

## Lab 1

# Value Function Iteration

**Lab Objective:** *This section teaches the fundamentals of Dynamic Programming using value function iteration.*

Often it is of interest to optimize decision making in some sequential process. For example, an oil company may need to decide how much oil to excavate and sell each month as prices change, a person entering retirement may need to decide how much of their savings to spend each year, or a model of economic growth may require a decision about how much to invest in capital versus how much to spend each year. In this lab we will formulate a general dynamic optimization problem. We will explore techniques for solving such a problem with both finite and infinite time horizons.

## The Sequential Problem, Finite Horizon

Suppose there are time periods  $t = 0, 1, \dots, T$  and at each time period we take an action  $c_t$ . Furthermore, at the beginning of each time period  $t$  we are in some state  $W_t$ . In many cases  $W_t$  might represent an available resource, such as money. At each time we receive some reward,  $u(W_t, c_t)$ , for taking action  $c_t$  given state  $W_t$ . We assume that rewards are worth more now than later. We let  $\beta \in (0, 1)$  represent what is called the discount factor, which gives the ratio of preference for rewards today versus rewards tomorrow. For example, receiving a dollar today is preferable to receiving a dollar in a year because taking a dollar today and putting it into a savings account results in having more than a dollar in a year. Lastly, over time our state variable  $W_t$  changes according to some rule depending on the previous state and our actions,

$$W_{t+1} = g(W_t, c_t). \tag{1.1}$$

Equation (1.1) is sometimes referred to as the law of motion, as it describes how we move from state to state. Mathematically such a problem can be represented as follows:

$$\text{maximize } \sum_{t=0}^T \beta^t u(W_t, c_t) \quad \text{s.t.} \quad W_{t+1} = g(W_t, c_t)$$

where our initial state,  $W_0$ , is given. There may also be restrictions on our choices  $c_t$ . For example, in many applications the state  $W_t$  represents the amount of some resource available, and  $c_t$  represents the amount we use up in time period  $t$ . In this case we would require  $c_t \in [0, W_t]$ .

For simplicity, let's assume that  $u$  is a function of  $c_t$  only (this is often, though not always, the case in practice). First let's consider the case that  $T = 0$ . So we maximize

$$u(c_0) \tag{1.2}$$

over  $c_0 \in [0, W_0]$ . In most cases  $u$  is increasing, which we will assume here. In this case it will be optimal to choose the largest value of  $c_0$  possible, that is,  $c_0 = W_0$ . Thinking of  $W_0$  as our available resources, this simply means that if we don't have future periods to consider, we will use all of it.

In fact, this is always true in the last period. In a problem with  $T$  periods we know that we will use all of our resources remaining in period  $T$ . Consider the two period problem:

$$\max \{u(c_0) + \beta u(c_1)\} \tag{1.3}$$

where  $c_0 \in [0, W_0]$ ,  $c_1 \in [0, W_1]$  and  $W_1 = g(W_0, c_0)$ . We know that in the last period we will use all of our remaining resources, so  $c_1 = W_1 = g(W_0, c_0)$ . Substituting gives

$$\max \{u(c_0) + \beta u(g(W_0, c_0))\}. \tag{1.4}$$

Now we need only determine  $c_0$ . Taking the derivative of (1.4) with respect to  $c_0$  and setting equal to zero gives the first order condition

$$u'(c_0) = -\beta u'(g(W_0, c_0))g_c(W_0, c_0) \tag{1.5}$$

where  $g_c$  is the partial derivative of  $g$  with respect to  $c_0$ .

Given a specific form for  $u$  we could solve for  $W_1$  and obtain the optimal solution. In fact, we can solve a problem of any length  $T$  in this manner, by starting at the last time period and working backward. We know that  $W_{T+1} = 0$ . Working backward in time we obtain an equation at each time step  $t < T$  by taking the derivative with respect to  $c_t$  and setting the equation equal to zero. This process is called backward induction. The equations at each time step, such as equation (1.5), are sometimes called the inter-temporal Euler equations. These equations, along with  $c_T = W_T$ , make  $T + 1$  equations to go with our  $T + 1$  unknowns  $\{c_0, c_1, c_2, \dots, c_T\}$ , where we can use the law of motion (1.1) to relate the  $c_t$  and  $W_t$ .

## The Recursive Problem, Finite Horizon

Approaching the problem sequentially like this can be somewhat messy. The dynamic programming approach we consider now is more easily adaptable to many situations. The key to the dynamic programming approach is to define our optimization problem in terms of subproblems. Notice that if we are in time period  $t$ , we face a problem of exactly the same form as the problem at time 0. We are in some state  $W_t$ , and want to maximize the sum from  $t$  to  $T$ . With this idea in mind, we define a function  $V_t(W_t)$  called the value function. The function  $V_t$  gives the

value of entering time  $t$  in state  $W_t$  and making optimal decisions moving forward. So

$$V_{t-1}(W_{t-1}) = \max_{c_t} \{u(c_{t-1}) + \beta V_t(g(W_{t-1}, c_{t-1}))\}.$$

This is called the Bellman Equation. The key to this formulation is that we decide what to do in period  $t - 1$  with the assumption that our actions in the remaining periods will be optimal. This is called the principle of optimality.

Let us consider a specific example from economics called The Cake Eating Problem. Suppose  $W_t$  represents the amount of cake available at time  $t$ . At each time period we can choose how much to consume. What we eat,  $c_t$ , gives us a reward. What we save,  $W_t - c_t$ , does not give us a reward (until it is eaten in a later period). The law of motion (1.1) becomes

$$W_{t+1} = g(W_t, c_t) = W_t - c_t \quad (1.6)$$

Now we have completely defined the problem. The Bellman Equation is

$$V_{t-1}(W_{t-1}) = \max_{c_t} \{u(c_{t-1}) + \beta V_t(W_{t-1} - c_{t-1})\}.$$

Notice that by the law of motion, each  $c_t$  is determined by  $W_t$  and  $W_{t+1}$ . In fact, rearranging (1.6), we have

$$c_t = W_t - W_{t+1}.$$

We can therefore rewrite the value function as

$$V_{t-1}(W_{t-1}) = \max_{W_t} \{u(W_{t-1} - W_t) + \beta V_t(W_t)\}. \quad (1.7)$$

We see that determining the optimal actions  $c_t$  is equivalent to determining the optimal states  $W_t$  in the above formulation. The solution to this problem is often called a *policy function*. A policy function determines an action based on the current state. Denoting the policy function by  $\psi$ , this can be written as

$$W_{t+1} = \psi_t(W_t).$$

The policy function gives the optimal amount of cake to leave for the next period (equivalent to the amount of consumption) given the amount of cake at the start of the period. In other words, it determines the choice of  $W_t$  that satisfies the max condition in (1.7).

As before, we know that in the last time period we should not save anything. So  $V_{T+1}(W_{T+1}) = 0$ , i.e. there is no value in leaving wealth for period  $T + 1$ . Stated in another way, our action at time  $T$  should be to eat all of the remaining cake  $W_T$ , so  $W_{T+1} = \psi_T(W_T) = 0$ . Plugging this result into the Bellman Equation gives us  $V_T(W_T) = u(W_T)$ . Now consider the value function equation for period  $T - 1$ :

$$\begin{aligned} V_{T-1}(W_{T-1}) &= \max_{W_T} \{u(W_{T-1} - W_T) + \beta V_T(W_T)\} \\ &= \max_{W_T} \{u(W_{T-1} - W_T) + \beta u(W_T)\}. \end{aligned}$$

We can determine this value by optimizing over  $W_T$ , where  $0 \leq W_T \leq W_{T-1}$ . Continuing backwards in this manner leads us to the solution of the original problem.

**Problem 1.** We recommend reading the entire problem before beginning to work on it, as many questions may be addressed further on. This applies to the other problems in this lab as well.

Follow the steps below to solve the problem described above. Take  $u(c_t) = \sqrt{c_t}$ . You will write a function called `eatCake` that takes parameters  $\beta$  (the discount factor),  $N$  (the number of discrete cake values to consider),  $W_{max}$  (the original size of the cake, set to the default value of 1), a keyword argument `finite` (set to default value `True`), a keyword argument  $T$  (the number of time periods, set to default value `None`), and a keyword argument `plot`, which indicates whether or not to plot the computed results. The function should return arrays representing the value function and the policy function (we describe how to compute these in the following steps).

1. Approximate the continuum of possible cake sizes by creating an array of evenly-spaced values that range from 0 to  $W_{max}$  inclusive. Let the number of possible cake values be given by  $N$ . In Python, this can be accomplished easily by using the `linspace` function in NumPy. You should obtain an array (call it  $w$ ) of the form

$$w = (w_1, w_2, \dots, w_N),$$

where  $w_1 = 0$  and  $w_N = W_{max}$ .

2. Note that in order to compute the value function, we need  $u(W_{t-1} - W_t)$ . We will pre-compute all such possible values and store them in an array, as follows: Create an  $N$  by  $N$  matrix that contains all possible values of  $W_{t-1} - W_t$  (where  $W_{t-1}$  corresponds to rows and  $W_t$  to columns). Make sure that  $c_t \geq 0$  is satisfied by replacing negative entries in the matrix with zeros. Then take the square root to get a matrix of  $u(W_{t-1} - W_t)$ . To make sure we do not choose  $W_{t-1} - W_t < 0$  when maximizing, replace the corresponding entries of the  $u(W_{t-1} - W_t)$  matrix with a large negative number (e.g.  $-10^{10}$ ). You should end up with a matrix whose  $(i, j)$ -th entry is equal to  $\sqrt{w_i - w_j}$  when  $i \geq j$ , and is equal to  $-10^{10}$  when  $i < j$ .
3. Next, create an  $N$  by  $T+2$  (corresponding to  $t = 0, 1, \dots, T+1$ ) matrix representing the value function for a given time  $t$  and state  $W_t$ . We can initialize it with zeros and begin filling in the columns starting with the last (which we know is zeros), as explained below.
4. Now we are ready to iterate backward and compute the value function for each time period. To find  $V_T$ , we first compute  $u(W_T - W_{T+1}) + \beta V_{T+1}(W_{T+1})$  for all values of  $W_T$  and  $W_{T+1}$ . This will result in an  $N$  by  $N$  matrix where the rows correspond to values of  $W_T$  and the columns correspond to values of  $W_{T+1}$ .

Now we maximize over choices of  $W_{T+1}$  (choosing how much to save for the next period). Then we will have a row vector representing the value

function for period  $T$  across all possible  $W_{T+1}$ . Iterate this procedure to fill in the value function for all  $t = T + 1, T, \dots, 0$ .

5. In each iteration, you maximize to find the value function at time  $t$ . Save the values of  $W_{t+1}$  that achieve the maximum. The result is an  $N$  by  $T + 1$  matrix whose  $(n, t)$  entry gives the optimal amount of cake to leave for period  $t + 1$ , given that we start period  $t$  with the  $n$ -th value of our vector of cake. This is the policy function.
6. If the keyword argument `plot` is set to `True`, plot the surface of the Value and Policy functions. This can be done by including the following import lines

```
>>> from matplotlib import pyplot as plt
>>> from matplotlib import cm
>>> from mpl_toolkits.mplot3d import Axes3D
```

and using the following code:

```
>>> W = np.linspace(0, Wmax, N)
>>> x = np.arange(0, N)
>>> y = np.arange(0, T+2)
>>> X, Y = np.meshgrid(x, y)
>>> fig1 = plt.figure()
>>> ax1 = Axes3D(fig1)
>>> ax1.plot_surface(W[X], Y, np.transpose(V), cmap=cm.coolwarm)
>>> plt.show()

>>> fig2 = plt.figure()
>>> ax2 = Axes3D(fig2)
>>> y = np.arange(0, T+1)
>>> X, Y = np.meshgrid(x, y)
>>> ax2.plot_surface(W[X], Y, np.transpose(psi), cmap=cm.coolwarm)
>>> plt.show()
```

where  $w$  is the vector of cake amounts,  $v$  is the value function, and  $\psi$  is the policy function.

7. Return the arrays giving the value function and the policy function.

Solve the problem using cake size 1, discount factor  $\beta = .9$ , number of time periods  $T = 10$ , and number of discrete cake values  $N = 100$ . You should also try plotting the value and policy functions for fixed time periods across  $W_t$ , or for fixed  $W_t$  across time, and make sure that these plots fit your intuition. See Figure 1.1. Your output should agree with the figure.

## The Recursive Problem, Infinite Horizon

Next we consider an infinite horizon problem. For simplicity, we continue with the example from the previous section. Suppose that rather than optimizing over

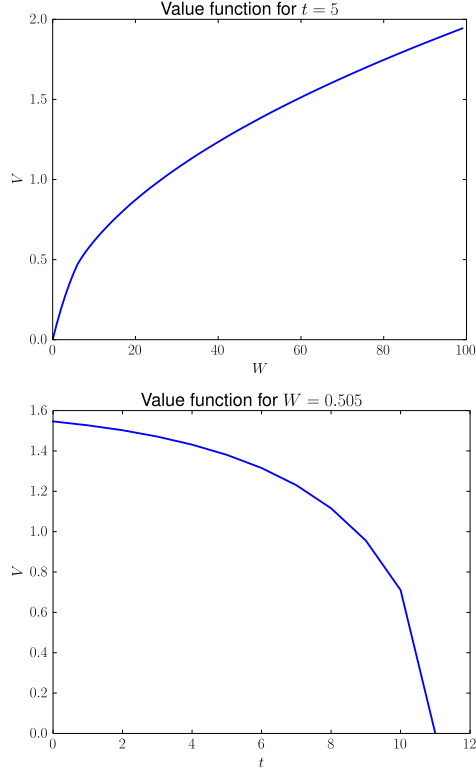


Figure 1.1: Slices of the finite horizon value function for fixed values of  $t$  and  $W$ , respectively.

$t = 0, 1, \dots, T$ , we wish to optimize over an infinite time horizon:

$$\text{maximize } \sum_{t=0}^{\infty} \beta^t u(W_t, c_t) \quad \text{s.t.} \quad W_{t+1} = g(W_t, c_t).$$

Since at any time  $t$ , there are an infinite number of periods remaining, one might suspect that the optimal policy will not depend on the current time  $t$ .

**Problem 2.** Compute the solution to Problem 1 with  $T = 1000$ , and the rest of the inputs the same. Plot the policy function across time for fixed  $W_t = 1$ . Notice that it is the same for all time periods, except those near the end time  $T$ .

As suggested by the results of Problem 2, the policy function for the infinite horizon problem does not depend on the time  $t$  (this can be proved). That is, at any time  $t$ , the optimal decision depends only on the amount of cake at the beginning of the period, not the value of  $t$ . So everything can now be written in terms of variables today and variables tomorrow. We will denote variables tomorrow

with a “'”.

$$V(W) = \max_{W' \in [0, W]} \{u(W - W') + \beta V(W')\} \quad (1.8)$$

Note that the value function  $V$  on the left-hand-side of (1.8) and on the right-hand-side are the same function.

Because the problem now has an infinite horizon, the nature of the solution is a little different. The solution to (1.8) is a policy function  $W' = \psi(W)$  that creates a fixed point in  $V$ . In other words, the solution is a policy function  $\psi(W)$  that makes the function  $V$  on the left-hand-side of (1.8) equal the function  $V$  on the right-hand-side.

Define  $C$  as an operator on any value function  $V_k(W)$ . Let  $C$  perform the following operation.

$$C(V_k(W)) \equiv \max_{W' \in [0, W]} \{u(W - W') + \beta V_k(W')\}. \quad (1.9)$$

Note that the value function on the right-hand-side of (1.9) and on the left-hand-side are the same function  $V_k$ , but have different inputs— $W$  versus  $W'$ . The operator  $C$  takes in a function  $V_k$ , and gives a new function which we will call  $V_{k+1}$ :

$$V_{k+1}(W) \equiv C(V_k(W)).$$

The value function  $V_{k+1}$  that results from the operation  $C$  is not necessarily the same as the value function that the system began with ( $V_k$ ). However, according to equation (1.8) we seek a  $V$  such that  $C(V) = V$ . The solution, then, is the fixed point in  $V$ .

$$C(V_k(W)) = V_{k+1}(W) = V_k(W) = V(W)$$

When trying to solve a fixed point equation, it is often very helpful to utilize the Contraction Mapping Principle, which guarantees the existence of a fixed point of a mapping, provided that the map sends any two distinct inputs to outputs that are strictly closer to each other than the inputs, in a controlled way. This principle also provides a constructive way to obtain the fixed point, namely by iterating the map. Fortunately, it can be shown that if  $u(\cdot)$  is real-valued, continuous, and bounded,  $\beta \in (0, 1)$ , and that the constraint set  $W' \in [0, W]$  is nonempty, compact-valued, and continuous, then the operator  $C$  is a contraction and thus we can obtain a solution  $V$  by iteration:

$$\lim_{k \rightarrow \infty} C^k(V_0(W)) = V_k(W) = V(W)$$

for any  $V_0$ .

Remember, in the infinite horizon problem both the value and policy functions do not depend on time. Computationally, this means that the value and policy functions in the infinite horizon problem are one dimensional.

**Problem 3.** Expand your `eatCake` function to solve the Cake Eating Problem with an infinite time horizon. If the keyword argument `finite` has the value `True`, then your function should behave as in Problem 1, solving the finite time horizon problem. However, if `finite = False`, solve the infinite time

horizon problem through the following steps. Both problems will require you to pre-compute the values  $u(W - W')$ , where  $W$  and  $W'$  range over the set of discrete cake amounts. Be sure to avoid replicating code by factoring it out. As in Problem 1, take  $u(c_t) = \sqrt{c_t}$ .

1. As in Problem 1, approximate the continuum of possible cake sizes by a column vector called  $W$  that ranges from 0 to  $W_{max}$  in  $N$  steps.
2. Initialize the value function  $V$  as a vector of zeros of length  $N$ . This is  $V_0$ . Perform one iteration of the contraction operation given in equation (1.9) to get a new value function  $V_1$  (this should be very similar to Problem 1). Determine the resulting policy function  $W' = \psi_1(W)$ . [HINT: The policy function should be a vector of length  $N$  of optimal future values of the cake  $W'$  given the current value of the cake  $W$ , and  $V_T$  should be an  $N$ -length vector representing the value of entering a period with cake size  $W$ .]
3. Measure the distance between the two value functions as the sum of the squared differences,

$$\delta_1 \equiv \|V_1(W) - V_0(W')\|_2^2 = (V_1 - V_0)^T (V_1 - V_0). \quad (1.10)$$

Defined in this way,  $\delta_1 \in [0, \infty)$ .

4. Write a loop that performs the contraction operation from steps 2 and 3 iteratively until the distance measure is very small ( $\delta_k < 10^{-9}$ ). The distance measure  $\delta_k$  being arbitrarily close to zero means you have converged to the fixed point  $V_k = V_{k+1} = V$ . (For fun, you can show that the policy function converges to the same function regardless of what you put in for your initial policy function value.)
5. If `plot = True`, plot the converged policy function vector ( $y$ -axis) as a function of the cake amounts ( $x$ -axis).
6. Return the value function and policy function arrays.

Compute the value function and policy function for the infinite time horizon problem with cake size 1, discount factor  $\beta = .9$ , and number of discrete cake values  $N = 100$ . The plot you generate should agree with Figure 1.2.

## Infinite Horizon, Stochastic, i.i.d.

In practice, dynamic programming problems often involve some level of uncertainty. For example, as time progresses prices may fluctuate, resources may vary, or preferences themselves may change. In this lab, we reexamine the cake eating problem, this time allowing for uncertainty.

We consider again the problem of optimizing a sequence of decisions over an infinite time horizon. We assume that the individual's preferences deviate each



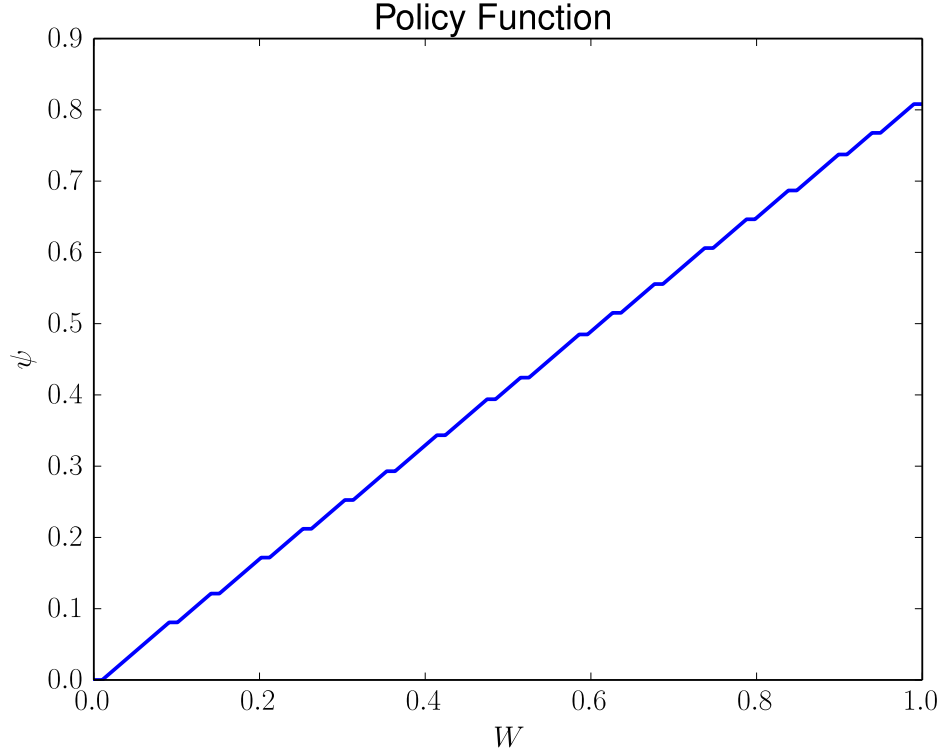


Figure 1.2: Policy function for infinite time horizon.

period according to some “shock”  $\varepsilon$ , where  $\varepsilon$  is a random variable. We assume that the shock terms  $\varepsilon$  for each time period are independent and identically distributed (i.i.d.). In effect, this means the probabilities associated with the  $\varepsilon$  are the same for any time  $t$  and do not depend on each other. We assume for now that the  $\varepsilon$  are distributed normally with mean  $\mu$  and variance  $\sigma^2$ . The Bellman equation can be easily rewritten in the following way to incorporate the uncertainty,

$$V(W, \varepsilon) = \max_{W' \in [0, W]} \{ \varepsilon u(W - W') + \beta E_{\varepsilon'} [V(W', \varepsilon')] \}, \quad (1.11)$$

where  $\varepsilon \sim N(\mu, \sigma^2)$  and  $E$  is the unconditional expectation operator over  $\varepsilon$ . Note that now the value function depends on two variables. It represents the value of entering the period with  $W$ , the amount of cake, and a preference shock of  $\varepsilon$ . For example, in a period where the realization of  $\varepsilon$  is higher, we will get more value from the cake eaten in the current period. Because we do not know the value of the shock in the next period  $\varepsilon'$ , we consider only the expected value for future time.

As it turns out, we can solve this problem in a manner similar to the infinite horizon deterministic cake-eating problem considered in the Value Function Iteration lab. It is worth noting that in this case, the value and policy functions will be two dimensional, as they will depend on both  $W$  and  $\varepsilon$ .

In order to deal with  $\varepsilon$  computationally, we would like to represent it as a vector of possible values it could take, along with the corresponding probabilities that it

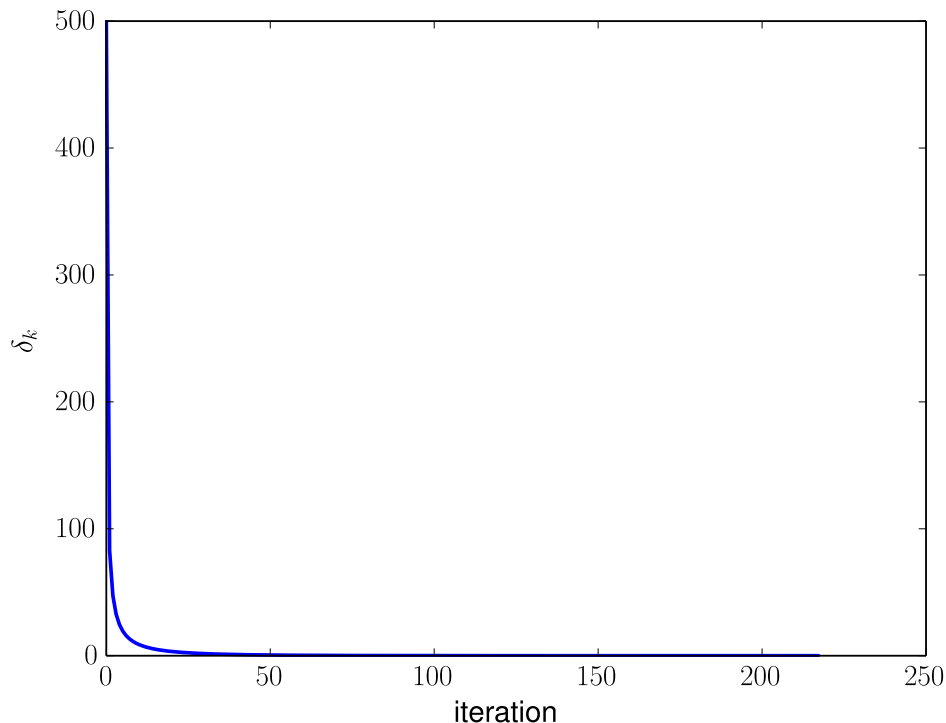


Figure 1.3: Due to the contraction mapping principle,  $\delta_k$  decreases as we perform iterations until it is small enough to meet our convergence tolerance.

takes each of those values. However,  $N(\mu, \sigma^2)$  is a continuous distribution, so we cannot represent every value  $\varepsilon$  could take. We need a discrete distribution that approximates  $N(\mu, \sigma^2)$ .

To do this, we choose  $N$  equally spaced points centered about the mean at which to approximate the distribution. Call these values  $\varepsilon_1, \dots, \varepsilon_N$ , and let the spacing between adjacent points be given by  $\delta$ . We can then break up the support of the distribution into  $N$  bins, where adjacent bins share a common endpoint. Call the endpoints of these bins  $v_1, \dots, v_{N+1}$ . By choosing the endpoints to be halfway between each  $\varepsilon_k$ , we have the formula

$$v_k = \varepsilon_k - \frac{1}{2}\delta, \quad k = 1, \dots, N$$

and

$$v_{N+1} = \varepsilon_N + \frac{1}{2}\delta.$$

We can then associate  $\varepsilon_k$  with the area under the curve from  $v_k$  to  $v_{k+1}$ . In Python, we can find the area using the function `norm.cdf` found in the `stats` package. The cdf (cumulative distribution function) gives the area under the curve from  $-\infty$  to a specified value. For example, in the following code, `eps` is the area under the curve from 0 to 1.

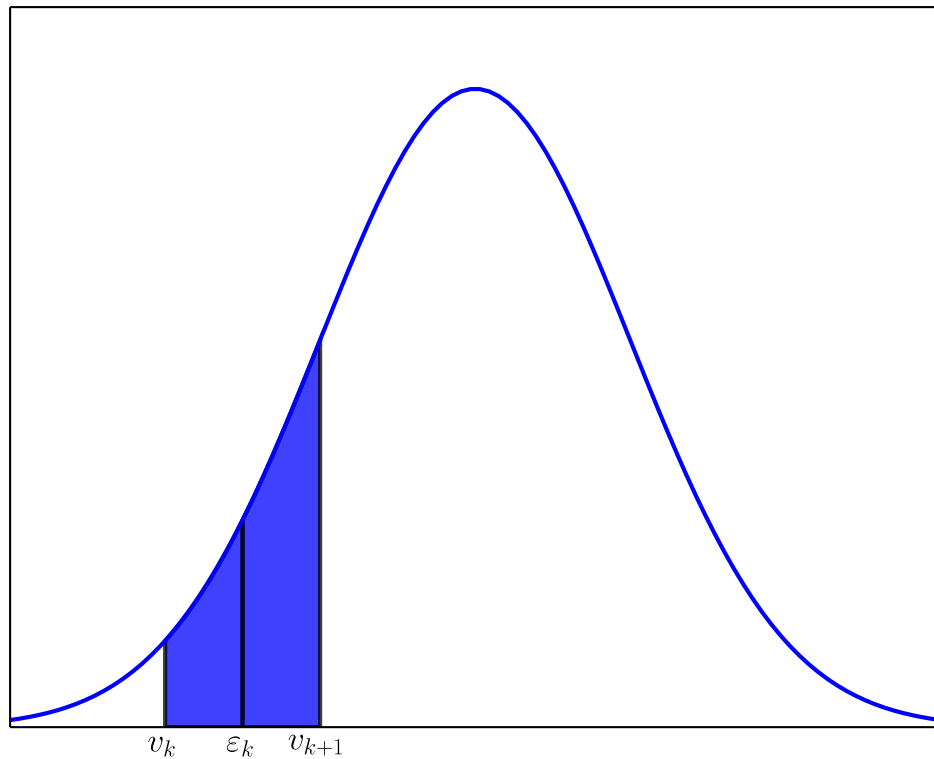


Figure 1.4: Discretization of  $N(\mu, \sigma^2)$ . We approximate  $P(\varepsilon = \varepsilon_k)$  by the area of the shaded region.

```
>>> from scipy import stats as st
>>> mu = 0
>>> sigma = 1
>>> eps = st.norm.cdf(1,loc=mu,scale=sigma) - st.norm.cdf(0,loc=mu,scale=sigma)
```

In general, it is sufficient to take our points  $\varepsilon_k$  ranging from  $\mu - 3\sigma$  to  $\mu + 3\sigma$ , as this range contains about 99.7% of the probability mass.

**Problem 4.** Write a function called `discretenorm` that accepts an integer  $K$  representing the number of discrete points desired, a mean  $\mu$ , and a standard deviation  $\sigma$ . It should return a length- $K$  vector of equally-spaced values ranging from  $\mu - 3\sigma$  to  $\mu + 3\sigma$  inclusive, and a length- $K$  vector containing the associated probabilities. Plot the approximation of  $N(0, 1)$  using different values of  $K$  to check that your results are plausible.

Now that we have a discrete distribution for  $\varepsilon$ , we can solve for the value and policy functions determined by (1.11).

**Problem 5.** Complete the following steps to solve the problem described above. Assume that the period utility function is  $u(c) = \sqrt{c}$ . Write a function `stochEatCake` that accepts parameters  $\beta$  (discount factor),  $N$  (number of discrete cake values), a tuple of values `e_params`,  $W_{max}$  (the original size of the cake, set to default value 1), a keyword argument `iid` (set to default value `True`), and a keyword argument `plot` (set to default value of `False`). Inside the function, carry out the steps outlined below.

The argument `e_params` is a tuple consisting of the values needed to generate the discrete approximation to  $\varepsilon$ . In the present case, this tuple consists (in order) of  $K$  (the number of discrete approximations of  $\varepsilon$ ),  $\mu$  (the mean of the shock term  $\varepsilon$ ), and  $\sigma$  (the standard deviation of the shock term  $\varepsilon$ ), since these are the arguments we need to pass to our `discretenorm` function.

1. First, compute an approximation of  $\varepsilon$  using the `discretenorm` function created in Problem 1. Use  $K$  equally spaced points to approximate  $N(\mu, \sigma^2)$ . Denote the resulting  $K$ -length vector of equally-spaced values by

$$e = (e_1, \dots, e_K),$$

and denote the  $K$ -length vector of the associated probabilities by

$$\Gamma = (\Gamma_1, \dots, \Gamma_K).$$

Note that  $\Gamma_k$  give the probability  $P(\varepsilon = e_k)$ .

Since the values needed for the `discretenorm` function are contained in the `e_params` input, we can feed these values directly into the function in the following way:

```
>>> e, gamma = discretenorm(*e_params)
```

The `*` operator essentially unpacks the values of a tuple or list.

2. As done previously, create a vector

$$w = (w_1, \dots, w_N)$$

of possible cake sizes. This should be a length- $N$  vector of equally spaced values from 0 to  $W_{max}$ , inclusive.

3. Represent the value function as a  $N \times K$  matrix  $v$ , satisfying

$$v_{i,j} = V(w_i, e_j).$$

(The rows correspond to different values of  $W$  and the columns correspond to different values of  $\varepsilon$ .) Initialize each entry of the matrix to 0.

Likewise, represent the policy function as a  $N \times K$  matrix  $p$ , satisfying

$$p_{i,j} = \psi(w_i, e_j).$$

Initialize all entries to 0.

4. In order to evaluate the value function equation, we need to pre-compute  $\varepsilon u(W - W')$  for all values of  $\varepsilon, W, W'$ . Begin by computing all possible values of  $u(W - W')$ , and storing these values in a  $N \times N$  array, as done before. Call this array  $u$ . Make sure that the upper triangular entries of this array are equal to zero, as these entries correspond to consuming more cake than is available, which is impossible.

The values  $\varepsilon u(W - W')$  will be represented by a three-dimensional array  $\hat{u}$  of size  $N \times N \times K$ , satisfying

$$\hat{u}_{i,j,k} = v_{i,j} e_k.$$

We can compute this array easily as follows:

```
>>> import numpy as np
>>> u_hat = np.repeat(u, K).reshape((N,N,K))*e
```

5. We also need to compute  $E_{\varepsilon'} [V(W', \varepsilon')]$  for each value of  $W'$ . The expected value is simply

$$E_{\varepsilon'} [V(W', \varepsilon')] = \sum_{k=1}^K \Gamma_k V(W', e'_k).$$

The result is a length  $N$  vector, call it  $E$ , satisfying

$$E_i = E_{\varepsilon'} [V(w_i, \varepsilon')] = \sum_{k=1}^K \Gamma_k v_{i,k}$$

This calculation can be done by multiplying  $\Gamma$  element-wise to each row of the value function matrix  $v$ , and then summing along the rows. Something like the following line of code should do the trick:

```
>>> E = (v*gamma).sum(axis=1)
```

6. We can now compute the value function contraction

$$C(V(W, \varepsilon)) \equiv \max_{W' \in [0, W]} \left\{ \varepsilon u(W - W') + \beta E_{\varepsilon'} [V(W', \varepsilon')] \right\}.$$

The first task is to create an  $N \times N \times K$  array  $c$  satisfying

$$c_{i,j,k} = \hat{u}_{i,j,k} + \beta E_j.$$

This can be done in any manner of ways. Below is a one-liner that does the job.

```
>>> c = np.swapaxes(np.swapaxes(u_hat, 1, 2) + beta*E, 1, 2)
```

Now, for any  $k$ , for all  $i < j$ , set  $c_{i,j,k}$  to a large negative number, say  $-10^{10}$ , so that when maximizing over this array, we do not choose to consume more cake than is available. Again, this can be done in a variety of different ways, but the following does the job concisely:

```
>>> c[np.triu_indices(N, k=1)] = -1e10
```

Finally, maximize over the second axis of  $c$  (which corresponds to different values of  $W'$ ) to obtain the updated value function matrix:

```
>>> v_new = np.max(c, axis=1)
```

You can likewise update your policy function matrix as follows:

```
>>> max_indices = np.argmax(c, axis=1)
>>> p = w[max_indices]
```

7. We now have our updated value function matrix  $v_{new}$  as well as the previous  $v$ , which we refer to here as  $v_{old}$ . As we iterate on the value function equation, we need a norm

$$\delta = \|v_{new} - v_{old}\|_2$$

that measures the distance between these two value functions to determine convergence. You may compute the norm using the SciPy function `scipy.linalg.norm`, or by direct calculation. At the end of each iteration, make sure to set  $v$  to  $v_{new}$ , so that the updates carry through the loop. Iterate on the contraction until  $\delta < 10^{-9}$ .

8. If `plot = True`, make a 3-D surface plot of the policy function for the converged problem  $W' = \psi(W, \varepsilon)$  which gives the value of the cake tomorrow as a function of the cake today and the taste shock today. Do the same for the value function. Example code to create the value function plot is provided below.

```
>>> x = np.arange(0,N)
>>> y = np.arange(0,K)
>>> X,Y = np.meshgrid(x,y)
>>> fig1 = plt.figure()
```

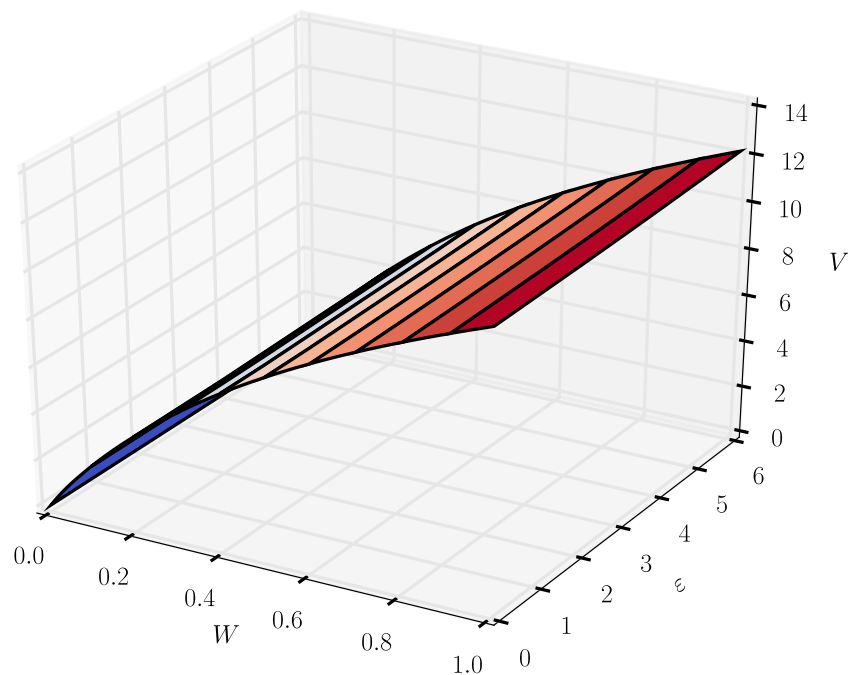


Figure 1.5: 3D surface representing the value function for the Stochastic Cake-Eating problem.

```
>>> ax1 = Axes3D(fig1)
>>> ax1.plot_surface(w[X], Y, v.T, cmap=cm.coolwarm)
>>> plt.show()
```

Creating the policy function plot is similar.

9. Return the converged value function matrix  $v$  and policy function matrix  $p$ .

Test your function using values  $\beta = .9$ ,  $N = 100$ ,  $K = 7$ ,  $\sigma = .5$ ,  $\mu = 4\sigma$ , and `plot = True`. The proper way to set this up and call the function is as follows:

```
>>> e_params = (7, 4*.5, .5)
>>> stuff = stochEatCake(.9, 100, e_params, plot=True)
```

## Infinite Horizon, Stochastic, AR(1)

In the previous example, we assumed that the shocks at time  $t$  were independent of what happened in previous periods. Often a shock may depend on recent events.

We will assume now that the shocks are persistent, meaning preferences in the current period are more likely to be close to what they were in the previous period. We can characterize the persistence by what is called an autoregressive process of order one, denoted AR(1). Such a process is defined as follows.

$$\varepsilon' = (1 - \rho)\mu + \rho\varepsilon + \nu' \quad \text{where} \quad \rho \in (0, 1) \quad \text{and} \quad \nu \sim N(0, \sigma^2). \quad (1.12)$$

Essentially, instead of allowing the shocks to have a mean which is independent of the past, the mean is now a weighted average (weighted by  $\rho$ ) of some  $\mu$  and the previous realization of the shock,  $\varepsilon$ . As it turns out, we can approximate this process by thinking of it as a Markov Chain. This means we need to determine a discrete set of points representing possible values of  $\varepsilon$  and a Markov transition matrix that gives the probabilities of moving from one value of  $\varepsilon$  to another. There are methods for determining the discrete approximation of  $\varepsilon$  with a Markov transition matrix. These methods are beyond the scope of this section, but you can use the file `tauchenhussey.py` to implement them in the next problem.

The Bellman equation becomes the following, in which the only change from the i.i.d. shock case is that the expectations operator is now conditional on the current shock  $\varepsilon$ :

$$V(W, \varepsilon) = \max_{W' \in [0, W]} \{ \varepsilon u(W - W') + \beta E_{\varepsilon' | \varepsilon} [V(W', \varepsilon')] \},$$

where  $\varepsilon'$  is distributed according to (1.12). Let  $\Gamma_{i,j} = P(\varepsilon'_j | \varepsilon_i)$  where  $\varepsilon'_j$  is the value of the shock in the next period and  $\varepsilon_i$  is the value of the shock in the current period. In other words,  $\Gamma$  is the Markov transition matrix.

The solution to this problem is of the same type as that in the i.i.d. case, since the only difference is the probability distributions of the  $\varepsilon$ .

**Problem 6.** Expand your `stochEatCake` function to handle the case of AR(1) shock terms. The function should handle this case for the parameter value `iid = False`, and should handle the previous case of normally distributed i.i.d. shock terms for the parameter value `iid = True`. You will need to add a few “if ... else” statements, as well as implement the steps outlined below, but most of the code will remain unchanged.

1. In the AR(1) case, the `e_params` argument should be a tuple of values needed to generate the arrays  $e$  and  $\Gamma$  that approximate the values and distribution of  $\varepsilon$  as a Markov chain. Use the file `tauchenhussey.py` to calculate these arrays. The provided Python function `tauchenhussey` produces the vector  $e$  of length  $M$  and an  $M \times M$  transition matrix  $\Gamma$ . Thus, you simply need the following lines of code, similar to the previous case.

```
>>> from tauchenhussey import tauchenhussey
>>> e, gamma = tauchenhussey(*e_params)
```

2. Because our values for  $e$  and  $\Gamma$  are different in the AR(1) case than in



the i.i.d. case, we must compute the expectation in a different manner. In particular, we need to compute the conditional expectation

$$E_{\varepsilon'|\varepsilon} \left[ V(W', \varepsilon') \right].$$

We obtain a two-dimensional array, since the expectation depends on both  $W'$  and on  $\varepsilon$ . The expectation can be computed by the matrix multiplication  $v\Gamma^T$ . Your code should match the following.

```
>>> E = v.dot(gamma.T)
```

3. The last difference comes in computing the array  $c$ . Fortunately, it is easier in this case. Recall that  $c$  gives the values for

$$\varepsilon u(W - W') + \beta E_{\varepsilon'|\varepsilon} \left[ V(W', \varepsilon') \right].$$

The array  $\hat{u}$  contains the values for the first term in the expression, and the array  $E$  contains the values for the expectation term. Hence, we obtain  $c$  by simple addition. Array broadcasting makes this work without problems.

```
>>> c = u_hat + beta*E
```

You will still need to set the upper triangular entries of  $c$  to a large negative number, just as in the previous case.

Those are the only differences. Let the following code snippet be a guideline for how to implement these differences.

```
>>> if iid:
>>>     # compute E as outlined in the previous problem
>>> else:
>>>     # compute E as outlined in the current problem
```

Now test your function with  $\beta = .9$ ,  $N = 100$ , `iid = False`, and `plot = True`. As inputs to `tauchenhussey`, let  $K = 7$ , the mean of the process  $\mu = 4\sigma$ ,  $\rho = 1/2$ ,  $\sigma = 1/2$ , and

$$baseSigma = (0.5 + \frac{\rho}{4})\sigma + (0.5 - \frac{\rho}{4})\frac{\sigma}{\sqrt{1 - \rho^2}}.$$

Your `e_params` parameter will therefore be a tuple of values containing (in order)  $K$ ,  $\mu$ ,  $\rho$ ,  $\sigma$ , and `baseSigma`.