Chapter 5

Function Approximation

In many computational economic applications, one must approximate an intractable real-valued function f with a computationally tractable function $\hat{f}$.

Two types of function approximation problems arise often in computational economic applications. In the *interpolation* or 'data fitting' problem one uncovers some properties satisfied by the function f and then selects an approximant $\hat{f}$ from a family of 'nice' functions that satisfies those properties. The data available about f is often just its value at a set of specified points. The data, however, could include the first or second derivative of f at some of the points. Interpolation methods were historically developed to approximate the value of mathematical and statistical functions from published tables of values. In most modern computational economic applications, however, the analyst is free to chose what data to obtain about the function to be approximated. Modern interpolation theory and practice is concerned with ways to optimally extract data from a function and with computationally efficient methods for constructing and working with its approximant.

In the *functional equation* problem, one must find a function f that satisfies

$$Tf = g$$

where T is an operator that maps a vector space of functions into itself and g is a known function in that space. In the equivalent *functional fixed-point* problem, one must find a function f such that

$$Tf = f.$$

Functional equations are common in dynamic economic analysis. For example, the Bellman equations that characterize the solutions of dynamic optimization models are functional fixed-point equations. Euler equations and fundamental asset pricing differential equations are also functional equations.

Functional equations are difficult to solve because the unknown is not simply a vector in $\Re^n$, but an entire function f whose domain contains an infinite number points. Moreover, the functional equation typically imposes an infinite number of conditions on the solution f. Except in special cases, functional equations lack analytic closed-form solutions and thus cannot be solved exactly. One must therefore settle for an approximate solution $\hat{f}$ that satisfies the functional equation as closely as possible. In many cases, one can compute accurate approximate solutions to functional equations using techniques that are natural extensions interpolation methods.

Numerical analysts have studied function approximation and functional equation problems extensively and have acquired substantial experience with different techniques for solving them. Below, the two most generally practical techniques for approximating functions are discussed: polynomial and spline interpolation. Univariate function interpolation methods are developed first and then are generalized to multivariate function methods. In the final section, we introduce the collocation method, a natural generalization of interpolation methods that may be used to solve a variety of functional equations.

5.1 Interpolation Principles

Interpolation is the most generally practical method for approximating a real-valued function f defined on an interval of the real line $\Re$. The first step in designing an interpolation scheme is to specify the properties of the original function f that one wishes the approximant $\hat{f}$ to replicate. The easiest and most common conditions imposed on the approximant is that it *interpolate* or match the value of the original function at selected *interpolation nodes* $x_1, x_2, \ldots, x_n$. The number n of interpolation nodes is called the *degree of interpolation.*

The second step in designing an interpolation scheme is to specify an n-dimensional subspace of functions from which the approximant is to be drawn. The subspace is characterized as the space spanned by the linear combinations of n linearly independent *basis functions* $\phi_1, \phi_2, \ldots, \phi_n$ selected

by the analyst. In other words, the approximant is of the form

$$\hat{f}(x) = \sum_{i=1}^{n} c_j \phi_j(x),$$

where $c_1, c_2, \ldots, c_n$ are *basis coefficients* to be determined. As we shall see, polynomials of increasing order are often used as basis functions, although other types of basis functions are also commonly used.

Given n interpolation nodes and n basis functions, constructing the approximant reduces to solving a linear equation. Specifically, one fixes the n unknown coefficients $c_1, c_2, \ldots, c_n$ of the approximant $\hat{f}$ by solving the *interpolation conditions*

$$\sum_{j=1}^{n} c_j \phi_j(x_i) = f(x_i) = y_i \qquad \forall i = 1, 2, \ldots, n.$$

Using matrix notation, the interpolation conditions equivalently may be written as the matrix linear *interpolation equation* whose unknown is the vector of basis coefficients c:

$$\Phi c = y.$$

Here,

$$\Phi_{ij} = \phi_j(x_j)$$

is the typical element of the *interpolation matrix* Φ. Equivalently, the interpolation conditions may be written as the linear transformation

$$c = P \star y$$

where $P = \Phi^{-1}$ is the *projection matrix*. For an interpolation scheme to be well-defined, the interpolation nodes and basis functions must be chosen such that the interpolation matrix is nonsingular and the projection matrix exists.

Interpolation schemes are not limited to using only function value information. In many applications, one may wish to interpolate both function values and derivatives at specified points. Suppose, for example, that one wishes to construct an approximant $\hat{f}$ that replicates the function's values at nodes $x_1, x_2, \ldots, x_{n_1}$ and its first derivatives at nodes $x'_1, x'_2, \ldots, x'_{n_2}$. This may be accomplished by selecting $n = n_1 + n_2$ basis functions and fixing the

basis coefficients $c_1, c_2, \ldots, c_n$ of the approximant by solving the interpolation conditions

$$\begin{aligned}
\sum_{j=1}^{n} c_j \phi_j(x_i) &= f(x_i), \qquad \forall i = 1, \ldots, n_1 \\
\sum_{j=1}^{n} c_j \phi_j'(x_i') &= f'(x_i'), \qquad \forall i = 1, \ldots, n_2
\end{aligned}$$

for the unknown coefficients c_j.

In developing an interpolation scheme, the analyst should chose interpolation nodes and basis functions that satisfy certain criteria. First, the approximant should be capable of producing an accurate approximation for the original function f. In particular, the interpolation scheme should allow the analyst to achieve, at least in theory, an arbitrarily accurate approximation by increasing the degree of approximation. Second, the approximant should be easy to compute. In particular, the interpolation equation should be well-conditioned and should possess a special structure that allows it to be solved quickly—diagonal, near diagonal, or orthogonal interpolation matrices are best. Third, the approximant should be easy to work with. In particular, the basis functions should be easy to evaluate, differentiate, and integrate.

Interpolation schemes may be classified as either *spectral* methods or *finite element* methods. A spectral method uses basis functions that are nonzero over the entire domain of the function being approximated. In contrast, a finite element method uses basis functions that are nonzero over only a subinterval of the domain of approximation. Polynomial interpolation, which uses polynomials of increasing degree as basis functions, is the most common spectral method. Spline interpolation, which uses basis functions that are polynomials of small degree over subintervals of the approximation domain, is the most common finite element method. Polynomial interpolation and two variants of spline interpolation are discussed below.

5.2 Polynomial Interpolation

According to the Weierstrass Theorem, any continuous real-valued function f defined on a bounded interval $[a, b]$ of the real line can be approximated to any degree of accuracy using a polynomial. More specifically, if $\epsilon > 0$, there

exists a polynomial p such that

$$||f - p|| = \sup_{x \in [a,b]} |f(x) - p(x)| < \epsilon.$$

The Weierstrass theorem provides strong motivation for using polynomials to approximate continuous functions. The theorem, however, is not very practical. It gives no guidance on how to find a good polynomial approximant. It does not even state what order polynomial is required to achieve the required level of accuracy.

One apparently reasonable way to construct a n^{th}-degree polynomial approximant for f is to form the $(n-1)^{th}$-order polynomial

$$p(x) = c_1 + c_2 x + c_3 x^2 + \ldots + c_n x^{n-1}$$

that interpolates f at the n uniformly spaced interpolation nodes

$$x_i = a + \frac{i-1}{n-1} \cdot (b-a) \qquad \forall i = 1, 2, \ldots, n.$$

Given the function values $y_i = f(x_i)$ at the interpolation nodes, the basis coefficient vector c is computed by solving the linear interpolation equation

$$c = \Phi \setminus y,$$

where

$$\Phi_{ij} = x_i^{j-1}.$$

In practice, however, this interpolation scheme is very ill-advised for two reasons. First, interpolation at uniformly spaced nodes often does not produce an accurate polynomial approximant. In fact, there are well-behaved functions for which uniform-node polynomial approximants rapidly deteriorate, rather than improve, as the degree of approximation n rises. A well known example is Runge's function, $f(x) = (1 + 25x^2)^{-1}$ on $[-1, 1]$. Second, the interpolation matrix Φ is a so-called Vandermonde matrix. Vandermonde matrices are notoriously ill-conditioned. Thus, efforts to compute the basis coefficients often fail due to rounding error, particularly as the degree of approximation is increased.

Numerical analysis theory and empirical experience both suggest that polynomial interpolation over a bounded interval $[a, b]$ should be performed using the *Chebychev nodes*

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos(\frac{n-i+0.5}{n}\pi), \qquad \forall i = 1, 2, \ldots, n.$$

As illustrated in figure 5.1, Chebychev nodes are not equally spaced. They are more closely spaced near the endpoints of the interpolation interval and less so near the center.

Figure 5.1: Chebychev nodes.

Chebychev-node polynomial approximants possess some strong theoretical properties. According to Rivlin's Theorem, Chebychev polynomial approximants are very nearly optimal polynomial approximants. Specifically, the approximation error associated with the n^{th}-degree Chebychev polynomial approximant cannot larger than $2\pi \log(n) + 2$ times the lowest error attainable with any other polynomial approximant of the same order. For $n < 100$, this factor is less than 5, which is very small when one considers that other polynomial interpolation schemes typically produce approximants with errors that are orders of magnitude, that is, powers of 10, larger. In practice, the accuracy afforded by the Chebychev polynomial approximant is often much better than indicated by Rivlin's bound.

Another theorem, Jackson's theorem, implies a more useful result. Specifically, if f is continuously differentiable, then the approximation error afforded by the n^{th}-degree Chebychev polynomial approximant p_n can be bounded above:

$$||f - p_n|| \leq \frac{6}{n}||f'||(b-a)(\log(n)/\pi + 1).$$

This error bound can often be accurately estimated in practice, giving the analyst a good indication of the accuracy afforded by the Chebychev polynomial approximant. More importantly, however, the error bound goes to zero as n rises. That is, unlike for uniform-node polynomial interpolation, one can achieve any desired degree of accuracy with a Chebychev-node polynomial interpolation by increasing the degree of approximation.

The best known basis for expressing polynomials is the monomial basis, which consists of the simple power functions $1, x, x^2, x^3, \ldots, x^n$. However, other polynomial bases exist. In particular, any sequence of n polynomials having exact orders $0, 1, 2, \ldots, n$ can serve as a basis for all polynomials of order n or less. This raises the question of whether some basis other

than the monomials makes a better choice for representing Chebychev-node polynomial approximants. Ideally, the interpolation equation associated with a basis should be well-conditioned and easy to solve.

Numerical analysis theory and practice suggest that Chebychev-node polynomial approximants should be expresses as linear combinations of the Chebychev polynomials, which are defined recursively as:

$$\phi_j(x) = T_j(2\frac{x-a}{b-a} - 1)$$

where, for $z \in [-1, 1]$,

$$\begin{aligned} T_1(z) &= 1 \\ T_2(z) &= z \\ T_3(z) &= 2z^2 - 1 \\ T_4(z) &= 4z^3 - 3z \\ &\vdots \\ T_j(z) &= 2zT_{j-1}(z) - T_{j-2}(z). \end{aligned}$$

The Chebychev polynomials also possess the alternate trigonometric definition

$$T_j(z) = \cos((j-1) \cdot \arccos(z)).$$

Chebychev polynomials are an excellent basis for interpolating functions at the Chebychev nodes because they yield extremely well-conditioned linear interpolation equation that can be accurately and efficiently solved, even for high degrees of interpolation. The interpolation matrix Φ associated with the Chebychev polynomial basis at the Chebychev nodes has typical element

$$\Phi_{ij} = \cos((n - i + 0.5)(j-1)\pi/n).$$

This matrix is orthogonal:

$$\Phi'\Phi = \text{diag}\{n, n/2, n/2, \ldots, n/2\}.$$

Moreover, for any degree of approximation n, the 2-norm condition number of the interpolation matrix Φ is $\sqrt{2}$, which is a very small condition number. This implies that the Chebychev basis coefficients can be computed accurately, regardless of the degree of interpolation.

	Chebychev Nodes and Chebychev Basis		Uniform Nodes and Monomial Basis	
Nodes	Approximation Error	Condition Number	Approximation Error	Condition Number
5	5.7e-01	1.414	6.2e+02	9.0e+02
10	3.2e-01	1.414	1.7e+06	5.1e+06
15	3.7e-02	1.414	5.9e+09	4.7e+10
20	1.1e-02	1.414	1.8e+13	4.9e+14
25	6.4e-04	1.414	5.7e+16	Inf

Table 5.1: Polynomial Approximation of $\exp(-x^2)$ on $[-1, 1]$

Table 5.1 gives the approximation errors associated with Chebychev interpolation and uniform-node monomial interpolation of the function $\exp(-x^2)$ for different number of nodes. The table also gives the 2-norm condition of the associated interpolation matrix. As one can see, the Chebychev approximation error falls with the number of interpolation nodes. The uniform-node monomial approximation error, on the other hand, grows with the number of nodes. This is largely due to the difficulty in accurately computing the coefficient vector as the condition of the interpolation matrix deteriorates.

Two computer routines for forming and evaluating Chebychev polynomial approximants are for computational economic analysis. One routine computes the vector of Chebychev nodes `xnodes` given the endpoints of the interpolation interval `a` and `b` and the degree of interpolation `n`:

```
function xnodes = nodecheb(n,a,b)
xnodes = 0.5*((a+b)-(b-a)*cos(((1:n)-0.5)*pi/n)');
```

The second routine takes as input an arbitrary vector `x`, the endpoints of the interpolation interval `a` and `b`, and the degree of interpolation `n`, and returns the values `phi` of the Chebychev basis functions in matrix form:

```
function phi = basecheb(x,n,a,b);
z = 2*(x-a)/(b-a) - 1;
phi = [ ones(size(z)) z ];
for j=3:n
   phi = [phi 2*z.*phi(:,j-1)-phi(:,j-2)];
```

```
end
```

The code can also be adapted to compute the first derivatives `phider` of the Chebychev basis polynomials at `x` by appending the following segment:

```
phi = [ ones(size(z)) z ];
phider = [ zeros(size(z)) ones(size(z)) ];
for j=3:n
   phider = [phider 2*z.*phider(:,j-1)+2*phi(:,j-1)-phider(:,j-2)];
end
phider = phider*2/(b-a);
```

Given the Chebychev node and Chebychev basis polynomial routines, Chebychev interpolation of a univariate function f on an interval $[a, b]$ in practice becomes straightforward. First, one selects the degree of interpolation n, and computes the Chebychev nodes `xnodes` using `nodecheb`:

```
xnodes = nodecheb(a,b,n)
```

Assuming that `func` is a Matlab function that returns a vector of values of the univariate function f at each element of an arbitrary vector `x`, one then computes the vector `c` of basis coefficients as follows:

```
c = basecheb(xnodes,n,a,b)\func(xnodes)
```

Once the basis coefficient vector `c` has been computed, the Chebychev approximant can subsequently be evaluated at an arbitrary vector of values `x` by evaluating the Chebychev basis functions at `x` and postmultiplying the resulting matrix by the coefficient vector:

```
y = basecheb(x,n,a,b)*c.
```

5.3 Linear Spline Interpolation

Splines are a rich, flexible class of functions that may be used instead of polynomials to approximate a real-valued function over a bounded interval. Generally, a k^{th} order spline consists of segments of polynomials of order k spliced together so as to preserve continuity of derivatives of order $k-1$. Two classes of splines are often employed in practice. A first-order or *linear spline* is a series of line segments spliced together to form a continuous function. A

third-order or *cubic spline* is a series of cubic polynomials segments spliced together to form a twice continuously differentiable function.

Linear spline approximants are the easiest approximants to construct and evaluate in practice, which largely explains their widespread popularity. An n^{th}-degree linear spline on the interval $[a, b]$ is any linear combination

$$\hat{f}(x) = \sum_{i=1}^{n} c_i \phi_i(x)$$

of the basis functions:

$$\phi_j(x) = \begin{cases} 1 - \frac{|x - t_j|}{w} & \text{if } |x - t_j| \leq w \\ 0 & \text{otherwise} \end{cases}$$

Here, $w = (b-a)/(n-1)$ and $t_j = a + (b-a)\frac{j-1}{n-1}$, $j = 1, 2, \ldots, n$, are called the knots of the n^{th}-degree linear spline. By construction, a linear spline is continuous on the interval $[a, b]$ and is linear on each subinterval $[t_i, t_{i+1}]$ defined by the knots.

The linear spline basis functions are popularly called the "hat" functions, for reasons that are made obvious in figure 5.2. Each hat function is zero everywhere, except over a narrow support element of width $2w$. The basis function achieves a maximum of 1 at the midpoint of its support element. At any point of $[a, b]$, at most two hat functions are nonzero.

Figure 5.2: Hat functions.

One can construct an n^{th}-degree linear spline approximant for a function f by interpolating its values at any n points of its domain, provided that the resulting interpolation matrix is nonsingular. However, if the interpolation nodes $x_1, x_2, \ldots, x_n$ are chosen to coincide with the spline knots $t_1, t_2, \ldots, t_n$, then computing the basis coefficients of the linear spline approximant becomes a trivial matter. If the interpolation nodes and knots coincide, then $\phi_i(x_j)$ equals one if $i = j$, but equals zero otherwise. That is, the interpolation matrix Φ is simply the identity matrix and the interpolation equation reduces to the trivial identity $c = y$ where y is the vector of function values

at the interpolation nodes. The linear spline approximant of f when nodes and knots coincide thus takes the form

$$\hat{f}(x) = \sum_{i=1}^{n} f(x_i)\phi_i(x).$$

When interpolation nodes and knots coincide, no computations other than function evaluations are required to form the linear spline approximant. For this reason linear spline interpolation nodes in practice are always chosen to be the spline's knots.

Evaluating a linear spline approximant and its derivative at an arbitrary point x is also very easy. Since at most two basis functions are nonzero at any point, only two basis function evaluations are required. Specifically, if i is the greatest integer less than $1 + (x - a)/w$, then x lies in the interval $[x_i, x_{i+1}]$. Thus,

$$\hat{f}(x) = (c_{i+1}(x - x_i) + c_i(x_{i+1} - x))/w$$

and

$$\hat{f}'(x) = (c_{i+1} - c_i)/w.$$

Higher order derivatives are zero, except at the knots, where they are undefined.

Two computer routines for forming and evaluating linear spline approximants are indispensable for computational economic analysis. One routine computes the vector of uniformly spaced nodes `xnodes` given the endpoints of the interpolation interval `a` and `b` and the degree of interpolation `n`:

```
function xnodes = nodeunif(n,a,b)
xnodes = a+(0:n-1)'*(b-a)/(n-1);
```

The second routine takes as input an arbitrary vector `x`, the endpoints of the interpolation interval `a` and `b`, and the degree of interpolation `n`, and returns the values `phi` of the linear spline basis functions in matrix form:

```
function phi = baselspl(x,n,a,b);
m = length(x);
xwid = (b-a)/(n-1);
j = floor((x-a)/xwid) + 1;
j = min(j,n-1); j = max(j,1);
theta = (x-a)/xwid - (j-1);
```

```
phi = zeros(m,n);
for i=1:m
   phi(i,j(i)+1) = theta(i);
   phi(i,j(i)) = 1 - theta(i);
end
```

The Matlab code employs the library subroutine `floor`, which gives the greatest integer that is less than or equal to its argument. The code can also be adapted to compute the first derivatives `phider` of the linear spline basis functions at `x` by appending the following segment:

```
phider = zeros(m,n);
for i=1:m
   phider(i,j(i)+1) = 1/xwid;
   phider(i,j(i)) = -1/xwid;
end
```

Given the uniform node and linear spline basis function routines, linear spline interpolation of a univariate function f on an interval $[a, b]$ in practice becomes straightforward. First, one selects the degree of interpolation n, and computes the uniformly spaced nodes `xnodes` using `nodeunif`:

```
xnodes = nodeunif(a,b,n)
```

Assuming that `func` is a Matlab function that returns a vector of values of the univariate function f at each element of an arbitrary vector `x`, one then computes the vector `c` of basis coefficients, which are simply the function values at the nodes, as follows:

```
c = func(xnodes)
```

Once the basis coefficient vector `c` has been computed, the linear spline approximant can subsequently be evaluated at an arbitrary vector of values `x` by evaluating the linear spline basis functions at `x` and postmultiplying the resulting matrix by the coefficient vector:

```
y = baselspl(x,n,a,b)*c.
```

Linear splines are attractive for their simplicity, but have certain limitations that sometimes make them a poor choice for computational economic applications. By construction, linear splines produce first derivatives that

are discontinuous step functions and second derivative that are zero everywhere, except at the interpolation nodes. Linear spline approximants thus typically do a very poor job of approximating the first derivative of a nonlinear function and are incapable of approximating its second derivative. In some economic applications, the derivative represents a measure of marginality that is of as much interest to the analyst as the function itself. In other applications, the derivative of the function may be needed to solve an iterative rootfinding problem. In such applications, linear spline functions are ill suited for function approximation.

5.4 Cubic Spline Interpolation

Cubic splines are piecewise cubic functions with continuous first and second derivatives. Cubic spline approximants offer a reasonable balance between the smoothness of polynomial approximants and the flexibility of linear spline approximants, and typically produce good approximations for both the function and its first and second derivatives.

Cubic spline basis functions are defined in terms of the canonical cubic B-spline

$$B(z) = \begin{cases} 0 & \text{if } z \leq -2 \\ \frac{4}{3}(1+z)^3 & \text{if } -2 \leq z \leq -1 \\ \frac{2}{3}(1-6z^2(1+z)) & \text{if } -1 \leq z \leq 0 \\ \frac{2}{3}(1-6z^2(1-z)) & \text{if } 0 \leq z \leq 1 \\ \frac{4}{3}(1-z)^3 & \text{if } 1 \leq z \leq 2 \\ 0 & \text{if } z \geq 2 \end{cases}$$

The cubic B-spline, which is pictured in figure *, is clearly piecewise cubic. Moreover, it may be easily verified that the cubic B-spline is twice continuously differentiable at the splice points -2, 1, 0, 1, and 2.

Figure 5.3: Canonical Cubic B-Spline.

Two types of cubic splines are used frequently in computational economic analysis. The "natural" spline is characterized by the additional condition

that the second derivatives vanish at the endpoints of the approximation interval. An n^{th}-degree natural cubic spline on the interval $[a, b]$ is any linear combination of the basis functions:

$$\phi_j(x) = \begin{cases} B((x-t_1)/w) + 2B((x-t_0)/w) & \text{if } j = 1 \\ B((x-t_2)/w) - 1B((x-t_0)/w) & \text{if } j = 2 \\ B((x-t_j)/w) & \text{if } 1 < j < n \\ B((x-t_{n-1})/w) - 1B((x-t_{n+1})/w) & \text{if } j = n-1 \\ B((x-t_n)/w) + 2B((x-t_{n+1})/w) & \text{if } j = n \end{cases}$$

Here, $w = (b-a)/(n-1)$ and $t_j = a + (b-a)\frac{j-1}{n-1}$, $j = 1, 2, \ldots, n$, are called the knots of the spline. By construction, a natural cubic spline is twice continuous on the interval $[a, b]$, is cubic on each subinterval $[t_i, t_{i+1}]$ defined by the knots, and has vanishing second derivatives at the knots t_1 and t_n.

The "not-a-knot" spline is characterized by the additional condition that the third derivatives at the outermost interior knots be continuous. An n^{th}-degree not-a-knot cubic spline on the interval $[a, b]$ is any linear combination of the basis functions:

$$\phi_j(x) = \begin{cases} B((x-t_1)/w) + 4B((x-t_0)/w) & \text{if } j = 1 \\ B((x-t_2)/w) - 6B((x-t_0)/w) & \text{if } j = 2 \\ B((x-t_3)/w) + 4B((x-t_0)/w) & \text{if } j = 3 \\ B((x-t_4)/w) - 1B((x-t_0)/w) & \text{if } j = 4 \\ B((x-t_j)/w) & \text{if } 4 < j < n-3 \\ B((x-t_{n-3})/w) - 1B((x-t_{n+1})/w) & \text{if } j = n-3 \\ B((x-t_{n-2})/w) + 4B((x-t_{n+1})/w) & \text{if } j = n-2 \\ B((x-t_{n-1})/w) - 6B((x-t_{n+1})/w) & \text{if } j = n-1 \\ B((x-t_n)/w) + 4B((x-t_{n+1})/w) & \text{if } j = n \end{cases}$$

By construction, a not-a-knot cubic spline is twice continuous on the interval $[a, b]$, is cubic on each subinterval $[t_i, t_{i+1}]$ defined by the knots, and has continuous third derivatives at the knots t_2 and t_{n-1}. As illustrated in figure 5.3, the not-a-knot cubic spline basis functions are each nonzero over a support element of width $4w$. Thus, at any point of $[a, b]$, at most four basis functions are nonzero.

Figure 5.4: Cubic spline basis functions.

One can construct an n^{th}-degree cubic spline approximant for a function f by interpolating its values at any n points of its domain, provided that the resulting interpolation matrix is nonsingular. However, if the interpolation nodes $x_1, x_2, \ldots, x_n$ are chosen to coincide with the spline knots $t_1, t_2, \ldots, t_n$, then it becomes much easier to compute the basis coefficients of the cubic spline approximant. If the interpolation nodes and knots coincide, the interpolation matrix Φ will be nearly tridiagonal, that is, for the most part, only the elements of the matrix one index removed from the diagonal will be nonzero.

The band diagonality of the interpolation matrix assures that the interpolation equation will be well-conditioned and allows sparse matrix storage and computation to be used to form and evaluate cubic spline approximants, substantially speeding up the process. If the interpolation matrix must be reused, one must resist the temptation to form and store its inverse, particularly if the size of the matrix is large. Inversion destroys the sparsity structure. More specifically, the inverse of the interpolation matrix will be dense, even though the interpolation matrix is not. When n is large, solving the sparse n by n linear equation using sparse L-U factorization will generally be less costly than performing the matrix-vector multiplication required with the dense inverse interpolation matrix.

Two computer routines for forming and evaluating cubic spline approximants are indispensable for computational economic analysis. One routine, previously introduced in the linear spline section under the name `nodeunif` computes the vector of uniformly spaced nodes `xnodes` given the endpoints of the interpolation interval `a` and `b` and the degree of interpolation `n`. The second routine takes as input an arbitrary vector `x`, the endpoints of the interpolation interval `a` and `b`, and the degree of interpolation `n`, and returns the values `phi` of the cubic spline basis functions in matrix form. The subroutine has a calling sequence of the form:

```
phi = basecspl(x,n,a,b);
```

The code can also be adapted to compute the first derivatives `phider` of the cubic spline basis functions at `x`. In the Matlab m-file subroutines accompanying this textbook, the not-a-knot condition is used to well-define the cubic spline approximant.

Given the uniform node and cubic spline basis routines, cubic spline interpolation of a univariate function f on an interval $[a, b]$ in practice becomes

straightforward. First, one selects the degree of interpolation n, and computes the uniform nodes `xnodes` using `nodeunif`:

```
xnodes = nodeunif(a,b,n)
```

Assuming that `func` is a Matlab function that returns a vector of values of the univariate function f at each element of an arbitrary vector `x`, one then computes the vector `c` of basis coefficients as follows:

```
c = basecspl(xnodes,n,a,b)\func(xnodes)
```

Once the basis coefficient vector `c` has been computed, the cubic spline approximant can subsequently be evaluated at an arbitrary vector of values `x` by evaluating the cubic spline basis functions at `x` and postmultiplying the resulting matrix by the coefficient vector:

```
y = basecspl(x,n,a,b)*c.
```

5.5 Choosing an Approximation Method

The most significant difference between spline and polynomial interpolation methods is that spline basis functions have narrow supports, but polynomial basis functions have supports that cover the entire interpolation interval. This can lead to big differences in the quality of approximation when the function being approximated is irregular. Discontinuities in the first or second derivatives can create problems for all interpolation schemes. However, spline functions, due to their narrow support, can often contain the effects of such discontinuities. Polynomial approximants, on the other hand, allow the ill effects of discontinuities to propagate over the entire interval of interpolation. Thus, when a function exhibits kinks, spline interpolation may be preferable to polynomial interpolation.

In order to illustrate the differences between spline and polynomial interpolation, we compare in table 5.2 the approximation error for four different functions and four different approximation schemes: linear spline interpolation, cubic spline interpolation, uniform node polynomial interpolation, and Chebychev polynomial interpolation. The results presented in the table lend support to certain rules of thumb used by experienced computational analysts. When comparing interpolation schemes of the same degree of approximation:

Function	Degree	Linear Spline	Cubic Spline	Uniform Polynomial	Chebychev Polynomial
$1 + x$	10	0.10E+00	0.30E−08	0.22E−14	0.89E−14
$+2x^2 - 3x^3$	20	0.26E−01	0.15E−08	0.10E−12	0.75E−14
	30	0.12E−01	0.10E−08	0.67E−10	0.30E−13
$\exp(-x)$	10	0.12E−01	0.11E−04	0.24E−09	0.27E−10
	20	0.32E−02	0.70E−06	0.24E−12	0.33E−14
	30	0.15E−02	0.14E−06	0.26E−10	0.16E−13
$(1 + 25x^2)^{-1}$	10	0.67E−01	0.22E−01	0.19E+01	0.11E+00
	20	0.42E−01	0.32E−02	0.60E+02	0.15E−01
	30	0.23E−01	0.82E−03	0.24E+04	0.21E−02
$\lvert x\rvert^{0.5}$	10	0.11E+00	0.18E+00	0.22E+01	0.22E+00
	20	0.79E−01	0.12E+00	0.45E+03	0.16E+00
	30	0.65E−01	0.10E+00	0.18E+06	0.13E+00

Table 5.2: Approximation Error for Selected Interpolation Methods

1. Chebychev node polynomial interpolation dominates uniform node polynomial interpolation.

2. Cubic spline interpolation dominates linear spline interpolation, except where the approximant exhibits a profound discontinuity.

3. Chebychev polynomial interpolation dominates cubic spline interpolation if the approximant is smooth and monotonic; otherwise, cubic or even linear spline interpolation may be preferred.

5.6 Multidimensional Interpolation

Interpolation schemes for multivariate functions may be developed by forming the appropriate products of univariate bases and nodes.

Consider first the problem of approximating a bivariate real-valued function $f(x, y)$ defined on a bounded interval $I = \{(x, y) \mid a_x \leq x \leq b_x, a_y \leq y \leq b_y\}$ in $\Re^2$. Suppose that ϕ_i^x, $i = 1, 2, \ldots, n_x$ and ϕ_j^y, $j = 1, 2, \ldots, n_y$ are basis functions for univariate functions defined on $[a_x, b_x]$ and $[a_y, b_y]$, respectively. Then an $n = n_x n_y$ degree basis for f on I may be constructed by letting

$$\phi_{ij}(x, y) = \phi_i^x(x)\phi_j^y(y) \qquad \forall i = 1, \ldots, n_x; j = 1, \ldots, n_y.$$

Similarly, a grid of $n = n_x n_y$ interpolation nodes can be constructed by taking the Cartesian product of univariate interpolation nodes. More specifically, if $x_1, x_2, \ldots x_{n_x}$ and $y_1, y_2, \ldots, y_{n_y}$ are n_x and n_y interpolation nodes in $[a_x, b_x]$ and $[a_y, b_y]$, respectively, then n nodes for interpolating f on I may be constructed by letting

$$\{(x_i, y_j) \mid i = 1, 2, \ldots, n_x; j = 1, 2, \ldots, n_y\}.$$

For example, suppose one wishes to approximate a function using a cubic polynomial in the x direction and a quadratic polynomial in the y direction. A tensor product basis constructed from the simple monomial basis of x and y comprises the following functions

$$1, x, y, xy, x^2, y^2, xy^2, x^2y, x^2y^2, x^3, x^3y, x^3y^2.$$

The dimension of the basis is 12. An approximant expressed in terms of the tensor product basis would take the form

$$\hat{f}(x, y) = \sum_{i=1}^{3} \sum_{j=1}^{2} c_{ij} x^{i-1} y^{j-1}.$$

Typically, tensor product node-basis schemes inherit the favorable qualities of their univariate node-basis parents. For example, if a bivariate linear spline basis is used and the interpolation nodes $\{x_i, y_j\}$ are chosen such that the x_i are uniformly spaced on $[a_x, b_x]$ and the y_j are uniformly spaced on $[a_y, b_y]$, then the interpolation matrix will be the identity matrix, just like in the univariate case. Also, if a bivariate Chebychev polynomial basis is used, and the bivariate nodes $\{x_i, y_j\}$ are chosen such that the x_i are the Chebychev nodes on $[a_x, b_x]$ and the y_j are the Chebychev nodes on $[a_y, b_y]$, then the interpolation matrix will be orthogonal.

Tensor product schemes can be developed similarly for higher than two dimensions. Consider the problem of interpolating a d-variate function $f(x_1, x_2, \ldots, x_d)$ on a d-dimensional interval $I = \{(x_1, x_2, \ldots, x_d) \mid a_i \leq x_i \leq b_i, i = 1, 2, \ldots, d\}$. If ϕ_{ij}, $j = 1, ..., n_i$ is a n_i degree univariate basis for real-valued functions of on $[a_i, b_i]$, then an approximant for f in the tensor product basis would take the following form:

$$\hat{f}(x) = \hat{f}(x_1, x_2, \ldots, x_d) = \sum_{j_1=1}^{n_1} \sum_{j_2=1}^{n_2} \ldots \sum_{j_d=1}^{n_d} \phi_{1j_1}(x_1)\phi_{2j_1}(x_2)\ldots\phi_{dj_1}(x_d).$$

Using tensor notation the approximating function an be written

$$\hat{f}(x) = [\phi_1(x_1) \diagdown \phi_2(x_2) \diagdown \ldots \diagdown \phi_d(x_d)] \star c.$$

where c is a column vector with $n = \Pi_{i=1}^{d} n_i$ elements.

To implement interpolation in multiple dimensions it is necessary to evaluate solve the interpolation equation. If Φ_i is the degree n_i interpolation matrix associated with variable x_i, then the interpolation conditions for the multivariate function can be written

$$[\Phi_1 \diagdown \Phi_2 \diagdown \ldots \diagdown \Phi_d)] \star c = f(x)$$

where $f(x)$ is an n by 1 vector of function values evaluated at the interpolation nodes x, properly stacked. Using a standard result from tensor matrix algebra, the this system may be solved by forming the projection matrix and postmultiplying it by the data vector:

$$c = [\Phi_1^{-1} \diagdown \Phi_2^{-1} \diagdown \ldots \diagdown \Phi_d^{-1}] \star f(x);$$

Hence there is no need to invert an n by n multivariate interpolation matrix to determine the interpolating coefficients. Instead, each of the univariate interpolation matrices may be inverted individually and then multiplied together.

This leads to substantial savings in storage and computational effort. For example, if the problem is 3-dimensional and the are 10 evaluation points in each dimension, to obtain the projection matrix for interpolation only three 10 by 10 matrices need to be inverted, rather than a single 1000 by 1000 matrix.

However, interpolation using tensor product schemes tends to become computationally more challenging as the dimensions rise. With a one-dimensional argument the number of interpolation nodes and the dimension of the interpolation matrix can generally be kept small with good results. For a relatively smooth function, Chebychev polynomial approximants of order 10 or less can often provide extremely accurate approximations to a function and its derivatives. If the function's argument is d-dimensional one could approximate the function using the same number of points in each dimension, but this increases the number of interpolation nodes to 10^d and the size of the interpolation matrix to 10^{2d} elements. Storing the interpolation matrix alone may be thus become difficult, to say nothing about the problems of inverting it.

The tendency of computational effort to grow exponentially with the dimension of the function being interpolated is known as the *curse of dimensionality*. In order to mitigate the effects of the curse requires that careful attention be paid to both storage and computational efficiency. A natural way to address both issues and one that is, in addition, a straightforward extension of one-dimensional problems, is to use tensor product bases.