3 Numerical Analysis

3.1 Overview

This chapter reviews numerical methods used to solve dynamic programming problems. This discussion provides a key link between the basic theory of dynamic programming and the empirical analysis of dynamic optimization problems. The need for numerical tools arises from the fact that dynamic programming problems generally do not have tractable closed form solutions. Hence techniques must be used to approximate the solutions of these problems. We present a variety of techniques in this chapter that are subsequently used in the macroeconomic applications studied in part II of this book.

The presentation starts by solving a stochastic cake-eating problem using a procedure called **value function iteration**. The same example is then used to illustrate alternative methods that operate on the policy function rather than the value function. Finally, a version of this problem is studied to illustrate the solution to dynamic discrete choice problems.

The appendix and the Web page for this book contain the programs used in this chapter. The applied researcher may find these useful templates for solving other problems. In the appendix we present several tools, such as numerical integration and interpolation techniques, that are useful numerical methods.

A number of articles and books have been devoted to numerical programing. For a more complete description, we refer the reader to Judd (1998), Amman et al. (1996), Press et al. (1986), and Taylor and Uhlig (1990).

3.2 Stochastic Cake-Eating Problem

We start with the stochastic cake-eating problem defined by

$$V(W, y) = \max_{0 \le c \le W+y} u(c) + \beta E_{y'|y} V(W', y') \qquad \text{for all } (W, y) \tag{3.1}$$

with $W' = R(W - c + y)$. Here there are two state variables: W, the size of the cake brought into the current period, and y, the stochastic endowment of additional cake. This is an example of a stochastic dynamic programming problem from the framework in (2.5.2).

We begin by analyzing the simple case where the endowment is iid: the shock today does not give any information on the shock tomorrow. In this case the consumer only cares about the total amount that can be potentially eaten, $X = W + y$, and not the particular origin of any piece of cake. In this problem there is only one state variable X. We can rewrite the problem as

$$V(X) = \max_{0 \le c \le X} u(c) + \beta E_{y'} V(X') \qquad \text{for all } X \tag{3.2}$$

with $X' = R(X - c) + y'$.

If the endowment is serially correlated, then the agent has to keep track of any variables that allow him to forecast future endowment. The state space will include X but also current and maybe past realizations of endowments. We present such a case in section 3.3 where we study a discrete cake eating problem. Chapter 6.1 also presents the continuous cake-eating problem with serially correlated shocks.

The control variable is c, the level of current consumption. The size of the cake evolves from one period to the next according to the transition equation. The goal is to evaluate the value $V(X)$ as well as the policy function for consumption, $c(X)$.

3.2.1 *Value Function Iterations*

This method works from the Bellman equation to compute the value function by backward iterations on an initial guess. While sometimes slower than competing methods, it is trustworthy in that it reflects the result, stated in chapter 2, that (under certain conditions) the solution of the Bellman equation can be reached by iterating the value function starting from an arbitrary initial value. We illustrate this approach here in solving (3.2).[1]

1. We present additional code for this approach in the context of the nonstochastic growth model presented in chapter 5.

In order to program value function iteration, there are several important steps:

1. Choosing a functional form for the utility function.
2. Discretizing the state and control variable.
3. Building a computer code to perform value function iteration
4. Evaluating the value and the policy function.

We discuss each steps in turn. These steps are indicated in the code for the stochastic cake-eating problem.

Functional Form and Parameterization

We need to specify the utility function. This is the only known **primitive** function in (3.2): recall that the value function is what we are solving for! The choice of this function depends on the problem and the data. The consumption literature has often worked with a constant relative risk aversion (CRRA) function:

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

The vector θ will represent the parameters. For the cake-eating problem (γ, β) are both included in θ. To solve for the value function, we need to assign particular values to these parameters as well as the exogenous return R. For now we assume that $\beta R = 1$ so that the growth in the cake is exactly offset by the consumers discounting of the future. The specification of the functional form and its parameterization are given in part I of the accompanying Matlab code for the cake-eating problem.

State and Control Space

We have to define the space spanned by the state and the control variables as well as the space for the endowment shocks. For each problem, specification of the state space is important. The computer cannot literally handle a continuous state space, so we have to approximate this continuous space by a discrete one. While the approximation is clearly better if the state space is very fine (i.e., has many points), this can be costly in terms of computation time. Thus there is a trade-off involved.

For the cake-eating problem, suppose that the cake endowment can take two values, low (y_L) and high (y_H). As the endowment is assumed to follow an iid process, denote the probability a shock y_i by π_i for $i = L, H$. The probability of transitions can be stacked in a transition matrix:

$$\pi = \begin{bmatrix} \pi_L & \pi_H \\ \pi_L & \pi_H \end{bmatrix} \qquad \text{with } \pi_L + \pi_H = 1.$$

In this discrete setting, the expectation in (3.2) is just a weighted sum, so the Bellman equation can be simply rewritten as

$$V(X) = \max_{0 \le c \le X} u(c) + \beta \sum_{i=L,H} \pi_i V(R(X-c) + y_i) \qquad \text{for all } X.$$

For this problem it turns out that the natural state space is given by $[\bar{X}_L, \bar{X}_H]$. This choice of the state space is based on the economics of the problem, which will be understood more completely after studying household consumption choices. Imagine, though, that endowment is constant at a level y_i for $i = L, H$. Then, given the assumption $\beta R = 1$, the cake level of the household will (trust us) eventually settle down to $\bar{X}_i$ for $i = L, H$. Since the endowment is stochastic and not constant, consumption and the size of the future cake will vary with realizations of the state variable X, but it turns out that X will never leave this interval.

The fineness of the grid is simply a matter of choice too. In the program let n_s be the number of elements in the state space. The program simply partitions the interval $[\bar{X}_L, \bar{X}_H]$ into n_s elements. In practice, the grid is usually uniform, with the distance between two consecutive elements being constant.[2]

Call the state space Ψ_S, and let i_s be an index:

$$\Psi_S = \{X^{i_s}\}_{i_s=1}^{n_s} \qquad \text{with } X^1 = \bar{X}_L, X^{n_s} = \bar{X}_H.$$

The control variable c takes values in $[\bar{X}_L, \bar{X}_H]$. These are the extreme levels of consumption given the state space for X. We discretize this space into a grid of size n_c, and call the control space $\Psi_C = \{c^{i_c}\}_{i_c=1}^{n_c}$.

Value Function Iteration and Policy Function

Here we must have a loop for the mapping $T(v(X))$, defined as

$$T(v(X)) = \max_c \; u(c) + \beta \sum_{i=L,H} \pi_i v_j(R(X-c) + y_i). \tag{3.3}$$

2. In some applications it can be useful to define a grid that is not uniformally spaced; see the discrete cake-eating problem in section 3.3.

In this expression $v(X)$ represents a candidate value function that is a proposed solution to (3.2). If $T(v(X)) = v(X)$, then indeed $v(X)$ is the unique solution to (3.2). Thus the solution to the dynamic programming problem is reduced to finding a fixed point of the mapping $T(v(X))$.

Starting with an initial guess $v_0(X)$, we compute a sequence of value functions $v_j(X)$:

$$v_{j+1}(X) = T(v_j(X)) = \max_c \; u(c) + \beta \sum_{i=L,H} \pi_i v_j(R(X-c) + y_i).$$

The iterations are stopped when $|v_{j+1}(X) - v_j(X)| < \varepsilon$ for all i_s, where ε is a small number. As $T(.)$ is a contraction mapping (see chapter 2), the initial guess $v_0(X)$ does not influence the convergence to the fixed point, so one can choose $v_0(X) = 0$, for instance. However, finding a good guess for $v_0(X)$ helps to decrease the computing time. By the contraction mapping property, we know that the convergence rate is geometric, parameterized by the discount rate β.

Let us review in more detail how the iteration is done in practice. At each iteration, the values $v_j(X)$ are stored in a $n_s \times 1$ matrix:

$$\mathbf{V} = \begin{bmatrix} v_j(X^1) \\ \vdots \\ v_j(X^{i_s}) \\ \vdots \\ v_j(X^{n_s}) \end{bmatrix}.$$

To compute v_{j+1}, we start by choosing a particular size for the cake at the start of the period, X^{i_s}. We then search among all the points in the control space Ψ_C for the point where $u(c) + \beta E v_j(X')$ is maximized. We will denote this point $c^{i_c^*}$. Finding next period's value involves calculating $v_j(R(X^{i_s} - c^{i_c^*}) + y_i)$, $i = L, H$. With the assumption of a finite state space, we look for the value $v_j(.)$ at the point nearest to $R(X^{i_s} - c^{i_c^*}) + y_i$. Once we have calculated the new value for $v_{j+1}(X^{i_s})$, we can proceed to compute similarly the value $v_{j+1}(.)$ for other sizes of the cake and other endowment at the start of the period. These new values are then stacked in $\mathbf{V}$. Figure 3.1 gives an example of how this can be programmed on a computer. (Note that the code is not written in a particular computer language, so one has to adapt the code to the appropriate syntax. The code for the value function iteration piece is part III of the Matlab code.)

```
i_s=1
do until i_s>n_s                  * Loop over all sizes of the
                                    total amount of cake X *
  c_L=X_L                         * Min value for consumption *
  c_H=X[i_s]                      * Max value for consumption *
  i_c=1
    do until i_c>n_c              * Loop over all consumption
                                    levels *
    c=c_L+(c_H-c_L)/n_c*(i_c-1)
    i_y=1
    EnextV=0                      * Initialize the next value
                                    to zero *
    do until i_y>n_y              * Loop over all possible
                                    realizations of the future
                                    endowment *
    nextX=R*(X[i_s]-c)+Y[i_y]     * Next period amount of
                                    cake *
    nextV=V(nextX)                * Here we use interpolation
                                    to find the next value
                                    function *

    EnextV=EnextV+nextV*Pi[i_y]   * Store the expected future
                                    value using the transition
                                    matrix *
    i_y=i_y+1
    endo                          * End of loop over
                                    endowment *
    aux[i_c]=u(c)+beta*EnextV     * Stores the value of a given
                                    consumption level *
    i_c=i_c+1
    endo                          * End of loop over
                                    consumption *
  newV[i_s,i_y]=max(aux)          * Take the max over all
                                    consumption levels *
  i_s=i_s+1
  endo                            * End of loop over size of
                                    cake *
V=newV                            * Update the new value
                                    function *
```

Figure 3.1
Stochastic cake-eating problem

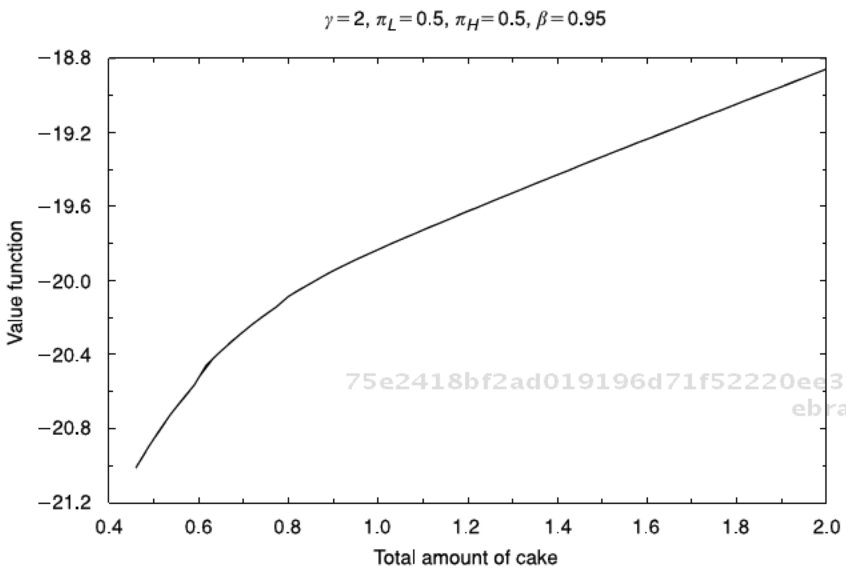

Figure 3.2
Value function, stochastic cake-eating problem

Once the value function iteration piece of the program is completed, the value function can be used to find the policy function, $c = c(X)$. This is done by collecting all the optimal consumption values $c^{i_c *}$ for every value of X^{i_s}. Here again, we only know the function $c(X)$ at the points of the grid. We can use interpolating methods to evaluate the policy function at other points. The value function and the policy function are displayed in figures 3.2 and 3.3 for particular values of the parameters.

As discussed above, approximating the value function and the policy rules by a finite state space requires a large number of points on this space (n_s has to be big). These numerical calculations are often extremely time-consuming. So we can reduce the number of points on the grid, while keeping a satisfactory accuracy, by using interpolations on this grid. When we have evaluated the function $v_j(R(X^{i_s} - c^{i_c^*}) + y_i)$, $i = L, H$, we use the nearest value on the grid to approximate $R(X^{i_s} - c^{i_c^*}) + y_i$. With a small number of points on the grid, this can be a very crude approximation. The accuracy of the computation can be increased by interpolating the function $v_j(.)$ (see the appendix for more details). The interpolation is based on the values in **V**.

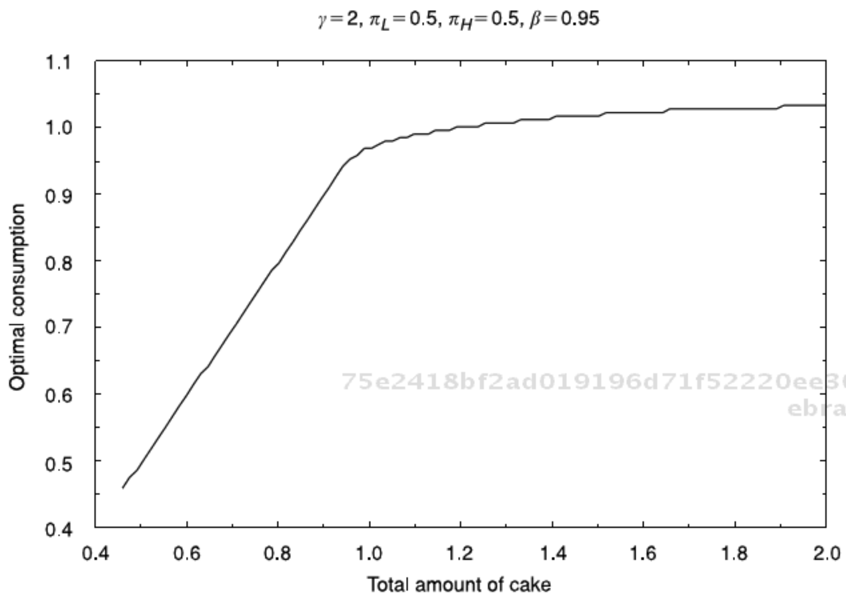

Figure 3.3
Policy function, stochastic cake-eating problem

3.2.2 *Policy Function Iterations*

The value function iteration method can be rather slow, as it converges at a rate β. Researchers have devised other methods that can be faster to compute the solution to the Bellman equation in an infinite horizon. The policy function iteration, also known as Howard's improvement algorithm, is one of these. We refer the reader to Judd (1998) or Ljungqvist and Sargent (2000) for more details.

This method starts with a guess of the policy function, in our case $c_0(X)$. This policy function is then used to evaluate the value of using this rule forever:

$$V_0(X) = u(c_0(X)) + \beta \sum_{i=L,H} \pi_i V_0(R(X - c_0(X)) + y_i) \qquad \text{for all } X.$$

This "policy evaluation step" requires solving a system of linear equations, given that we have approximated $R(X - c^0(X)) + y_i$ by an X on our grid. Next we do a "policy improvement step" to compute $c_1(X)$:

$$c_1(X) = \arg\max_c \left[u(c) + \beta \sum_{i=L,H} \pi_i V_0(R(X-c) + y_i) \right] \qquad \text{for all } X.$$

Given this new rule, the iterations are continued to find $V_1(\ )$, $c_2(\ ), \ldots, c_{j+1}(\ )$ until $|c_{j+1}(X) - c_j(X)|$ is small enough. The convergence rate is much faster than the value function iteration method. However, solving the "policy evaluation step" can sometimes be quite time-consuming, especially when the state space is large. Once again, the computation time can be much reduced if the initial guess $c_0(X)$ is close to the true policy rule $c(X)$.

3.2.3 *Projection Methods*

These methods compute directly the policy function without calculating the value functions. They use the first-order conditions (Euler equation) to back out the policy rules. The continuous cake problem satisfies the first-order Euler equation

$$u'(c_t) = \beta R E_t u'(c_{t+1})$$

if the desired consumption level is less than the total resources $X = W + y$. If there is a corner solution, then the optimal consumption level is $c(X) = X$. Taking into account the corner solution, we can rewrite the Euler equation as

$$u'(c_t) = \max[u'(X_t), \beta R E_t u'(c_{t+1})].$$

We know that by the iid assumption, the problem has only one state variable X, so the consumption function can be written $c = c(X)$. As we consider the stationary solution, we drop the subscript t in the next equation. The Euler equation can then be reformulated as

$$u'(c(X)) - \max[u'(X), \beta R E_{y'} u'(c(R(X - c(X)) + y'))] = 0 \qquad (3.4)$$

or

$$F(c(X)) = 0. \qquad (3.5)$$

The goal is to find an approximation $\hat{c}(X)$ of $c(X)$, for which (3.5) is approximately satisfied. The problem is thus reduced to find the zero of F, where F is an operator over function spaces. This can be done with a minimizing algorithm. There are two issues to resolve. First, we need to find a good approximation of $c(X)$. Second, we have to define a metric to evaluate the fit of the approximation.

Solving for the Policy Rule

Let $\{p_i(X)\}$ be a base of the space of continuous functions, and let $\Psi = \{\psi_i\}$ be a set of parameters. We can approximate $c(X)$ by

$$\hat{c}(X, \Psi) = \sum_{i=1}^{n} \psi_i p_i(X).$$

There is an infinite number of bases to chose from. A simple one is to consider polynomials in X so that $\hat{c}(X, \Psi) = \psi_0 + \psi_1 X + \psi_2 X^2 + \cdots$. Although this choice is intuitive, it is not usually the best choice. In the function space this base is not an orthogonal base, which means that some elements tend to be collinear.

Orthogonal bases will yield more efficient and precise results.[3] The chosen base should be computationally simple. Its elements should "look like" the function to approximate, so that the function $c(X)$ can be approximated with a small number of base functions. Any knowledge of the shape of the policy function will be to a great help. If, for instance, this policy function has a kink, a method based only on a series of polynomials will have a hard time fitting it. It would require a large number of powers of the state variable to come somewhere close to the solution.

Having chosen a method to approximate the policy rule, we now have to be more precise about what "bringing $F(\hat{c}(X, \Psi))$ close to zero" means. To be more specific, we need to define some operators on the space of continuous functions. For any weighting function $g(x)$, the inner product of two integrable functions f_1 and f_2 on a space A is defined as

$$\langle f_1, f_2 \rangle = \int_A f_1(x) f_2(x) g(x)\, dx. \tag{3.6}$$

Two functions f_1 and f_2 are said to be orthogonal, conditional on a weighting function $g(x)$, if $\langle f_1, f_2 \rangle = 0$. The weighting function indicates where the researcher wants the approximation to be good. We are using the operator $\langle .,. \rangle$ and the weighting function to construct a metric to evaluate how close $F(\hat{c}(X, \Psi))$ is to zero. This will be done

3. Popular orthogonal bases are Chebyshev, Legendre, or Hermite polynomials.

by solving for Ψ such that

$$\langle F(\hat{c}(X,\Psi)), f(X)\rangle = 0,$$

where $f(X)$ is some known function. We next review three methods that differ in their choice for this function $f(X)$.

First, a simple choice for $f(X)$ is $F(\hat{c}(X,\Psi))$ itself. This defines the **least square metric** as

$$\min_{\Psi} \langle F(\hat{c}(X,\Psi)), F(\hat{c}(X,\Psi))\rangle.$$

By the **collocation method**, detailed later in this section, we can choose to find Ψ as

$$\min_{\Psi} \langle F(\hat{c}(X,\Psi)), \delta(X - X_i)\rangle, \qquad i = 1, \dots, n,$$

where $\delta(X - X_i)$ is the mass point function at point X_i, meaning that $\delta(X) = 1$ if $X = X_i$ and $\delta(X) = 0$ elsewhere. Another possibility is to define

$$\min_{\Psi} \langle F(\hat{c}(X,\Psi)), p_i(X)\rangle, \qquad i = 1, \dots, n,$$

where $p_i(X)$ is a base of the function space. This is called the **Galerkin method**. An application of this method can be seen below, where the base is taken to be "tent" functions.

Figure 3.4 displays a segment of the computer code that calculates the residual function $F(\hat{c}(X,\Psi))$ when the consumption rule is approximated by a second-order polynomial. This can then be used in one of the proposed methods.

Collocation Methods

Judd (1992) presents in some detail this method applied to the growth model. The function $c(X)$ is approximated using Chebyshev polynomials. These polynomials are defined on the interval $[0,1]$ and take the form

$$p_i(X) = \cos(i \arccos(X)), \qquad X \in [0,1], i = 0, 1, 2, \dots .$$

For $i = 0$, this polynomial is a constant. For $i = 1$, the polynomial is equal to X. As these polynomials are only defined on the $[0,1]$ interval, one can usually scale the state variables appropriately.[4] The

4. The polynomials are also defined recursively by $p_i(X) = 2Xp_{i-1}(X) - p_{i-2}(X)$, $i \geq 2$, with $p_0(0) = 1$ and $p(X,1) = X$.

```
procedure c(x)                  * Here we define an
cc=psi_0+psi_1*x+psi_2*x*x        approximation for the
return(cc)                        consumption function based
endprocedure                      on a second-order
                                  polynomial *

i_s=1
do until i_s>n_s                * Loop over all sizes of the
                                  total amount of cake *
  utoday=U′(c(X[i_s]))          * Marginal utility of
                                  consuming *
  ucorner=U′(X[i_s])            * Marginal utility if corner
                                  solution *
  EnextU=0                      * Initialize expected future
  i_y=1                           marginal utility *
  do until i_y>n_y              * Loop over all possible
                                  realizations of the future
                                  endowment *
  nextX=R(X[i_s]-               * Next amount of cake *
  c(X[i_s]))+Y̅[i_y]
  nextU=U′(c(nextX))            * Next marginal utility of
                                  consumption *
  EnextU=EnextU+nextU*Pi[i_y]   * Here we compute the expected
                                  future marginal utility of
                                  consumption using the
                                  transition matrix Pi *
  i_y=i_y+1
  endo                          * End of loop over endowment *
F[i_s]=utoday-
max(ucorner,beta*EnextU)
i_s=i_s+1
endo                            * End of loop over size of
                                  cake *
```

Figure 3.4
Stochastic cake-eating problem, projection method

policy function can then be expressed as

$$\hat{c}(X,\Psi) = \sum_{i=1}^{n} \psi_i p_i(X).$$

Next the method finds Ψ, which minimizes

$$\langle F(\hat{c}(X,\Psi)), \delta(X - X_i)\rangle, \qquad i = 1,\ldots,n,$$

where $\delta(\ )$ is the mass point function. Hence the method requires that $F(\hat{c}(X,\Psi))$ is zero at some particular points X_i and not over the whole range $[\overline{X}_L, \overline{X}_H]$. The method is more efficient if these points are chosen to be the zeros of the base elements $p_i(X)$, here $X_i = \cos(\pi/2i)$. This method is referred to as an orthogonal collocation method. Ψ is

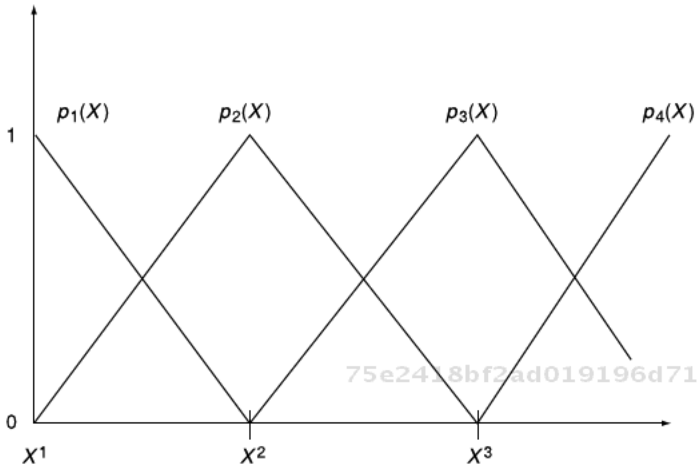

Figure 3.5
Basis functions, finite element method

the solution to a system of nonlinear equations:

$$F(\hat{c}(X_i, \Psi)) = 0, \qquad i = 1, \ldots, n.$$

This method is good at approximating policy functions that are relatively smooth. A drawback is that the Chebyshev polynomials tend to display oscillations at higher orders. The resulting policy function $c(X)$ will also tend to fluctuate. There is no particular rule for choosing n, the highest order of the Chebyshev polynomial. Obviously the higher n is, the better will be the approximation, but this comes at the cost of increased computation.

Finite Element Methods

McGrattan (1996) illustrates the finite element method with the stochastic growth model (see also Reddy 1993 for an in-depth discussion of finite elements).

To start, the state variable X is discretized over a grid $\{X^{i_s}\}_{i_s=1}^{n_s}$. The finite element method is based on the following functions:

$$p_{i_s}(X) = \begin{cases} \dfrac{X - X^{i_s-1}}{X^{i_s} - X^{i_s-1}} & \text{if } X \in [X^{i_s-1}, X^{i_s}], \\ \dfrac{X^{i_s+1} - X}{X^{i_s+1} - X^{i_s}} & \text{if } X \in [X^{i_s}, X^{i_s+1}], \\ 0 & \text{elsewhere.} \end{cases}$$

The function $p_{i_s}(X)$ is a simple function in $[0,1]$, as illustrated in figure 3.5. It is in fact a simple linear interpolation (and an order two spline; see the appendix for more on these techniques). On the interval $[X^{i_s}, X^{i_s+1}]$, the function $\hat{c}(X)$ is equal to the weighted sum of $p_{i_s}(X)$ and $p_{i_s+1}(X)$. Here the residual function satisfies

$$\langle F(\hat{c}(X,\Psi)), p_i(X)\rangle = 0, \qquad i = 1,\dots,n.$$

Equivalently, we could choose a constant weighting function:

$$\int_0^{\bar{X}} p_{i_s}(X) F(\hat{c}(X))\, dX = 0, \qquad i_s = 1,\dots,n_s.$$

This gives a system with n_s equations and n_s unknowns, $\{\psi_{i_s}\}_{i_s=1}^{n_s}$. This nonlinear system can be solved to find the weights $\{\psi_{i_s}\}$. To solve the system, the integral can be computed numerically using numerical techniques; see the appendix. As in the collocation method, the choice of n_s is the result of a trade-off between increased precision and a higher computational burden.

3.3 Stochastic Discrete Cake-Eating Problem

We present here another example of a dynamic programming model. It differs from the one presented in section 3.2 in two ways. First, the decision of the agent is not continuous (how much to eat) but discrete (eat or wait). Second, the problem has two state variables as the exogenous shock is serially correlated.

The agent is endowed with a cake of size W. In each period the agent has to decide whether or not to eat the entire cake. Even if not eaten, the cake shrinks by a factor ρ each period. The agent also experiences taste shocks, possibly serially correlated, and which follow an autoregressive process of order one. The agent observes the current taste shock at the beginning of the period, before the decision to eat the cake is taken. However, the future shocks are unobserved by the agent, introducing a stochastic element into the problem. Although the cake is shrinking, the agent might decide to postpone the consumption decision until a period with a better realization of the taste shock. The program of the agent can be written in the form

$$V(W,\varepsilon) = \max[\varepsilon u(W), \beta E_{\varepsilon'|\varepsilon} V(\rho W, \varepsilon')], \tag{3.7}$$

where $V(W,\varepsilon)$ is the intertemporal value of a cake of size W conditional of the realization ε of the taste shock. Here $E_{\varepsilon'}$ denotes the expectation with respect to the future shock ε, conditional on the value of ε. The policy function is a function $d(W, \varepsilon)$ that takes a value of zero if the agent decides to wait or one if the cake is eaten. We can also define a threshold $\varepsilon^*(W)$ such that

$$\begin{cases} d(W,\varepsilon) = 1 & \text{if } \varepsilon > \varepsilon^*(W), \\ d(W,\varepsilon) = 0 & \text{otherwise.} \end{cases}$$

As in section 3.2 the problem can be solved by value function iterations. However, the problem is discrete, so we cannot use the projection technique as the decision rule is not a smooth function but a step function.

3.3.1 *Value Function Iterations*

As before, we have to define, first, the functional form for the utility function, and we need to discretize the state space. We will consider $\rho < 1$, so the cake shrinks with time and W is naturally bounded between $\overline{W}$, the initial size and 0. In this case the size of the cake takes only values equal to $\rho^t \overline{W}$, $t \geq 0$. Hence $\Psi_S = \{\rho^i \overline{W}\}$ is a judicious choice for the state space. Contrary to an equally spaced grid, this choice ensures that we do not need to interpolate the value function outside of the grid points.

Next, we need to discretize the second state variable, ε. The shock is supposed to come from a continuous distribution, and it follows an autoregressive process of order one. We discretize ε in I points $\{\varepsilon_i\}_{i=1}^{I}$ following a technique presented by Tauchen (1986) and summarized in the appendix. In fact we approximate an autoregressive process by a Markov chain. The method determines the optimal discrete points $\{\varepsilon_i\}_{i=1}^{I}$ and the transition matrix $\pi_{ij} = \text{Prob}(\varepsilon_t = \varepsilon_i | \varepsilon_{t-1} = \varepsilon_j)$ such that the Markov chain mimics the AR(1) process. Of course, the approximation is only good if I is big enough.

In the case where $I = 2$, we have to determine two grid points ε_L and ε_H. The probability that a shock ε_L is followed by a shock ε_H is denoted by π_{LH}. The probability of transitions can be stacked in a transition matrix:

$$\pi = \begin{bmatrix} \pi_{LL} & \pi_{LH} \\ \pi_{HL} & \pi_{HH} \end{bmatrix}$$

with the constraints that the probability of reaching either a low or a high state next period is equal to one: $\pi_{LL} + \pi_{LH} = 1$ and $\pi_{HL} + \pi_{HH} =$

```
i_s=2
do until i_s>n_s              * Loop over all sizes of the
                                cake *
  i_e=1
  do until i_e>2              * Loop over all possible
                                realizations of the taste shock
                                *
  ueat=u(W[i_s],e[i_e])       * Utility of doing the eating now
                                *
  nextV1=V[i_s-1,1]           * Next period value if low taste
                                shock *
  nextV2=V[i_s-1,2]           * Next period value if high taste
                                shock *
  EnextV=nextV1*p[i_e,1]+
  nextV2*p[i_e,2]
  newV[i_s,i_e]
  =max(ueat,beta*EnextV)
                              * Take the max between eating now
                                or waiting *
  i_e=i_e+1
  endo                        * End of loop over taste shock *
i_s=i_s+1
endo                          * End of loop over size of cake *
V=newV                        * Update the new value function *
```

Figure 3.6
Discrete cake-eating problem

1. For a given size of the cake $W^{i_s} = \rho^{i_s}\overline{W}$ and a given shock ε_j, $j = L$ or H, it is easy to compute the first term $\varepsilon_j u(\rho^{i_s}\overline{W})$. To compute the second term we need to calculate the expected value of tomorrow's cake. Given a guess for the value function of next period, $v(.,.)$, the expected value is

$$E_{\varepsilon' \mid \varepsilon_j} v(\rho^{i_s+1}\overline{W}) = \pi_{jL} v(\rho^{i_s+1}\overline{W}, \varepsilon_L) + \pi_{jH} v(\rho^{i_s+1}\overline{W}, \varepsilon_H).$$

The recursion is started backward with an initial guess for $V(.,.)$. For a given state of the cake W_{i_s} and a given shock ε_j, the new value function is calculated from equation (3.7). The iterations are stopped when two successive value functions are close enough. In numerical computing the value function is stored as a matrix $\mathbf{V}$ of size $n_W \times n_\varepsilon$, where n_W and n_ε are the number of points on the grid for W and ε. At each iteration the matrix is updated with the new guess for the value function. Figure 3.6 gives an example of a computer code that obtains the value function $v_{j+1}(W, \varepsilon)$ given the value $v_j(W, \varepsilon)$.

The way we have computed the grid, the next period value is simple to compute as it is given by $\mathbf{V}[i_s - 1, .]$. This rule is valid if

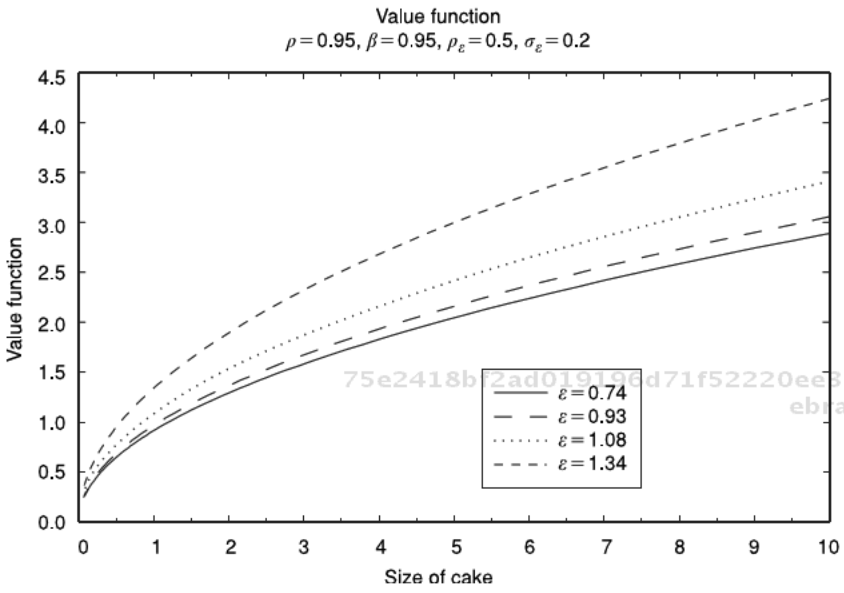

Figure 3.7
Value function, discrete cake-eating problem

$i_s > 1$. Computing $\mathbf{V}[1, .]$ is more of a problem. One way is to use an extrapolation method to approximate the values, given the knowledge of $\mathbf{V}[i_s, .]$, $i_s > 1$.

Figure 3.7 shows the value function for particular parameters. The utility function is taken to be $u(c, \varepsilon) = \ln(\varepsilon c)$, and $\ln(\varepsilon)$ is supposed to follow an AR(1) process with mean zero, autocorrelation $\rho_\varepsilon = 0.5$ and with an unconditional variance of 0.2. We have discretized ε into four grid points.

Figure 3.8 shows the decision rule, and the function $\varepsilon^*(W)$. This threshold was computed as the solution of:

$$u(W, \varepsilon^*(W)) = \beta E_{\varepsilon' | \varepsilon} V(\rho W, \varepsilon'),$$

which is the value of the taste shock that makes the agent indifferent between waiting and eating, given the size of the cake W.

We return later in this book to examples of discrete choice models. In particular, we refer the readers to the models presented in sections 8.5 and 7.3.3.

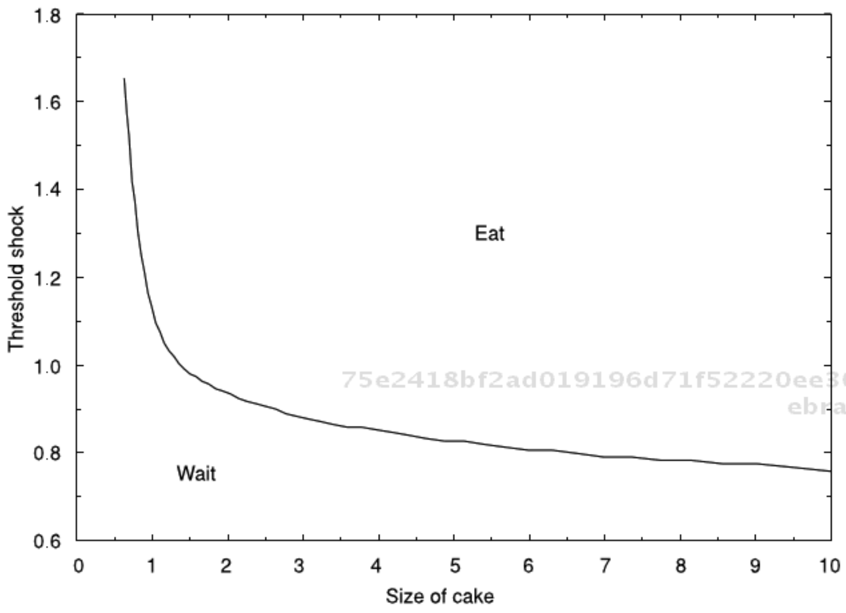

Figure 3.8
Decision rule, discrete cake-eating problem

3.4 Extensions and Conclusion

In this chapter we reviewed the common techniques used to solve the dynamic programming problems of chapter 2. We applied these techniques to both deterministic and stochastic problems, to continuous and discrete choice models. These methods can be applied as well to more complicated problems.

3.4.1 Larger State Spaces

The two examples we have studied in sections 3.2 and 3.3 have small state spaces. In empirical applications the state space often needs to be much larger if the model is to confront real data. For instance, the endowment shocks might be serially correlated or the interest rate R might also be a stochastic and persistent process.

For the value function iteration method, this means that the successive value functions have to be stacked in a multidimensional matrix. Also the value function has to be interpolated in several dimensions. The techniques in the appendix can be extended to deal with this problem. However, the value function iteration method quickly encounters the "curse of dimensionality." If every state variable is discretized into n_s grid points, the value function has to be evaluated by N^{n_s} points, where N is the number of state variables. This demands an increasing amount of computer memory and so slows down the computation. A solution to this problem is to evaluate the value function for a subset of the points in the state space and then to interpolate the value function elsewhere. This solution was implemented by Keane and Wolpin (1994).

Projection methods are better at handling larger state spaces. Suppose that the problem is characterized by N state variables $\{X_1, \dots, X_N\}$. The approximated policy function can be written as

$$\hat{c}(X_1, \dots, X_N) = \sum_{j=1}^{N} \sum_{i_j=1}^{n_j} \psi_{i_j}^{j} p_{i_j}(X_j).$$

The problem is then characterized by auxiliary parameters $\{\psi_i^j\}$.

EXERCISE 3.1 Suppose that $u(c) = c^{1-\gamma}/(1-\gamma)$. Construct the code to solve for the stochastic cake-eating problem using the value function iteration method. Plot the policy function as a function of the size of the cake and the stochastic endowment for $\gamma = \{0.5, 1, 2\}$. Compare the level and slope of the policy functions for different values of γ. How do you interpret the results?

EXERCISE 3.2 Consider, again, the discrete cake-eating problem of section 3.3. Construct the code to solve for this problem, with iid taste shocks, using $u(c) = \ln(c)$, $\varepsilon_L = 0.8$, $\varepsilon_H = 1.2$, $\pi_L = 0.3$, and $\pi_H = 0.7$. Map the decision rule as a function of the size of the cake.

EXERCISE 3.3 Consider an extension of the discrete cake-eating problem of section 3.3. The agent can now choose among three actions: eat the cake, store it in fridge 1 or in fridge 2. In fridge 1, the cake shrinks by a factor ρ: $W' = \rho W$. In fridge 2, the cake diminish by a fixed amount: $W' = W - \kappa$. The program of the agent is characterized as

$$V(W, \varepsilon) = \max[V^{\text{Eat}}(W, \varepsilon), V^{\text{Fridge 1}}(W, \varepsilon), V^{\text{Fridge 2}}(W, \varepsilon)]$$

$$\text{with} \begin{cases} V^{\text{Eat}}(W, \varepsilon) = \varepsilon u(W), \\ V^{\text{Fridge 1}}(W, \varepsilon) = \beta E_{\varepsilon'} V(\rho W, \varepsilon'), \\ V^{\text{Fridge 2}}(W, \varepsilon) = \beta E_{\varepsilon'} V(W - \kappa, \varepsilon'). \end{cases}$$

Construct the code to solve for this problem, using $u(c) = \ln(c)$, $\varepsilon_L = 0.8$, $\varepsilon_H = 1.2$, $\pi_L = 0.5$, and $\pi_H = 0.5$. When will the agent switch from one fridge to the other?

EXERCISE 3.4 Consider the stochastic cake-eating problem. Suppose that the discount rate β is a function of the amount of cake consumed: $\beta = \Phi(\beta_1 + \beta_2 c)$, where β_1 and β_2 are known parameters and $\Phi(\ )$ is the normal cumulative distribution function. Construct the code to solve for this new problem using value function iterations. Suppose $\gamma = 2$, $\beta_1 = 1.65$, $\pi_L = \pi_H = 0.5$, $y_L = 0.8$, $y_H = 1.2$, and $\beta_2 = -1$. Plot the policy rule $c = c(X)$. Compare the result with that of the case where the discount rate is independent of the quantity consumed. How would you interpret the fact that the discount rate depends on the amount of cake consumed?

3.5 Appendix: Additional Numerical Tools

In this appendix we provide some useful numerical tools that are often used in solving dynamic problems. We present interpolation methods, numerical integration methods, as well as a method to approximate serially correlated processes by a Markov process. The last subsection is devoted to simulations.

3.5.1 *Interpolation Methods*

We briefly review three simple interpolation methods. For further readings, see, for instance, Press et al. (1986) or Judd (1996).

When solving the value function or the policy function, we often have to calculate the value of these functions outside of the points of the grid. This requires one to be able to interpolate the function. Using a good interpolation method can also save computer time and space since fewer grid points are needed to approximate the functions. Let us denote $f(x)$ the function to approximate. We assume that we know this function at a number of grid points x_i, $i = 1, \dots, I$. We denote by $f_i = f(x_i)$ the values of the function at these grid points. We are interested in finding an approximate function $\hat{f}(x)$ such that $\hat{f}(x) \simeq f(x)$, based on the observations $\{x_i, f_i\}$. We present three different methods and use as an example the function $f(x) = x\sin(x)$. Figure 3.9 displays the results for all the methods.

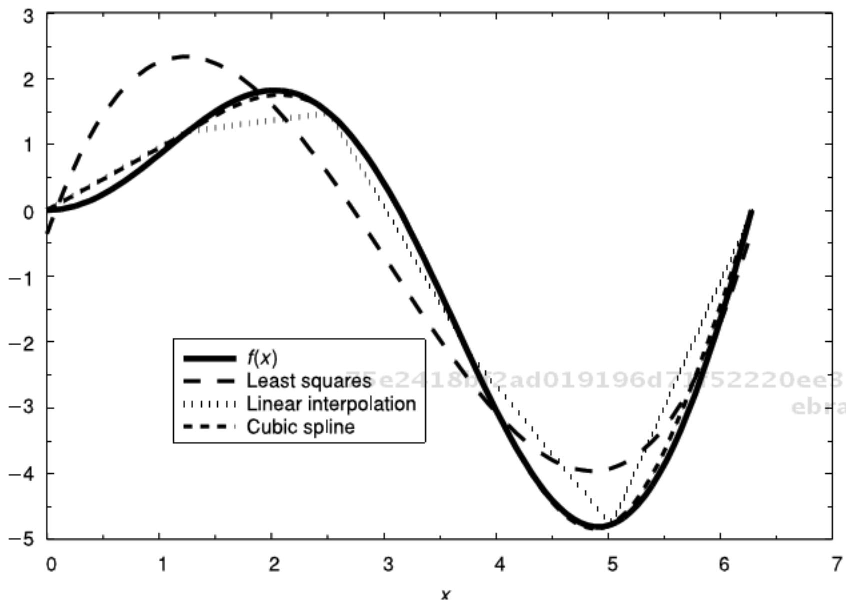

Figure 3.9
Approximation methods

Least Squares Interpolation

A natural way to approximate $f(\ )$ is to use an econometric technique, such as OLS, to "estimate" the function $\hat{f}(.)$. The first step is to assume a functional form for $\hat{f}$. For instance, we can approximate f with a polynomial in x such as

$$\hat{f}(x) = \alpha_0 + \alpha_1 x + \cdots + \alpha_N x^N, \qquad N < I.$$

By regressing f_i on x_i, we can easily recover the parameters α_n. In practice, this method is often not very good, unless the function f is well behaved. Higher-order polynomials tend to fluctuate and can occasionally give an extremely poor fit. This is particularly true when the function is extrapolated outside of the grid points, when $x > x_I$ or $x < x_1$. The least square method is a global approximation method. As such, the fit can be on average satisfactory but mediocre almost everywhere. This can be seen in the example in figure 3.9.

Linear Interpolation

This method fits the function f with piecewise linear functions on the intervals $[x_{i-1}, x_i]$. For any value of x in $[x_{i-1}, x_i]$, an approximation $\hat{f}(x)$ of $f(x)$ can be found as

$$\hat{f}(x) = f_{i-1} + \frac{f_i - f_{i-1}}{x_i - x_{i-1}}(x - x_{i-1}).$$

A finer grid will give a better approximation of $f(x)$. When x is greater than x_I, using this rule can lead to numerical problems as the expression above may not be accurate. Note that the approximation function $\hat{f}$ is continuous but not differentiable at the grid points. This can be an undesirable feature as this nondifferentiability can be translated to the value function or the policy function.

The linear interpolation method can be extended for multivariate functions. For instance, we can approximate the function $f(x, y)$ given data on $\{x_i, y_j, f_{ij}\}$. Denote $d_x = (x - x_i)/(x_{i-1} - x_i)$ and $d_y = (y - y_i)/(y_{i-1} - y_i)$. The approximation can be written as

$$\hat{f}(x, y) = d_x d_y f_{i-1,j-1} + (1 - d_x)\, d_y f_{i,j-1} + d_x(1 - d_y) f_{i-1,j} + (1 - d_x)(1 - d_y) f_{i,j}.$$

The formula can be extended to higher dimensions as well.

Spline Methods

This method extends the linear interpolation by fitting piecewise polynomials while ensuring that the resulting approximate function $\hat{f}$ is both continuous and differentiable at the grid points x_i. We restrict ourself to cubic splines for simplicity, but the literature on splines is very large (e.g., see De Boor 1978). The approximate function is expressed as

$$\hat{f}_i(x) = f_i + a_i(x - x_{i-1}) + b_i(x - x_{i-1})^2 + c_i(x - x_{i-1})^3, \qquad x \in [x_{i-1}, x_i].$$

Here for each point on the grid, we have to determine three parameters $\{a_i, b_i, c_i\}$, so in total there is $3 \times I$ parameters to compute. However, imposing the continuity of the function and of its derivative up to the second order reduces the number of coefficients:

$$\hat{f}_i(x) = \hat{f}_{i+1}(x),$$
$$\hat{f}_i'(x) = \hat{f}_{i+1}'(x),$$
$$\hat{f}_i''(x) = \hat{f}_{i+1}''(x).$$

It is also common practice to apply $\hat{f}_1''(x_1) = \hat{f}_I''(x_I) = 0$. With these constraints, the number of coefficients to compute is down to I. Some algebra gives

$$\begin{cases} a_i = \dfrac{f_i - f_{i-1}}{x_i - x_{i-1}} - b_i(x_i - x_{i-1}) - c_i(x_i - x_{i-1})^2, \qquad i = 1, \dots, I, \\ c_i = \dfrac{b_{i+1} - b_i}{3(x_i - x_{i-1})}, \qquad i = 1, \dots, I-1, \\ c_I = -\dfrac{b_I}{3(x_I - x_{I-1})}, \\ a_i + 2b_i(x_i - x_{i-1}) + 3c_i(x_i - x_{i-1})^2 = a_{i+1}. \end{cases}$$

Solving this system of equation leads to expressions for the coefficients $\{a_i, b_i, c_i\}$. Figure 3.9 shows that the cubic spline is a very good approximation to the function f.

3.5.2 *Numerical Integration*

Numerical integration is often required in dynamic programming problems to solve for the expected value function or to "integrate out" an unobserved state variable. For instance, solving the Bellman equation (3.3) requires one to calculate $Ev(X') = \int v(X')\,dF(X')$, where $F(.)$ is the cumulative density of the next period cash-on-hand X. In econometric applications some important state variables might not be observed. For this reason one may need to compute the decision rule unconditional of this state variable. For instance, in the stochastic cake-eating problem of section 3.2, if X is not observed, one could compute $\bar{c} = \int c(X)\,dF(X)$, which is the unconditional mean of consumption, and match it with observed consumption. We present three methods that can be used when numerical integration is needed.

Quadrature Methods

There are a number of quadrature methods. We briefly detail the Gauss-Legendre method (more detailed information can be found in Press et al. 1986). The integral of a function f is approximated as

$$\int_{-1}^{1} f(x)\,dx \simeq w_1 f(x_1) + \cdots + w_n f(x_n), \tag{3.8}$$

where w_i and x_i are n weights and nodes to be determined. Integration over a different domain can be easily handled by operating a change of the integration variable. The weights and the nodes are computed such that (3.8) is exactly satisfied for polynomials of degree $2n-1$ or less. For instance, if $n=2$, denote $f_i(x)=x^{i-1}$, $i=1,\dots,4$. The weights and nodes satisfy

$$w_1 f_1(x_1)+w_2 f_1(x_2)=\int_{-1}^{1} f_1(x)\,dx,$$

$$w_1 f_2(x_1)+w_2 f_2(x_2)=\int_{-1}^{1} f_2(x)\,dx,$$

$$w_1 f_3(x_1)+w_2 f_3(x_2)=\int_{-1}^{1} f_3(x)\,dx,$$

$$w_1 f_4(x_1)+w_2 f_4(x_2)=\int_{-1}^{1} f_4(x)\,dx.$$

This is a system of four equations with four unknowns. The solutions are $w_1=w_2=1$ and $x_2=-x_1=0.578$. For larger values of n, the computation is similar. By increasing the number of nodes n, the precision increases. Notice that the nodes are not necessarily equally spaced. The weights and the value of the nodes are published in the literature for commonly used values of n.

Approximating an Autoregressive Process with a Markov Chain

In this discussion we follow Tauchen (1986) and Tauchen and Hussey (1991) and show how to approximate an autoregressive process of order one by a first-order Markov process. This way we can simplify the computation of expected values in the value function iteration framework.

To return to the value function in the cake-eating problem, we need to calculate the expected value given ε:

$$V(W,\varepsilon)=\max[\varepsilon u(W), E_{\varepsilon'|\varepsilon}V(\rho W,\varepsilon')].$$

The calculation of an integral at each iteration is cumbersome. So we discretize the process ε_t, into N points ε^i, $i=1,\dots,N$. Now we can replace the expected value by

$$V(W,\varepsilon^i)=\max\left[\varepsilon u(W), \sum_{j=1}^{N}\pi_{i,j}V(\rho W,\varepsilon^j)\right], \qquad i=1,\dots,N.$$

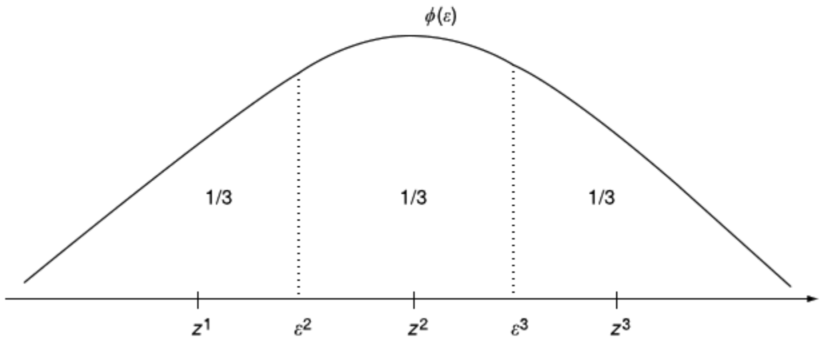

Figure 3.10
Example of discretization, $N = 3$

As in the quadrature method, the method involves finding nodes ε^j and weights $\pi_{i,j}$. As we will see below, the ε^i and the $\pi_{i,j}$ can be computed prior to the iterations.

Suppose that ε_t follows an AR(1) process, with an unconditional mean μ and an autocorrelation ρ:

$$\varepsilon_t = \mu(1-\rho) + \rho\varepsilon_{t-1} + u_t, \tag{3.9}$$

where u_t is a normally distributed shock with variance σ^2. To discretize this process, we need to determine three different objects. First, we need to discretize the process ε_t into N intervals. Second, we need to compute the conditional mean of ε_t within each intervals, which we denote by z^i, $i, \dots, N$. Third, we need to compute the probability of transition between any of these intervals, $\pi_{i,j}$. Figure 3.10 shows the plot of the distribution of ε and the cut-off points ε^i as well as the conditional means z^i.

We start by discretizing the real line into N intervals, defined by the limits $\varepsilon^1, \dots, \varepsilon^{N+1}$. As the process ε_t is unbounded, $\varepsilon^1 = -\infty$ and $\varepsilon^{N+1} = +\infty$. The intervals are constructed such that ε_t has an equal probability of $1/N$ of falling into them. Given the normality assumption, the cut-off points $\{\varepsilon^i\}_{i=1}^{N+1}$ are defined as

$$\Phi\left(\frac{\varepsilon^{i+1}-\mu}{\sigma_\varepsilon}\right) - \Phi\left(\frac{\varepsilon^{i}-\mu}{\sigma_\varepsilon}\right) = \frac{1}{N}, \qquad i = 1, \dots, N, \tag{3.10}$$

where $\Phi(\ )$ is the cumulative of the normal density and σ_ε is the standard deviation of ε equal to $\sigma/\sqrt{(1-\rho)}$. Working recursively, we get

$$\varepsilon^i = \sigma_\varepsilon \Phi^{-1}\left(\frac{i-1}{N}\right) + \mu.$$

Now that we have defined the intervals, we want to find the average value of ε within a given interval. We denote this value by z^i, which is computed as the mean of ε_t conditional on $\varepsilon_t \in [\varepsilon^i, \varepsilon^{i+1}]$:

$$z^i = E(\varepsilon_t | \varepsilon_t \in [\varepsilon^i, \varepsilon^{i+1}]) = \sigma_\varepsilon \frac{\phi((\varepsilon^i - \mu)/\sigma_\varepsilon) - \phi((\varepsilon^{i+1} - \mu)/\sigma_\varepsilon)}{\Phi((\varepsilon^{i+1} - \mu)/\sigma_\varepsilon) - \Phi((\varepsilon^i - \mu)/\sigma_\varepsilon)} + \mu.$$

From (3.10), we know that the expression simplifies to

$$z^i = N\sigma_\varepsilon\left(\phi\left(\frac{\varepsilon^i - \mu}{\sigma_\varepsilon}\right) - \phi\left(\frac{\varepsilon^{i+1} - \mu}{\sigma_\varepsilon}\right)\right) + \mu.$$

Next we define the transition probability as

$$\pi_{i,j} = P(\varepsilon_t \in [\varepsilon^j, \varepsilon^{j+1}] \,|\, \varepsilon_{t-1} \in [\varepsilon^i, \varepsilon^{i+1}])$$

$$\pi_{i,j} = \frac{N}{\sqrt{2\pi\sigma_\varepsilon}} \int_{\varepsilon^i}^{\varepsilon^{i+1}} e^{-(u-\mu)^2/(2\sigma_\varepsilon^2)} \left[\Phi\left(\frac{\varepsilon^{j+1} - \mu(1-\rho) - \rho u}{\sigma}\right) - \Phi\left(\frac{\varepsilon^j - \mu(1-\rho) - \rho u}{\sigma}\right)\right] du.$$

The computation of $\pi_{i,j}$ requires the computation of a nontrivial integral. This can be done numerically. Note that if $\rho = 0$, meaning ε is an iid process, the expression above is simply

$$\pi_{i,j} = \frac{1}{N}.$$

We can now define a Markov process z_t that will mimic an autoregressive process of order one, as defined in (3.9). z_t takes its values in $\{z^i\}_{i=1}^N$ and the transition between period t and $t+1$ is defined as

$$P(z_t = z^j | z_{t-1} = z^i) = \pi_{i,j}.$$

By increasing N, the discretization becomes finer and the Markov process gets closer to the real autoregressive process.

Example For N = 3, $\rho = 0.5$, $\mu = 0$, and $\sigma = 1$, we have

$$z^1 = -1.26, \quad z^2 = 0, \quad z^3 = 1.26,$$

and

$$\pi = \begin{bmatrix} 0.55 & 0.31 & 0.14 \\ 0.31 & 0.38 & 0.31 \\ 0.14 & 0.31 & 0.55 \end{bmatrix}.$$

3.5.3 *How to Simulate the Model*

Once the value function is computed, the estimation or the evaluation of the model often requires the simulation of the behavior of the agent through time. If the model is stochastic, the first step is to generate a series for the shocks, for $t = 1, \dots, T$. Then we go from period to period and use the policy function to find out the optimal choice for this period. We also update the state variable and proceed to next period.

How to Program a Markov Process

The Markov process is characterized by grid points, $\{z^i\}$ and by a transition matrix π, with elements $\pi_{ij} = \text{Prob}(y_t = z^j / y_{t-1} = z^i)$. We start in period 1. The process z_t is initialized at, say, z^i. Next, we have to assign a value for z_2. To this end, using the random generator of the computer, we draw a uniform variable u in $[0, 1]$. The state in period 2, j, is defined as

```
t=1
oldind=1                          * Variable to keep track of
                                    state in period t-1 *
y[t]=z[oldind]                    * Initialize first period *
  do until t>T                    * Loop over all time periods *
    u=uniform(0,1)                * Generate a uniform random
                                    variable *
    sum=0                         * Will contain the cumulative
                                    sum of pi *
    ind=1                         * Index over all possible
                                    values for process *
      do until u<=sum             * Loop to find out the state
                                    in period t *
        sum=sum+pi[oldind,ind]    * Cumulative sum of pi *
        ind=ind+1
      endo
  y[t]=z[ind]                     * State in period t *
  oldind=ind                      * Keep track of lagged state *
  t=t+1
  endo
```

Figure 3.11
Simulation of a Markov process

$$\sum_{l=1}^{j} \pi_{i,l} < u \leq \sum_{l=1}^{j+1} \pi_{i,l},$$

or $j = 1$ if $u < \pi_{i,1}$. The values for the periods ahead are constructed in a similar way. Figure 3.11 presents a computer code that will construct iteratively the values for T periods.

How to Simulate the Model

For the model we need to initialize all stochastic processes that are the exogenous shock and the state variables. The state variables can be initialized to their long-run values or to some other value. Often the model is simulated over a long number of periods and the first periods are discarded to get rid of initial condition problems.

The value of the state variables and the shock in period 1 are used to determine the choice variable in period 1. In the case of the continuous stochastic cake-eating problem of section 3.2, we would construct $c_1 = c(X_1)$. Next we can generate the values of the state variable in period 2, $X_2 = R(X_1 - c_1) + y_2$, where y_2 is calculated using the method described in section 3.5.3 above. This procedure would be repeated over T periods to successively construct all the values for the choice variables and the state variables.