18 Value Function Iteration

Lab Objective: *Many questions have optimal answers that change over time. Sequential decision making problems are among this classification. In this lab you we learn how to solve sequential decision making problems, also known as dynamic optimization problems. We teach these fundamentals by solving the finite-horizon cake eating problem.*

Dynamic optimization answers different questions than optimization techniques we have studied thus far. For example, an oil company might want to know how much oil to excavate in one day in order to maximize profit. If each day is considered in isolation, then the strategy to optimize profit is to maximize excavation in order to maximize profits. However, in reality oil prices change from day to day as supply increases or decreases, and so maximizing excavation may in fact lead to less profit. On the other hand, if the oil company considers how production on one day will affect subsequent decisions, they may be able to maximize their profits. In this lab we explore techniques for solving such a problem.

The Cake Eating Problem

Rather than maximizing oil profits, we focus on solving a general problem that can be applied in many areas called the cake eating problem. Given a cake of a certain size, how do we eat it to maximize our enjoyment (also known as utility) over time? Some people may prefer to eat all of their cake at once and not save any for later. Others may prefer to eat a little bit at a time. These preferences are expressed with a utility function. Our task is to find an optimal strategy given a smooth, strictly increasing and concave utility function, u. Precisely, given a cake of size W and some amount of consumption $c_0 \in [0, W]$, the utility gained is given by

$$u(c_0).$$

For this lab we restrict our attention to utility functions that have the point $u(0) = 0$. Although any size of W could be used, for simplicity of this lab assume that W has size 1. To further simplify the problem assume that W is cut into N equally-sized pieces. If we want to maximize utility in one time period, we consume the entire cake. How do we maximize utility over several days?

Discount Factors

A person or firm typically has a *time preference* for saving or consuming. For example, a dollar today can be invested and yield interest, whereas a dollar received next year does not include the accrued interest. In this lab, cake in the present yields more utility than cake in the future. We can model this by multiplying future utility by a discount factor $\beta \in (0, 1)$. For example, if we were to consume c_0 cake at time 0 and c_1 cake at time 1, with $c_0 = c_1$ then the utility gained at time 0 is larger than the utility at time 1.

$$u(c_0) > \beta u(c_0).$$

The Optimization Problem

If we are to consume a cake of size W over $T + 1$ time periods, then our consumption at each step is represented as a vector

$$[c_0, c_1, \dots, c_T]^\mathsf{T}$$

where

$$\sum_{i=0}^{T} c_i = W.$$

This vector is called a *policy vector*. The optimization problem is to

$$\begin{aligned} \max_{c_t} \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{subject to } \sum_{t=0}^{T} c_t = W \\ c_t \geq 0. \end{aligned}$$

Problem 1. Write a function called `graph_policy()` that will accept a policy vector $\mathbf{c}$, a utility function $u(x)$, and a discount factor β. Return the total utility gained with the policy input. Also display a plot of the total cumulative utility gained over time. Ensure that the policy that the user passes in sums to 1. Otherwise, raise a `ValueError`. It might seem obvious what sort of policy will yield the most utility, but the truth may surprise you. See Figure 18.1 for some examples.

```
# The policy functions used in the Figure below.
>>> pol1 = np.array([1, 0, 0, 0, 0])
>>> pol2 = np.array([0, 0, 0, 0, 1])
>>> pol3 = np.array([0.2, 0.2, 0.2, 0.2, 0.2])
>>> pol4 = np.array([.4, .3, .2, .1, 0])
```

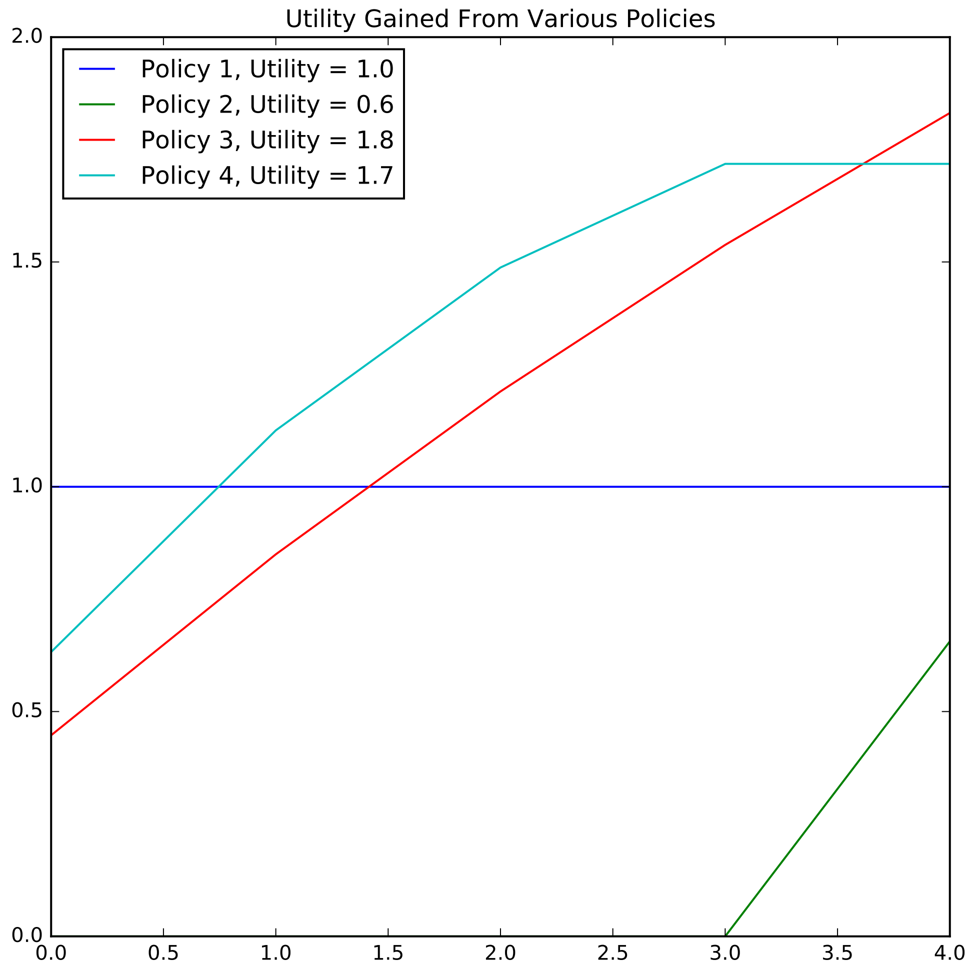

Figure 18.1: Plots for various policies with $u(x) = \sqrt{x}$ and $\beta = 0.9$. Policy 1 eats all of the cake in the first step while policy 2 eats all of the cake in the last step. Their difference in utility demonstrate the effect of the discount factor on waiting to eat. Policy 3 eats the same amount of cake at each step, while policy 4 begins by eating a lot of the cake but eats less and less as time goes on until the cake runs out.

The Value Function

The cake eating problem is an optimization problem and the value function is used to solve it. The value function $V(a, b, W)$ is the highest utility we can achieve.

$$
\begin{aligned}
V(a,b,W) = \max_{c_t} \sum_{t=a}^{b} \beta^t u(c_t) \\
\text{subject to } \sum_{t=a}^{b} c_t = W \\
c_t \geq 0.
\end{aligned}
$$

The value function gives the utility gained from following an optimal policy from time a to time b. $V(a,b,\frac{W}{2})$ gives how much utility we will gain by proceeding optimally from $t=a$ if half of a cake of size W was eaten before time $t=a$.

By using the optimal value in the future, we can determine the optimal value for the present. In other words, we must iterate backwards to solve the value problem. Let W_i represent the total amount of cake left at time $t=i$. Observe that $W_{i+1} \leq W_i$ for all i, because our problem does not allow for the creation of more cake. The value function can be expressed as

$$
V(t,T,W_t) = \max_{W_{t+1}} \left(u(W_t - W_{t+1}) + \beta V(t,T-1,W_{t+1}) \right). \tag{18.1}
$$

$u(W_t - W_{t+1})$ is the value gained from eating $W_t - W_{t+1}$ cake. $\beta V(t,T-1,W_{t+1})$ is the value of saving W_{t+1} cake until later. Recall that the utility function $u(x)$ and β are known.

In order to solve the problem iteratively, W is split into N equally-sized pieces, meaning that W_t only has $N+1$ possible values. Programmatically, $V(t,T,W_t)$ can be solved by trying each possible W_{t+1} and choosing the one that gives the highest utility. Knowing the maximum utility in the future allows us to calculate the maximum utility in the present.

Problem 2. Write a helper function to assist in solving the value function. Assume our cake has volume 1 and N equally-sized pieces. Write a method that accepts N and a utility function $u(x)$. Create a partition vector whose entries correspond to possible amounts of cake. For example, if split a cake into 4 pieces, the vector is

$$
\mathbf{w} = [0, 0.25, 0.5, 0.75, 1.0]^{\mathsf{T}}.
$$

Construct and return a matrix whose $(ij)^{th}$ entry is the amount of utility gained by starting with i pieces and saving j pieces (where i and j start at 0). In other words, the $(ij)^{th}$ entry should be $u(w_i - w_j)$.

Set impossible situations to 0 (i.e., eating more cake than you have available). The resulting lower triangular matrix is the *consumption matrix*.

For example, the following matrix results with $N=4$ and $u(x)=\sqrt{x}$.

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
u(0.25) & 0 & 0 & 0 & 0 \\
u(0.5) & u(0.25) & 0 & 0 & 0 \\
u(0.75) & u(0.5) & u(0.25) & 0 & 0 \\
u(1) & u(0.75) & u(0.5) & u(0.25) & 0
\end{bmatrix}.
$$

Solving the Optimization Problem

At each time t, W_t can only have $N+1$ values, which will be represented as $w_i = \frac{i}{N}$, which is i pieces of cake remaining. For example, if $N = 4$ then w_3 represents having three pieces of cake left and $w_3 = 0.75$.

The $(N+1) \times (T+1)$ matrix, A, that solves the value function is called the *value function matrix*. We will calculate the value function matrix step-by-step. A_{ij} is the value of having w_i cake at time j. Like the consumption matrix, i and j start at 0. It should be noted that $A_{0j} = 0$ because there is never any value in having w_0 cake, $u(w_0) = u(0) = 0$.

Initially we do not know how much cake to eat at $t = 0$: should we eat one piece of cake (w_1), or perhaps all of the cake (w_N)? Indeed there may be many scenarios to consider. It may not be obvious which option is best and that option may change depending on the discount factor β.

Instead of asking how much cake to eat at some time t, we should ask how valuable w_i cake is at time t? At some time t, there may be numerous decisions, but at the last time period, the only decision to make is how much cake to eat at $t = T$.

Since there is no value in having any cake left over when time runs out, the decision at time T is obvious: eat the rest of the cake. The amount of utility gained from having w_i cake at time T is given by $u(w_i)$. This utility is A_{iT}. Written in the form of (18.1),

$$A_{iT} = V(0, 0, w_i) = \max_{w_j} \left(u(w_i - w_j) + \beta V(0, -1, w_j)\right) = u(w_i). \tag{18.2}$$

This happens because $V(0, -1, w_j) = 0$. As mentioned, there is no value in saving cake so this equation is maximized when $w_j = 0$. All possible values of w_i are calculated so that the value of having w_i cake at time T is known.

Achtung!

Given a time interval from $t = 0$ to $t = T$ it is true that the true utility of waiting until time T to eat w_i cake is actually $\beta^T u(W_i)$, and can be verified by inspecting the difference of policies 1 and 2 in Figure 18.1. However, programmatically the problem is solved backwards by beginning with $t = T$ as an isolated state and calculating its value. This is why the value function above is $V(0, 0, W_i)$ and not $V(T, T, W_i)$. After calculating $t = T$, $t = T - 1$ is introduced, and its value is calculated by considering the utility from eating $w_i - w_j$ cake at time $t = T - 1$, plus the utility of β times the value of w_j at time T. We then proceed iteratively backwards, considering $t = T - 2$ and considering its utility plus the utility of β times the value at time $T - 1$.

Problem 3. Write a function called `eat_cake()` that accepts the stopping time T, the number of equal sized pieces that divides the cake N, a discount factor β, and a utility function $u(x)$. Return the value function matrix with all zeros except for the last column. The spec file indicates returning a policy matrix as well, for now return a matrix of zeros.

For example, the following matrix results with $T = 3$, $N = 4$, $\beta = 0.9$, and $u(x) = \sqrt{x}$.

$$\begin{bmatrix} 0 & 0 & 0 & u(0) \\ 0 & 0 & 0 & u(0.25) \\ 0 & 0 & 0 & u(0.5) \\ 0 & 0 & 0 & u(0.75) \\ 0 & 0 & 0 & u(1) \end{bmatrix}.$$

We can evaluate the next column of the value function matrix, $A_{i(T-1)}$, by modifying (18.2) as follows,

$$A_{i(T-1)} = V(0, 1, w_i) = \max_{w_j} \left(u(w_i - w_j) + \beta V(0, 0, w_j)\right) = \max_{w_j} \left(u(w_i - w_j) + \beta A_{jT}\right). \tag{18.3}$$

Remember that there is a limited set of possibilities for w_j, and we only need to consider options such that $w_j \leq w_i$. Instead of doing these one by one for each w_i, we can compute the options for each w_i simultaneously by creating a matrix. This information is stored in an $(N + 1) \times (N + 1)$ matrix known as the *current value matrix*, or CV^t, where the $(ij)^{th}$ entry is the value of eating $i - j$ pieces of cake at time t and saving j pieces of cake until the next period. For $t = T - 1$,

$$CV_{ij}^{T-1} = u(w_i - w_j) + \beta A_{jT}. \tag{18.4}$$

The largest entry in the i^{th} row of CV^{T-1} is the optimal value that the value function can attain at $T - 1$, given that we start with w_i cake. The maximal values of each row of CV^{T-1} become the column of the value function matrix, A, at time $T - 1$. Because we know the last column of A, we may iterate backwards to fill in the rest of A.

Achtung!

The notation CV^t does not mean raising the matrix to the t^{th} power, it indicates what time period we are in. All of the CV^t could be grouped together into a three-dimensional matrix, CV, that has dimensions $(N + 1) \times (N + 1) \times (T + 1)$. Although this is possible, we will not use CV in this lab, and will instead only consider CV^t for any given time t.

$$CV^2 = \begin{bmatrix} \boxed{0} & 0 & 0 & 0 & 0 \\ \boxed{0.5} & 0.45 & 0 & 0 & 0 \\ 0.707 & \boxed{0.95} & 0.636 & 0 & 0 \\ 0.866 & \boxed{1.157} & 1.136 & 0.779 & 0 \\ 1 & 1.316 & \boxed{1.343} & 1.279 & 0.9 \end{bmatrix}$$

This is CV^2 where $T = 3$, $\beta = .9$, $N = 4$, and $u(x) = \sqrt{x}$. The maximum value of each row, circled in red, is used in the 3^{rd} column of A. Remember that A's column index begins at 0, so the 3^{rd} column represents $t = 2$. See Figure 18.2.

Now that the column of A corresponding to $t = T - 1$ has been calculated, we repeat the process for $T - 2$ and so on until we have calculated each column of A. In summary, at each time step t, find

CV^t and then set A_{it} as the maximum value of the i^{th} row of CV^t. Generalizing (18.3) and (18.4) shows

$$CV^t_{ij} = u(w_i - w_j) + \beta A_{j(t+1)}. \qquad A_{it} = \max_j \left(CV^t_{ij} \right). \tag{18.5}$$

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \\ 0.95 & 0.95 & 0.95 & 0.707 \\ 1.355 & 1.355 & 1.157 & 0.866 \\ 1.7195 & 1.562 & 1.343 & 1 \end{bmatrix}.$$

Figure 18.2: The value function matrix where $T = 3$, $\beta = .9$, $N = 4$, and $u(x) = \sqrt{x}$. The bottom left entry indicates the highest utility that can be achieved is 1.7195.

Problem 4. Complete the function `eat_cake()` to determine the entire value function matrix. Starting from the next to last column, iterate backwards by

- calculating the current value matrix for time t using (18.5),
- finding the largest value in each row of the current value matrix,
- filling in the corresponding column of A with these values.

With the value function matrix constructed, the optimization problem is solved in some sense. The value function matrix contains the maximum possible utility to be gained. However, it is not immediately apparent what policy should be followed by only inspecting the value function matrix, A. The $(N + 1) \times (T + 1)$ policy matrix, P, is used to find the optimal policy. The $(ij)^{th}$ entry of the policy matrix indicates how much cake to eat at time j if we have i pieces of cake. Like A and CV, i and j begin at 0.

The last column of P is a straightforward calculation similar to last column of A. $P_{iT} = w_i$, because at time T we know that the remainder of the cake should be eaten. Recall that the column of A corresponding to t was calculated by the maximum values of CV^t. The column of P for time t is calculated by taking $w_i - w_j$, where j is the smallest index corresponding to the maximum value of CV^t,

$$P_{it} = w_i - w_j.$$

$$\text{where } j = \{\, min\{j\} \mid CV^t_{ij} \geq CV^t_{ik} \,\forall\, k \in [0, 1, \ldots, N] \,\}$$

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.5 \\ 0.25 & 0.25 & 0.5 & 0.75 \\ 0.25 & 0.5 & 0.5 & 1. \end{bmatrix}$$

An example of P where $T = 3$, $\beta = .9$, $N = 4$, and $u(x) = \sqrt{x}$. The optimal policy is found by starting at $i = N$, $j = 0$ and eating as much cake as the $(ij)^{th}$ entry indicates, as traced out by the red arrows. The blue arrows trace out the policy that would occur if we only had 2 time intervals. What would be the optimal policy if we had 3 time intervals?

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} \\ \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.5} \\ \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.25} & \sqrt{0.75} \\ \underline{\sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} + \beta^3\sqrt{0.25}} & \sqrt{0.5} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \underline{\sqrt{0.5} + \beta\sqrt{0.5}} & \sqrt{1} \end{bmatrix}$$

The non-simplified version of Figure 18.2. Notice that the value of A_{ij} is equal to following optimal path if you start at P_{ij}. A_{40} has the same values traced by the red arrows in P above and A_{42} has the same values traced by the blue arrows.

Problem 5. Modify `eat_cake()` to determine the policy matrix. Initialize the matrix as zeros and fill it in starting from the last column at the same time that you calculate the value function matrix. (Hint: You may find `np.argmax()` useful.)

Problem 6. The $(ij)^{th}$ entry of the policy matrix tells us how much cake to eat at time j if we start with i pieces. Use this information to write a function that will find the optimal policy for starting with a cake of size 1 split into N pieces given the stopping time T, the utility function u, and a discount factor β. Use `graph_policy()` to plot the optimal policy. See Figure 18.3 for an example.

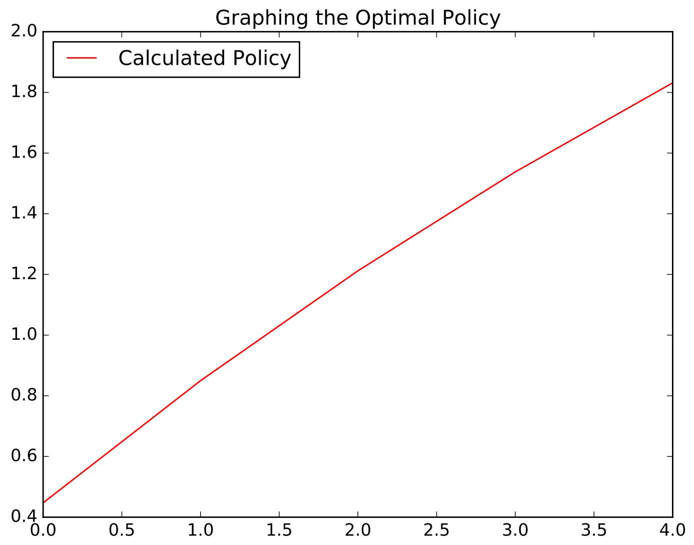

Figure 18.3: The graph of the optimal policy (Policy 3 from Figure 18.1) where $T = 4$, $\beta = .9$, $N = 5$, and $u(x) = \sqrt{x}$. It achieves a value of roughly 1.83.

A summary of the arrays generated in this lab is given below, in the order that they were generated in the lab:

Consumption matrix: Equal to $u(u_i - w_j)$, the utility gained when you start with i pieces and end with j pieces.

Value Function Matrix, A: How valuable it is to have w_i pieces at time j.

Current Value Matrix, CV^t: How valuable each possible decision is at time t.

Policy Matrix, P: The amount of cake you should eat at time t.