How to read this lecture...

Time Iteration with Euler Equations

- Overview
- The Euler Equation
- Comparison with Value Function Iteration
- Implementation
- Exercises
- Solutions

Overview

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

In that lecture we solved the associated discounted 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 that's more efficient than value function iteration for this and some other closely related applications

In a subsequent lecture we'll see that the numerical implementation part of the Euler equation method can be further adjusted to obtain even more efficiency

The Euler Equation

Let's take the model set out in the stochastic growth model lecture and add the assumptions that

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 c^*

We know that c^* is a v^* greedy policy, so that $c^*(y)$ is the maximizer in (1)

The conditions above imply that

- c^* 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 $0 < c^*(y) < y$ for all strictly positive y, and
- the value function is strictly concave and continuously differentiable, with

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

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

To see why (2) might be valid, write the Bellman equation in the equivalent form

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

differentiate naively with respect to y, and then evaluate at the optimum

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 iteriority of the optimal policy imply that optimal consumption satisfies the first order condition associated with (1), which is

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

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

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

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 c^*

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

The Coleman Operator

Recall the Bellman operator

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

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 (i.e., $0 < \sigma(y) < y$ for all strictly positive y)

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

1. The operator K takes as its argument a $\sigma \in \mathscr{P}$ 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 operator** to acknowledge the work of [Col90] (although many people have studied this and other closely related iterative techniques)

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 c^* is a fixed point

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

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

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

Is the Coleman 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 $\sigma \in \mathscr{P}$, 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 $K\sigma \in \mathscr{P}$ whenever $\sigma \in \mathscr{P}$

Comparison with Value Function Iteration

How does Euler equation time iteration compare with value function iteration?

Both can be used to compute the optimal policy, but is one faster or more accurate?

There are two parts to this story

First, on a theoretical level, the two methods are essentially isomorphic

In particular, they converge at the same rate

We'll prove this in just a moment

The other side to the story is the speed of the numerical implementation

It turns out that, once we actually implement these two routines, time iteration is faster and more accurate than value function iteration

More on this below

Equivalent Dynamics

Let's talk about the theory first

To explain the connection between the two algorithms, it helps to understand the notion of equivalent dynamics

(This concept is very helpful in many other contexts as well)

Suppose that we have a function g: $X \to X$ where X is a given set

The pair (X, g) is sometimes called a **dynamical system** and we associate it with trajectories of the form

$$x_{t+1} = g(x_t), \qquad x_0 \text{ given}$$

Equivalently, $x_t = g^t(x_0)$, where g is the t-th composition of g with itself

Here's the picture

$$x_0 \xrightarrow{g} g(x_0) \xrightarrow{g} g^2(x_0) \xrightarrow{g} g^3(x_0) \xrightarrow{g} \cdots$$

Now let another function h: $Y \to Y$ where Y is another set

Suppose further that

- there exists a bijection τ from X to Y
- the two functions **commute** under τ, which is to say that $\tau(g(x)) = h(\tau(x))$ for all $x \in X$

The last statement can be written more simply as

$$\tau \circ g = h \circ \tau$$

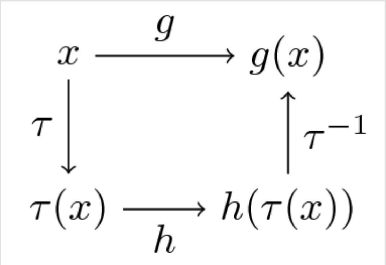
or, by applying τ^{-1} to both sides

$$g = \tau^{-1} \circ h \circ \tau \tag{8}$$

Here's a commutative diagram that illustrates

$$\begin{array}{ccc} X & \xrightarrow{g} & X \\ {\scriptstyle \tau}\downarrow & & \uparrow{\scriptstyle \tau^{-1}} \\ Y & \xrightarrow[h]{} & Y \end{array}$$

Here's a similar figure that traces out the action of the maps on a point $x \in X$



Now, it's easy to check from (8) that $g^2 = \tau^{-1} \circ h^2 \circ \tau$ holds

In fact, if you like proofs by induction, you won't have trouble showing that

$$g^n = \tau^{-1} \circ h^n \circ \tau$$

is valid for all n

What does this tell us?

It tells us that the following are equivalent

- iterate n times with g, starting at x
- shift x to Y using τ, iterate n times with h starting at $\tau(x)$, and shift the result $h^n(\tau(x))$ back to X using τ^{-1}

We end up with exactly the same object

Back to Economics

Have you guessed where this is leading?

What we're going to show now is that the operators T and K commute under a certain bijection

The implication is that they have exactly the same rate of convergence

To make life a little easier, we'll assume in the following analysis (although not always in our applications) that $u(0) = 0$

A Bijection

Let $\mathscr{V}$ be all strictly concave, continuously differentiable functions v mapping $\mathbb{R}_+$ to itself and satisfying $v(0) = 0$ and $v'(y) > u'(y)$ for all positive y

For $v \in \mathscr{V}$ let

$$Mv := h \circ v' \qquad \text{where } h := (u')^{-1}$$

Although we omit details, $\sigma := Mv$ is actually the unique v-greedy policy

- See proposition 12.1.18 of EDTC

It turns out that M is a bijection from $\mathscr{V}$ to $\mathscr{P}$

A (solved) exercise below asks you to confirm this

Commutative Operators

It is an additional solved exercise (see below) to show that T and K commute under M, in the sense that

$$M \circ T = K \circ M \tag{9}$$

In view of the preceding discussion, this implies that

$$T^n = M^{-1} \circ K^n \circ M$$

Hence, T and K converge at exactly the same rate!

Implementation

We've just shown that the operators T and K have the same rate of convergence

However, it turns out that, once numerical approximation is taken into account, significant differences arises

In particular, the image of policy functions under K can be calculated faster and with greater accuracy than the image of value functions under T

Our intuition for this result is that

- the Coleman operator exploits more information because it uses first order and envelope conditions
- policy functions generally have less curvature than value functions, and hence admit more accurate approximations based on grid point information

The Operator

Here's some code that implements the Coleman operator

```
import numpy as np
from scipy.optimize import brentq

def coleman_operator(g, grid, beta, u_prime, f, f_prime, shocks, Kg=None):
    """
    The approximate Coleman operator, which takes an existing guess g of the
    optimal consumption policy and computes and returns the updated function
    Kg on the grid points.  An array to store the new set of values Kg is
    optionally supplied (to avoid having to allocate new arrays at each
    iteration).  If supplied, any existing data in Kg will be overwritten.

    Parameters
    ----------
    g : array_like(float, ndim=1)
        The value of the input policy function on grid points
    grid : array_like(float, ndim=1)
        The set of grid points
    beta : scalar
        The discount factor
    u_prime : function
        The derivative u'(c) of the utility function
    f : function
        The production function f(k)
    f_prime : function
        The derivative f'(k)
    shocks : numpy array
        An array of draws from the shock, for Monte Carlo integration (to
        compute expectations).
    Kg : array_like(float, ndim=1) optional (default=None)
        Array to write output values to

    """
    # === Apply linear interpolation to g === #
    g_func = lambda x: np.interp(x, grid, g)

    # == Initialize Kg if necessary == #
    if Kg is None:
        Kg = np.empty_like(g)

    # == solve for updated consumption value
    for i, y in enumerate(grid):
        def h(c):
            vals = u_prime(g_func(f(y - c) * shocks)) * f_prime(y - c) * sho
            return u_prime(c) - beta * np.mean(vals)
        c_star = brentq(h, 1e-10, y - 1e-10)
        Kg[i] = c_star

    return Kg
```

It has some similarities to the code for the Bellman operator in our optimal growth lecture

For example, it evaluates integrals by Monte Carlo and approximates functions using linear interpolation

Let's see that Bellman operator code again, since we'll use it in some tests below

```
import numpy as np
from scipy.optimize import fminbound


def bellman_operator(w, grid, beta, u, f, shocks, Tw=None, compute_policy=0)
    """
    The approximate Bellman operator, which computes and returns the
    updated value function Tw on the grid points.  An array to store
    the new set of values Tw is optionally supplied (to avoid having to
    allocate new arrays at each iteration).  If supplied, any existing data
    Tw will be overwritten.

    Parameters
    ----------
    w : array_like(float, ndim=1)
        The value of the input function on different grid points
    grid : array_like(float, ndim=1)
        The set of grid points
    beta : scalar
        The discount factor
    u : function
        The utility function
    f : function
        The production function
    shocks : numpy array
        An array of draws from the shock, for Monte Carlo integration (to
        compute expectations).
    Tw : array_like(float, ndim=1) optional (default=None)
        Array to write output values to
    compute_policy : Boolean, optional (default=False)
        Whether or not to compute policy function

    """
    # === Apply linear interpolation to w === #
    w_func = lambda x: np.interp(x, grid, w)

    # == Initialize Tw if necessary == #
    if Tw is None:
        Tw = np.empty_like(w)

    if compute_policy:
        sigma = np.empty_like(w)

    # == set Tw[i] = max_c { u(c) + beta E w(f(y  - c) z)} == #
    for i, y in enumerate(grid):
        def objective(c):
            return - u(c) - beta * np.mean(w_func(f(y - c) * shocks))
        c_star = fminbound(objective, 1e-10, y)
        if compute_policy:
```

```
            sigma[i] = c_star
        Tw[i] = - objective(c_star)

    if compute_policy:
        return Tw, sigma
    else:
        return Tw
```

Testing on the Log / Cobb–Douglas case

As we did for value function iteration, let's start by testing our method in the presence of a model that does have an analytical solution

We assume the following imports

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

Now let's bring in the log-linear growth model we used in the value function iteration lecture

```
class LogLinearOG:
    """
    Log linear optimal growth model, with log utility, CD production and
    multiplicative lognormal shock, so that

        y = f(k, z) = z k^alpha

    with z ~ LN(mu, s).

    The class holds parameters and true value and policy functions.
    """

    def __init__(self, alpha=0.4, beta=0.96, mu=0, s=0.1):

        self.alpha, self.beta, self.mu, self.s = alpha, beta, mu, s

        # == Some useful constants == #
        self.ab = alpha * beta
        self.c1 = np.log(1 - self.ab) / (1 - beta)
        self.c2 = (mu + alpha * np.log(self.ab)) / (1 - alpha)
        self.c3 = 1 / (1 - beta)
        self.c4 = 1 / (1 - self.ab)

    def u(self, c):
        " Utility "
        return np.log(c)

    def u_prime(self, c):
        return 1 / c

    def f(self, k):
        " Deterministic part of production function.  "
        return k**self.alpha

    def f_prime(self, k):
        return self.alpha * k**(self.alpha - 1)

    def c_star(self, y):
        " True optimal policy.  "
        return (1 - self.alpha * self.beta) * y

    def v_star(self, y):
        " True value function. "
        return self.c1 + self.c2 * (self.c3 - self.c4) + self.c4 * np.log(y)
```

Next we generate an instance

```
lg = LogLinearOG()

# == Unpack parameters / functions for convenience == #
alpha, beta, mu, s = lg.alpha, lg.beta, lg.mu, lg.s
v_star, c_star = lg.v_star, lg.c_star
u, u_prime, f, f_prime = lg.u, lg.u_prime, lg.f, lg.f_prime
```

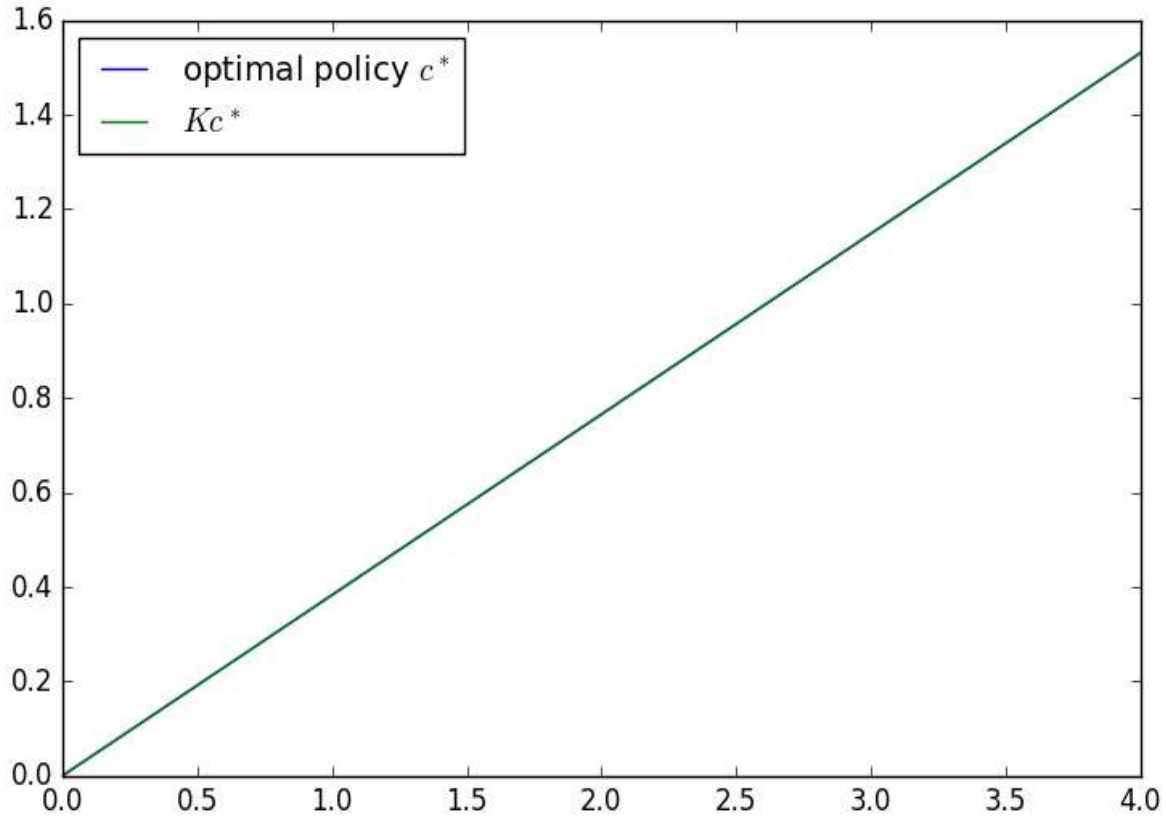
We also need a grid and some shock draws for Monte Carlo integration

```
grid_max = 4          # Largest grid point
grid_size = 200       # Number of grid points
shock_size = 250      # Number of shock draws in Monte Carlo integral

grid = np.linspace(1e-5, grid_max, grid_size)
shocks = np.exp(mu + s * np.random.randn(shock_size))
```

As a preliminary test, let's see if $Kc^* = c^*$, as implied by the theory

```
c_star_new = coleman_operator(c_star(grid),
             grid, beta, u_prime, f, f_prime, shocks)

fig, ax = plt.subplots()

ax.plot(grid, c_star(grid), label="optimal policy $c^*$")
ax.plot(grid, c_star_new, label="$Kc^*$")

ax.legend(loc='upper left')
plt.show()
```

Here's the result:



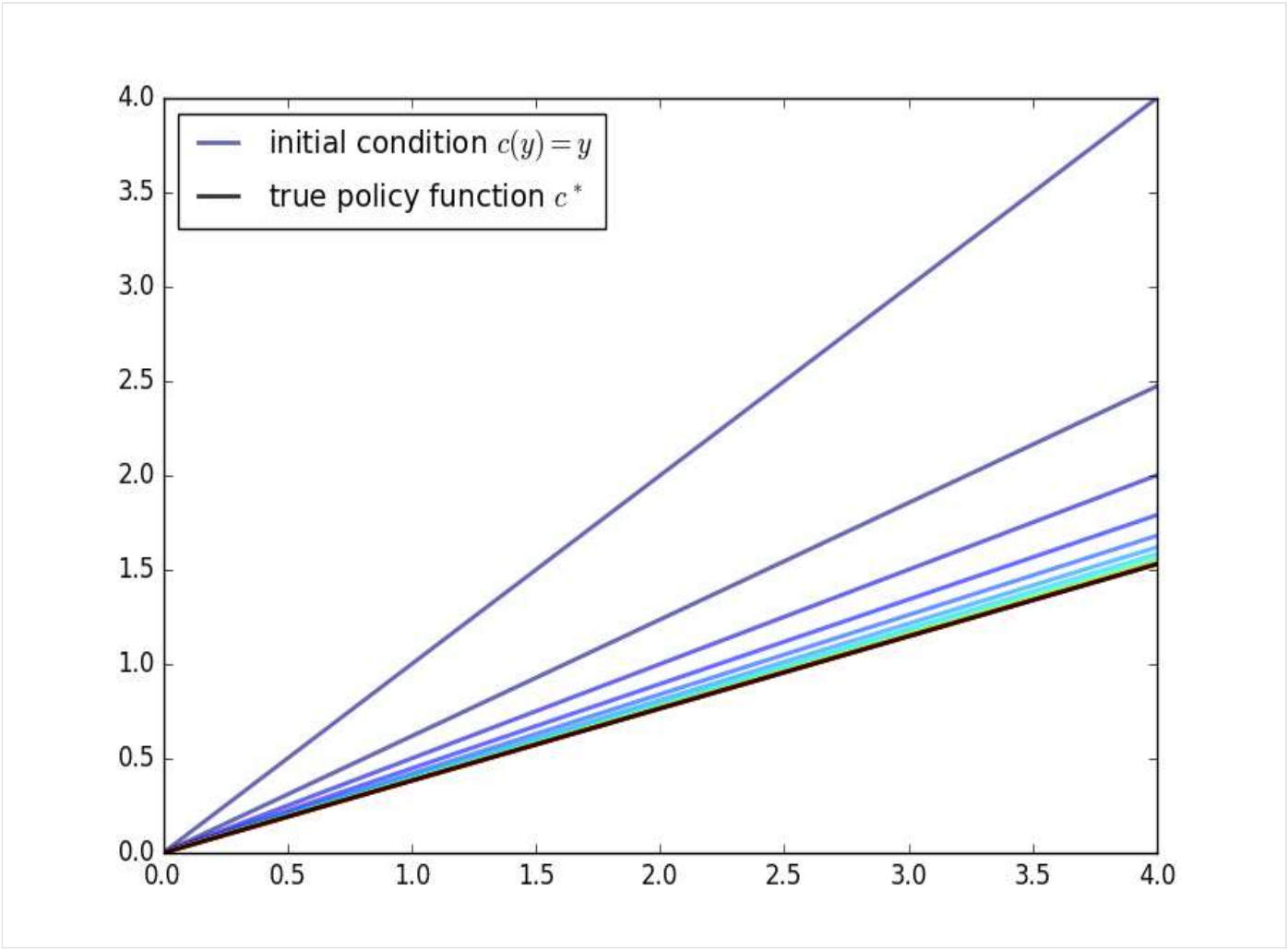
We can't really distinguish the two plots, so we are looking good, at least for this test

Next let's try iterating from an arbitrary initial condition and see if we converge towards c^*

The initial condition we'll use is the one that eats the whole pie: $c(y) = y$

```
g = grid
n = 15
fig, ax = plt.subplots(figsize=(9, 6))
lb = 'initial condition $c(y) = y$'
ax.plot(grid, g, color=plt.cm.jet(0), lw=2, alpha=0.6, label=lb)
for i in range(n):
    new_g = coleman_operator(g, grid, beta, u_prime, f, f_prime, shocks)
    g = new_g
    ax.plot(grid, g, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

lb = 'true policy function $c^*$'
ax.plot(grid, c_star(grid), 'k-', lw=2, alpha=0.8, label=lb)
ax.legend(loc='upper left')

plt.show()
```

We see that the policy has converged nicely, in only a few steps



Now let's compare the accuracy of iteration using the Coleman and Bellman operators

We'll generate

1. $K^n c$ where $c(y) = y$
2. $(M \circ T^n \circ M^{-1})c$ where $c(y) = y$

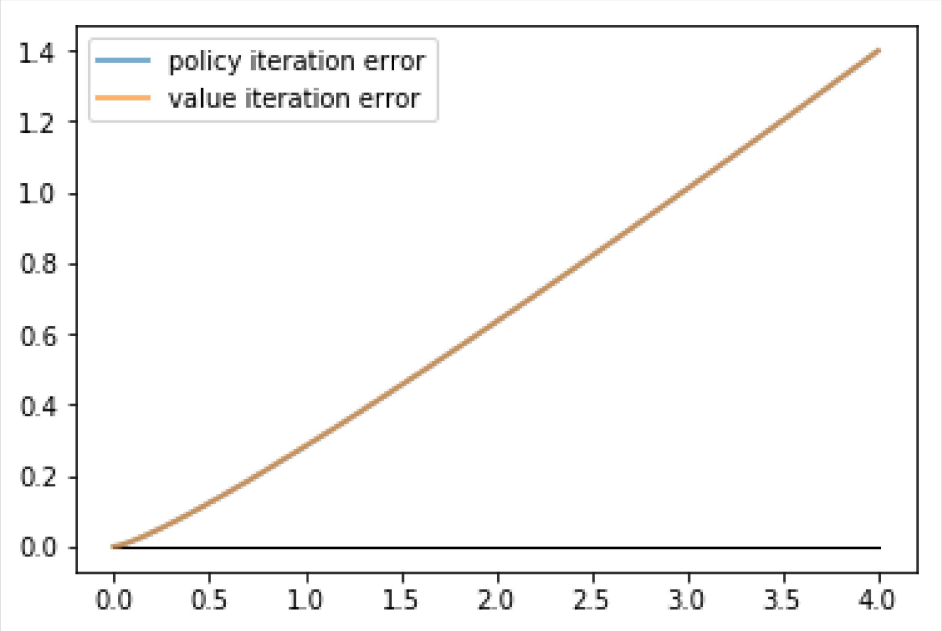
In each case we'll compare the resulting policy to c^*

The theory on equivalent dynamics says we will get the same policy function and hence the same errors

But in fact we expect the first method to be more accurate for reasons discussed above

```
g_init = grid
w_init = u(grid)
sim_length = 20

g, w = g_init, w_init
for i in range(sim_length):
    new_g = coleman_operator(g, grid, beta, u_prime, f, f_prime, shocks)
    new_w = bellman_operator(w, grid, beta, u, f, shocks)
    g, w = new_g, new_w

new_w, vf_g = bellman_operator(w, grid, beta, u, f, shocks, compute_policy=T

fig, ax = plt.subplots()

pf_error = c_star(grid) - g
vf_error = c_star(grid) - vf_g

ax.plot(grid, 0 * grid, 'k-', lw=1)
ax.plot(grid, pf_error, lw=2, alpha=0.6, label="policy iteration error")
ax.plot(grid, vf_error, lw=2, alpha=0.6, label="value iteration error")

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

Here's the result, which shows the errors in each case



As you can see, time iteration is much more accurate for a given number of iterations

Exercises

Exercise 1

Show that (9) is valid. In particular,

- Let v be strictly concave and continuously differentiable on $(0, \infty)$
- Fix $y \in (0, \infty)$ and show that $MTv(y) = KMv(y)$

Exercise 2

Show that M is a bijection from $\mathscr{V}$ to $\mathscr{P}$

Exercise 3

Consider the same model as above but with the CRRA utility function

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

Iterate 20 times with Bellman iteration and Euler equation time iteration

- start time iteration from $c(y) = y$
- start value function iteration from $v(y) = u(y)$
- set $\gamma = 1.5$

Compare the resulting policies and check that they are close

Exercise 4

Do the same exercise, but now, rather than plotting results, time how long 20 iterations takes in each case

Solutions

Solution to Exercise 1

Let T, K, M, v and y be as stated in the exercise

Using the envelope theorem, one can show that $(Tv)'(y) = u'(c(y))$ where $c(y)$ solves

$$u'(c(y)) = \beta \int v'(f(y - c(y))z) f'(y - c(y)) z \phi(dz) \tag{10}$$

Hence $MTv(y) = (u')^{-1}(u'(c(y))) = c(y)$

On the other hand, $KMv(y)$ is the $c(y)$ that solves

$$
\begin{aligned}
u'(c(y)) &= \beta \int (u' \circ (Mv))(f(y - c(y))z) f'(y - c(y)) z \phi(dz) \\
&= \beta \int (u' \circ ((u')^{-1} \circ v'))(f(y - c(y))z) f'(y - c(y)) z \phi(dz) \\
&= \beta \int v'(f(y - c(y))z) f'(y - c(y)) z \phi(dz)
\end{aligned}
$$

We see that $c(y)$ is the same in each case

Solution to Exercise 2

We need to show that M is a bijection from $\mathscr{V}$ to $\mathscr{P}$

To see this, first observe that, in view of our assumptions above, u' is a strictly decreasing continuous bijection from $(0, \infty)$ to itself

It follows that h has the same properties

Moreover, for fixed $v \in \mathscr{V}$, the derivative v' is a continuous, strictly decreasing function

Hence, for fixed $v \in \mathscr{V}$, the map $Mv = h \circ v'$ is strictly increasing and continuous, taking values in $(0, \infty)$

Moreover, interiority holds because v' strictly dominates u', implying that

$$
(Mv)(y) = h(v'(y)) < h(u'(y)) = y
$$

In particular, $\sigma(y) := (Mv)(y)$ is an element of $\mathscr{P}$

To see that each $\sigma \in \mathscr{P}$ has a preimage $v \in \mathscr{V}$ with $Mv = \sigma$, fix any $\sigma \in \mathscr{P}$

Let $v(y) := \int_0^y u'(\sigma(x)) dx$ with $v(0) = 0$

With a small amount of effort you will be able to show that $v \in \mathscr{V}$ and $Mv = \sigma$

It's also true that M is one-to-one on $\mathscr{V}$

To see this, suppose that v and w are elements of $\mathscr{V}$ satisfying $Mv = Mw$

Then $v(0) = w(0) = 0$ and $v' = w'$ on $(0, \infty)$

The fundamental theorem of calculus then implies that $v = w$ on $\mathbb{R}_+$

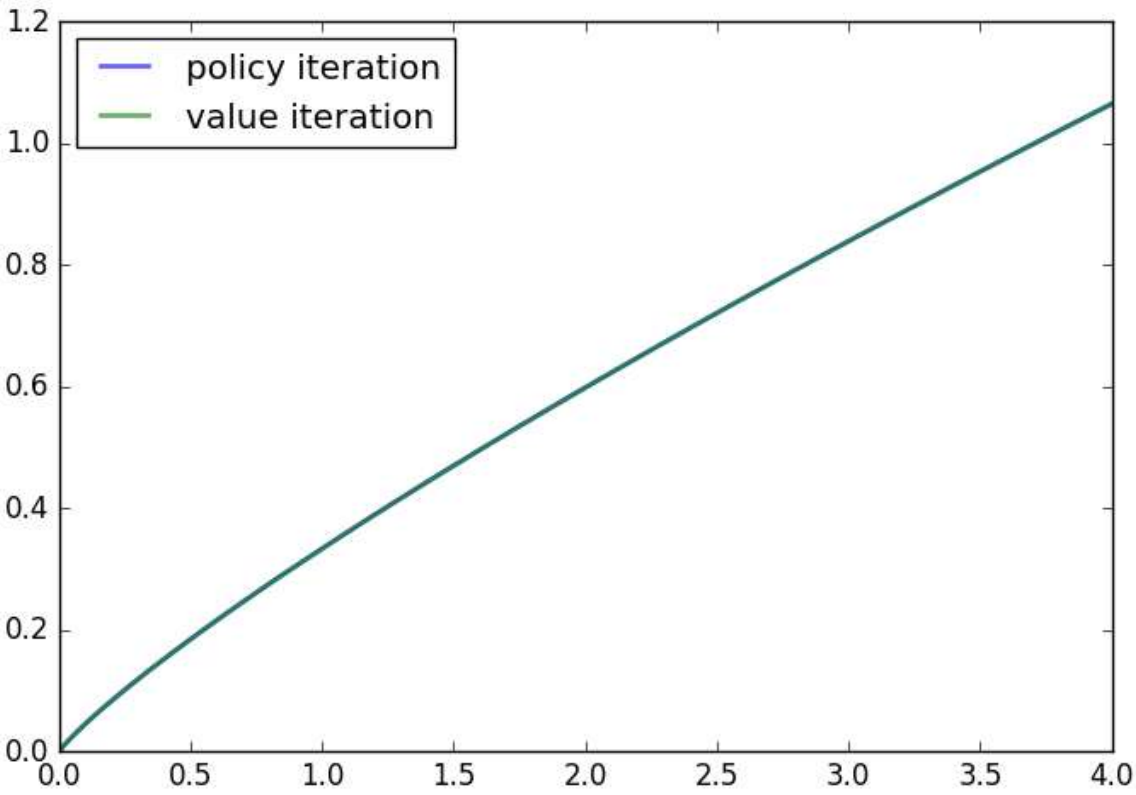
Solution to Exercise 3

Here's the code, which will execute if you've run all the code above

```
## Define the model

alpha = 0.65
beta = 0.95
mu = 0
s = 0.1
grid_min = 1e-6
grid_max = 4
grid_size = 200
shock_size = 250

gamma = 1.5   # Preference parameter

def f(k):
    return k**alpha

def f_prime(k):
    return alpha * k**(alpha - 1)

def u(c):
    return (c**(1 - gamma) - 1) / (1 - gamma)

def u_prime(c):
    return c**(-gamma)

grid = np.linspace(grid_min, grid_max, grid_size)
shocks = np.exp(mu + s * np.random.randn(shock_size))

## Let's make convenience functions based around these primitives

def crra_bellman(w):
    return bellman_operator(w, grid, beta, u, f, shocks)

def crra_coleman(g):
    return coleman_operator(g, grid, beta, u_prime, f, f_prime, shocks)

## Iterate with K and T, compare policies

g_init = grid
w_init = u(grid)
sim_length = 20

g, w = g_init, w_init
for i in range(sim_length):
    new_g = crra_coleman(g)
    new_w = crra_bellman(w)
    g, w = new_g, new_w

new_w, vf_g = bellman_operator(w, grid, beta, u, f, shocks, compute_policy=T
```

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

ax.plot(grid, g, lw=2, alpha=0.6, label="policy iteration")
ax.plot(grid, vf_g, lw=2, alpha=0.6, label="value iteration")

ax.legend(loc="upper left")
plt.show()
```

Here's the resulting figure



The policies are indeed close

Solution to Exercise 4

Here's the code

It assumes that you've just run the code from the previous exercise

```
g_init = grid
w_init = u(grid)
sim_length = 100

print("Timing value function iteration")

w = w_init
qe.util.tic()
for i in range(sim_length):
    new_w = crra_bellman(w)
    w = new_w
qe.util.toc()


print("Timing Euler equation time iteration")

g = g_init
qe.util.tic()
for i in range(sim_length):
    new_g = crra_coleman(g)
    g = new_g
qe.util.toc()
```

If you run this you'll find that the two operators execute at about the same speed

However, as we saw above, time iteration is numerically far more accurate for a given number of iterations

Next topic
The Endogenous Grid Method

Previous topic
A Stochastic Optimal Growth Model