off strategy, then we obtain maximal lifetime rewards from our current state x.

Hence, $v(x)$ equals the right hand side of (5), as claimed.

33.3.2 An Analytical Solution

It has been shown that, with u as the CRRA utility function in (1), the function

$$v^*(x_t) = \left(1 - \beta^{1/\gamma}\right)^{-\gamma} u(x_t) \tag{6}$$

solves the Bellman equation and hence is equal to the value function.

You are asked to confirm that this is true in the exercises below.

The solution (6) depends heavily on the CRRA utility function.

In fact, if we move away from CRRA utility, usually there is no analytical solution at all.

In other words, beyond CRRA utility, we know that the value function still satisfies the Bellman equation, but we do not have a way of writing it explicitly, as a function of the state variable and the parameters.

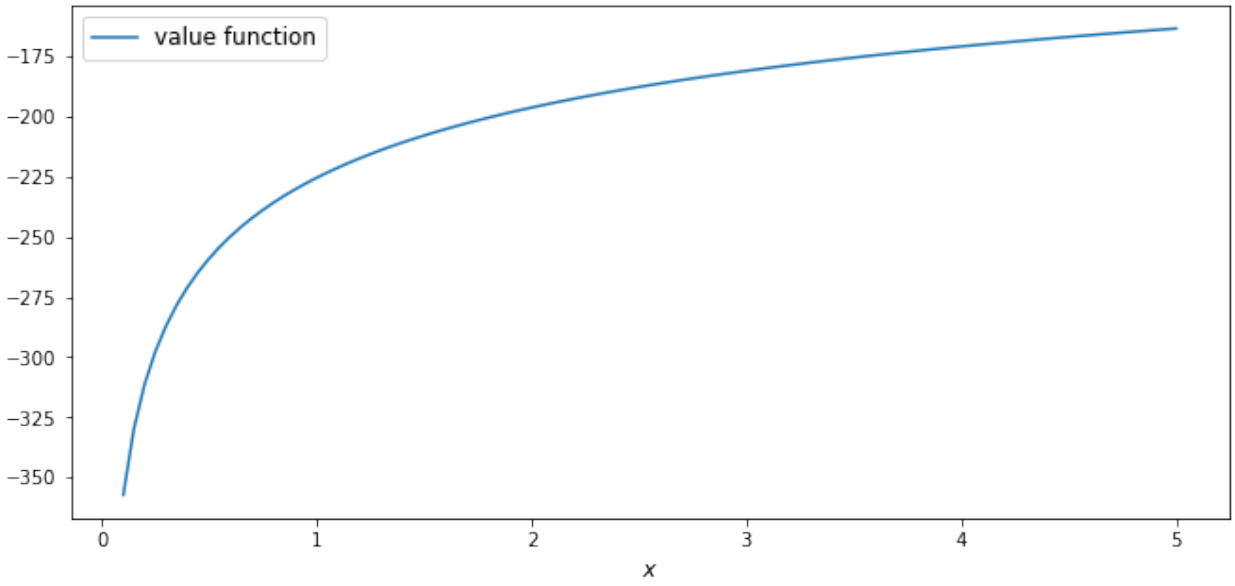
We will deal with that situation numerically when the time comes.

Here is a Python representation of the value function:

```
def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * u(x, γ)
```

And here's a figure showing the function for fixed parameters:

```
β, γ = 0.95, 1.2
x_grid = np.linspace(0.1, 5, 100)

fig, ax = plt.subplots()

ax.plot(x_grid, v_star(x_grid, β, γ), label='value function')

ax.set_xlabel('$x$', fontsize=12)
ax.legend(fontsize=12)

plt.show()
```



33.4 The Optimal Policy

Now that we have the value function, it is straightforward to calculate the optimal action at each state.

We should choose consumption to maximize the right hand side of the Bellman equation (5).

$$c^* = \arg\max_c \{u(c) + \beta v(x - c)\}$$

We can think of this optimal choice as a function of the state x, in which case we call it the **optimal policy**.

We denote the optimal policy by σ^*, so that

$$\sigma^*(x) := \arg\max_c \{u(c) + \beta v(x - c)\} \quad \text{for all } x$$

If we plug the analytical expression (6) for the value function into the right hand side and compute the optimum, we find that

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x \tag{7}$$

Now let's recall our intuition on the impact of parameters.

We guessed that the consumption rate would be decreasing in both parameters.

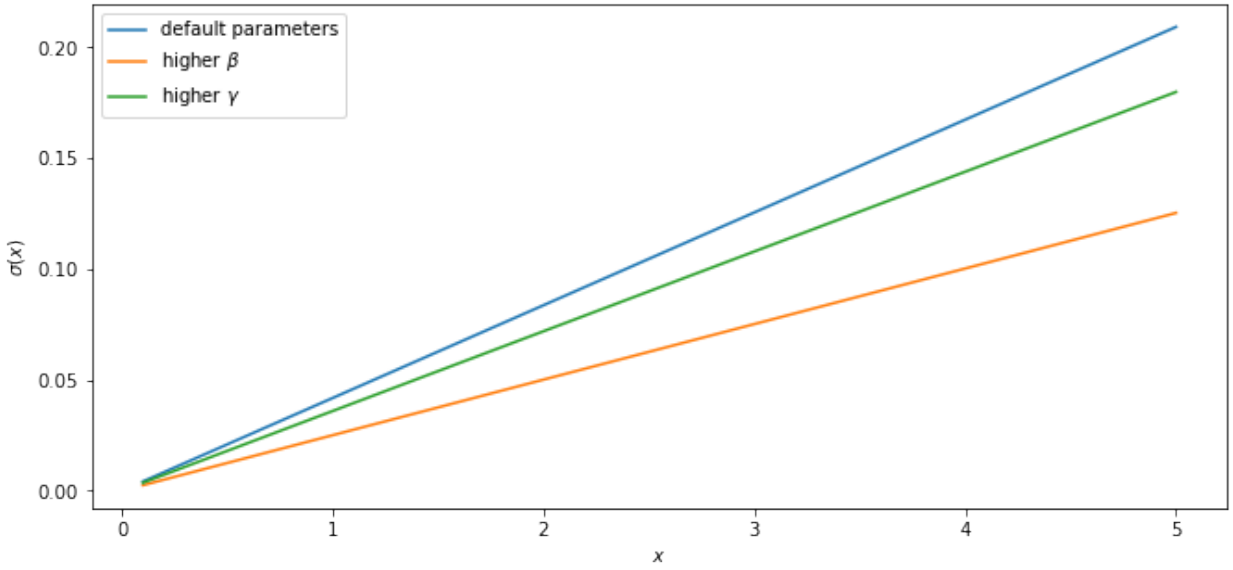
This is in fact the case, as can be seen from (7).

Here's some plots that illustrate.

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x
```

Continuing with the values for β and γ used above, the plot is

```
fig, ax = plt.subplots()
ax.plot(x_grid, c_star(x_grid, β, γ), label='default parameters')
ax.plot(x_grid, c_star(x_grid, β + 0.02, γ), label=r'higher $\beta$')
ax.plot(x_grid, c_star(x_grid, β, γ + 0.2), label=r'higher $\gamma$')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



33.5 The Euler Equation

In the discussion above we have provided a complete solution to the cake eating problem in the case of CRRA utility.

There is in fact another way to solve for the optimal policy, based on the so-called **Euler equation**.

Although we already have a complete solution, now is a good time to study the Euler equation.

This is because, for more difficult problems, this equation provides key insights that are hard to obtain by other methods.

33.5.1 Statement and Implications

The Euler equation for the present problem can be stated as

$$u'(c_t^*) = \beta u'(c_{t+1}^*) \tag{8}$$

This is necessary condition for the optimal path.

It says that, along the optimal path, marginal rewards are equalized across time, after appropriate discounting.

This makes sense: optimality is obtained by smoothing consumption up to the point where no marginal gains remain.

We can also state the Euler equation in terms of the policy function.

A **feasible consumption policy** is a map $x \mapsto \sigma(x)$ satisfying $0 \leq \sigma(x) \leq x$.

The last restriction says that we cannot consume more than the remaining quantity of cake.

A feasible consumption policy σ is said to **satisfy the Euler equation** if, for all $x > 0$,

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \tag{9}$$

Evidently (9) is just the policy equivalent of (8).

It turns out that a feasible policy is optimal if and only if it satisfies the Euler equation.

In the exercises, you are asked to verify that the optimal policy (7) does indeed satisfy this functional equation.

Note: A **functional equation** is an equation where the unknown object is a function.

For a proof of sufficiency of the Euler equation in a very general setting, see proposition 2.2 of [MST20].

The following arguments focus on necessity, explaining why an optimal path or policy should satisfy the Euler equation.

33.5.2 Derivation I: A Perturbation Approach

Let's write c as a shorthand for consumption path $\{c_t\}_{t=0}^{\infty}$.

The overall cake-eating maximization problem can be written as

$$\max_{c \in F} U(c) \quad \text{where } U(c) := \sum_{t=0}^{\infty} \beta^t u(c_t)$$

and F is the set of feasible consumption paths.

We know that differentiable functions have a zero gradient at a maximizer.

So the optimal path $c^* := \{c_t^*\}_{t=0}^{\infty}$ must satisfy $U'(c^*) = 0$.

Note: If you want to know exactly how the derivative $U'(c^*)$ is defined, given that the argument c^* is a vector of infinite length, you can start by learning about Gateaux derivatives. However, such knowledge is not assumed in what follows.

In other words, the rate of change in U must be zero for any infinitesimally small (and feasible) perturbation away from the optimal path.

So consider a feasible perturbation that reduces consumption at time t to $c_t^* - h$ and increases it in the next period to $c_{t+1}^* + h$.

Consumption does not change in any other period.

We call this perturbed path c^h.

By the preceding argument about zero gradients, we have

$$\lim_{h \to 0} \frac{U(c^h) - U(c^*)}{h} = U'(c^*) = 0$$

Recalling that consumption only changes at t and $t+1$, this becomes

$$\lim_{h \to 0} \frac{\beta^t u(c_t^* - h) + \beta^{t+1} u(c_{t+1}^* + h) - \beta^t u(c_t^*) - \beta^{t+1} u(c_{t+1}^*)}{h} = 0$$

After rearranging, the same expression can be written as

$$\lim_{h \to 0} \frac{u(c_t^* - h) - u(c_t^*)}{h} + \beta \lim_{h \to 0} \frac{u(c_{t+1}^* + h) - u(c_{t+1}^*)}{h} = 0$$

or, taking the limit,

$$-u'(c_t^*) + \beta u'(c_{t+1}^*) = 0$$

This is just the Euler equation.

33.5.3 Derivation II: Using the Bellman Equation

Another way to derive the Euler equation is to use the Bellman equation (5).

Taking the derivative on the right hand side of the Bellman equation with respect to c and setting it to zero, we get

$$u'(c) = \beta v'(x - c) \tag{10}$$

To obtain $v'(x - c)$, we set $g(c, x) = u(c) + \beta v(x - c)$, so that, at the optimal choice of consumption,

$$v(x) = g(c, x) \tag{11}$$

Differentiating both sides while acknowledging that the maximizing consumption will depend on x, we get

$$v'(x) = \frac{\partial}{\partial c} g(c, x) \frac{\partial c}{\partial x} + \frac{\partial}{\partial x} g(c, x)$$

When $g(c, x)$ is maximized at c, we have $\frac{\partial}{\partial c} g(c, x) = 0$.

Hence the derivative simplifies to

$$v'(x) = \frac{\partial g(c, x)}{\partial x} = \frac{\partial}{\partial x} \beta v(x - c) = \beta v'(x - c) \tag{12}$$

(This argument is an example of the Envelope Theorem.)

But now an application of (10) gives

$$u'(c) = v'(x) \tag{13}$$

Thus, the derivative of the value function is equal to marginal utility.

Combining this fact with (12) recovers the Euler equation.

33.6 Exercises

33.6.1 Exercise 1

How does one obtain the expressions for the value function and optimal policy given in (6) and (7) respectively?

The first step is to make a guess of the functional form for the consumption policy.

So suppose that we do not know the solutions and start with a guess that the optimal policy is linear.

In other words, we conjecture that there exists a positive θ such that setting $c_t^* = \theta x_t$ for all t produces an optimal path.

Starting from this conjecture, try to obtain the solutions (6) and (7).

In doing so, you will need to use the definition of the value function and the Bellman equation.

33.7 Solutions

33.7.1 Exercise 1

We start with the conjecture $c_t^* = \theta x_t$, which leads to a path for the state variable (cake size) given by

$$x_{t+1} = x_t(1-\theta)$$

Then $x_t = x_0(1-\theta)^t$ and hence

$$\begin{aligned} v(x_0) &= \sum_{t=0}^{\infty} \beta^t u(\theta x_t) \\ &= \sum_{t=0}^{\infty} \beta^t u(\theta x_0 (1-\theta)^t) \\ &= \sum_{t=0}^{\infty} \theta^{1-\gamma} \beta^t (1-\theta)^{t(1-\gamma)} u(x_0) \\ &= \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} u(x_0) \end{aligned}$$

From the Bellman equation, then,

$$\begin{aligned} v(x) &= \max_{0 \le c \le x} \left\{ u(c) + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot u(x-c) \right\} \\ &= \max_{0 \le c \le x} \left\{ \frac{c^{1-\gamma}}{1-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot \frac{(x-c)^{1-\gamma}}{1-\gamma} \right\} \end{aligned}$$

From the first order condition, we obtain

$$c^{-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}(-1) = 0$$

or

$$c^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}$$

With $c = \theta x$ we get

$$(\theta x)^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x(1-\theta))^{-\gamma}$$

Some rearrangement produces

$$\theta = 1 - \beta^{\frac{1}{\gamma}}$$

This confirms our earlier expression for the optimal policy:

$$c_t^* = \left(1 - \beta^{\frac{1}{\gamma}}\right) x_t$$

Substituting θ into the value function above gives

$$v^*(x_t) = \frac{\left(1-\beta^{\frac{1}{\gamma}}\right)^{1-\gamma}}{1-\beta\left(\beta^{\frac{1-\gamma}{\gamma}}\right)} u(x_t)$$

Rearranging gives

$$v^*(x_t) = \left(1-\beta^{\frac{1}{\gamma}}\right)^{-\gamma} u(x_t)$$

Our claims are now verified.

CHAPTER

THIRTYFOUR

CAKE EATING II: NUMERICAL METHODS

Contents

In addition to what's in Anaconda, this lecture will require the following library:

```
!pip install interpolation
```

34.1 Overview

In this lecture we continue the study of *the cake eating problem*.

The aim of this lecture is to solve the problem using numerical methods.

At first this might appear unnecessary, since we already obtained the optimal policy analytically.

However, the cake eating problem is too simple to be useful without modifications, and once we start modifying the problem, numerical methods become essential.

Hence it makes sense to introduce numerical methods now, and test them on this simple problem.

Since we know the analytical solution, this will allow us to assess the accuracy of alternative numerical methods.

We will use the following imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
from scipy.optimize import minimize_scalar, bisect
```

34.2 Reviewing the Model

You might like to *review the details* before we start.

Recall in particular that the Bellman equation is

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for all } x \geq 0. \tag{1}$$

where u is the CRRA utility function.

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

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Our first aim is to obtain these analytical solutions numerically.

34.3 Value Function Iteration

The first approach we will take is **value function iteration**.

This is a form of **successive approximation**, and was discussed in our *lecture on job search*.

The basic idea is:

1. Take an arbitary intial guess of v.
2. Obtain an update w defined by

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

3. Stop if w is approximately equal to v, otherwise set $v = w$ and go back to step 2.

Let's write this a bit more mathematically.

34.3.1 The Bellman Operator

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

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

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

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

As we discuss in more detail in later lectures, one can use Banach's contraction mapping theorem to prove that the sequence of functions $T^n v$ converges to the solution to the Bellman equation.

34.3.2 Fitted Value Function Iteration

Both consumption c and the state variable x are continuous.

This causes complications when it comes to numerical work.

For example, we need to store each function $T^n v$ in order to compute the next iterate $T^{n+1} v$.

But this means we have to store $T^n v(x)$ at infinitely many x, which is, in general, impossible.

To circumvent this issue we will use fitted value function iteration, as discussed previously in *one of the lectures* on job search.

The process looks like this:

1. Begin with an array of values $\{v_0, \ldots, v_I\}$ representing the values of some initial function v on the grid points $\{x_0, \ldots, x_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(x_i)$ on each grid point x_i by repeatedly solving the maximization problem in the Bellman equation.
4. Unless some stopping condition is satisfied, set $\{v_0, \ldots, v_I\} = \{T\hat{v}(x_0), \ldots, T\hat{v}(x_I)\}$ and go to step 2.

In step 2 we'll use continuous piecewise linear interpolation.

34.3.3 Implementation

The `maximize` function below is a small helper function that converts a SciPy minimization routine into a maximization routine.

```
def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.  The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """

    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

We'll store the parameters β and γ in a class called `CakeEating`.

The same class will also provide a method called `state_action_value` that returns the value of a consumption choice given a particular state and guess of v.

```
class CakeEating:

    def __init__(self,
                 β=0.96,           # discount factor
                 γ=1.5,            # degree of relative risk aversion
                 x_grid_min=1e-3,  # exclude zero for numerical stability
                 x_grid_max=2.5,   # size of cake
```

(continues on next page)

(continued from previous page)

```
                 x_grid_size=120):

        self.β, self.γ = β, γ

        # Set up grid
        self.x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)

    # Utility function
    def u(self, c):

        γ = self.γ

        if γ == 1:
            return np.log(c)
        else:
            return (c ** (1 - γ)) / (1 - γ)

    # first derivative of utility function
    def u_prime(self, c):

        return c ** (-self.γ)

    def state_action_value(self, c, x, v_array):
        """
        Right hand side of the Bellman equation given x and c.
        """

        u, β = self.u, self.β
        v = lambda x: interp(self.x_grid, v_array, x)

        return u(c) + β * v(x - c)
```

We now define the Bellman operation:

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

    * ce is an instance of CakeEating
    * v is an array representing a guess of the value function

    """
    v_new = np.empty_like(v)

    for i, x in enumerate(ce.x_grid):
        # Maximize RHS of Bellman equation at state x
        v_new[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[1]

    return v_new
```

After defining the Bellman operator, we are ready to solve the model.

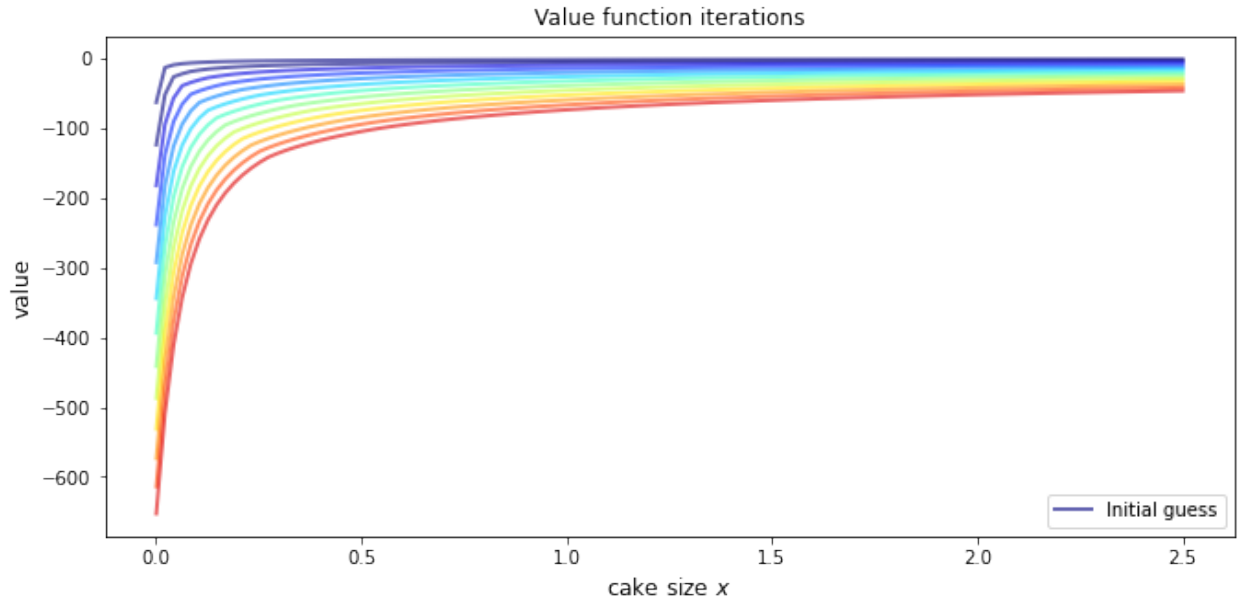
Let's start by creating a `CakeEating` instance using the default parameterization.

```
ce = CakeEating()
```

Now let's see the iteration of the value function in action.

We start from guess v given by $v(x) = u(x)$ for every x grid point.

```
x_grid = ce.x_grid
v = ce.u(x_grid)        # Initial guess
n = 12                  # Number of iterations

fig, ax = plt.subplots()

ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial guess')

for i in range(n):
    v = T(v, ce)  # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

ax.legend()
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('cake size $x$', fontsize=12)
ax.set_title('Value function iterations')

plt.show()
```



To do this more systematically, we introduce a wrapper function called `compute_value_function` that iterates until some convergence conditions are satisfied.

```
def compute_value_function(ce,
                           tol=1e-4,
                           max_iter=1000,
                           verbose=True,
                           print_skip=25):

    # Set up loop
    v = np.zeros(len(ce.x_grid)) # Initial guess
    i = 0
    error = tol + 1
```

(continues on next page)

(continued from previous page)

```
    while i < max_iter and error > tol:
        v_new = T(v, ce)

        error = np.max(np.abs(v - v_new))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        v = v_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

Now let's call it, noting that it takes a little while to run.

```
v = compute_value_function(ce)
```

```
Error at iteration 25 is 23.8003755134813.
```

```
Error at iteration 50 is 8.577577195046615.
```

```
Error at iteration 75 is 3.091330659691039.
```

```
Error at iteration 100 is 1.1141054204751981.
```

```
Error at iteration 125 is 0.4015199357729671.
```

```
Error at iteration 150 is 0.14470646660561215.
```

```
Error at iteration 175 is 0.052151735472762084.
```

```
Error at iteration 200 is 0.018795314242879613.
```

```
Error at iteration 225 is 0.006773769545588948.
```

```
Error at iteration 250 is 0.0024412443051460286.
```

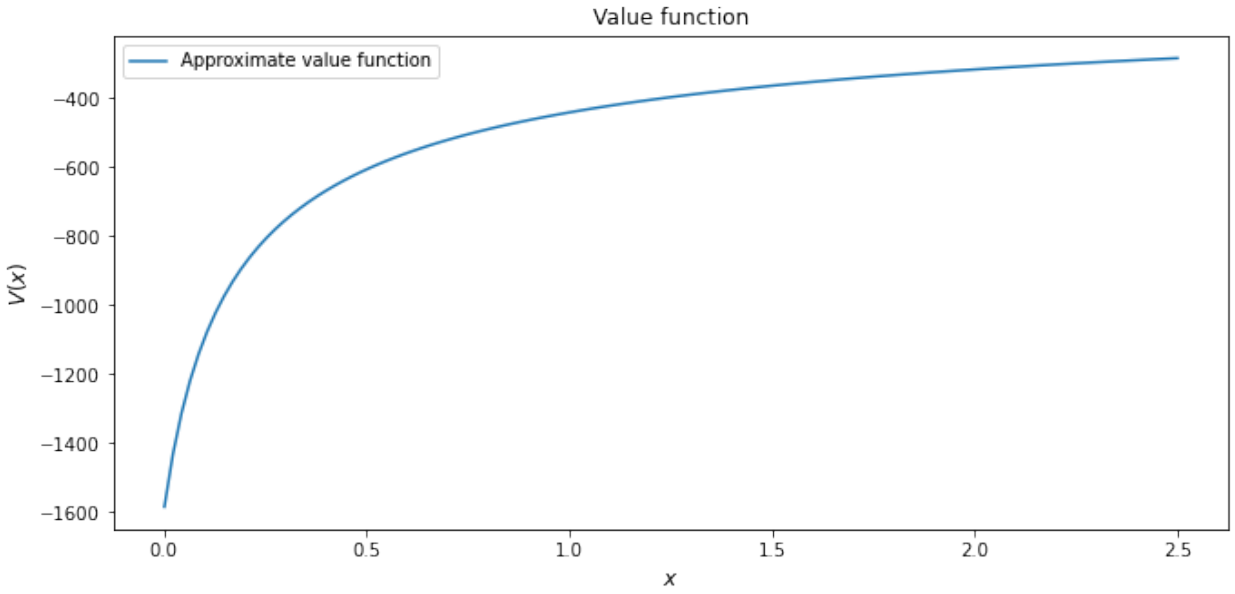
```
Error at iteration 275 is 0.000879816432870939.
```

```
Error at iteration 300 is 0.00031708295398402697.
```

```
Error at iteration 325 is 0.00011427565573285392.

Converged in 329 iterations.
```

Now we can plot and see what the converged value function looks like.

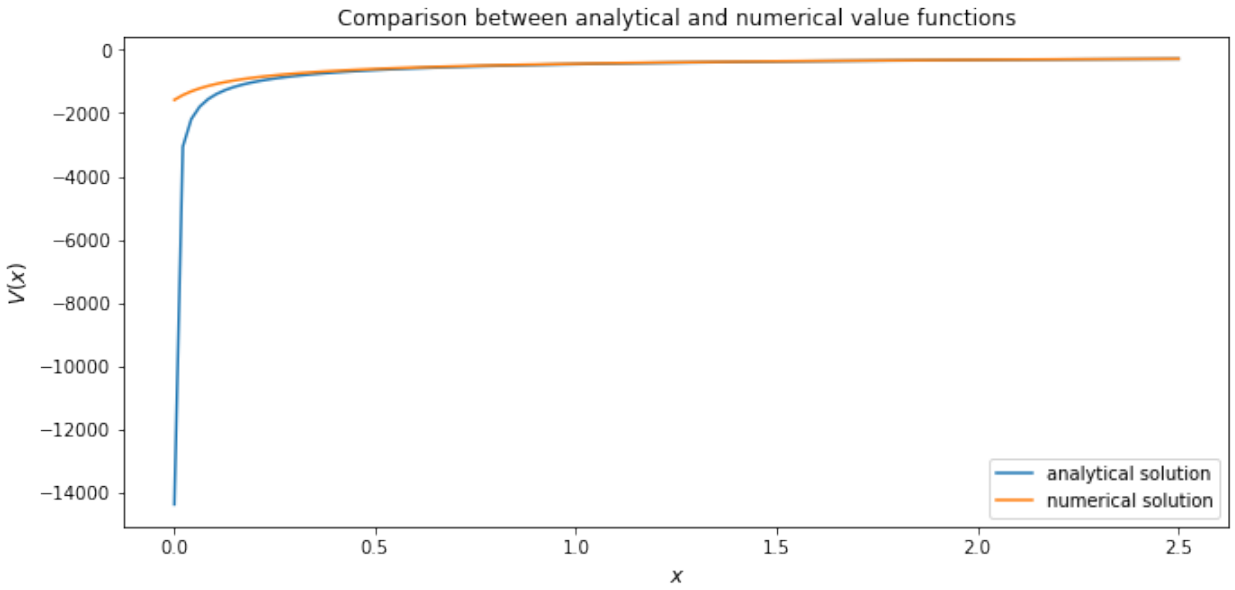
```
fig, ax = plt.subplots()

ax.plot(x_grid, v, label='Approximate value function')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.set_title('Value function')
ax.legend()
plt.show()
```



Next let's compare it to the analytical solution.

```
v_analytical = v_star(ce.x_grid, ce.β, ce.γ)
```

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

ax.plot(x_grid, v_analytical, label='analytical solution')
ax.plot(x_grid, v, label='numerical solution')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.legend()
ax.set_title('Comparison between analytical and numerical value functions')
plt.show()
```



The quality of approximation is reasonably good for large x, but less so near the lower boundary.

The reason is that the utility function and hence value function is very steep near the lower boundary, and hence hard to approximate.

34.3.4 Policy Function

Let's see how this plays out in terms of computing the optimal policy.

In the *first lecture on cake eating*, the optimal consumption policy was shown to be

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x$$

Let's see if our numerical results lead to something similar.

Our numerical strategy will be to compute

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

on a grid of x points and then interpolate.

For v we will use the approximation of the value function we obtained above.

Here's the function:

```
def σ(ce, v):
    """
    The optimal policy function. Given the value function,
    it finds optimal consumption in each state.

    * ce is an instance of CakeEating
    * v is a value function array

    """
    c = np.empty_like(v)

    for i in range(len(ce.x_grid)):
```

(continues on next page)

(continued from previous page)

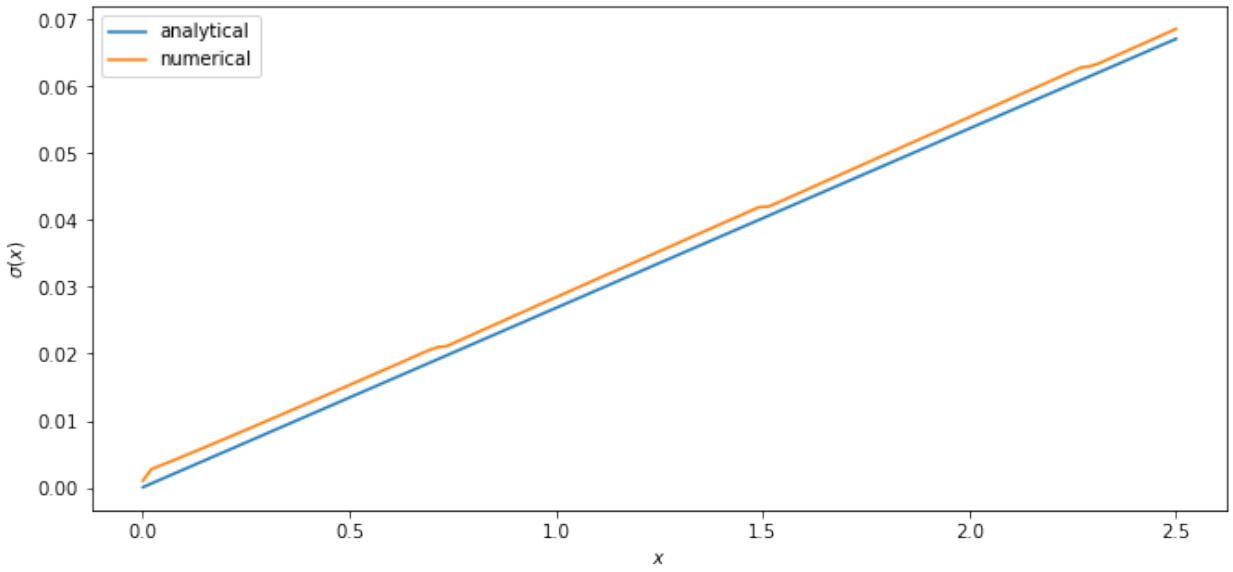
```
        x = ce.x_grid[i]
        # Maximize RHS of Bellman equation at state x
        c[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[0]

    return c
```

Now let's pass the approximate value function and compute optimal consumption:

```
c = σ(ce, v)
```

Let's plot this next to the true analytical solution

```
c_analytical = c_star(ce.x_grid, ce.β, ce.γ)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label='analytical')
ax.plot(ce.x_grid, c, label='numerical')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



The fit is reasonable but not perfect.

We can improve it by increasing the grid size or reducing the error tolerance in the value function iteration routine.

However, both changes will lead to a longer compute time.

Another possibility is to use an alternative algorithm, which offers the possibility of faster compute time and, at the same time, more accuracy.

We explore this next.

34.4 Time Iteration

Now let's look at a different strategy to compute the optimal policy.

Recall that the optimal policy satisfies the Euler equation

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \quad \text{for all } x > 0 \tag{2}$$

Computationally, we can start with any initial guess of σ_0 and now choose c to solve

$$u'(c) = \beta u'(\sigma_0(x - c))$$

Choosing c to satisfy this equation at all $x > 0$ produces a function of x.

Call this new function σ_1, treat it as the new guess and repeat.

This is called **time iteration**.

As with value function iteration, we can view the update step as action of an operator, this time denoted by K.

- In particular, $K\sigma$ is the policy updated from σ using the procedure just described.
- We will use this terminology in the exercises below.

The main advantage of time iteration relative to value function iteration is that it operates in policy space rather than value function space.

This is helpful because the policy function has less curvature, and hence is easier to approximate.

In the exercises you are asked to implement time iteration and compare it to value function iteration.

You should find that the method is faster and more accurate.

This is due to

1. the curvature issue mentioned just above and
2. the fact that we are using more information — in this case, the first order conditions.

34.5 Exercises

34.5.1 Exercise 1

Try the following modification of the problem.

Instead of the cake size changing according to $x_{t+1} = x_t - c_t$, let it change according to

$$x_{t+1} = (x_t - c_t)^\alpha$$

where α is a parameter satisfying $0 < \alpha < 1$.

(We will see this kind of update rule when we study optimal growth models.)

Make the required changes to value function iteration code and plot the value and policy functions.

Try to reuse as much code as possible.

34.5.2 Exercise 2

Implement time iteration, returning to the original case (i.e., dropping the modification in the exercise above).

34.6 Solutions

34.6.1 Exercise 1

We need to create a class to hold our primitives and return the right hand side of the Bellman equation.

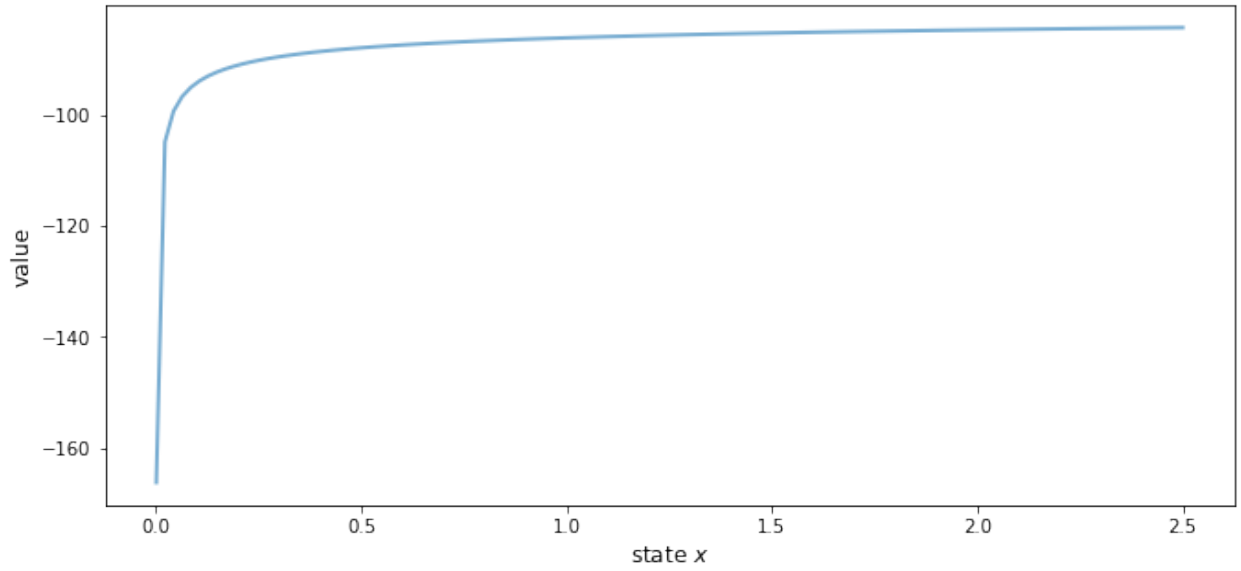
We will use inheritance to maximize code reuse.

```
class OptimalGrowth(CakeEating):
    """
    A subclass of CakeEating that adds the parameter α and overrides
    the state_action_value method.
    """

    def __init__(self,
                 β=0.96,           # discount factor
                 γ=1.5,            # degree of relative risk aversion
                 α=0.4,            # productivity parameter
                 x_grid_min=1e-3,  # exclude zero for numerical stability
                 x_grid_max=2.5,   # size of cake
                 x_grid_size=120):

        self.α = α
        CakeEating.__init__(self, β, γ, x_grid_min, x_grid_max, x_grid_size)

    def state_action_value(self, c, x, v_array):
        """
        Right hand side of the Bellman equation given x and c.
        """

        u, β, α = self.u, self.β, self.α
        v = lambda x: interp(self.x_grid, v_array, x)

        return u(c) + β * v((x - c)**α)
```

```
og = OptimalGrowth()
```

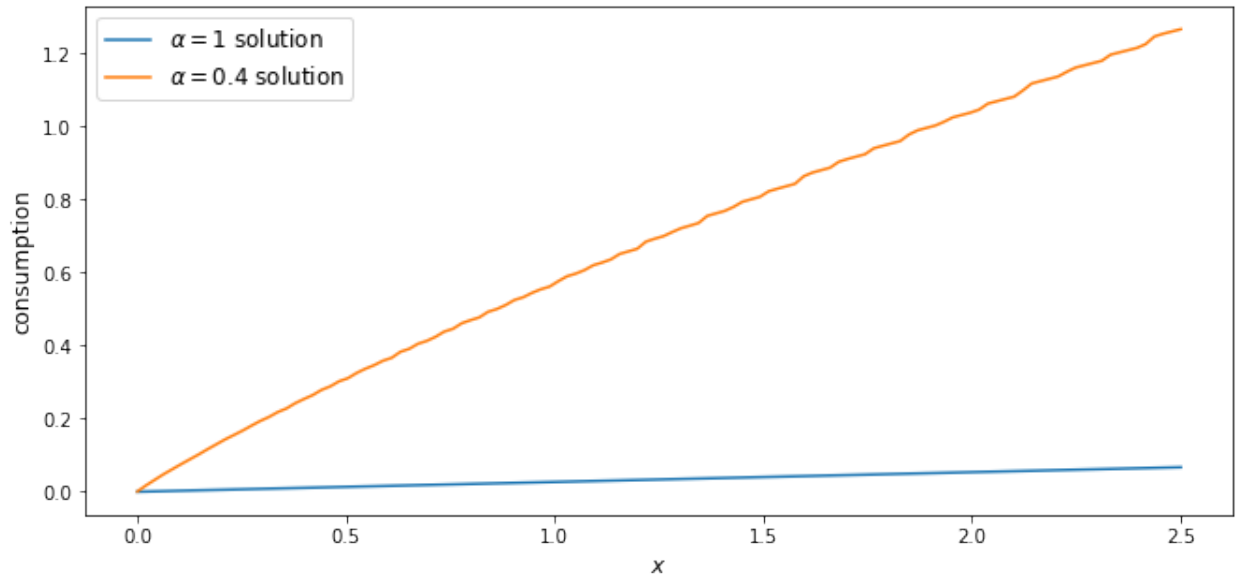
Here's the computed value function.

```
v = compute_value_function(og, verbose=False)

fig, ax = plt.subplots()

ax.plot(x_grid, v, lw=2, alpha=0.6)
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('state $x$', fontsize=12)

plt.show()
```



Here's the computed policy, combined with the solution we derived above for the standard cake eating case $\alpha = 1$.

```
c_new = σ(og, v)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label=r'$\alpha=1$ solution')
ax.plot(ce.x_grid, c_new, label=fr'$\alpha={og.α}$ solution')

ax.set_ylabel('consumption', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)

ax.legend(fontsize=12)

plt.show()
```



Consumption is higher when $\alpha < 1$ because, at least for large x, the return to savings is lower.

34.6.2 Exercise 2

Here's one way to implement time iteration.

```
def K(σ_array, ce):
    """
    The policy function operator. Given the policy function,
    it updates the optimal consumption using Euler equation.

    * σ_array is an array of policy function values on the grid
    * ce is an instance of CakeEating

    """

    u_prime, β, x_grid = ce.u_prime, ce.β, ce.x_grid
    σ_new = np.empty_like(σ_array)

    σ = lambda x: interp(x_grid, σ_array, x)

    def euler_diff(c, x):
        return u_prime(c) - β * u_prime(σ(x - c))

    for i, x in enumerate(x_grid):

        # handle small x separately --- helps numerical stability
        if x < 1e-12:
            σ_new[i] = 0.0

        # handle other x
        else:
            σ_new[i] = bisect(euler_diff, 1e-10, x - 1e-10, x)

    return σ_new
```

```
def iterate_euler_equation(ce,
                           max_iter=500,
                           tol=1e-5,
                           verbose=True,
                           print_skip=25):

    x_grid = ce.x_grid

    σ = np.copy(x_grid)        # initial guess

    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        σ_new = K(σ, ce)

        error = np.max(np.abs(σ_new - σ))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        σ = σ_new
```

(continues on next page)

(continued from previous page)

```
    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return σ
```

```
ce = CakeEating(x_grid_min=0.0)
c_euler = iterate_euler_equation(ce)
```

```
Error at iteration 25 is 0.0036456675931543225.
```

```
Error at iteration 50 is 0.0008283185047067848.
Error at iteration 75 is 0.0003079132300957147.
```

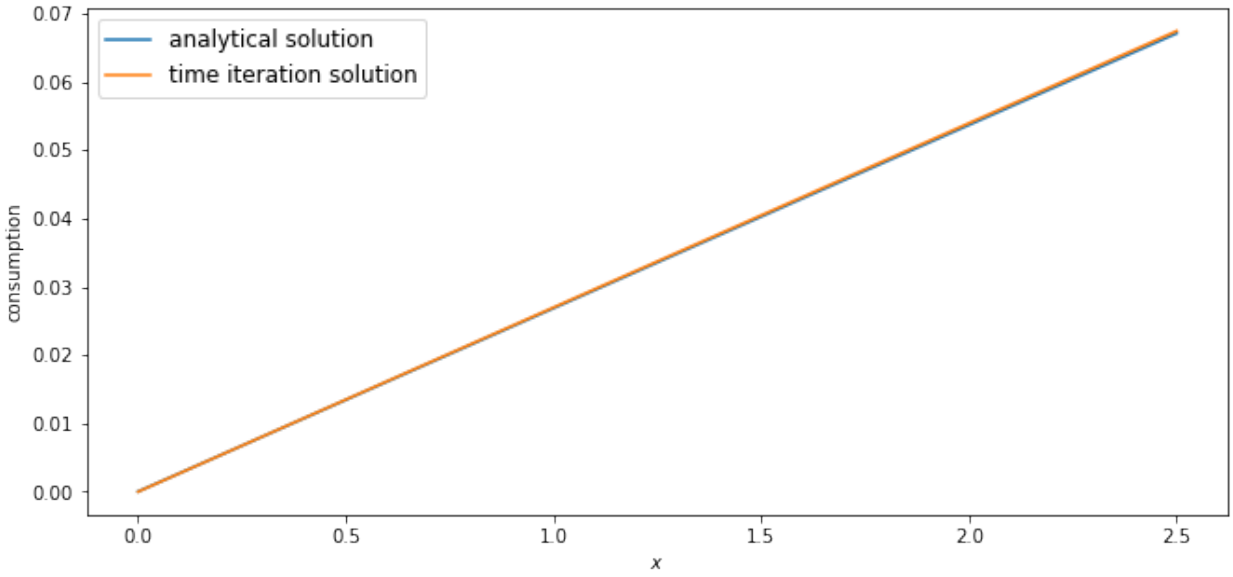
```
Error at iteration 100 is 0.00013555502390599772.
```

```
Error at iteration 125 is 6.417740905302616e-05.
```

```
Error at iteration 150 is 3.1438019047758115e-05.
Error at iteration 175 is 1.5658492883291464e-05.
```

```
Converged in 192 iterations.
```

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

ax.plot(ce.x_grid, c_analytical, label='analytical solution')
ax.plot(ce.x_grid, c_euler, label='time iteration solution')

ax.set_ylabel('consumption')
ax.set_xlabel('$x$')
ax.legend(fontsize=12)

plt.show()
```


analytical solution
time iteration solution
consumption
x
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.0
0.5
1.0
1.5
2.0
2.5

CHAPTER

THIRTYFIVE

OPTIMAL GROWTH I: THE STOCHASTIC OPTIMAL GROWTH MODEL

Contents

35.1 Overview

In this lecture, we're going to study a simple optimal growth model with one agent.

The model is a version of the standard one sector infinite horizon growth model studied in

- [SLP89], chapter 2
- [LS18], section 3.1
- EDTC, chapter 1
- [Sun96], chapter 12

It is an extension of the simple *cake eating problem* we looked at earlier.

The extension involves

- nonlinear returns to saving, through a production function, and
- stochastic returns, due to shocks to production.

Despite these additions, the model is still relatively simple.

We regard it as a stepping stone to more sophisticated models.

We solve the model using dynamic programming and a range of numerical techniques.

In this first lecture on optimal growth, the solution method will be value function iteration (VFI).

While the code in this first lecture runs slowly, we will use a variety of techniques to drastically improve execution time over the next few lectures.

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import minimize_scalar
```

35.2 The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t.

This output can either be consumed or invested.

When the good is invested, it is transformed one-for-one into capital.

The resulting capital stock, denoted here by k_{t+1}, will then be used for production.

Production is stochastic, in that it also depends on a shock ξ_{t+1} realized at the end of the current period.

Next period output is

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is called the production function.

The resource constraint is

$$k_{t+1} + c_t \leq y_t \tag{1}$$

and all variables are required to be nonnegative.

35.2.1 Assumptions and Comments

In what follows,

- The sequence $\{\xi_t\}$ is assumed to be IID.
- The common distribution of each ξ_t will be denoted by ϕ.
- The production function f is assumed to be increasing and continuous.
- Depreciation of capital is not made explicit but can be incorporated into the production function.

While many other treatments of the stochastic growth model use k_t as the state variable, we will use y_t.

This will allow us to treat a stochastic model while maintaining only one state variable.

We consider alternative states and timing specifications in some of our other lectures.

35.2.2 Optimization

Taking y_0 as given, the agent wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(c_t)\right] \tag{2}$$

subject to

$$y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t \tag{3}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0,1)$ is a discount factor.

In (3) we are assuming that the resource constraint (1) holds with equality — which is reasonable because u is strictly increasing and no output will be wasted at the optimum.

In summary, the agent's aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is

1. nonnegative,
2. feasible in the sense of (1),
3. optimal, in the sense that it maximizes (2) relative to all other feasible consumption sequences, and
4. *adapted*, in the sense that the action c_t depends only on observable outcomes, not on future outcomes such as ξ_{t+1}.

In the present context

- y_t is called the *state* variable — it summarizes the "state of the world" at the start of each period.
- c_t is called the *control* variable — a value chosen by the agent each period after observing the state.

35.2.3 The Policy Function Approach

One way to think about solving this problem is to look for the best **policy function**.

A policy function is a map from past and present observables into current action.

We'll be particularly interested in **Markov policies**, which are maps from the current state y_t into a current action c_t.

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy.

In other words, the current state y_t provides a sufficient statistic for the history in terms of making an optimal decision today.

This is quite intuitive, but if you wish you can find proofs in texts such as [SLP89] (section 4.1).

Hereafter we focus on finding the best Markov policy.

In our context, a Markov policy is a function $\sigma : \mathbb{R}_+ \to \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call σ a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{4}$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ.

Each $\sigma \in \Sigma$ determines a continuous state Markov process $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \tag{5}$$

This is the time path for output when we choose and stick with the policy σ.

We insert this process into the objective function to get

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{6}$$

This is the total expected present value of following policy σ forever, given initial income y_0.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

35.2.4 Optimality

The σ associated with a given policy σ is the mapping defined by

$$v_\sigma(y) = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{7}$$

when $\{y_t\}$ is given by (5) with $y_0 = y$.

In other words, it is the lifetime value of following policy σ starting at initial condition y.

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} v_\sigma(y) \tag{8}$$

The value function gives the maximal value that can be obtained from state y, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if it attains the supremum in (8) for all $y \in \mathbb{R}_+$.

35.2.5 The Bellman Equation

With our assumptions on utility and production functions, the value function as defined in (8) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

$$v(y) = \max_{0 \le c \le y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{9}$$

This is a *functional equation in* v.

The term $\int v(f(y-c)z)\phi(dz)$ can be understood as the expected next period value when

- v is used to measure value
- the state is y
- consumption is set to c

As shown in EDTC, theorem 10.1.11 and a range of other texts

The value function v^* satisfies the Bellman equation

In other words, (9) holds when $v = v^*$.

The intuition is that maximal value from a given state can be obtained by optimally trading off

- current reward from a given action, vs
- expected discounted future value of the state resulting from that action

The Bellman equation is important because it gives us more information about the value function.

It also suggests a way of computing the value function, which we discuss below.

35.2.6 Greedy Policies

The primary importance of the value function is that we can use it to compute optimal policies.

The details are as follows.

Given a continuous function v on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is v-**greedy** if $\sigma(y)$ is a solution to

$$\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \tag{10}$$

for every $y \in \mathbb{R}_+$.

In other words, $\sigma \in \Sigma$ is v-greedy if it optimally trades off current and future rewards when v is taken to be the value function.

In our setting, we have the following key result

- A feasible consumption policy is optimal if and only if it is v^*-greedy.

The intuition is similar to the intuition for the Bellman equation, which was provided after (9).

See, for example, theorem 10.1.11 of EDTC.

Hence, once we have a good approximation to v^*, we can compute the (approximately) optimal policy by computing the corresponding greedy policy.

The advantage is that we are now solving a much lower dimensional optimization problem.

35.2.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(An operator is a map that sends functions into functions.)

The Bellman operator is denoted by T and defined by

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{11}$$

In other words, T sends the function v into the new function Tv defined by (11).

By construction, the set of solutions to the Bellman equation (9) *exactly coincides with* the set of fixed points of T.

For example, if $Tv = v$, then, for any $y \geq 0$,

$$v(y) = Tv(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

which says precisely that v is a solution to the Bellman equation.

It follows that v^* is a fixed point of T.

35.2.8 Review of Theoretical Results

One can also show that T is a contraction mapping on the set of continuous bounded functions on $\mathbb{R}_+$ under the supremum distance

$$\rho(g, h) = \sup_{y \geq 0} |g(y) - h(y)|$$

See EDTC, lemma 10.1.18.

Hence, it has exactly one fixed point in this set, which we know is equal to the value function.

It follows that

- The value function v^* is bounded and continuous.
- Starting from any bounded and continuous v, the sequence $v, Tv, T^2v, \ldots$ generated by iteratively applying T converges uniformly to v^*.

This iterative method is called **value function iteration**.

We also know that a feasible policy is optimal if and only if it is v^*-greedy.

It's not too hard to show that a v^*-greedy policy exists (see EDTC, theorem 10.1.11 if you get stuck).

Hence, at least one optimal policy exists.

Our problem now is how to compute it.

35.2.9 Unbounded Utility

The results stated above assume that the utility function is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Unfortunately, they tend to be case-specific, as opposed to valid for a large range of applications.

Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word "bounded").

Consult, for example, section 12.2 of EDTC, [Kam12] or [MdRV10].

35.3 Computation

Let's now look at computing the value function and the optimal policy.

Our implementation in this lecture will focus on clarity and flexibility.

Both of these things are helpful, but they do cost us some speed — as you will see when you run the code.

Later we will sacrifice some of this clarity and flexibility in order to accelerate our code with just-in-time (JIT) compilation.

The algorithm we will use is fitted value function iteration, which was described in earlier lectures *the McCall model* and *cake eating*.

The algorithm will be

1. Begin with an array of values $\{v_1, \ldots, v_I\}$ representing the values of some initial function v on the grid points $\{y_1, \ldots, y_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(y_i)$ on each grid point y_i by repeatedly solving (11).
4. Unless some stopping condition is satisfied, set $\{v_1, \ldots, v_I\} = \{T\hat{v}(y_1), \ldots, T\hat{v}(y_I)\}$ and go to step 2.

35.3.1 Scalar Maximization

To maximize the right hand side of the Bellman equation (9), we are going to use the `minimize_scalar` routine from SciPy.

Since we are maximizing rather than minimizing, we will use the fact that the maximizer of g on the interval $[a, b]$ is the minimizer of $-g$ on the same interval.

To this end, and to keep the interface tidy, we will wrap `minimize_scalar` in an outer function as follows:

```
def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.  The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """

    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

35.3.2 Optimal Growth Model

We will assume for now that ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ where

- ζ is standard normal,
- μ is a shock location parameter and
- s is a shock scale parameter.

We will store this and other primitives of the optimal growth model in a class.

The class, defined below, combines both parameters and a method that realizes the right hand side of the Bellman equation (9).

```
class OptimalGrowthModel:

    def __init__(self,
                 u,            # utility function
                 f,            # production function
                 β=0.96,       # discount factor
                 μ=0,          # shock location parameter
                 s=0.1,        # shock scale parameter
```

(continues on next page)

(continued from previous page)

```
                grid_max=4,
                grid_size=120,
                shock_size=250,
                seed=1234):

        self.u, self.f, self.β, self.μ, self.s = u, f, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-4, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))

    def state_action_value(self, c, y, v_array):
        """
        Right hand side of the Bellman equation.
        """

        u, f, β, shocks = self.u, self.f, self.β, self.shocks

        v = interp1d(self.grid, v_array)

        return u(c) + β * np.mean(v(f(y - c) * shocks))
```

In the second last line we are using linear interpolation.

In the last line, the expectation in (11) is computed via Monte Carlo, using the approximation

$$\int v(f(y-c)z)\phi(dz) \approx \frac{1}{n}\sum_{i=1}^{n} v(f(y-c)\xi_i)$$

where $\{\xi_i\}_{i=1}^n$ are IID draws from ϕ.

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator — see, e.g., [PalS13].)

35.3.3 The Bellman Operator

The next function implements the Bellman operator.

(We could have added it as a method to the `OptimalGrowthModel` class, but we prefer small classes rather than monolithic ones for this kind of numerical work.)

```
def T(v, og):
    """
    The Bellman operator.  Updates the guess of the value function
    and also computes a v-greedy policy.

      * og is an instance of OptimalGrowthModel
      * v is an array representing a guess of the value function

    """
    v_new = np.empty_like(v)
```

(continues on next page)

(continued from previous page)

```
    v_greedy = np.empty_like(v)

    for i in range(len(grid)):
        y = grid[i]

        # Maximize RHS of Bellman equation at state y
        c_star, v_max = maximize(og.state_action_value, 1e-10, y, (y, v))
        v_new[i] = v_max
        v_greedy[i] = c_star

    return v_greedy, v_new
```

35.3.4 An Example

Let's suppose now that

$$f(k) = k^\alpha \quad \text{and} \quad u(c) = \ln c$$

For this particular problem, an exact analytical solution is available (see [LS18], section 3.1.2), with

$$v^*(y) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha \ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln y \tag{12}$$

and optimal consumption policy

$$\sigma^*(y) = (1-\alpha\beta)y$$

It is valuable to have these closed-form solutions because it lets us check whether our code works for this particular case.

In Python, the functions above can be expressed as:

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

Next let's create an instance of the model with the above primitives and assign it to the variable `og`.

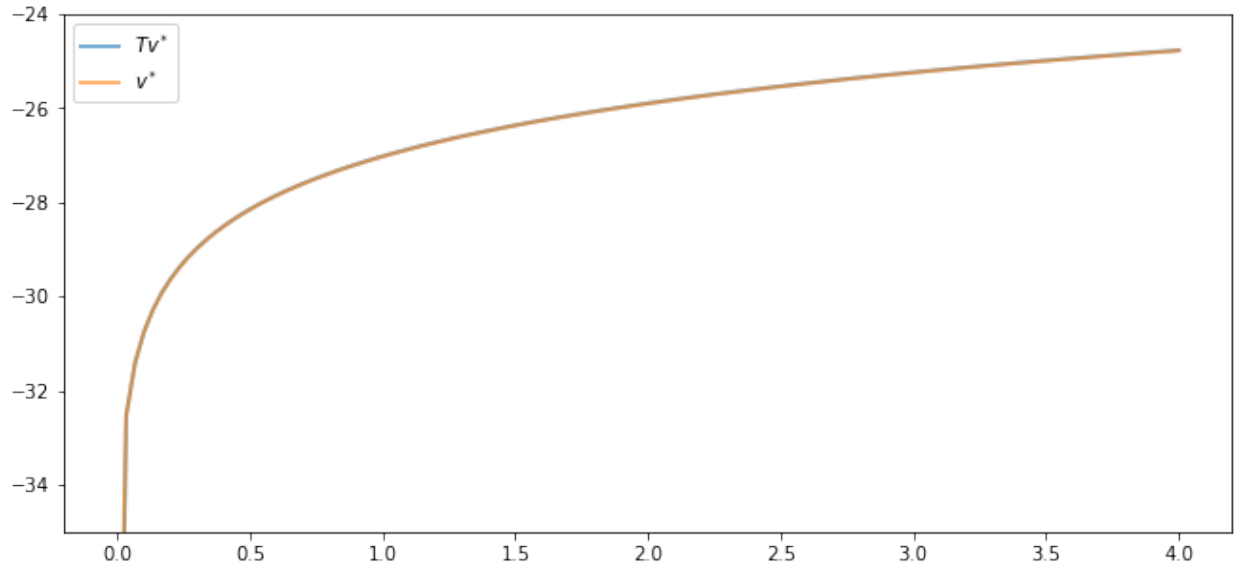
```
α = 0.4
def fcd(k):
    return k**α

og = OptimalGrowthModel(u=np.log, f=fcd)
```

Now let's see what happens when we apply our Bellman operator to the exact solution v^* in this case.

In theory, since v^* is a fixed point, the resulting function should again be v^*.

In practice, we expect some small numerical error.

```
grid = og.grid

v_init = v_star(grid, α, og.β, og.μ)     # Start at the solution
v_greedy, v = T(v_init, og)              # Apply T once

fig, ax = plt.subplots()
ax.set_ylim(-35, -24)
ax.plot(grid, v, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(grid, v_init, lw=2, alpha=0.6, label='$v^*$')
ax.legend()
plt.show()
```



The two functions are essentially indistinguishable, so we are off to a good start.

Now let's have a look at iterating with the Bellman operator, starting from an arbitrary initial condition.

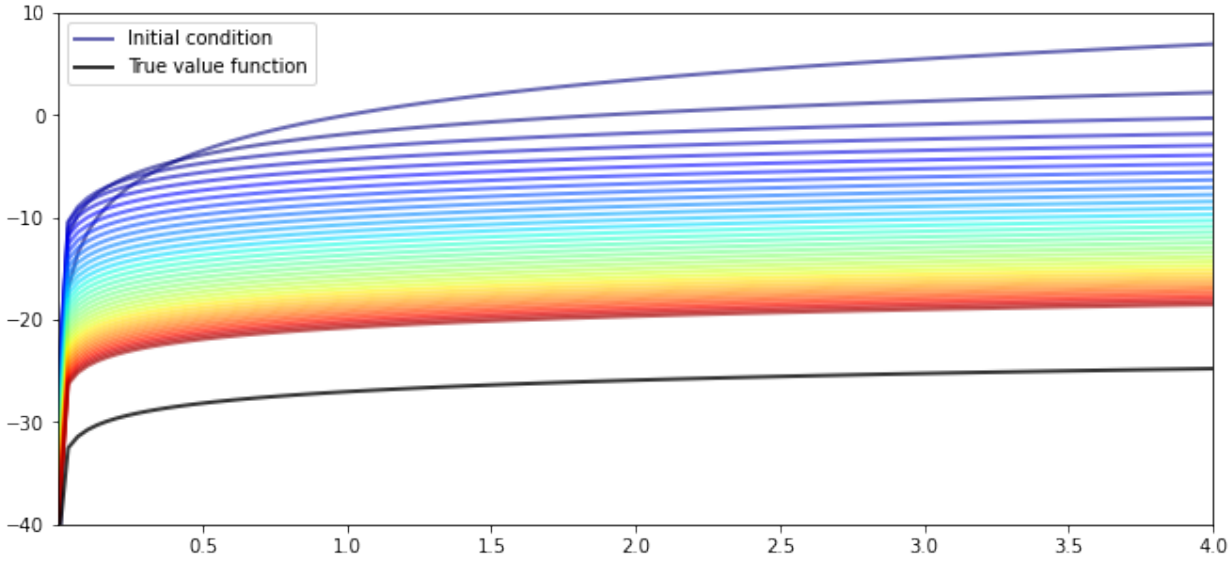
The initial condition we'll start with is, somewhat arbitrarily, $v(y) = 5 \ln(y)$.

```
v = 5 * np.log(grid)  # An initial condition
n = 35

fig, ax = plt.subplots()

ax.plot(grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial condition')

for i in range(n):
    v_greedy, v = T(v, og)  # Apply the Bellman operator
    ax.plot(grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

ax.plot(grid, v_star(grid, α, og.β, og.μ), 'k-', lw=2,
        alpha=0.8, label='True value function')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
ax.set(ylim=(-40, 10), xlim=(np.min(grid), np.max(grid)))
plt.show()
```



The figure shows

1. the first 36 functions generated by the fitted value function iteration algorithm, with hotter colors given to higher iterates
2. the true value function v^* drawn in black

The sequence of iterates converges towards v^*.

We are clearly getting closer.

35.3.5 Iterating to Convergence

We can write a function that iterates until the difference is below a particular tolerance level.

```
def solve_model(og,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):
    """
    Solve model by iterating with the Bellman operator.

    """

    # Set up loop
    v = og.u(og.grid)  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_greedy, v_new = T(v, og)
```

(continues on next page)

(continued from previous page)

```
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return v_greedy, v_new
```

Let's use this function to compute an approximate solution at the defaults.

```
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.40975776844490497.
```

```
Error at iteration 50 is 0.1476753540823772.
```

```
Error at iteration 75 is 0.05322171277213883.
```

```
Error at iteration 100 is 0.019180930548646558.
```

```
Error at iteration 125 is 0.006912744396029069.
```

```
Error at iteration 150 is 0.002491330384817303.
```

```
Error at iteration 175 is 0.000897867291303811.
```

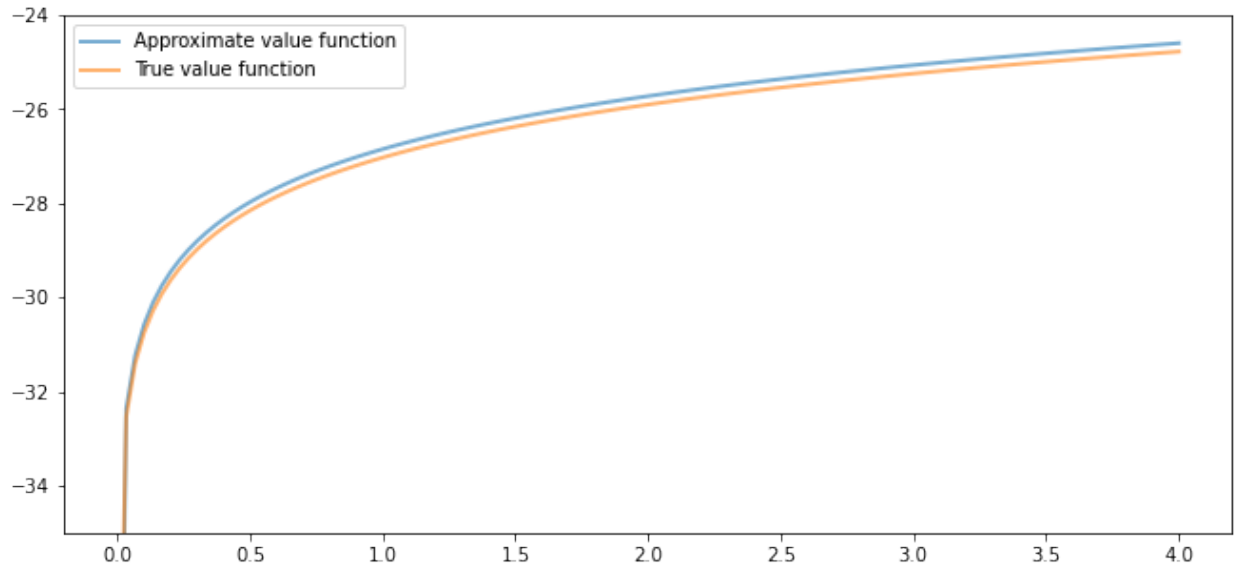
```
Error at iteration 200 is 0.00032358842396718046.
```

```
Error at iteration 225 is 0.00011662020561331587.
```

```
Converged in 229 iterations.
```

Now we check our result by plotting it against the true value:

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

ax.plot(grid, v_solution, lw=2, alpha=0.6,
        label='Approximate value function')

ax.plot(grid, v_star(grid, α, og.β, og.μ), lw=2,
        alpha=0.6, label='True value function')

ax.legend()
ax.set_ylim(-35, -24)
plt.show()
```



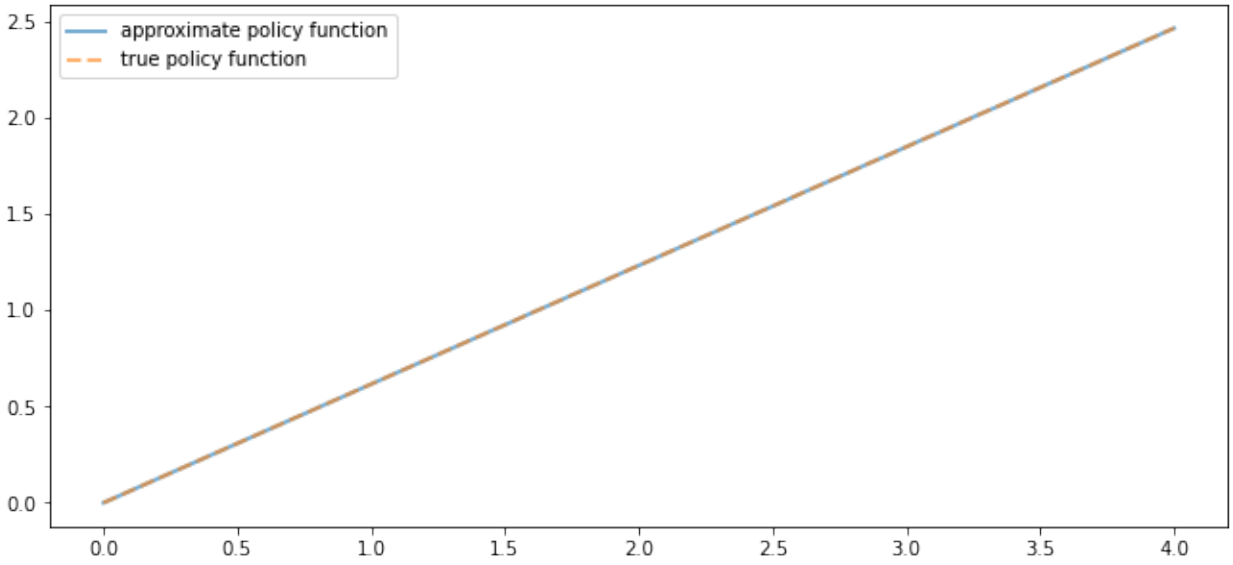
The figure shows that we are pretty much on the money.

35.3.6 The Policy Function

The policy `v_greedy` computed above corresponds to an approximate optimal policy.

The next figure compares it to the exact solution, which, as mentioned above, is $\sigma(y) = (1 - \alpha\beta)y$

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

ax.plot(grid, v_greedy, lw=2,
        alpha=0.6, label='approximate policy function')

ax.plot(grid, σ_star(grid, α, og.β), '--',
        lw=2, alpha=0.6, label='true policy function')

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



The figure shows that we've done a good job in this instance of approximating the true policy.

35.4 Exercises

35.4.1 Exercise 1

A common choice for utility function in this kind of work is the CRRA specification

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

Maintaining the other defaults, including the Cobb-Douglas production function, solve the optimal growth model with this utility specification.

Setting $\gamma = 1.5$, compute and plot an estimate of the optimal policy.

Time how long this function takes to run, so you can compare it to faster code developed in the *next lecture*.

35.4.2 Exercise 2

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the model specification in the previous exercise.

(As before, we will compare this number with that for the faster code developed in the *next lecture*.)

35.5 Solutions

35.5.1 Exercise 1

Here we set up the model.

```
γ = 1.5    # Preference parameter

def u_crra(c):
    return (c**(1 - γ) - 1) / (1 - γ)

og = OptimalGrowthModel(u=u_crra, f=fcd)
```

Now let's run it, with a timer.

```
%%time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.5528151810417512.
```

```
Error at iteration 50 is 0.19923228425590978.
```

```
Error at iteration 75 is 0.07180266113800826.
```

```
Error at iteration 100 is 0.025877443335843964.
```

```
Error at iteration 125 is 0.009326145618970827.
```

```
Error at iteration 150 is 0.003361112262005861.
```

```
Error at iteration 175 is 0.0012113338243295857.
```

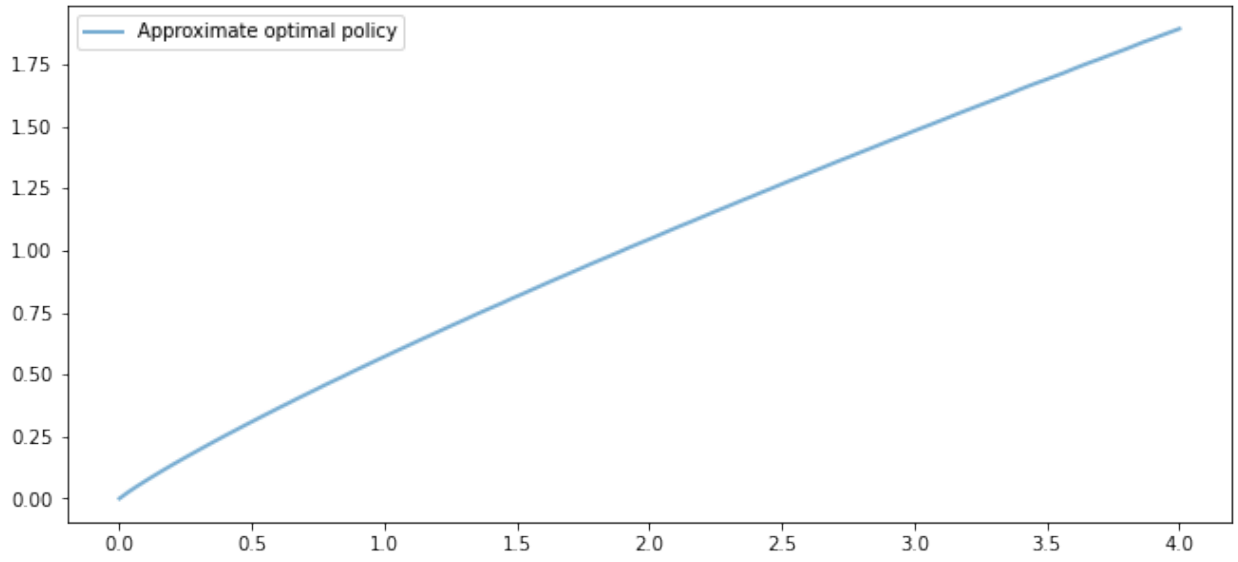
```
Error at iteration 200 is 0.0004365607333056687.
```

```
Error at iteration 225 is 0.00015733505506432266.
```

```
Converged in 237 iterations.
CPU times: user 42.5 s, sys: 12 ms, total: 42.5 s
Wall time: 42.5 s
```

Let's plot the policy function just to see what it looks like:

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

ax.plot(grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate optimal policy')

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



35.5.2 Exercise 2

Let's set up:

```
og = OptimalGrowthModel(u=u_crra, f=fcd)
v = og.u(og.grid)
```

Here's the timing:

```
%%time

for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new
```

```
CPU times: user 3.6 s, sys: 2 µs, total: 3.6 s
Wall time: 3.6 s
```

CHAPTER

THIRTYSIX

OPTIMAL GROWTH II: ACCELERATING THE CODE WITH NUMBA

Contents

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

```
!conda install -y quantecon
!pip install interpolation
```

36.1 Overview

Previously, we studied a stochastic optimal growth model with one representative agent.

We solved the model using dynamic programming.

In writing our code, we focused on clarity and flexibility.

These are important, but there's often a trade-off between flexibility and speed.

The reason is that, when code is less flexible, we can exploit structure more easily.

(This is true about algorithms and mathematical problems more generally: more specific problems have more structure, which, with some thought, can be exploited for better results.)

So, in this lecture, we are going to accept less flexibility while gaining speed, using just-in-time (JIT) compilation to accelerate our code.

Let's start with some imports:

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

(continues on next page)

(continued from previous page)

```
from numba import jit, njit, prange, float64, int32
from numba.experimental import jitclass
from quantecon.optimize.scalar_maximization import brent_max
```

We are using an interpolation function from interpolation.py because it helps us JIT-compile our code.

The function `brent_max` is also designed for embedding in JIT-compiled code.

These are alternatives to similar functions in SciPy (which, unfortunately, are not JIT-aware).

36.2 The Model

The model is the same as discussed in our *previous lecture* on optimal growth.

We will start with log utility:

$$u(c) = \ln(c)$$

We continue to assume that

- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

We will once again use value function iteration to solve the model.

In particular, the algorithm is unchanged, and the only difference is in the implementation itself.

As before, we will be able to compare with the true solutions

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

36.3 Computation

We will again store the primitives of the optimal growth model in a class.

But now we are going to use Numba's `@jitclass` decorator to target our class for JIT compilation.

Because we are going to use Numba to compile our class, we need to specify the data types.

You will see this as a list called `opt_growth_data` above our class.

Unlike in the *previous lecture*, we hardwire the production and utility specifications into the class.

This is where we sacrifice flexibility in order to gain more speed.

```
opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

The class includes some methods such as `u_prime` that we do not need now but will use in later lectures.

36.3.1 The Bellman Operator

We will use JIT compilation to accelerate the Bellman operator.

First, here's a function that returns the value of a particular consumption choice `c`, given state `y`, as per the Bellman equation (9).

```
@njit
def state_action_value(c, y, v_array, og):
    """
    Right hand side of the Bellman equation.

     * c is consumption
     * y is income
     * og is an instance of OptimalGrowthModel
     * v_array represents a guess of the value function on the grid

    """

    u, f, β, shocks = og.u, og.f, og.β, og.shocks

    v = lambda x: interp(og.grid, v_array, x)

    return u(c) + β * np.mean(v(f(y - c) * shocks))
```

Now we can implement the Bellman operator, which maximizes the right hand side of the Bellman equation:

```
@jit(nopython=True)
def T(v, og):
    """
    The Bellman operator.

     * og is an instance of OptimalGrowthModel
     * v is an array representing a guess of the value function

    """

    v_new = np.empty_like(v)
    v_greedy = np.empty_like(v)

    for i in range(len(og.grid)):
        y = og.grid[i]

        # Maximize RHS of Bellman equation at state y
        result = brent_max(state_action_value, 1e-10, y, args=(y, v, og))
        v_greedy[i], v_new[i] = result[0], result[1]

    return v_greedy, v_new
```

We use the `solve_model` function to perform iteration until convergence.

```
def solve_model(og,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):
    """
    Solve model by iterating with the Bellman operator.
```

(continues on next page)

(continued from previous page)

```
    """

    # Set up loop
    v = og.u(og.grid)  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_greedy, v_new = T(v, og)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return v_greedy, v_new
```

Let's compute the approximate solution at the default parameters.

First we create an instance:

```
og = OptimalGrowthModel()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
%%time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.41372668361362486.
```

```
Error at iteration 50 is 0.14767653072604503.
```

```
Error at iteration 75 is 0.053221715530341385.
```

```
Error at iteration 100 is 0.019180931418517844.
```

```
Error at iteration 125 is 0.006912744709509866.
```

```
Error at iteration 150 is 0.0024913304978220197.
```

```
Error at iteration 175 is 0.0008978673320463315.
```

```
Error at iteration 200 is 0.0003235884386789678.
```

```
Error at iteration 225 is 0.00011662021094238639.
```

(continues on next page)

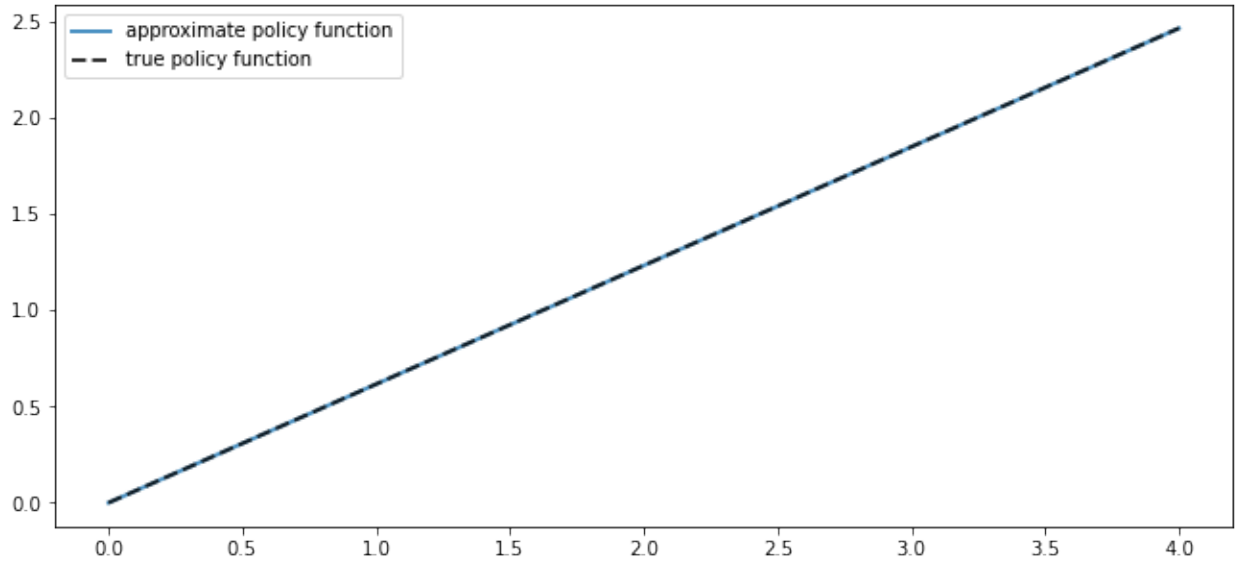
(continued from previous page)

```
Converged in 229 iterations.
CPU times: user 8.37 s, sys: 11.3 ms, total: 8.38 s
Wall time: 8.37 s
```

You will notice that this is *much* faster than our *original implementation*.

Here is a plot of the resulting policy, compared with the true policy:

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

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



Again, the fit is excellent — this is as expected since we have not changed the algorithm.

The maximal absolute deviation between the two policies is

```
np.max(np.abs(v_greedy - σ_star(og.grid, og.α, og.β)))
```

```
0.0010480495344911134
```

36.4 Exercises

36.4.1 Exercise 1

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the default parameterization.

36.4.2 Exercise 2

Modify the optimal growth model to use the CRRA utility specification.

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

Set `γ = 1.5` as the default value and maintaining other specifications.

(Note that `jitclass` currently does not support inheritance, so you will have to copy the class and change the relevant parameters and methods.)

Compute an estimate of the optimal policy, plot it and compare visually with the same plot from the *analogous exercise* in the first optimal growth lecture.

Compare execution time as well.

36.4.3 Exercise 3

In this exercise we return to the original log utility specification.

Once an optimal consumption policy σ is given, income follows

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}$$

The next figure shows a simulation of 100 elements of this sequence for three different discount factors (and hence three different policies).

In each sequence, the initial condition is $y_0 = 0.1$.

The discount factors are `discount_factors = (0.8, 0.9, 0.98)`.

We have also dialed down the shocks a bit with `s = 0.05`.

Otherwise, the parameters and primitives are the same as the log-linear model discussed earlier in the lecture.

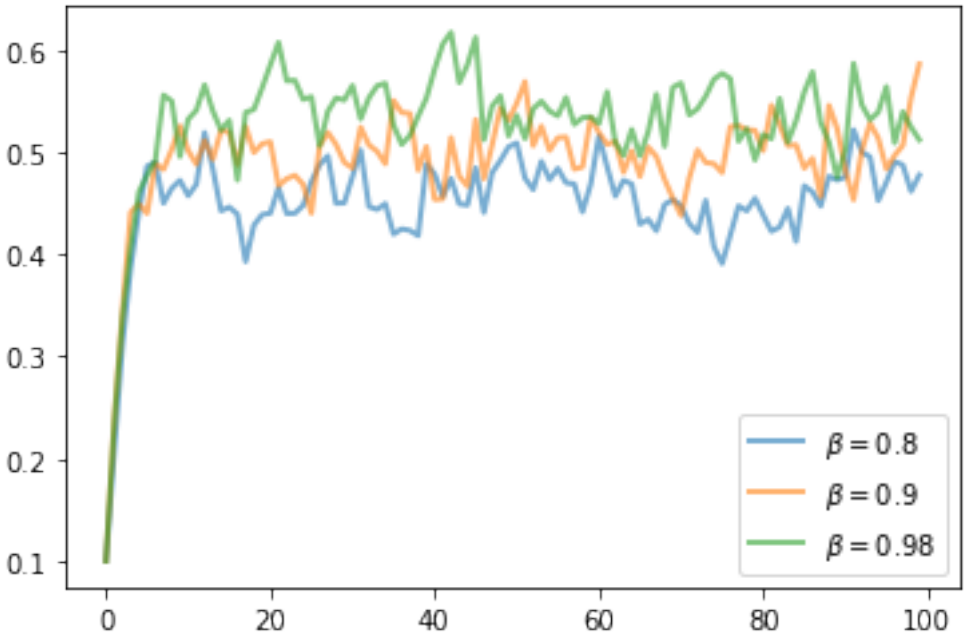
Notice that more patient agents typically have higher wealth.

Replicate the figure modulo randomness.

36.5 Solutions

36.5.1 Exercise 1

Let's set up the initial condition.




```
v = og.u(og.grid)
```

Here's the timing:

```
%%time

for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new
```

```
CPU times: user 514 ms, sys: 0 ns, total: 514 ms
Wall time: 513 ms
```

Compared with our *timing* for the non-compiled version of value function iteration, the JIT-compiled code is usually an order of magnitude faster.

36.5.2 Exercise 2

Here's our CRRA version of `OptimalGrowthModel`:

```
opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('γ', float64),          # Preference parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
```

(continues on next page)

(continued from previous page)

```
class OptimalGrowthModel_CRRA:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 γ=1.5,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.γ, self.μ, self.s = α, β, γ, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function."
        return k**self.α

    def u(self, c):
        "The utility function."
        return c**(1 - self.γ) / (1 - self.γ)

    def f_prime(self, k):
        "Derivative of f."
        return self.α * (k**(self.α - 1))

    def u_prime(self, c):
        "Derivative of u."
        return c**(-self.γ)

    def u_prime_inv(c):
        return c**(-1 / self.γ)
```

Let's create an instance:

```
og_crra = OptimalGrowthModel_CRRA()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
%%time
v_greedy, v_solution = solve_model(og_crra)
```

```
Error at iteration 25 is 1.6201897527216715.
```

```
Error at iteration 50 is 0.4591060470565935.
```

```
Error at iteration 75 is 0.1654235221617455.
```

```
Error at iteration 100 is 0.05961808343499797.
```

```
Error at iteration 125 is 0.0214861615317119.
```

```
Error at iteration 150 is 0.007743542074422294.
```

```
Error at iteration 175 is 0.0027907471405086426.
```

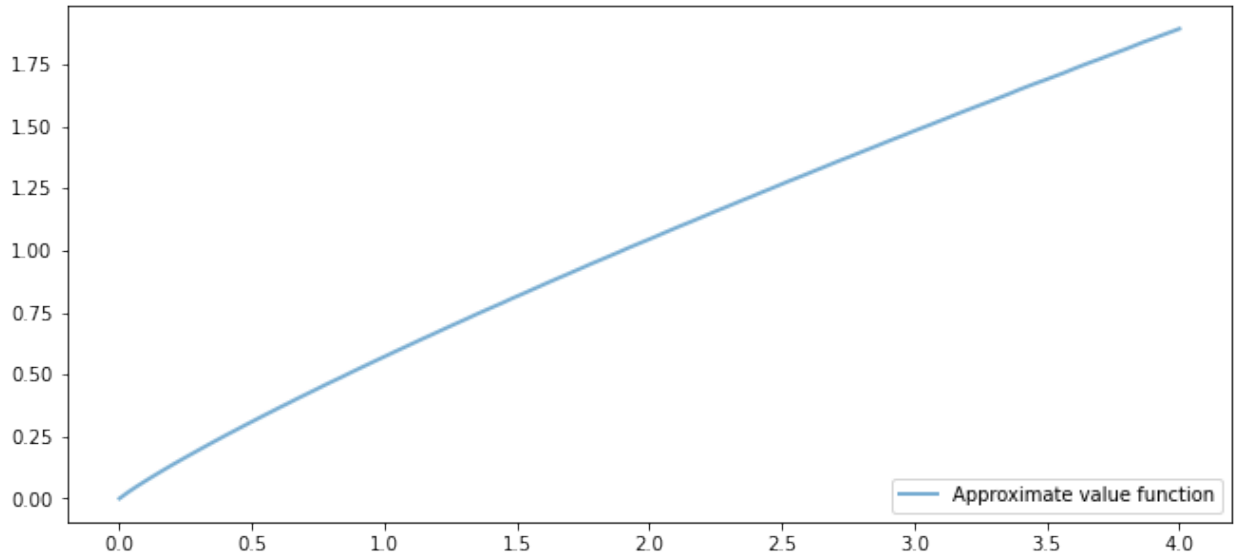
```
Error at iteration 200 is 0.001005776107220413.
```

```
Error at iteration 225 is 0.0003624784085332067.
```

```
Error at iteration 250 is 0.00013063602790452933.

Converged in 257 iterations.
CPU times: user 8.73 s, sys: 23.7 ms, total: 8.76 s
Wall time: 8.74 s
```

Here is a plot of the resulting policy:

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

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate value function')

ax.legend(loc='lower right')
plt.show()
```



This matches the solution that we obtained in our non-jitted code, *in the exercises*.

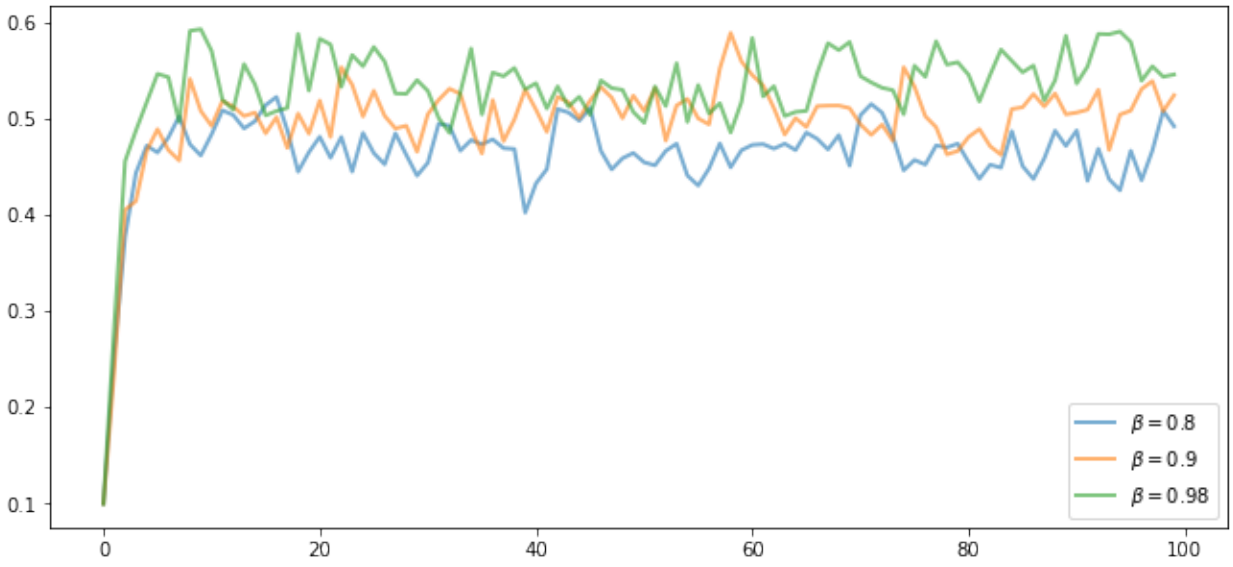
Execution time is an order of magnitude faster.

36.5.3 Exercise 3

Here's one solution:

```
def simulate_og(σ_func, og, y0=0.1, ts_length=100):
    '''
    Compute a time series given consumption policy σ.
    '''
    y = np.empty(ts_length)
    ξ = np.random.randn(ts_length-1)
    y[0] = y0
    for t in range(ts_length-1):
        y[t+1] = (y[t] - σ_func(y[t]))**og.α * np.exp(og.μ + og.s * ξ[t])
    return y
```

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

for β in (0.8, 0.9, 0.98):

    og = OptimalGrowthModel(β=β, s=0.05)

    v_greedy, v_solution = solve_model(og, verbose=False)

    # Define an optimal policy function
    σ_func = lambda x: interp(og.grid, v_greedy, x)
    y = simulate_og(σ_func, og)
    ax.plot(y, lw=2, alpha=0.6, label=rf'$\beta = {β}$')

ax.legend(loc='lower right')
plt.show()
```



CHAPTER

THIRTYSEVEN

OPTIMAL GROWTH III: TIME ITERATION

Contents

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

```
!conda install -y quantecon
!pip install interpolation
```

37.1 Overview

In this lecture, we'll continue our *earlier* study of the stochastic optimal growth model.

In that lecture, we solved the associated dynamic programming problem using value function iteration.

The beauty of this technique is its broad applicability.

With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand.

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives.

We'll use this structure to obtain an Euler equation based method.

This will be an extension of the time iteration method considered in our elementary lecture on *cake eating*.

In a *subsequent lecture*, we'll see that time iteration can be further adjusted to obtain even more efficiency.

Let's start with some imports:

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

(continues on next page)

(continued from previous page)

```
import numpy as np
import quantecon as qe
from interpolation import interp
from quantecon.optimize import brentq
from numba import njit, float64
from numba.experimental import jitclass
```

37.2 The Euler Equation

Our first step is to derive the Euler equation, which is a generalization of the Euler equation we obtained in the *lecture on cake eating*.

We take the model set out in *the stochastic growth model lecture* and add the following assumptions:

1. u and f are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
4. $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$

The last two conditions are usually called **Inada conditions**.

Recall the Bellman equation

$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\} \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{1}$$

Let the optimal consumption policy be denoted by σ^*.

We know that σ^* is a v^*-greedy policy so that $\sigma^*(y)$ is the maximizer in (1).

The conditions above imply that

- σ^* is the unique optimal policy for the stochastic optimal growth model
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that $0 < \sigma^*(y) < y$ for all strictly positive y, and
- the value function is strictly concave and continuously differentiable, with

$$(v^*)'(y) = u'(\sigma^*(y)) := (u' \circ \sigma^*)(y) \tag{2}$$

The last result is called the **envelope condition** due to its relationship with the envelope theorem.

To see why (2) holds, write the Bellman equation in the equivalent form

$$v^*(y) = \max_{0 \leq k \leq y} \left\{ u(y-k) + \beta \int v^*(f(k)z)\phi(dz) \right\},$$

Differentiating with respect to y, and then evaluating at the optimum yields (2).

(Section 12.1 of EDTC contains full proofs of these results, and closely related discussions can be found in many other texts.)

Differentiability of the value function and interiority of the optimal policy imply that optimal consumption satisfies the first order condition associated with (1), which is

$$u'(\sigma^*(y)) = \beta \int (v^*)'(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{3}$$

Combining (2) and the first-order condition (3) gives the **Euler equation**

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{4}$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \tag{5}$$

over interior consumption policies σ, one solution of which is the optimal policy σ^*.

Our aim is to solve the functional equation (5) and hence obtain σ^*.

37.2.1 The Coleman-Reffett Operator

Recall the Bellman operator

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y - c)z) \phi(dz) \right\} \tag{6}$$

Just as we introduced the Bellman operator to solve the Bellman equation, we will now introduce an operator over policies to help us solve the Euler equation.

This operator K will act on the set of all $\sigma \in \Sigma$ that are continuous, strictly increasing and interior.

Henceforth we denote this set of policies by $\mathcal{P}$

1. The operator K takes as its argument a $\sigma \in \mathcal{P}$ and
2. returns a new function $K\sigma$, where $K\sigma(y)$ is the $c \in (0, y)$ that solves.

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{7}$$

We call this operator the **Coleman-Reffett operator** to acknowledge the work of [Col90] and [Ref96].

In essence, $K\sigma$ is the consumption policy that the Euler equation tells you to choose today when your future consumption policy is σ.

The important thing to note about K is that, by construction, its fixed points coincide with solutions to the functional equation (5).

In particular, the optimal policy σ^* is a fixed point.

Indeed, for fixed y, the value $K\sigma^*(y)$ is the c that solves

$$u'(c) = \beta \int (u' \circ \sigma^*)(f(y - c)z) f'(y - c) z \phi(dz)$$

In view of the Euler equation, this is exactly $\sigma^*(y)$.

37.2.2 Is the Coleman-Reffett Operator Well Defined?

In particular, is there always a unique $c \in (0, y)$ that solves (7)?

The answer is yes, under our assumptions.

For any $\sigma \in \mathcal{P}$, the right side of (7)

- is continuous and strictly increasing in c on $(0, y)$

- diverges to $+\infty$ as $c \uparrow y$

The left side of (7)

- is continuous and strictly decreasing in c on $(0, y)$
- diverges to $+\infty$ as $c \downarrow 0$

Sketching these curves and using the information above will convince you that they cross exactly once as c ranges over $(0, y)$.

With a bit more analysis, one can show in addition that $K\sigma \in \mathcal{P}$ whenever $\sigma \in \mathcal{P}$.

37.2.3 Comparison with VFI (Theory)

It is possible to prove that there is a tight relationship between iterates of K and iterates of the Bellman operator.

Mathematically, the two operators are *topologically conjugate*.

Loosely speaking, this means that if iterates of one operator converge then so do iterates of the other, and vice versa.

Moreover, there is a sense in which they converge at the same rate, at least in theory.

However, it turns out that the operator K is more stable numerically and hence more efficient in the applications we consider.

Examples are given below.

37.3 Implementation

As in our *previous study*, we continue to assume that

- $u(c) = \ln c$
- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

This will allow us to compare our results to the analytical solutions

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

As discussed above, our plan is to solve the model using time iteration, which means iterating with the operator K.

For this we need access to the functions u' and f, f'.

These are available in a class called `OptimalGrowthModel` that we constructed in an *earlier lecture*.

```
opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

Now we implement a method called `euler_diff`, which returns

$$u'(c) - \beta \int (u' \circ \sigma)(f(y-c)z) f'(y-c) z \phi(dz) \tag{8}$$

```
@njit
def euler_diff(c, σ, y, og):
    """
    Set up a function such that the root with respect to c,
    given y and σ, is equal to Kσ(y).

    """

    β, shocks, grid = og.β, og.shocks, og.grid
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime

    # First turn σ into a function via interpolation
    σ_func = lambda x: interp(grid, σ, x)

    # Now set up the function we need to find the root of.
    vals = u_prime(σ_func(f(y - c) * shocks)) * f_prime(y - c) * shocks
    return u_prime(c) - β * np.mean(vals)
```

The function `euler_diff` evaluates integrals by Monte Carlo and approximates functions using linear interpolation.

We will use a root-finding algorithm to solve (8) for c given state y and σ, the current guess of the policy.

Here's the operator K, that implements the root-finding step.

```
@njit
def K(σ, og):
    """
    The Coleman-Reffett operator

     Here og is an instance of OptimalGrowthModel.
    """

    β = og.β
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime
    grid, shocks = og.grid, og.shocks

    σ_new = np.empty_like(σ)
    for i, y in enumerate(grid):
        # Solve for optimal c at y
        c_star = brentq(euler_diff, 1e-10, y-1e-10, args=(σ, y, og))[0]
        σ_new[i] = c_star

    return σ_new
```

37.3.1 Testing

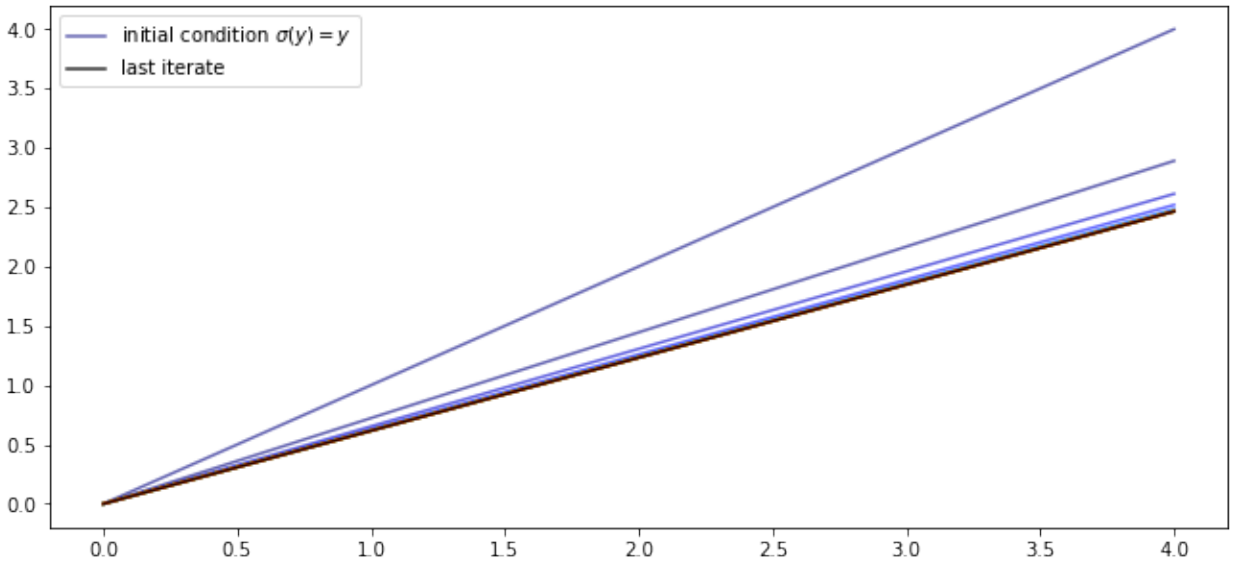
Let's generate an instance and plot some iterates of K, starting from $\sigma(y) = y$.

```
og = OptimalGrowthModel()
grid = og.grid

n = 15
σ = grid.copy()  # Set initial condition

fig, ax = plt.subplots()
lb = 'initial condition $\sigma(y) = y$'
```

(continues on next page)

(continued from previous page)

```
ax.plot(grid, σ, color=plt.cm.jet(0), alpha=0.6, label=lb)

for i in range(n):
    σ = K(σ, og)
    ax.plot(grid, σ, color=plt.cm.jet(i / n), alpha=0.6)

# Update one more time and plot the last iterate in black
σ = K(σ, og)
ax.plot(grid, σ, color='k', alpha=0.8, label='last iterate')

ax.legend()

plt.show()
```



We see that the iteration process converges quickly to a limit that resembles the solution we obtained in *the previous lecture*.

Here is a function called `solve_model_time_iter` that takes an instance of `OptimalGrowthModel` and returns an approximation to the optimal policy, using time iteration.

```
def solve_model_time_iter(model,     # Class with model information
                          σ,         # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

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

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
```

(continues on next page)

(continued from previous page)

```
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

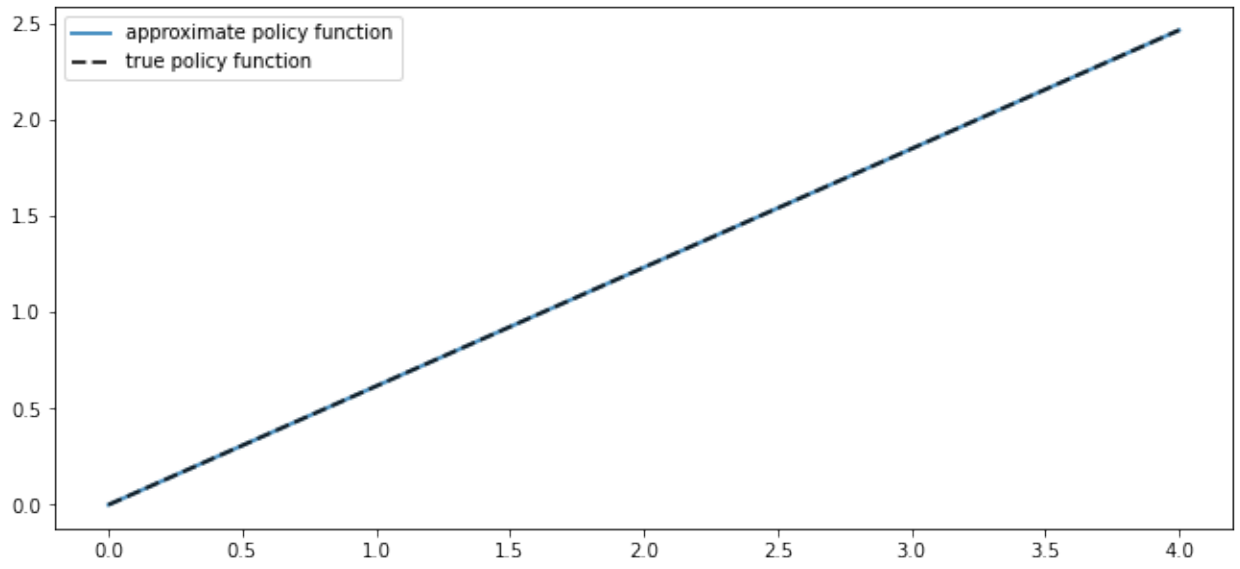
Let's call it:

```
σ_init = np.copy(og.grid)
σ = solve_model_time_iter(og, σ_init)
```

```
Converged in 11 iterations.
```

Here is a plot of the resulting policy, compared with the true policy:

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

ax.plot(og.grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



Again, the fit is excellent.

The maximal absolute deviation between the two policies is

```
np.max(np.abs(σ - σ_star(og.grid, og.α, og.β)))
```

```
2.5329106213334285e-05
```

How long does it take to converge?

```
%%timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

```
196 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)
```

Convergence is very fast, even compared to our *JIT-compiled value function iteration*.

Overall, we find that time iteration provides a very high degree of efficiency and accuracy, at least for this model.

37.4 Exercises

37.4.1 Exercise 1

Solve the model with CRRA utility

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

Set `γ = 1.5`.

Compute and plot the optimal policy.

37.5 Solutions

37.5.1 Exercise 1

We use the class `OptimalGrowthModel_CRRA` from our *VFI lecture*.

```
opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('γ', float64),          # Preference parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel_CRRA:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 γ=1.5,
                 grid_max=4,
```

(continues on next page)

(continued from previous page)

```
                grid_size=120,
                shock_size=250,
                seed=1234):

        self.α, self.β, self.γ, self.μ, self.s = α, β, γ, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function."
        return k**self.α

    def u(self, c):
        "The utility function."
        return c**(1 - self.γ) / (1 - self.γ)

    def f_prime(self, k):
        "Derivative of f."
        return self.α * (k**(self.α - 1))

    def u_prime(self, c):
        "Derivative of u."
        return c**(-self.γ)

    def u_prime_inv(c):
        return c**(-1 / self.γ)
```

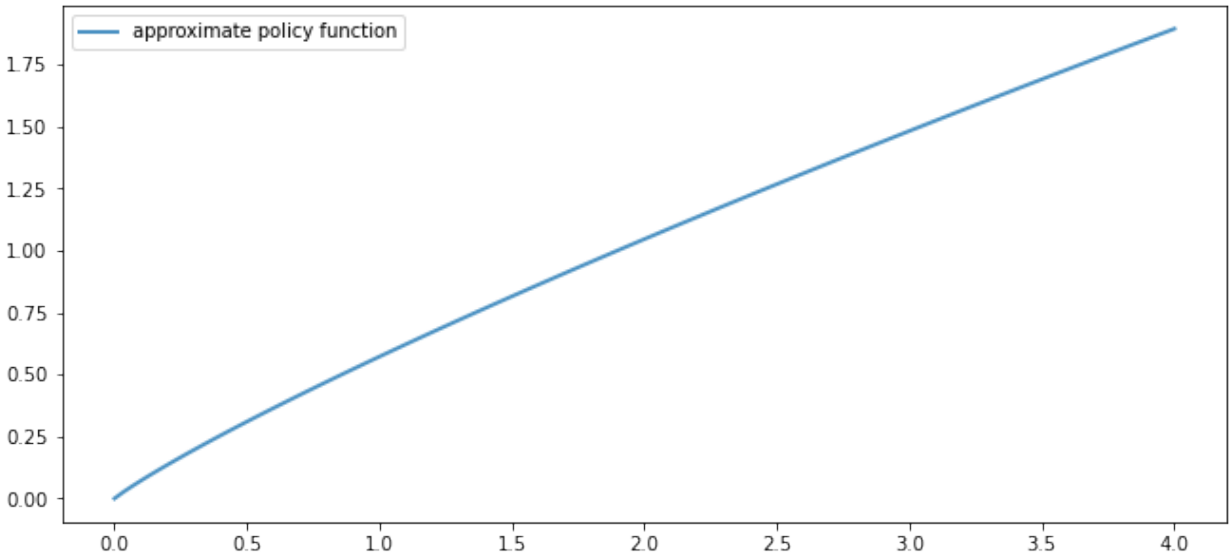
Let's create an instance:

```
og_crra = OptimalGrowthModel_CRRA()
```

Now we solve and plot the policy:

```
%%time
σ = solve_model_time_iter(og_crra, σ_init)


fig, ax = plt.subplots()

ax.plot(og.grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

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

```
Converged in 13 iterations.
```




```
CPU times: user 2.26 s, sys: 12.3 ms, total: 2.28 s
Wall time: 2.27 s
```

CHAPTER

THIRTYEIGHT

OPTIMAL GROWTH IV: THE ENDOGENOUS GRID METHOD

Contents

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

```
!conda install -y quantecon
!pip install interpolation
```

38.1 Overview

Previously, we solved the stochastic optimal growth model using

1. *value function iteration*
2. *Euler equation based time iteration*

We found time iteration to be significantly more accurate and efficient.

In this lecture, we'll look at a clever twist on time iteration called the **endogenous grid method** (EGM).

EGM is a numerical method for implementing policy iteration invented by Chris Carroll.

The original reference is [Car06].

Let's start with some standard imports:

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

38.2 Key Idea

Let's start by reminding ourselves of the theory and then see how the numerics fit in.

38.2.1 Theory

Take the model set out in *the time iteration lecture*, following the same terminology and notation.

The Euler equation is

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{1}$$

As we saw, the Coleman-Reffett operator is a nonlinear operator K engineered so that σ^* is a fixed point of K.

It takes as its argument a continuous strictly increasing consumption policy $\sigma \in \Sigma$.

It returns a new function $K\sigma$, where $(K\sigma)(y)$ is the $c \in (0, \infty)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{2}$$

38.2.2 Exogenous Grid

As discussed in *the lecture on time iteration*, to implement the method on a computer, we need a numerical approximation.

In particular, we represent a policy function by a set of values on a finite grid.

The function itself is reconstructed from this representation when necessary, using interpolation or some other method.

Previously, to obtain a finite representation of an updated consumption policy, we

- fixed a grid of income points $\{y_i\}$
- calculated the consumption value c_i corresponding to each y_i using (2) and a root-finding routine

Each c_i is then interpreted as the value of the function $K\sigma$ at y_i.

Thus, with the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation.

Iteration then continues…

38.2.3 Endogenous Grid

The method discussed above requires a root-finding routine to find the c_i corresponding to a given income value y_i.

Root-finding is costly because it typically involves a significant number of function evaluations.

As pointed out by Carroll [Car06], we can avoid this if y_i is chosen endogenously.

The only assumption required is that u' is invertible on $(0, \infty)$.

Let $(u')^{-1}$ be the inverse function of u'.

The idea is this:

- First, we fix an *exogenous* grid $\{k_i\}$ for capital ($k = y - c$).
- Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ \sigma)(f(k_i)z) \, f'(k_i) \, z \, \phi(dz) \right\} \tag{3}$$

- Finally, for each c_i we set $y_i = c_i + k_i$.

It is clear that each (y_i, c_i) pair constructed in this manner satisfies (2).

With the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation as before.

The name EGM comes from the fact that the grid $\{y_i\}$ is determined **endogenously**.

38.3 Implementation

As *before*, we will start with a simple setting where

- $u(c) = \ln c$,
- production is Cobb-Douglas, and
- the shocks are lognormal.

This will allow us to make comparisons with the analytical solutions

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

We reuse the `OptimalGrowthModel` class

```
opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
```

(continues on next page)

(continued from previous page)

```
                grid_size=120,
                shock_size=250,
                seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

38.3.1 The Operator

Here's an implementation of K using EGM as described above.

```
@njit
def K(σ_array, og):
    """
    The Coleman-Reffett operator using EGM

    """

    # Simplify names
    f, β = og.f, og.β
    f_prime, u_prime = og.f_prime, og.u_prime
    u_prime_inv = og.u_prime_inv
    grid, shocks = og.grid, og.shocks

    # Determine endogenous grid
    y = grid + σ_array  # y_i = k_i + c_i
```

(continues on next page)

(continued from previous page)

```
    # Linear interpolation of policy using endogenous grid
    σ = lambda x: interp(y, σ_array, x)

    # Allocate memory for new consumption array
    c = np.empty_like(grid)

    # Solve for updated consumption value
    for i, k in enumerate(grid):
        vals = u_prime(σ(f(k) * shocks)) * f_prime(k) * shocks
        c[i] = u_prime_inv(β * np.mean(vals))

    return c
```

Note the lack of any root-finding algorithm.

38.3.2 Testing

First we create an instance.

```
og = OptimalGrowthModel()
grid = og.grid
```

Here's our solver routine:

```
def solve_model_time_iter(model,     # Class with model information
                          σ,         # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

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

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

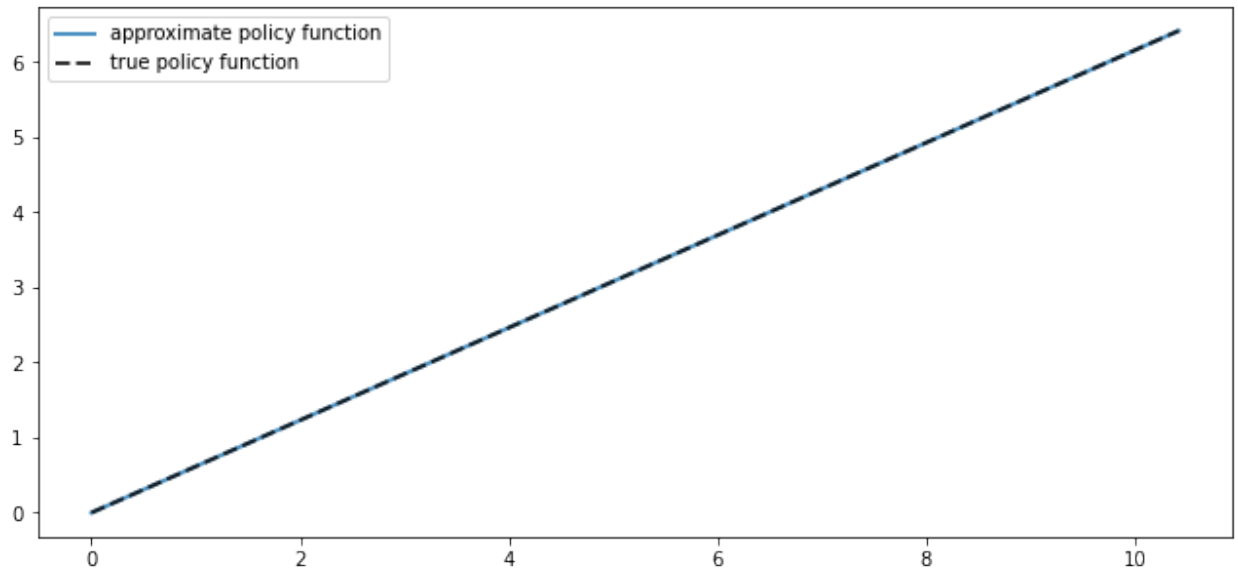
Let's call it:

```
σ_init = np.copy(grid)
σ = solve_model_time_iter(og, σ_init)
```

```
Converged in 12 iterations.
```

Here is a plot of the resulting policy, compared with the true policy:

```
y = grid + σ  # y_i = k_i + c_i

fig, ax = plt.subplots()

ax.plot(y, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(y, σ_star(y, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



The maximal absolute deviation between the two policies is

```
np.max(np.abs(σ - σ_star(y, og.α, og.β)))
```

```
1.530274914252061e-05
```

How long does it take to converge?

```
%%timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

```
30 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)
```

```

```

Relative to time iteration, which as already found to be highly efficient, EGM has managed to shave off still more run time without compromising accuracy.

This is due to the lack of a numerical root-finding step.

We can now solve the optimal growth model at given parameters extremely fast.

CHAPTER

THIRTYNINE

THE INCOME FLUCTUATION PROBLEM I: BASIC MODEL

Contents

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

```
!conda install -y quantecon
!pip install interpolation
```

39.1 Overview

In this lecture, we study an optimal savings problem for an infinitely lived consumer—the "common ancestor" described in [LS18], section 1.3.

This is an essential sub-problem for many representative macroeconomic models

- [Aiy94]
- [Hug93]
- etc.

It is related to the decision problem in the *stochastic optimal growth model* and yet differs in important ways.

For example, the choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.

Moreover, in this and the following lectures, we will inject more realistic features such as correlated shocks.

To solve the model we will use Euler equation based time iteration, which proved to be *fast and accurate* in our investigation of the *stochastic optimal growth model*.

Time iteration is globally convergent under mild assumptions, even when utility is unbounded (both above and below).

We'll need the following imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon.optimize import brent_max, brentq
from interpolation import interp
from numba import njit, float64
from numba.experimental import jitclass
from quantecon import MarkovChain
```

39.1.1 References

Our presentation is a simplified version of [MST20].

Other references include [Dea91], [DH10], [Kuh13], [Rab02], [Rei09] and [SE77].

39.2 The Optimal Savings Problem

Let's write down the model and then discuss how to solve it.

39.2.1 Set-Up

Consider a household that chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

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

subject to

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

Here

- $\beta \in (0, 1)$ is the discount factor
- a_t is asset holdings at time t, with borrowing constraint $a_t \geq 0$
- c_t is consumption
- Y_t is non-capital income (wages, unemployment compensation, etc.)
- $R := 1 + r$, where $r > 0$ is the interest rate on savings

The timing here is as follows:

1. At the start of period t, the household chooses consumption c_t.
2. Labor is supplied by the household throughout the period and labor income Y_{t+1} is received at the end of period t.
3. Financial income $R(a_t - c_t)$ is received at the end of period t.
4. Time shifts to $t + 1$ and the process repeats.

Non-capital income Y_t is given by $Y_t = y(Z_t)$, where $\{Z_t\}$ is an exogeneous state process.

As is common in the literature, we take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix P.

We further assume that

1. $\beta R < 1$
2. u is smooth, strictly increasing and strictly concave with $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$

The asset space is $\mathbb{R}_+$ and the state is the pair $(a, z) \in \mathsf{S} := \mathbb{R}_+ \times \mathsf{Z}$.

A *feasible consumption path* from $(a, z) \in \mathsf{S}$ is a consumption sequence $\{c_t\}$ such that $\{c_t\}$ and its induced asset path $\{a_t\}$ satisfy

1. $(a_0, z_0) = (a, z)$
2. the feasibility constraints in (1), and
3. measurability, which means that c_t is a function of random outcomes up to date t but not after.

The meaning of the third point is just that consumption at time t cannot be a function of outcomes are yet to be observed.

In fact, for this problem, consumption can be chosen optimally by taking it to be contingent only on the current state.

Optimality is defined below.

39.2.2 Value Function and Euler Equation

The *value function* $V \colon \mathsf{S} \to \mathbb{R}$ is defined by

$$V(a, z) := \max \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{2}$$

where the maximization is overall feasible consumption paths from (a, z).

An *optimal consumption path* from (a, z) is a feasible consumption path from (a, z) that attains the supremum in (2).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{3}$$

and

$$c_t < a_t \implies u'(c_t) = \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{4}$$

When $c_t = a_t$ we obviously have $u'(c_t) = u'(a_t)$,

When c_t hits the upper bound a_t, the strict inequality $u'(c_t) > \beta R \, \mathbb{E}_t u'(c_{t+1})$ can occur because c_t cannot increase sufficiently to attain equality.

(The lower boundary case $c_t = 0$ never arises at the optimum because $u'(0) = \infty$.)

With some thought, one can show that (3) and (4) are equivalent to

$$u'(c_t) = \max \left\{ \beta R \, \mathbb{E}_t u'(c_{t+1}) \, , \; u'(a_t) \right\} \tag{5}$$

39.2.3 Optimality Results

As shown in [MST20],

1. For each $(a, z) \in \mathsf{S}$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equality (5) and the transversality condition

$$\lim_{t \to \infty} \beta^t \, \mathbb{E} \left[u'(c_t) a_{t+1} \right] = 0 \tag{6}$$

Moreover, there exists an *optimal consumption function* $\sigma^* \colon \mathsf{S} \to \mathbb{R}_+$ such that the path from (a, z) generated by

$$(a_0, z_0) = (a, z), \quad c_t = \sigma^*(a_t, Z_t) \quad \text{and} \quad a_{t+1} = R(a_t - c_t) + Y_{t+1}$$

satisfies both (5) and (6), and hence is the unique optimal path from (a, z).

Thus, to solve the optimization problem, we need to compute the policy σ^*.

39.3 Computation

There are two standard ways to solve for σ^*

1. time iteration using the Euler equality and
2. value function iteration.

Our investigation of the cake eating problem and stochastic optimal growth model suggests that time iteration will be faster and more accurate.

This is the approach that we apply below.

39.3.1 Time Iteration

We can rewrite (5) to make it a statement about functions rather than random variables.

In particular, consider the functional equation

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta R \, \mathbb{E}_z (u' \circ \sigma)[R(a - \sigma(a, z)) + \hat{Y}, \, \hat{Z}] \, , \; u'(a) \right\} \tag{7}$$

where

- $(u' \circ \sigma)(s) := u'(\sigma(s))$.
- $\mathbb{E}_z$ conditions on current state z and $\hat{X}$ indicates next period value of random variable X and
- σ is the unknown function.

We need a suitable class of candidate solutions for the optimal consumption policy.

The right way to pick such a class is to consider what properties the solution is likely to have, in order to restrict the search space and ensure that iteration is well behaved.

To this end, let $\mathcal{C}$ be the space of continuous functions $\sigma \colon \mathbf{S} \to \mathbb{R}$ such that σ is increasing in the first argument, $0 < \sigma(a, z) \leq a$ for all $(a, z) \in \mathbf{S}$, and

$$\sup_{(a,z) \in \mathbf{S}} |(u' \circ \sigma)(a, z) - u'(a)| < \infty \tag{8}$$

This will be our candidate class.

In addition, let $K \colon \mathcal{C} \to \mathcal{C}$ be defined as follows.

For given $\sigma \in \mathcal{C}$, the value $K\sigma(a, z)$ is the unique $c \in [0, a]$ that solves

$$u'(c) = \max \left\{ \beta R \, \mathbb{E}_z (u' \circ \sigma) \left[R(a - c) + \hat{Y}, \, \hat{Z} \right], \; u'(a) \right\} \tag{9}$$

We refer to K as the Coleman–Reffett operator.

The operator K is constructed so that fixed points of K coincide with solutions to the functional equation (7).

It is shown in [MST20] that the unique optimal policy can be computed by picking any $\sigma \in \mathcal{C}$ and iterating with the operator K defined in (9).

39.3.2 Some Technical Details

The proof of the last statement is somewhat technical but here is a quick summary:

It is shown in [MST20] that K is a contraction mapping on $\mathcal{C}$ under the metric

$$\rho(c, d) := \| \, u' \circ \sigma_1 - u' \circ \sigma_2 \, \| := \sup_{s \in S} | \, u'(\sigma_1(s)) - u'(\sigma_2(s)) \, | \qquad (\sigma_1, \sigma_2 \in \mathcal{C})$$

which evaluates the maximal difference in terms of marginal utility.

(The benefit of this measure of distance is that, while elements of $\mathcal{C}$ are not generally bounded, ρ is always finite under our assumptions.)

It is also shown that the metric ρ is complete on $\mathcal{C}$.

In consequence, K has a unique fixed point $\sigma^* \in \mathcal{C}$ and $K^n c \to \sigma^*$ as $n \to \infty$ for any $\sigma \in \mathcal{C}$.

By the definition of K, the fixed points of K in $\mathcal{C}$ coincide with the solutions to (7) in $\mathcal{C}$.

As a consequence, the path $\{c_t\}$ generated from $(a_0, z_0) \in S$ using policy function σ^* is the unique optimal path from $(a_0, z_0) \in S$.

39.4 Implementation

We use the CRRA utility specification

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

The exogeneous state process $\{Z_t\}$ defaults to a two-state Markov chain with state space $\{0, 1\}$ and transition matrix P.

Here we build a class called `IFP` that stores the model primitives.

```
ifp_data = [
    ('R', float64),              # Interest rate 1 + r
    ('β', float64),              # Discount factor
    ('γ', float64),              # Preference parameter
    ('P', float64[:, :]),        # Markov matrix for binary Z_t
    ('y', float64[:]),           # Income is Y_t = y[Z_t]
    ('asset_grid', float64[:])   # Grid (array)
]

@jitclass(ifp_data)
class IFP:

    def __init__(self,
```

(continues on next page)

(continued from previous page)

```
                 r=0.01,
                 β=0.96,
                 γ=1.5,
                 P=((0.6, 0.4),
                    (0.05, 0.95)),
                 y=(0.0, 2.0),
                 grid_max=16,
                 grid_size=50):

        self.R = 1 + r
        self.β, self.γ = β, γ
        self.P, self.y = np.array(P), np.array(y)
        self.asset_grid = np.linspace(0, grid_max, grid_size)

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

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

Next we provide a function to compute the difference

$$u'(c) - \max\left\{\beta R \, \mathbb{E}_z (u' \circ \sigma)\left[R(a-c) + \hat{Y}, \, \hat{Z}\right], \, u'(a)\right\} \tag{10}$$

```
@njit
def euler_diff(c, a, z, σ_vals, ifp):
    """
    The difference between the left- and right-hand side
    of the Euler Equation, given current policy σ.

        * c is the consumption choice
        * (a, z) is the state, with z in {0, 1}
        * σ_vals is a policy represented as a matrix.
        * ifp is an instance of IFP

    """

    # Simplify names
    R, P, γ, β, γ  = ifp.R, ifp.P, ifp.y, ifp.β, ifp.γ
    asset_grid, u_prime = ifp.asset_grid, ifp.u_prime
    n = len(P)

    # Convert policy into a function by linear interpolation
    def σ(a, z):
        return interp(asset_grid, σ_vals[:, z], a)

    # Calculate the expectation conditional on current z
    expect = 0.0
    for z_hat in range(n):
        expect += u_prime(σ(R * (a - c) + y[z_hat], z_hat)) * P[z, z_hat]

    return u_prime(c) - max(β * R * expect, u_prime(a))
```

Note that we use linear interpolation along the asset grid to approximate the policy function.

The next step is to obtain the root of the Euler difference.

```
@njit
def K(σ, ifp):
    """
    The operator K.

    """
    σ_new = np.empty_like(σ)
    for i, a in enumerate(ifp.asset_grid):
        for z in (0, 1):
            result = brentq(euler_diff, 1e-8, a, args=(a, z, σ, ifp))
            σ_new[i, z] = result.root

    return σ_new
```

With the operator K in hand, we can choose an initial condition and start to iterate.

The following function iterates to convergence and returns the approximate optimal policy.

```
def solve_model_time_iter(model,     # Class with model information
                          σ,         # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

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

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

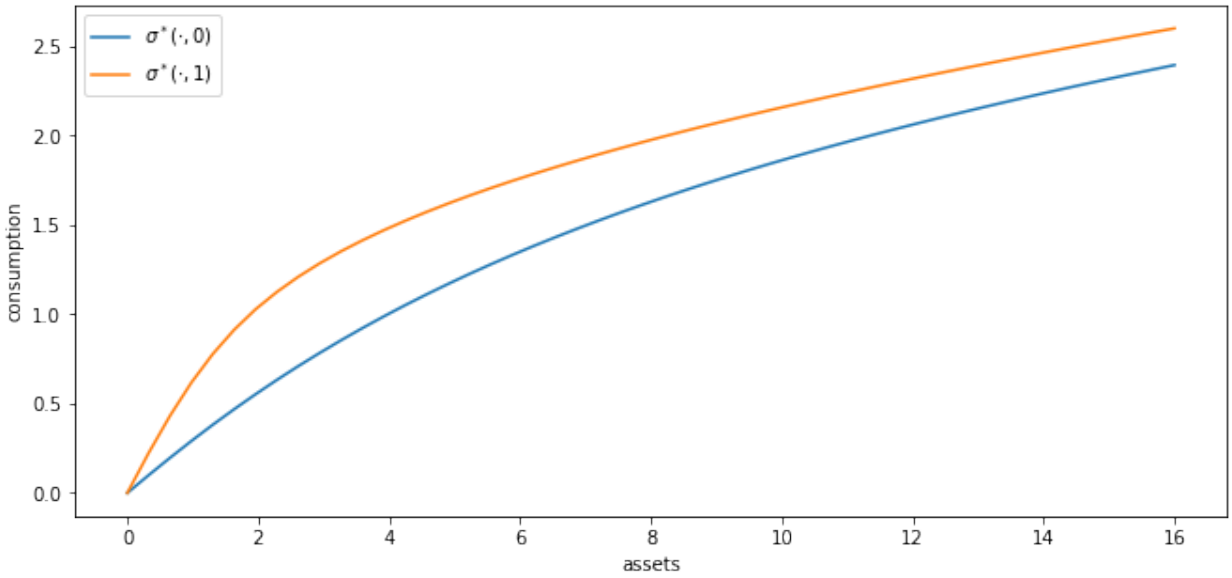
Let's carry this out using the default parameters of the `IFP` class:

```
ifp = IFP()

# Set up initial consumption policy of consuming all assets at all z
z_size = len(ifp.P)
a_grid = ifp.asset_grid
a_size = len(a_grid)
σ_init = np.repeat(a_grid.reshape(a_size, 1), z_size, axis=1)

σ_star = solve_model_time_iter(ifp, σ_init)
```

```
Error at iteration 25 is 0.011629589188246303.
Error at iteration 50 is 0.0003857183099462702.

Converged in 60 iterations.
```

Here's a plot of the resulting policy for each exogeneous state z.

```
fig, ax = plt.subplots()
for z in range(z_size):
    label = rf'$\sigma^*(\cdot, {z})$'
    ax.plot(a_grid, σ_star[:, z], label=label)
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



The following exercises walk you through several applications where policy functions are computed.

39.4.1 A Sanity Check

One way to check our results is to

- set labor income to zero in each state and
- set the gross interest rate R to unity.

In this case, our income fluctuation problem is just a cake eating problem.

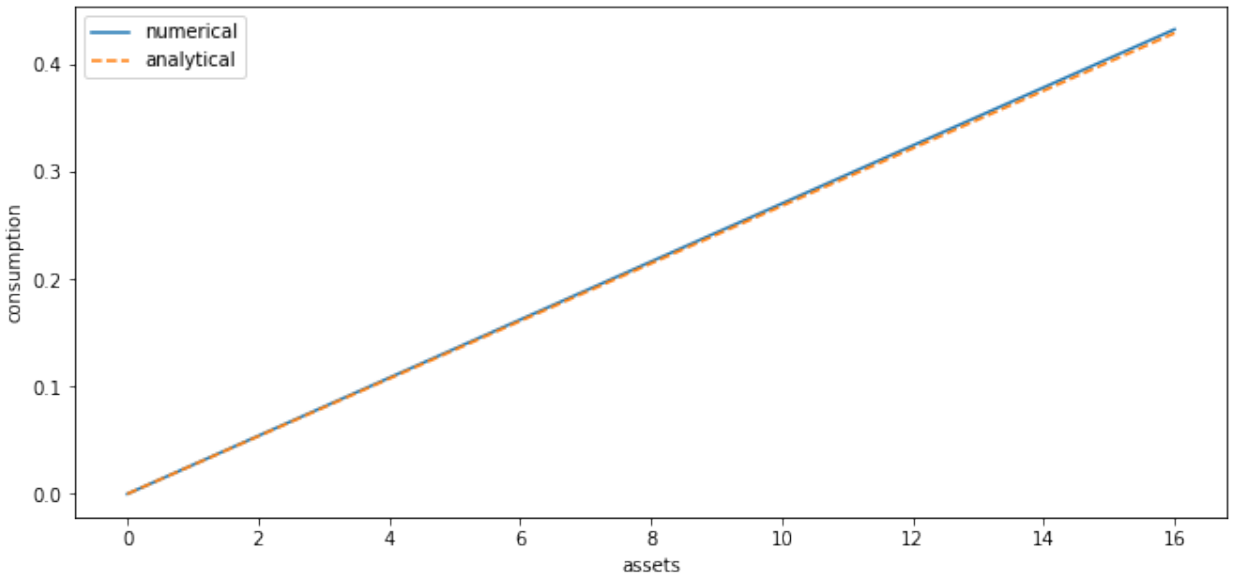
We know that, in this case, the value function and optimal consumption policy are given by

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Let's see if we match up:

```
ifp_cake_eating = IFP(r=0.0, y=(0.0, 0.0))

σ_star = solve_model_time_iter(ifp_cake_eating, σ_init)

fig, ax = plt.subplots()
ax.plot(a_grid, σ_star[:, 0], label='numerical')
ax.plot(a_grid, c_star(a_grid, ifp.β, ifp.γ), '--', label='analytical')

ax.set(xlabel='assets', ylabel='consumption')
ax.legend()

plt.show()
```

```
Error at iteration 25 is 0.023332272630545492.
Error at iteration 50 is 0.005301238424249566.
Error at iteration 75 is 0.0019706324625650695.
Error at iteration 100 is 0.0008675521337956349.
Error at iteration 125 is 0.00041073542212266556.
Error at iteration 150 is 0.00020120334010526042.
Error at iteration 175 is 0.00010021430795065234.

Converged in 176 iterations.
```



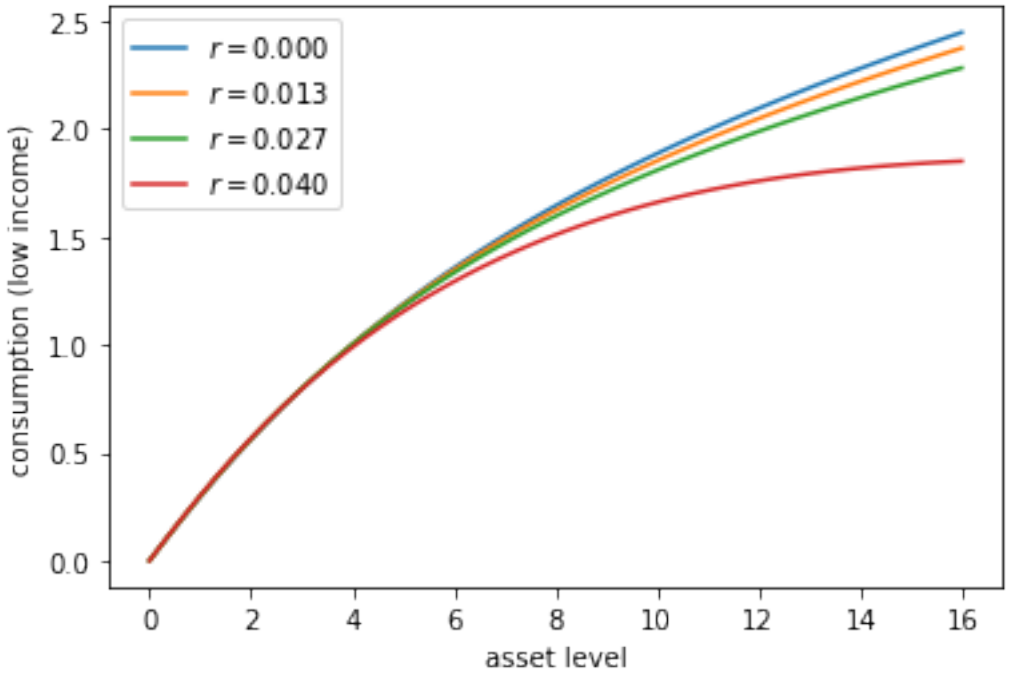
Success!

39.5 Exercises

39.5.1 Exercise 1

Let's consider how the interest rate affects consumption.

Reproduce the following figure, which shows (approximately) optimal consumption policies for different interest rates



- Other than `r`, all parameters are at their default values.
- `r` steps through `np.linspace(0, 0.04, 4)`.
- Consumption is plotted against assets for income shock fixed at the smallest value.

The figure shows that higher interest rates boost savings and hence suppress consumption.

39.5.2 Exercise 2

Now let's consider the long run asset levels held by households under the default parameters.

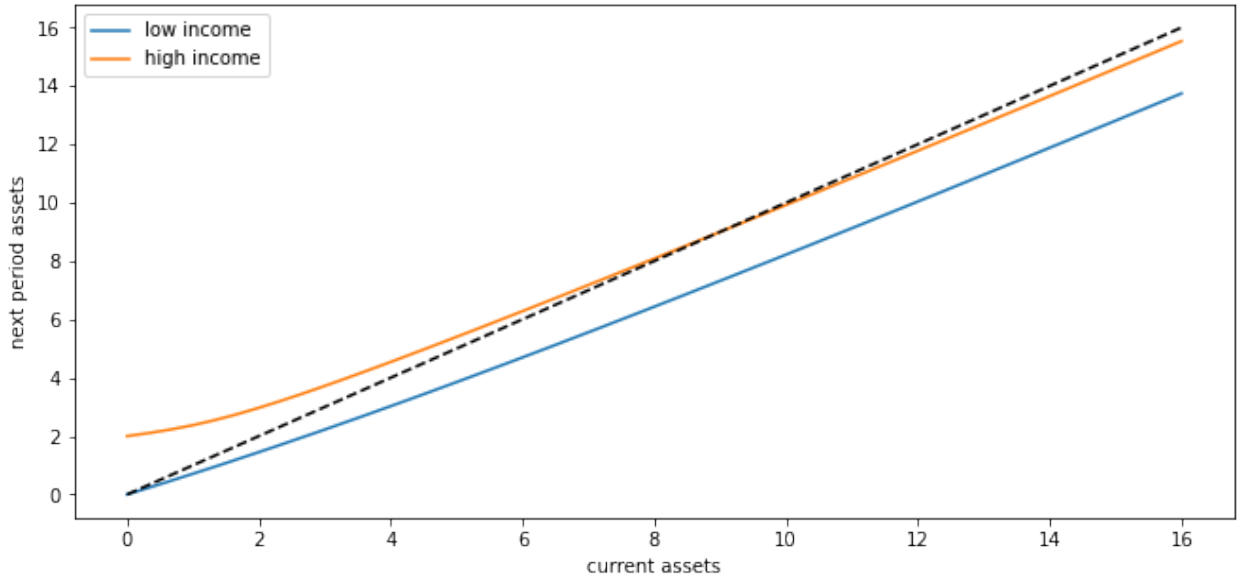
The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal

```
ifp = IFP()

σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
a = ifp.asset_grid
R, y = ifp.R, ifp.y

fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('low income', 'high income')):
    ax.plot(a, R * (a - σ_star[:, z]) + y[z] , label=lb)
```

(continues on next page)

(continued from previous page)

```
ax.plot(a, a, 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

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



The unbroken lines show the update function for assets at each z, which is

$$a \mapsto R(a - \sigma^*(a, z)) + y(z)$$

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

In fact there is a unique stationary distribution of assets that we can calculate by simulation

- Can be proved via theorem 2 of [HP92].
- It represents the long run dispersion of assets across households when households have idiosyncratic shocks.

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series.

Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram it.

Your task is to generate such a histogram.

- Use a single time series $\{a_t\}$ of length 500,000.
- Given the length of this time series, the initial condition (a_0, z_0) will not matter.
- You might find it helpful to use the `MarkovChain` class from `quantecon`.

39.5.3 Exercise 3

Following on from exercises 1 and 2, let's look at how savings and aggregate asset holdings vary with the interest rate

- Note: [LS18] section 18.6 can be consulted for more background on the topic treated in this exercise.

For a given parameterization of the model, the mean of the stationary distribution of assets can be interpreted as aggregate capital in an economy with a unit mass of *ex-ante* identical households facing idiosyncratic shocks.

Your task is to investigate how this measure of aggregate capital varies with the interest rate.

Following tradition, put the price (i.e., interest rate) on the vertical axis.

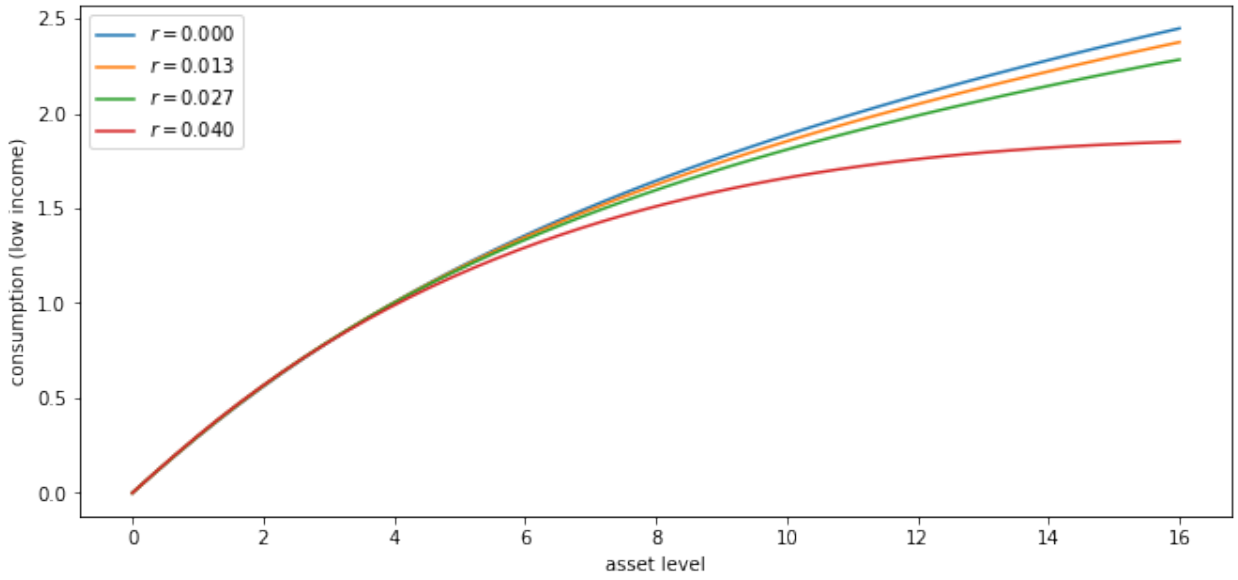
On the horizontal axis put aggregate capital, computed as the mean of the stationary distribution given the interest rate.

39.6 Solutions

39.6.1 Exercise 1

Here's one solution:

```
r_vals = np.linspace(0, 0.04, 4)

fig, ax = plt.subplots()
for r_val in r_vals:
    ifp = IFP(r=r_val)
    σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
    ax.plot(ifp.asset_grid, σ_star[:, 0], label=f'$r = {r_val:.3f}$')

ax.set(xlabel='asset level', ylabel='consumption (low income)')
ax.legend()
plt.show()
```



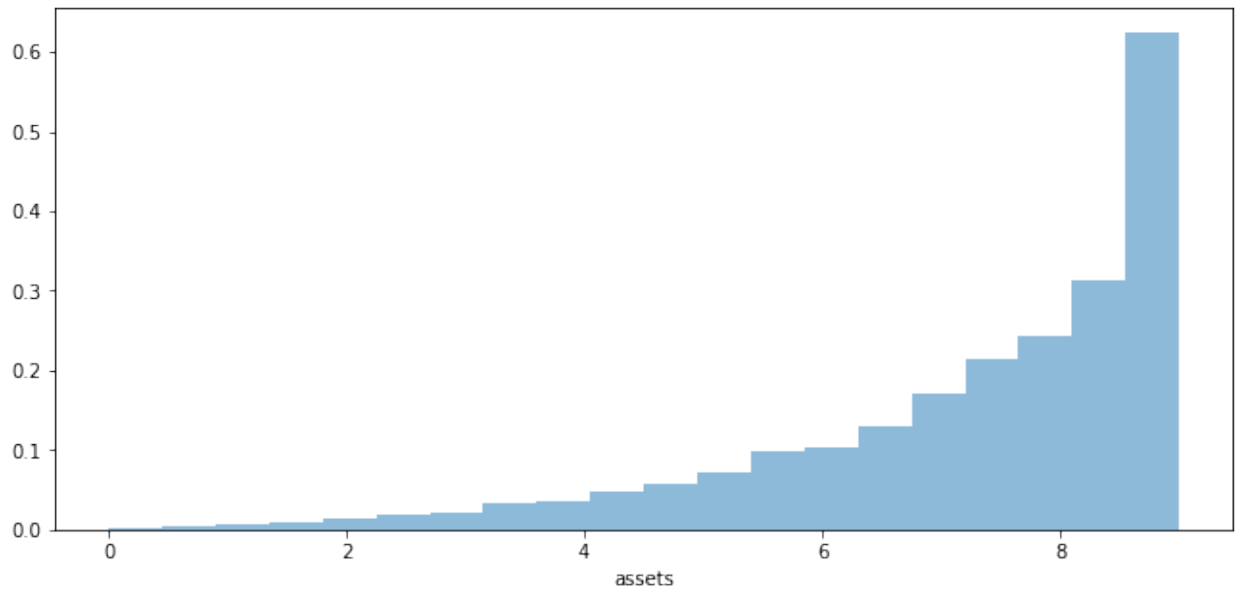
39.6.2 Exercise 2

First we write a function to compute a long asset series.

```
def compute_asset_series(ifp, T=500_000, seed=1234):
    """
    Simulates a time series of length T for assets, given optimal
    savings behavior.

    ifp is an instance of IFP
    """
    P, y, R = ifp.P, ifp.y, ifp.R  # Simplify names

    # Solve for the optimal policy
    σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
    σ = lambda a, z: interp(ifp.asset_grid, σ_star[:, z], a)

    # Simulate the exogeneous state process
    mc = MarkovChain(P)
    z_seq = mc.simulate(T, random_state=seed)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        a[t+1] = R * (a[t] - σ(a[t], z)) + y[z]
    return a
```

Now we call the function, generate the series and then histogram it:

```
ifp = IFP()
a = compute_asset_series(ifp)

fig, ax = plt.subplots()
ax.hist(a, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```



The shape of the asset distribution is unrealistic.

Here it is left skewed when in reality it has a long right tail.

In a *subsequent lecture* we will rectify this by adding more realistic features to the model.

39.6.3 Exercise 3

Here's one solution

```
M = 25
r_vals = np.linspace(0, 0.02, M)
fig, ax = plt.subplots()

asset_mean = []
for r in r_vals:
    print(f'Solving model at r = {r}')
    ifp = IFP(r=r)
    mean = np.mean(compute_asset_series(ifp, T=250_000))
    asset_mean.append(mean)
ax.plot(asset_mean, r_vals)

ax.set(xlabel='capital', ylabel='interest rate')

plt.show()
```

```
Solving model at r = 0.0
```

```
Solving model at r = 0.0008333333333333334
```

```
Solving model at r = 0.0016666666666666668
```

```
Solving model at r = 0.0025
```

```
Solving model at r = 0.0033333333333333335
```

```
Solving model at r = 0.004166666666666667
```

```
Solving model at r = 0.005
```

```
Solving model at r = 0.005833333333333334
```

```
Solving model at r = 0.006666666666666667
```

```
Solving model at r = 0.007500000000000001
```

```
Solving model at r = 0.008333333333333333
```

```
Solving model at r = 0.009166666666666667
```

```
Solving model at r = 0.01
```

```
Solving model at r = 0.010833333333333334
```

```
Solving model at r = 0.011666666666666667
```

```
Solving model at r = 0.0125
```

```
Solving model at r = 0.013333333333333334
```

```
Solving model at r = 0.014166666666666668
```

```
Solving model at r = 0.015000000000000001
```

```
Solving model at r = 0.015833333333333335
```

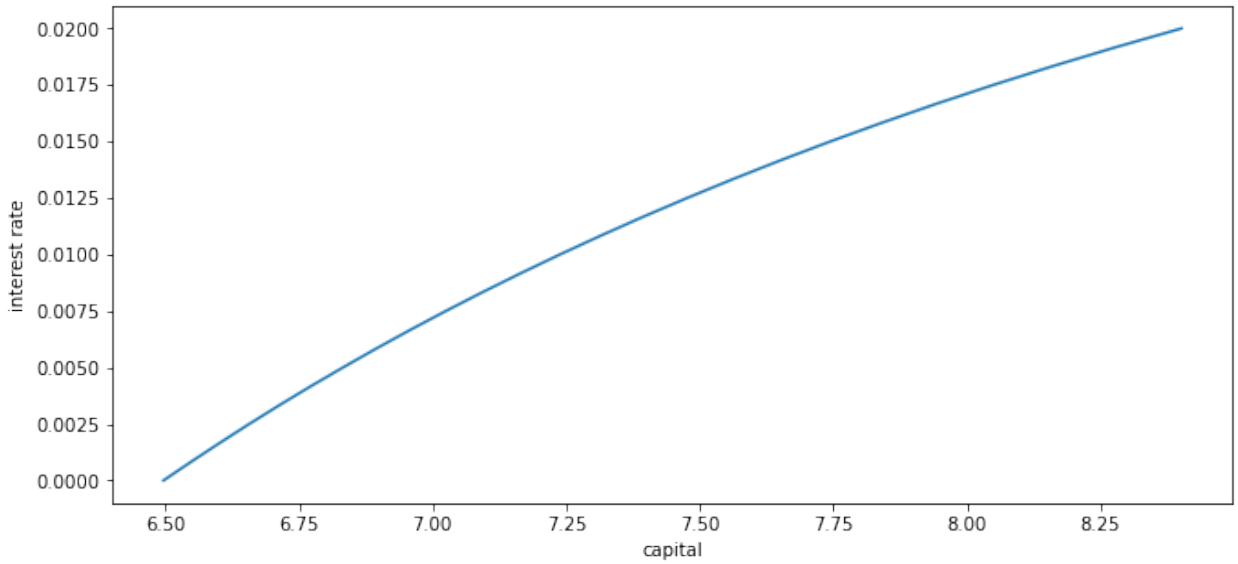
```
Solving model at r = 0.016666666666666666
```

```
Solving model at r = 0.0175
```

```
Solving model at r = 0.018333333333333333
```

```
Solving model at r = 0.01916666666666667
```

```
Solving model at r = 0.02
```



As expected, aggregate savings increases with the interest rate.

CHAPTER

FORTY

THE INCOME FLUCTUATION PROBLEM II: STOCHASTIC RETURNS ON ASSETS

Contents

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

```
!conda install -y quantecon
!pip install interpolation
```

40.1 Overview

In this lecture, we continue our study of the *income fluctuation problem*.

While the interest rate was previously taken to be fixed, we now allow returns on assets to be state-dependent.

This matches the fact that most households with a positive level of assets face some capital income risk.

It has been argued that modeling capital income risk is essential for understanding the joint distribution of income and wealth (see, e.g., [BBZ15] or [ST19b]).

Theoretical properties of the household savings model presented here are analyzed in detail in [MST20].

In terms of computation, we use a combination of time iteration and the endogenous grid method to solve the model quickly and accurately.

We require the following imports:

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

(continues on next page)

(continued from previous page)

```
import numpy as np
from quantecon.optimize import brent_max, brentq
from interpolation import interp
from numba import njit, float64
from numba.experimental import jitclass
from quantecon import MarkovChain
```

40.2 The Savings Problem

In this section we review the household problem and optimality results.

40.2.1 Set Up

A household chooses a consumption-asset path $\{(c_t, a_t)\}$ to maximize

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t u(c_t)\right\} \tag{1}$$

subject to

$$a_{t+1} = R_{t+1}(a_t - c_t) + Y_{t+1} \text{ and } 0 \leq c_t \leq a_t, \tag{2}$$

with initial condition $(a_0, Z_0) = (a, z)$ treated as given.

Note that $\{R_t\}_{t\geq 1}$, the gross rate of return on wealth, is allowed to be stochastic.

The sequence $\{Y_t\}_{t\geq 1}$ is non-financial income.

The stochastic components of the problem obey

$$R_t = R(Z_t, \zeta_t) \quad \text{and} \quad Y_t = Y(Z_t, \eta_t), \tag{3}$$

where

- the maps R and Y are time-invariant nonnegative functions,
- the innovation processes $\{\zeta_t\}$ and $\{\eta_t\}$ are IID and independent of each other, and
- $\{Z_t\}_{t\geq 0}$ is an irreducible time-homogeneous Markov chain on a finite set Z

Let P represent the Markov matrix for the chain $\{Z_t\}_{t\geq 0}$.

Our assumptions on preferences are the same as our *previous lecture* on the income fluctuation problem.

As before, $\mathbb{E}_z \hat{X}$ means expectation of next period value $\hat{X}$ given current value $Z = z$.

40.2.2 Assumptions

We need restrictions to ensure that the objective (1) is finite and the solution methods described below converge.

We also need to ensure that the present discounted value of wealth does not grow too quickly.

When $\{R_t\}$ was constant we required that $\beta R < 1$.

Now it is stochastic, we require that

$$\beta G_R < 1, \quad \text{where} \quad G_R := \lim_{n \to \infty} \left(\mathbb{E} \prod_{t=1}^{n} R_t \right)^{1/n} \tag{4}$$

Notice that, when $\{R_t\}$ takes some constant value R, this reduces to the previous restriction $\beta R < 1$

The value G_R can be thought of as the long run (geometric) average gross rate of return.

More intuition behind (4) is provided in [MST20].

Discussion on how to check it is given below.

Finally, we impose some routine technical restrictions on non-financial income.

$$\mathbb{E}\, Y_t < \infty \text{ and } \mathbb{E}\, u'(Y_t) < \infty$$

One relatively simple setting where all these restrictions are satisfied is the IID and CRRA environment of [BBZ15].

40.2.3 Optimality

Let the class of candidate consumption policies $\mathcal{C}$ be defined *as before*.

In [MST20] it is shown that, under the stated assumptions,

- any $\sigma \in \mathcal{C}$ satisfying the Euler equation is an optimal policy and
- exactly one such policy exists in $\mathcal{C}$.

In the present setting, the Euler equation takes the form

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta\, \mathbb{E}_z\, \hat{R}\, (u' \circ \sigma)[\hat{R}(a - \sigma(a, z)) + \hat{Y},\, \hat{Z}],\, u'(a) \right\} \tag{5}$$

(Intuition and derivation are similar to our *earlier lecture* on the income fluctuation problem.)

We again solve the Euler equation using time iteration, iterating with a Coleman–Reffett operator K defined to match the Euler equation (5).

40.3 Solution Algorithm

40.3.1 A Time Iteration Operator

Our definition of the candidate class $\sigma \in \mathcal{C}$ of consumption policies is the same as in our *earlier lecture* on the income fluctuation problem.

For fixed $\sigma \in \mathcal{C}$ and $(a, z) \in \mathbf{S}$, the value $K\sigma(a, z)$ of the function $K\sigma$ at (a, z) is defined as the $\xi \in (0, a]$ that solves

$$u'(\xi) = \max \left\{ \beta\, \mathbb{E}_z\, \hat{R}\, (u' \circ \sigma)[\hat{R}(a - \xi) + \hat{Y},\, \hat{Z}],\, u'(a) \right\} \tag{6}$$

The idea behind K is that, as can be seen from the definitions, $\sigma \in \mathcal{C}$ satisfies the Euler equation if and only if $K\sigma(a, z) = \sigma(a, z)$ for all $(a, z) \in \mathbf{S}$.

This means that fixed points of K in $\mathcal{C}$ and optimal consumption policies exactly coincide (see [MST20] for more details).

40.3.2 Convergence Properties

As before, we pair $\mathcal{C}$ with the distance

$$\rho(c, d) := \sup_{(a,z)\in \mathbf{S}} |(u' \circ c)(a, z) - (u' \circ d)(a, z)|,$$

It can be shown that

1. $(\mathcal{C}, \rho)$ is a complete metric space,
2. there exists an integer n such that K^n is a contraction mapping on $(\mathcal{C}, \rho)$, and
3. The unique fixed point of K in $\mathcal{C}$ is the unique optimal policy in $\mathcal{C}$.

We now have a clear path to successfully approximating the optimal policy: choose some $\sigma \in \mathcal{C}$ and then iterate with K until convergence (as measured by the distance ρ).

40.3.3 Using an Endogenous Grid

In the study of that model we found that it was possible to further accelerate time iteration via the *endogenous grid method*.

We will use the same method here.

The methodology is the same as it was for the optimal growth model, with the minor exception that we need to remember that consumption is not always interior.

In particular, optimal consumption can be equal to assets when the level of assets is low.

Finding Optimal Consumption

The endogenous grid method (EGM) calls for us to take a grid of *savings* values s_i, where each such s is interpreted as $s = a - c$.

For the lowest grid point we take $s_0 = 0$.

For the corresponding a_0, c_0 pair we have $a_0 = c_0$.

This happens close to the origin, where assets are low and the household consumes all that it can.

Although there are many solutions, the one we take is $a_0 = c_0 = 0$, which pins down the policy at the origin, aiding interpolation.

For $s > 0$, we have, by definition, $c < a$, and hence consumption is interior.

Hence the max component of (5) drops out, and we solve for

$$c_i = (u')^{-1} \left\{ \beta \, \mathbb{E}_z \hat{R} (u' \circ \sigma) [\hat{R} s_i + \hat{Y}, \, \hat{Z}] \right\} \tag{7}$$

at each s_i.

Iterating

Once we have the pairs $\{s_i, c_i\}$, the endogenous asset grid is obtained by $a_i = c_i + s_i$.

Also, we held $z \in \mathsf{Z}$ in the discussion above so we can pair it with a_i.

An approximation of the policy $(a, z) \mapsto \sigma(a, z)$ can be obtained by interpolating $\{a_i, c_i\}$ at each z.

In what follows, we use linear interpolation.

40.3.4 Testing the Assumptions

Convergence of time iteration is dependent on the condition $\beta G_R < 1$ being satisfied.

One can check this using the fact that G_R is equal to the spectral radius of the matrix L defined by

$$L(z, \hat{z}) := P(z, \hat{z}) \int R(\hat{z}, x)\phi(x)dx$$

This identity is proved in [MST20], where ϕ is the density of the innovation ζ_t to returns on assets.

(Remember that Z is a finite set, so this expression defines a matrix.)

Checking the condition is even easier when $\{R_t\}$ is IID.

In that case, it is clear from the definition of G_R that G_R is just $\mathbb{E}R_t$.

We test the condition $\beta \mathbb{E}R_t < 1$ in the code below.

40.4 Implementation

We will assume that $R_t = \exp(a_r \zeta_t + b_r)$ where a_r, b_r are constants and $\{\zeta_t\}$ is IID standard normal.

We allow labor income to be correlated, with

$$Y_t = \exp(a_y \eta_t + Z_t b_y)$$

where $\{\eta_t\}$ is also IID standard normal and $\{Z_t\}$ is a Markov chain taking values in $\{0, 1\}$.

```
ifp_data = [
    ('γ', float64),              # utility parameter
    ('β', float64),              # discount factor
    ('P', float64[:, :]),        # transition probs for z_t
    ('a_r', float64),            # scale parameter for R_t
    ('b_r', float64),            # additive parameter for R_t
    ('a_y', float64),            # scale parameter for Y_t
    ('b_y', float64),            # additive parameter for Y_t
    ('s_grid', float64[:]),      # Grid over savings
    ('η_draws', float64[:]),     # Draws of innovation η for MC
    ('ζ_draws', float64[:])      # Draws of innovation ζ for MC
]
```

```
@jitclass(ifp_data)
class IFP:
    """
    A class that stores primitives for the income fluctuation
    problem.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self,
                 γ=1.5,
                 β=0.96,
                 P=np.array([(0.9, 0.1),
                             (0.1, 0.9)]),
                 a_r=0.1,
                 b_r=0.0,
                 a_y=0.2,
                 b_y=0.5,
                 shock_draw_size=50,
                 grid_max=10,
                 grid_size=100,
                 seed=1234):

        np.random.seed(seed)  # arbitrary seed

        self.P, self.γ, self.β = P, γ, β
        self.a_r, self.b_r, self.a_y, self.b_y = a_r, b_r, a_y, b_y
        self.η_draws = np.random.randn(shock_draw_size)
        self.ζ_draws = np.random.randn(shock_draw_size)
        self.s_grid = np.linspace(0, grid_max, grid_size)

        # Test stability assuming {R_t} is IID and adopts the lognormal
        # specification given below.  The test is then β E R_t < 1.
        ER = np.exp(b_r + a_r**2 / 2)
        assert β * ER < 1, "Stability condition failed."

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

    # Inverse of marginal utility
    def u_prime_inv(self, c):
        return c**(-1/self.γ)

    def R(self, z, ζ):
        return np.exp(self.a_r * ζ + self.b_r)

    def Y(self, z, η):
        return np.exp(self.a_y * η + (z * self.b_y))
```

Here's the Coleman-Reffett operator based on EGM:

```
@njit
def K(a_in, σ_in, ifp):
    """
    The Coleman--Reffett operator for the income fluctuation problem,
    using the endogenous grid method.

        * ifp is an instance of IFP
        * a_in[i, z] is an asset grid
        * σ_in[i, z] is consumption at a_in[i, z]
    """

    # Simplify names
```

(continues on next page)

(continued from previous page)

```
    u_prime, u_prime_inv = ifp.u_prime, ifp.u_prime_inv
    R, Y, P, β = ifp.R, ifp.Y, ifp.P, ifp.β
    s_grid, η_draws, ζ_draws = ifp.s_grid, ifp.η_draws, ifp.ζ_draws
    n = len(P)

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_in[:, z], σ_in[:, z], a)

    # Allocate memory
    σ_out = np.empty_like(σ_in)

    # Obtain c_i at each s_i, z, store in σ_out[i, z], computing
    # the expectation term by Monte Carlo
    for i, s in enumerate(s_grid):
        for z in range(n):
            # Compute expectation
            Ez = 0.0
            for z_hat in range(n):
                for η in ifp.η_draws:
                    for ζ in ifp.ζ_draws:
                        R_hat = R(z_hat, ζ)
                        Y_hat = Y(z_hat, η)
                        U = u_prime(σ(R_hat * s + Y_hat, z_hat))
                        Ez += R_hat * U * P[z, z_hat]
            Ez = Ez / (len(η_draws) * len(ζ_draws))
            σ_out[i, z] =  u_prime_inv(β * Ez)

    # Calculate endogenous asset grid
    a_out = np.empty_like(σ_out)
    for z in range(n):
        a_out[:, z] = s_grid + σ_out[:, z]

    # Fixing a consumption-asset pair at (0, 0) improves interpolation
    σ_out[0, :] = 0
    a_out[0, :] = 0

    return a_out, σ_out
```

The next function solves for an approximation of the optimal consumption policy via time iteration.

```
def solve_model_time_iter(model,        # Class with model information
                          a_vec,        # Initial condition for assets
                          σ_vec,        # Initial condition for consumption
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

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

    while i < max_iter and error > tol:
        a_new, σ_new = K(a_vec, σ_vec, model)
        error = np.max(np.abs(σ_vec - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
```

(continues on next page)

(continued from previous page)

```
            print(f"Error at iteration {i} is {error}.")
        a_vec, σ_vec = np.copy(a_new), np.copy(σ_new)

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return a_new, σ_new
```

Now we are ready to create an instance at the default parameters.

```
ifp = IFP()
```

Next we set up an initial condition, which corresponds to consuming all assets.

```
# Initial guess of σ = consume all assets
k = len(ifp.s_grid)
n = len(ifp.P)
σ_init = np.empty((k, n))
for z in range(n):
    σ_init[:, z] = ifp.s_grid
a_init = np.copy(σ_init)
```

Let's generate an approximation solution.

```
a_star, σ_star = solve_model_time_iter(ifp, a_init, σ_init, print_skip=5)
```

```
Error at iteration 5 is 0.5081944529506561.
```

```
Error at iteration 10 is 0.1057246950930697.
```

```
Error at iteration 15 is 0.03658262202883744.
```

```
Error at iteration 20 is 0.013936729965906114.
```

```
Error at iteration 25 is 0.005292165269711546.
```

```
Error at iteration 30 is 0.0019748126990770665.
```

```
Error at iteration 35 is 0.0007219210463285108.
```

```
Error at iteration 40 is 0.0002690544496094971.
```

```
Error at iteration 45 is 9.163966595426842e-05.

Converged in 45 iterations.
```

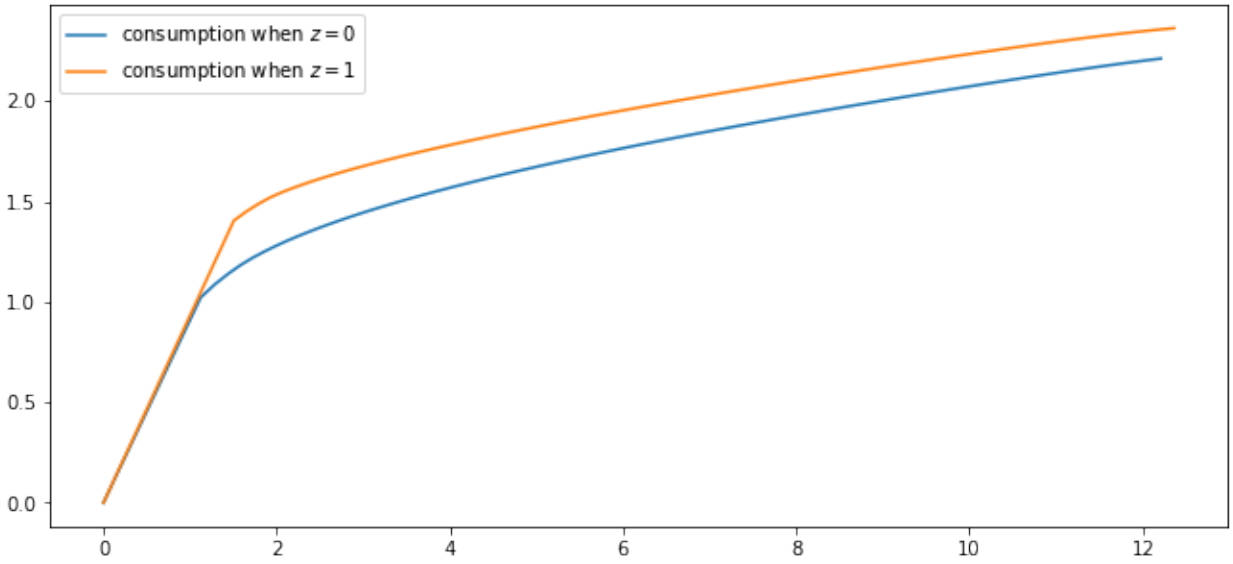
Here's a plot of the resulting consumption policy.

```
fig, ax = plt.subplots()
for z in range(len(ifp.P)):
```

(continues on next page)

(continued from previous page)

```
    ax.plot(a_star[:, z], σ_star[:, z], label=f"consumption when $z={z}$")

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



Notice that we consume all assets in the lower range of the asset space.

This is because we anticipate income Y_{t+1} tomorrow, which makes the need to save less urgent.

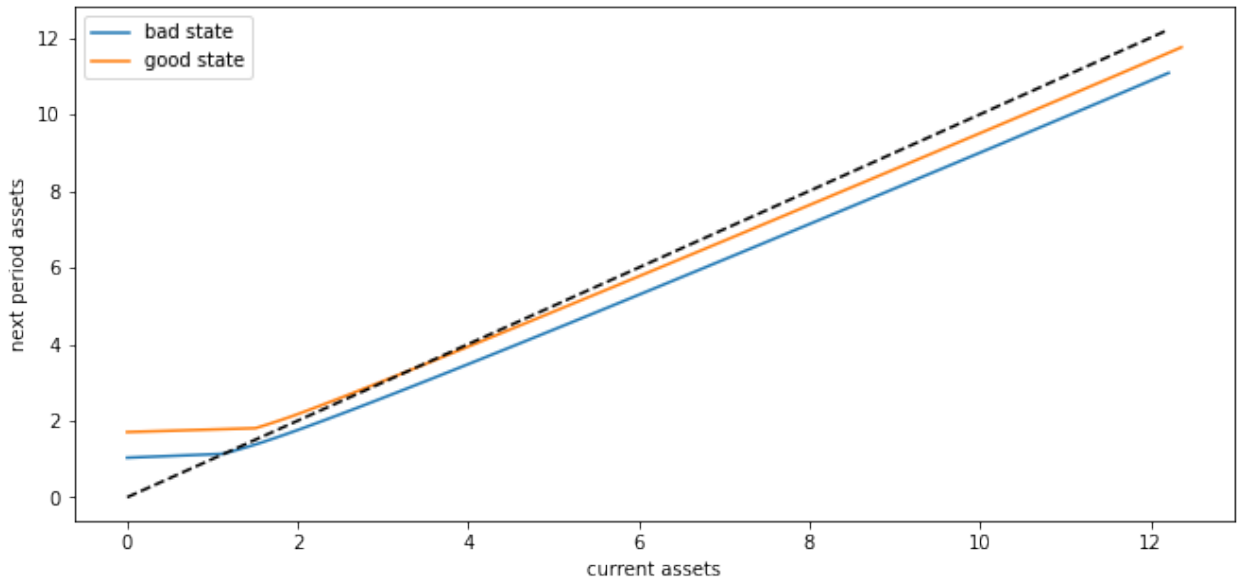
Can you explain why consuming all assets ends earlier (for lower values of assets) when $z = 0$?

40.4.1 Law of Motion

Let's try to get some idea of what will happen to assets over the long run under this consumption policy.

As with our *earlier lecture* on the income fluctuation problem, we begin by producing a 45 degree diagram showing the law of motion for assets

```
# Good and bad state mean labor income
Y_mean = [np.mean(ifp.Y(z, ifp.η_draws)) for z in (0, 1)]
# Mean returns
R_mean = np.mean(ifp.R(z, ifp.ζ_draws))

a = a_star
fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('bad state', 'good state')):
    ax.plot(a[:, z], R_mean * (a[:, z] - σ_star[:, z]) + Y_mean[z] , label=lb)

ax.plot(a[:, 0], a[:, 0], 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

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



The unbroken lines represent, for each z, an average update function for assets, given by

$$a \mapsto \bar{R}(a - \sigma^*(a, z)) + \bar{Y}(z)$$

Here

- $\bar{R} = \mathbb{E} R_t$, which is mean returns and
- $\bar{Y}(z) = \mathbb{E}_z Y(z, \eta_t)$, which is mean labor income in state z.

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

40.5 Exercises

40.5.1 Exercise 1

Let's repeat our *earlier exercise* on the long-run cross sectional distribution of assets.

In that exercise, we used a relatively simple income fluctuation model.

In the solution, we found the shape of the asset distribution to be unrealistic.

In particular, we failed to match the long right tail of the wealth distribution.

Your task is to try again, repeating the exercise, but now with our more sophisticated model.

Use the default parameters.

40.6 Solutions

40.6.1 Exercise 1

First we write a function to compute a long asset series.

Because we want to JIT-compile the function, we code the solution in a way that breaks some rules on good programming style.

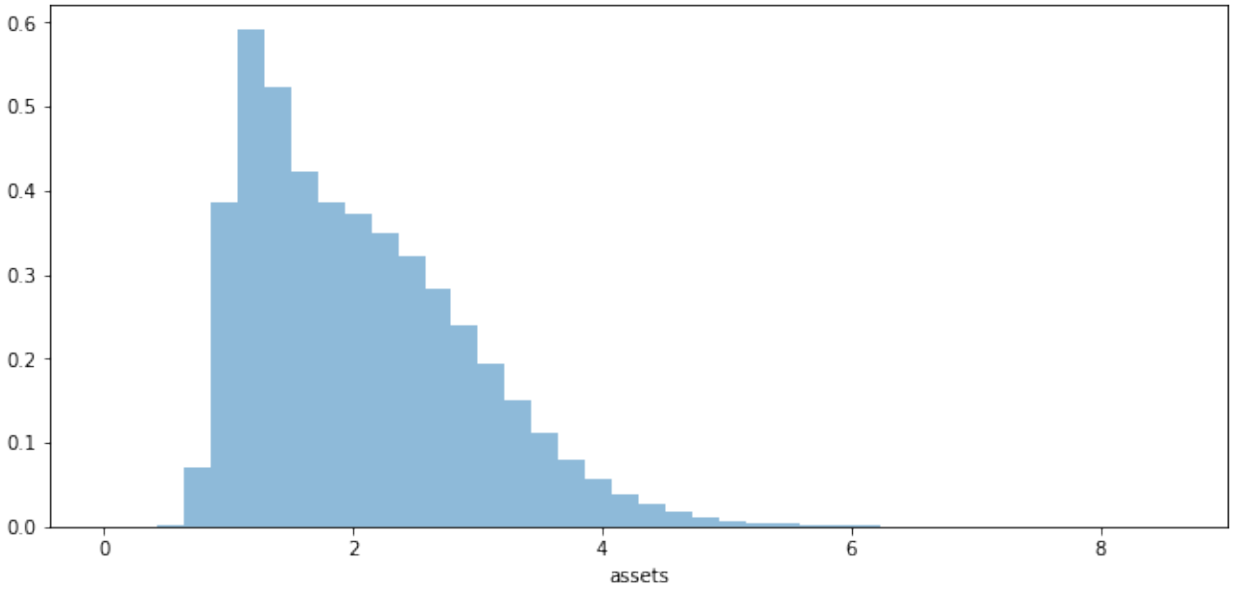
For example, we will pass in the solutions `a_star, σ_star` along with `ifp`, even though it would be more natural to just pass in `ifp` and then solve inside the function.

The reason we do this is that `solve_model_time_iter` is not JIT-compiled.

```
@njit
def compute_asset_series(ifp, a_star, σ_star, z_seq, T=500_000):
    """
    Simulates a time series of length T for assets, given optimal
    savings behavior.

        * ifp is an instance of IFP
        * a_star is the endogenous grid solution
        * σ_star is optimal consumption on the grid
        * z_seq is a time path for {Z_t}

    """

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_star[:, z], σ_star[:, z], a)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        ζ, η = np.random.randn(), np.random.randn()
        R = ifp.R(z, ζ)
        Y = ifp.Y(z, η)
        a[t+1] = R * (a[t] - σ(a[t], z)) + Y
    return a
```

Now we call the function, generate the series and then histogram it, using the solutions computed above.

```
T = 1_000_000
mc = MarkovChain(ifp.P)
z_seq = mc.simulate(T, random_state=1234)

a = compute_asset_series(ifp, a_star, σ_star, z_seq, T=T)

fig, ax = plt.subplots()
ax.hist(a, bins=40, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```



Now we have managed to successfully replicate the long right tail of the wealth distribution.

Here's another view of this using a horizontal violin plot.

```
fig, ax = plt.subplots()
ax.violinplot(a, vert=False, showmedians=True)
ax.set(xlabel='assets')
plt.show()
```