Chapter 2

Polynomial Interpolation

In the problem of *data approximation*, we are given some points $(x_1, y_1), \ldots, (x_n, y_n)$ and are asked to find a function $\phi(x)$ that "captures the trend" of the data. If the trend is one of decay, then we may seek a ϕ of the form $a_1e^{-\lambda_1 x} + a_2e^{-\lambda_2 x}$. If the trend of the data is oscillatory, then a trigonometric approximant might be appropriate. Other settings may require a low-degree polynomial. Regardless of the type of function used, there are many different metrics for success, e.g., least squares.

A special form of the approximation problem ensues if we insist that ϕ actually "goes through" the data, as shown in Figure 2.1. This means that $\phi(x_i) = y_i$, $i = 1{:}n$ and we say that ϕ *interpolates* the data. The polynomial interpolation problem is particularly important:

> *Given $x_1, \ldots, x_n$ (distinct) and $y_1, \ldots, y_n$, find a polynomial $p_{n-1}(x)$ of degree $n-1$ (or less) such that $p_{n-1}(x_i) = y_i$ for $i = 1{:}n$.*

Thus, $p_2(x) = 1 + 4x - 2x^2$ interpolates the points $(-2, -15)$, $(3, -5)$, and $(1, 3)$.

Each (x_i, y_i) pair can be regarded as a snapshot of some function $f(x)$: $y_i = f(x_i)$. The function f may be explicitly available, as when we want to interpolate $\sin(x)$ at $x = 0, \pi/2$, and π with a quadratic. On other occasions, f is implicitly defined, as when we want to interpolate the solution to a differential equation at a discrete number of points.

The discussion of polynomial interpolation revolves around how it can be represented, computed, and evaluated:

- How do we *represent* the interpolant $p_{n-1}(x)$? Instead of expressing the interpolant in terms of the "usual" basis polynomials 1, x, and x^2, we could use the alternative basis 1, $(x+2)$, and $(x+2)(x-3)$. Thus,
$$p_2(x) = -15 + 2(x+2) - 2(x+2)(x-3)$$
is another way to express the quadratic interpolant of the data $(-2, -15)$, $(3, -5)$, and $(1, 3)$. Different bases have different computational virtues.

- Once we have settled on a representation for the polynomial interpolant, how do we determine the associated coefficients? It turns out that this aspect of the problem involves the solution of a linear system of equations with a highly structured coefficient matrix.

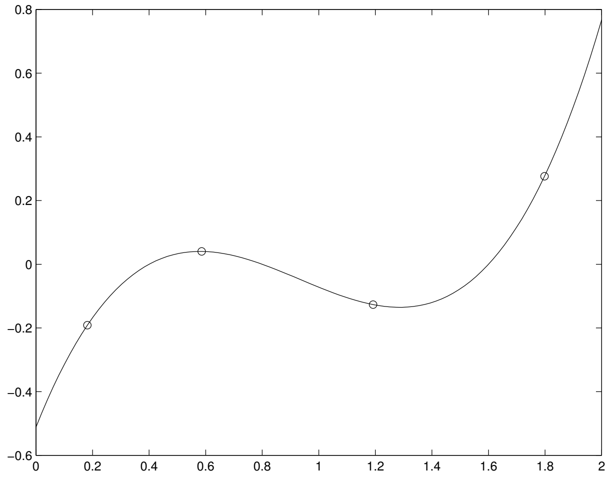


FIGURE 2.1 *The interpolation of four data points with a cubic polynomial*

- After we have computed the coefficients, how can the interpolant be evaluated with efficiency? For example, if the interpolant is to be plotted then we are led to the problem of evaluating a polynomial on a vector of values.

In MATLAB these issues can be handled by `polyfit` and `polyval`. The script

```
x = [-2 3 1];
y = [-15 -5 3];
a = polyfit(x,y,2)
xvals = linspace(-3,2,100);
pvals = polyval(a,xvals);
plot(xvals,pvals)
```

plots the polynomial interpolant of the data $(-2,-15)$, $(3,-5)$, and $(1,3)$. The interpolant is given by $p(x) = 1 + 4x - 2x^2$ and the call to `polyfit` computes a representation of this polynomial. In particular, `a` is assigned the vector `[-2 4 1]`.

In general, if `x` and `y` are n-vectors, then `a = polyfit(x,y,n-1)` assigns a length-n vector to `a` with the property that the polynomial

$$p(x) = a_n + a_{n-1}x + a_{n-2}x^2 + \cdots + a_1 x^{n-1}$$

interpolates the data $(x_1, y_1), \ldots, (x_n, y_n)$.

The function `polyval` is used to evaluate polynomials in the MATLAB representation. In the above script `polyval(a,xvals)` is a vector of interpolant evaluations.

In this chapter we start with what we call the "Vandermonde" approach to the polynomial interpolation problem. The Newton representation is considered in §2.2 and accuracy issues in §2.3. Divided differences, inverse interpolation, interpolation in the plane, and trigonmetric interpolation are briefly discussed in §2.4.

2.1 The Vandermonde Approach

In the Vandermonde approach, the interpolant is expressed as a linear combination of 1, x, x^2, etc. Although monomials are not the best choice for a basis, our familiarity with this way of "doing business" with polynomials makes them a good choice to initiate the discussion.

2.1.1 A Four-point Interpolation Problem

Let us find a cubic polynomial

$$p_3(x) = a_1 + a_2x + a_3x^2 + a_4x^3$$

that interpolates the four data points $(-2, 10)$, $(-1, 4)$, $(1, 6)$, and $(2, 3)$. Note that this is the "reverse" of MATLAB 's convention for representing polynomials. [1] Each point of interpolation leads to a linear equation that relates the four unknowns a_1, a_2, a_3, and a_4:

$$\begin{array}{lcl} p_3(-2) = 10 & \Rightarrow & a_1 - 2a_2 + 4a_3 - 8a_4 = 10 \\ p_3(-1) = 4 & \Rightarrow & a_1 - a_2 + a_3 - a_4 = 4 \\ p_3(1) = 6 & \Rightarrow & a_1 + a_2 + a_3 + a_4 = 6 \\ p_3(2) = 3 & \Rightarrow & a_1 + 2a_2 + 4a_3 + 8a_4 = 3 \end{array}$$

Expressing these four equations in matrix/vector terms gives

$$\begin{bmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \\ 6 \\ 3 \end{bmatrix}.$$

The solution $a = [4.5000\ 1.9167\ 0.5000\ \ -0.9167]^T$ to this 4-by-4 system can be found as follows:

```
y = [10; 4; 6; 3];
V = [1 -2 4 -8; 1 -1 1 -1; 1 1 1 1; 1 2 4 8];
a = V\y;
```

2.1.2 The General n Case

From this example, it looks like the polynomial interpolation problem reduces to a linear equation problem. For general n, the goal is to determine $a_1, \ldots, a_n$ so that if

$$p_{n-1}(x) = a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1},$$

then

$$p_{n-1}(x_i) \ = \ a_1 + a_2x_i + a_3x_i^2 + \cdots + a_nx_i^{n-1} \ = \ y_i$$

for $i = 1{:}n$. By writing these equations in matrix-vector form, we obtain

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}.$$

Designate the matrix of coefficients by V. The solvability of the interpolation problem hinges on the nonsingularity of V. Suppose there is a vector c such that $Vc = 0$. It follows that the polynomial

$$q(x) = c_1 + c_2x + \cdots + c_nx^{n-1}$$

is zero at $x = x_1, \ldots, x = x_n$. This says that we have a degree $n - 1$ polynomial with n roots. The only way that this can happen is if q is the zero polynomial (i.e., $c = 0$). Thus V is nonsingular because the only vector that it zeros is the zero vector.

[1] MATLAB would represent the sought-after cubic as $p_3 = a_4 + a_3x + a_2x^2 + a_1x^3$. Our chosen style is closer to what one would find in a typical math book: $p_3(x) = a_0 + a_1x + a_2x^2 + a_3x^3$.

2.1.3 Setting Up and Solving the System

Let us examine the construction of the Vandermonde matrix V. Our first method is based on the observation that the ith row of V involves powers of x_i and that the powers increase from 0 to $n-1$ as the row is traversed from left to right. A conventional double-loop approach gives

```
n = length(x); V = zeros(n,n);
for i=1:n
   % Set up row i.
   for j=1:n
      V(i,j) = x(i)^(j-1);
   end
end
```

Algorithms that operate on a two-dimensional array in row-by-row fashion are *row oriented*.

The inner-loop in the preceding script can be *vectorized* because MATLAB supports pointwise exponentiation. For example, `u = [1 2 3 4] .^[3 5 2 3]` assigns to `u` the row vector [1 32 9 64]. The i-th row of V requires exponentiating the scalar x_i to each of the values in the row vector $0{:}n-1 = (0,\ 1,\ \ldots,\ n-1)$. Thus, `row = (x(i)*ones(1,n)).^(0:n-1)` assigns the vector $(1, x_i, x_i^2, \ldots, x_i^{n-1})$ to `row`, precisely the values that make up the ith row of V. The ith row of a matrix V may be referenced by `V(i,:)`, and so we obtain

```
n = length(x); V = zeros(n,n);
for i=1:n
   % Set up the i-th row of V.
   V(i,:)  = (x(i)*ones(1,n)).^(0:n-1);
end
```

By reversing the order of the loops in the original set-up script, we obtain a *column oriented* algorithm:

```
n = length(x); V = zeros(n,n);
for j=1:n
   % Set up column j.
   for i=1:n
      V(i,j) = x(i)^(j-1);
   end
end
```

If $j > 1$, then $V(i,j)$ is the product of $x(i)$ and $V(i,j-1)$, the matrix entry to its left. This suggests that the required exponentiations can be obtained through repeated multiplication:

```
n = length(x);
V = ones(n,n);
for j=2:n
   % Set up column j.
   for i=1:n
      V(i,j) = x(i)*V(i,j-1)
   end
end
```

The generation of the jth column involves *pointwise vector multiplication*:

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} .* \begin{bmatrix} v_{1,j-1} \\ \vdots \\ v_{n,j-1} \end{bmatrix} = \begin{bmatrix} v_{1,j} \\ \vdots \\ v_{n,j} \end{bmatrix}.$$

This may be implemented by `V(:,j) = x .* V(:,j-1)`. Basing our final implementation on this, we obtain

```
  function a = InterpV(x,y)
% a = InterpV(x,y)
% This computes the Vandermonde polynomial interpolant where
% x is a column n-vector with distinct components and y is a
% column n-vector.
%
% a is a column n-vector with the property that if
%
%        p(x) = a(1) + a(2)x + ... a(n)x^(n-1)
% then
%        p(x(i)) = y(i), i=1:n

n = length(x);
V = ones(n,n);
for j=2:n
   % Set up column j.
   V(:,j) = x.*V(:,j-1);
end
a = V\y;
```

Column-oriented, matrix-vector implementations will generally be favored in this text. One reason for doing this is simply to harmonize with the traditions of linear algebra, which is usually taught with a column-oriented perspective.

2.1.4 Nested Multiplication

We now consider the evaluation of $p_{n-1}(x) = a_1 + \cdots + a_n x^{n-1}$ at $x = z$, assuming that `z` and `a(1:n)` are available. The routine approach

```
n = length(a);
zpower = 1;
pVal = a(1);
for i=2:n
   zpower = z*zpower;
   pVal = pVal + a(i)*zpower;
end
```

assigns the value of $p_{n-1}(z)$ to `pVal`.

A more efficient algorithm is based on a nested organization of the polynomial, which we illustrate for the case $n = 4$:

$$p_3(x) = a_1 + a_2x + a_3x^2 + a_4x^3 = ((a_4x + a_3)x + a_2)x + a_1.$$

Note that the fragment

```
pVal = a(4);
pVal = z*pVal + a(3);
pVal = z*pVal + a(2);
pVal = z*pVal + a(1);
```

assigns the value of $p_3(z)$ to `pVal`. For general n, this nested multiplication idea takes on the following form:

```
n = length(a);
pVal = a(n);
for i=n-1:-1:1
   pVal = z*pVal + a(i);
end
```

This is widely known as *Horner's rule.*

Before we encapsulate the Horner idea in a MATLAB function, let us examine the case when the interpolant is to be evaluated at many different points. To be precise, suppose `z(1:m)` is initialized and that for $i = 1{:}m$, we want to assign the value of $p_{n-1}(z(i))$ to `pVal(i)`. One obvious approach is merely to repeat the preceding Horner iteration at each point. Instead, we develop a vectorized implementation that can be obtained if we think about the "simultaneous" evaluation of the interpolants at each z_i. Suppose $m = 5$ and $n = 4$ (i.e, the case when a cubic interpolant is to be evaluated at five different points). The first step in the five applications of the Horner idea may be summarized as follows:

$$\begin{bmatrix} \texttt{pVal(1)} \\ \texttt{pVal(2)} \\ \texttt{pVal(3)} \\ \texttt{pVal(4)} \\ \texttt{pVal(5)} \end{bmatrix} = \begin{bmatrix} \texttt{a(4)} \\ \texttt{a(4)} \\ \texttt{a(4)} \\ \texttt{a(4)} \\ \texttt{a(4)} \end{bmatrix}.$$

In vector terms `pVal = a(n)*ones(m,1)`. The next step requires a multiply-add of the following form:

$$\begin{bmatrix} \texttt{pVal(1)} \\ \texttt{pVal(2)} \\ \texttt{pVal(3)} \\ \texttt{pVal(4)} \\ \texttt{pVal(5)} \end{bmatrix} = \begin{bmatrix} \texttt{z(1)*pVal(1)} \\ \texttt{z(2)*pVal(2)} \\ \texttt{z(3)*pVal(3)} \\ \texttt{z(4)*pVal(4)} \\ \texttt{z(5)*pVal(5)} \end{bmatrix} + \begin{bmatrix} \texttt{a(3)} \\ \texttt{a(3)} \\ \texttt{a(3)} \\ \texttt{a(3)} \\ \texttt{a(3)} \end{bmatrix}.$$

That is,

```
pVal = z.*pVal + a(3)
```

The pattern is clear for the cubic case:

```
pVal = a(4)*ones(m,1);
pVal = z .* pVal + a(3);
pVal = z .* pVal + a(2);
pVal = z .* pVal + a(1);
```

From this we generalize to the following:

```
  function pVal = HornerV(a,z)
% pVal = HornerV(a,z)
% evaluates the Vandermonde interpolant on z where
% a is an n-vector and z is an m-vector.
%
% pVal is a vector the same size as z with the property that if
%
%         p(x) = a(1) + .. +a(n)x^(n-1)
% then
%         pVal(i) = p(z(i))  ,  i=1:m.

n = length(a);
m = length(z);
pVal = a(n)*ones(size(z));
for k=n-1:-1:1
   pVal = z.*pVal + a(k);
end
```

Each update of `pval` requires $2m$ flops so approximately $2mn$ flops are required in total.

As an application, here is a script that displays cubic interpolants of $\sin(x)$ on $[0, 2\pi]$. The abscissas are chosen randomly.

```
% Script File: ShowV
% Plots 4 random cubic interpolants of sin(x) on [0,2pi].
% Uses the Vandermonde method.

close all
x0 = linspace(0,2*pi,100)';
y0 = sin(x0);
for eg=1:4
   x = 2*pi*sort(rand(4,1));
   y = sin(x);
   a = InterpV(x,y);
   pVal = HornerV(a,x0);
   subplot(2,2,eg)
   plot(x0,y0,x0,pVal,'--',x,y,'*')
   axis([0 2*pi -2 2])
end
```

Figure 2.2 displays a sample output.

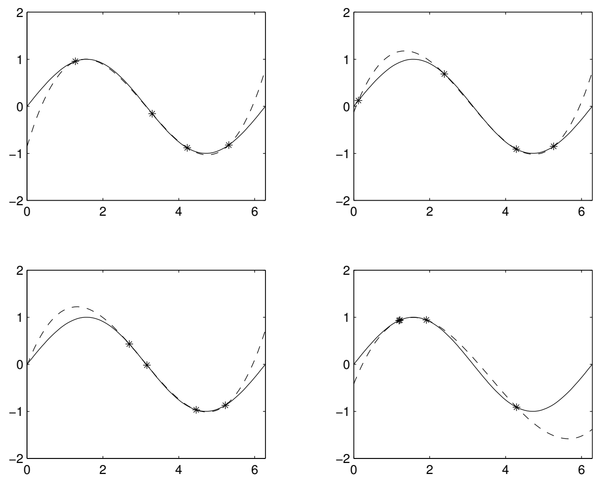


FIGURE 2.2 *Random cubic interpolants of* $\sin(x)$ *on* $[0, 2\pi]$

Problems

P2.1.1 Instead of expressing the polynomial interpolant in terms of the basis functions $1, x, \ldots, x^{n-1}$, we can work with the alternative representation

$$p_{n-1}(x) = \sum_{k=1}^{n} a_k \left(\frac{x-u}{v}\right)^{k-1}.$$

Here u and v are scalars that serve to shift and scale the x-range. Generalize `InterpV` so that it can be called with either two, three, or four arguments. A call of the form `a = InterpV(x,y)` should assume that $u = 0$ and $v = 1$. A call of the form `a = InterpV(x,y,u)` should assume that $v = 1$ and that `u` houses the shift factor. A call of the form `a = InterpV(x,y,u,v)` should assume that `u` and `v` house the shift and scale factors, respectively.

P2.1.2 A polynomial of the form

$$p(x) = a_1 + a_2x^2 + \cdots + a_mx^{2m-2}$$

is said to be *even*, while a polynomial of the form

$$p(x) = a_1x + a_3x^3 + \cdots + a_mx^{2m-1}$$

is said to be *odd*. Generalize `HornerV(a,z)` so that it has an optional third argument `type` that indicates whether or not the underlying polynomial is even or odd. In particular, a call of the form `HornerV(a,z,'even')` should assume that a_k is the coefficient of x^{2k-2}. A call of the form `HornerV(a,z,'odd')` should assume that a_k is the coefficient of x^{2k-1}.

P2.1.3 Assume that `z` and `a(1:n)` are initialized and define

$$p(x) = a_1 + a_2x + \cdots + a_nx^{n-1}.$$

Write a script that evaluates (1) $p(z)/p(-z)$, (2) $p(z) + p(-z)$, (3) $p'(z)$, (4) $\int_0^1 p(x)dx$, and (e) $\int_{-z}^{z} p(x)dx$. Make effective use of `HornerV`.

P2.1.4 (a) Assume that `L` (scalar), `R` (scalar), and `c(1:4)` are given. Write a script that computes `a(1:4)` so that if $p(x) = a_1 + a_2x + a_3x^2 + a_4x^3$, then $p(L) = c_1$, $p'(L) = c_2$, $p''(L) = c_3$, and $p(R) = c_4$. Use \ to solve any linear system that arises in your method. (b) Write a function `a = TwoPtInterp(L,cL,R,cR)` that returns the coefficients of a polynomial $p(x) = a_1 + a_2x + \cdots + a_nx^n$ that satisfies $p^{(k-1)}(L) =$ `cL(k)` for $k = 1$:`length(cL)` and $p^{(k-1)}(R) =$ `cR(k)` for $k = 1$:`length(cR)`. The degree of p should be one less than the total number of end conditions. (The problem of determining a cubic polynomial whose value and slope are prescribed at two points is discussed in detail in §3.2.1. It is referred to as the *cubic Hermite interpolation problem.*)

P2.1.5 Write a function `PlotDerPoly(x,y)` that plots the derivative of the polynomial interpolant of the data (x_i, y_i), $i = 1{:}n$. Assume that $x_1 < \cdots < x_n$ and the plot should be across the interval $[x_1, x_n]$. Use `polyfit` and `polyval`.

2.2 The Newton Representation

We now look at a form of the polynomial interpolant that is generally more useful than the Vandermonde representation.

2.2.1 A Four-Point Example

To motivate the idea, consider once again the problem of interpolating the four points (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4) with a cubic polynomial $p_3(x)$. However, instead of expressing the interpolant in terms of the "canonical" basis 1, x, x^2, and x^3, we use the basis 1, $(x-x_1)$, $(x-x_1)(x-x_2)$, and $(x-x_1)(x-x_2)(x-x_3)$. This means that we are looking for coefficients c_1, c_2, c_3, and c_4 so that if

$$p_3(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + c_4(x - x_1)(x - x_2)(x - x_3), \tag{2.1}$$

then $y_i = p_3(x_i) = y_i$ for $i = 1{:}4$. In expanded form, these four equations state that

$$\begin{aligned}
y_1 &= c_1 \\
y_2 &= c_1 + c_2(x_2 - x_1) \\
y_3 &= c_1 + c_2(x_3 - x_1) + c_3(x_3 - x_1)(x_3 - x_2) \\
y_4 &= c_1 + c_2(x_4 - x_1) + c_3(x_4 - x_1)(x_4 - x_2) + c_4(x_4 - x_1)(x_4 - x_2)(x_4 - x_3).
\end{aligned}$$

By rearranging these equations, we obtain the following four-step solution process:

$$\begin{aligned}
c_1 &= y_1 \\
c_2 &= \frac{y_2 - c_1}{x_2 - x_1} \\
c_3 &= \frac{y_3 - (c_1 + c_2(x_3 - x_1))}{(x_3 - x_1)(x_3 - x_2)} \\
c_4 &= \frac{y_4 - (c_1 + c_2(x_4 - x_1) + c_3(x_4 - x_1)(x_4 - x_2))}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)}.
\end{aligned}$$

This sequential solution process is made possible by the clever choice of the basis polynomials and the result is the *Newton representation* of the interpolating polynomial.

To set the stage for the general-n algorithm, we redo the $n = 4$ case using matrix-vector notation to discover a number of simplifications. The starting point is the system of equations that we obtained previously which can be expressed in the following form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & (x_2 - x_1) & 0 & 0 \\ 1 & (x_3 - x_1) & (x_3 - x_1)(x_3 - x_2) & 0 \\ 1 & (x_4 - x_1) & (x_4 - x_1)(x_4 - x_2) & (x_4 - x_1)(x_4 - x_2)(x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}.$$

From this we see immediately that $c_1 = y_1$. We can eliminate c_1 from equations 2, 3, and 4 by subtracting equation 1 from equations 2, 3, and 4:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & (x_2 - x_1) & 0 & 0 \\ 0 & (x_3 - x_1) & (x_3 - x_1)(x_3 - x_2) & 0 \\ 0 & (x_4 - x_1) & (x_4 - x_1)(x_4 - x_2) & (x_4 - x_1)(x_4 - x_2)(x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 - y_1 \\ y_3 - y_1 \\ y_4 - y_1 \end{bmatrix}.$$

If we divide equations 2, 3, and 4 by $(x_2 - x_1)$, $(x_3 - x_1)$, and $(x_4 - x_1)$, respectively, then the system transforms to

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & (x_3 - x_2) & 0 \\ 0 & 1 & (x_4 - x_2) & (x_4 - x_2)(x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{31} \\ y_{41} \end{bmatrix},$$

where y_{21}, y_{31}, and y_{41} are defined by

$$y_{21} = \frac{y_2 - y_1}{x_2 - x_1} \qquad y_{31} = \frac{y_3 - y_1}{x_3 - x_1} \qquad y_{41} = \frac{y_4 - y_1}{x_4 - x_1}.$$

Notice that

$$\begin{bmatrix} y_{21} \\ y_{31} \\ y_{41} \end{bmatrix} = \left(\begin{bmatrix} y_2 \\ y_3 \\ y_4 \end{bmatrix} - \begin{bmatrix} y_1 \\ y_1 \\ y_1 \end{bmatrix} \right) ./ \left(\begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix} - \begin{bmatrix} x_1 \\ x_1 \\ x_1 \end{bmatrix} \right) = (y(2{:}4) - y(1))./(x(2{:}4) - x(1)).$$

The key point is that we have reduced the size of problem by one. The remaining unknowns satisfy a 3-by-3 system:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & (x_3 - x_2) & 0 \\ 1 & (x_4 - x_2) & (x_4 - x_2)(x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_{21} \\ y_{31} \\ y_{41} \end{bmatrix}.$$

This is exactly the system obtained were we to seek the coefficients of the quadratic

$$q(x) = c_2 + c_3(x - x_2) + c_4(x - x_2)(x - x_3)$$

that interpolates the data (x_2, y_{21}), (x_3, y_{31}), and (x_4, y_{41}).

2.2.2 The General n Case

For general n, we see that if $c_1 = y_1$ and

$$q(x) = c_2 + c_3(x - x_2) + \cdots + c_n(x - x_2) \cdots (x - x_{n-1})$$

interpolates the data

$$\left(x_i, \frac{y_i - y_1}{x_i - x_1}\right) \qquad i = 2{:}n,$$

then

$$p(x) = c_1 + (x - x_1)q(x)$$

interpolates $(x_1, y_1), \ldots, (x_n, y_n)$. This is easy to verify. Indeed, for $j = 1{:}n$

$$p(x_j) = c_1 + (x_j - x_1)q(x_j) = y_1 + (x_j - x_1)\frac{y_j - y_1}{x_j - x_1} = y_j.$$

This sets the stage for a recursive formulation of the whole process:

```
  function c = InterpNRecur(x,y)
% c = InterpNRecur(x,y)
% The Newton polynomial interpolant.
% x is a column n-vector with distinct components and y is
% a column n-vector. c is  a column n-vector with the property that if
%
%      p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
% then
%      p(x(i)) = y(i), i=1:n.

n = length(x); c = zeros(n,1); c(1) = y(1);
if n > 1
   c(2:n) = InterpNRecur(x(2:n),(y(2:n)-y(1))./(x(2:n)-x(1)));
end
```

If $n = 1$, then the constant interpolant $p(x) \equiv y_1$ is returned (i.e., $c_1 = y_1$.) Otherwise, the final c-vector is a "stacking" of y_1 and the solution to the reduced problem. The recursive call obtains the coefficients of the interpolant $q(x)$ mentioned earlier.

To develop a nonrecursive implementation, we return to our four-point example and the equation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & (x_3 - x_2) & 0 \\ 0 & 1 & (x_4 - x_2) & (x_4 - x_2)(x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{31} \\ y_{41} \end{bmatrix}.$$

From this we see that $c_2 = y_{21}$. Now subtract equation 2 from equation 3 and divide by $(x_3 - x_2)$. Next, subtract equation 2 from equation 4 and divide by $(x_4 - x_2)$. With these operations we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & (x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{321} \\ y_{421} \end{bmatrix},$$

where

$$y_{321} = \frac{y_{31} - y_{21}}{x_3 - x_2} \qquad y_{421} = \frac{y_{41} - y_{21}}{x_4 - x_2}.$$

At this point we see that $c_3 = y_{321}$. Finally, by subtracting the third equation from the fourth equation and dividing by $(x_4 - x_3)$, we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{321} \\ y_{4321} \end{bmatrix},$$

where

$$y_{4321} = \frac{y_{421} - y_{321}}{x_4 - x_3}.$$

Clearly, $c_4 = y_{4321}$. The pattern for the general n case should be apparent:

```
for k=1:n-1
    $c_k = y_k$
    for $j = k+1{:}n$
        Subtract equation $k$ from equation $j$ and divide the result by $(x_j - x_k)$.
    end
end
$c_n = y_n$
```

However, when updating the equations *we need only keep track of the changes in the y-vector.* For example,

$$\begin{aligned} y(k+1{:}n) \quad &\leftarrow \quad \left(\begin{bmatrix} y_{k+1} \\ \vdots \\ y_n \end{bmatrix} - \begin{bmatrix} y_k \\ \vdots \\ y_k \end{bmatrix} \right) ./ \left(\begin{bmatrix} x_{k+1} \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} x_k \\ \vdots \\ x_k \end{bmatrix} \right) \\ &= \quad (y(k+1{:}n) - y(k)) \;./\; (x(k+1) - x(k)). \end{aligned}$$

This leads to

```
  function c = InterpN(x,y)
% c = InterpN(x,y)
% The Newton polynomial interpolant.
% x is a column n-vector with distinct components and y is
% a column n-vector. c is a column n-vector with the property that if
%
%     p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
% then
%     p(x(i)) = y(i), i=1:n.
n = length(x);
for k = 1:n-1
   y(k+1:n) = (y(k+1:n)-y(k)) ./ (x(k+1:n) - x(k));
end
c = y;
```

2.2.3 Nested Multiplication

As with the Vandermonde representation, the Newton representation permits an efficient nested multiplication scheme. For example, to evaluate $p_3(x)$ at $x = z$, we have the nesting

$$p_3(x) = ((c_4(x - x_3) + c_3)(x - x_2) + c_2)(x - x_1) + c_1.$$

The fragment

```
pVal = c(4);
pVal = (z-x(3))*pVal + c(3);
pVal = (z-x(2))*pVal + c(2);
pVal = (z-x(1))*pVal + c(1);
```

assigns the value of $p_3(z)$ to `pVal`. If `z` is a vector, then this becomes

```
pVal = c(4)*ones(size(z));
pVal = (z-x(3)).*pVal + c(3);
pVal = (z-x(2)).*pVal + c(2);
pVal = (z-x(1)).*pVal + c(1);
```

In general, we have

```
  function pVal = HornerN(c,x,z)
% pVal = HornerN(c,x,z)
% Evaluates the Newton interpolant on z where c and x are n-vectors, z is an
% m-vector, and pVal is a vector the same size as z with the property that if
%
%        p(x) = c(1) +  c(2)(x-x(1))+ ... + c(n)(x-x(1))...(x-x(n-1))
% then
%        pVal(i) = p(z(i)) , i=1:m.

n = length(c);
pVal = c(n)*ones(size(z));
for k=n-1:-1:1
   pVal = (z-x(k)).*pVal + c(k);
end
```

The script `ShowN` illustrates `HornerN` and `InterpN`.

Problems

P2.2.1 Write a MATLAB function `a = N2V(c,x)`, where `c` is a column n-vector, `x` is a column $(n-1)$-vector and `a` is a column n-vector, so that if

$$p(x) = c_1 + c_2(x-x_1) + \cdots + c_n(x-x_1)(x-x_2)\cdots(x-x_{n-1}),$$

then

$$p(x) = a_1 + a_2x + \cdots + a_nx^{n-1}.$$

In other words, `N2V` converts from the Newton representation to the Vandermonde representation.

P2.2.2 Suppose we are given the data (x_i, y_i), $i = 1{:}n$. Assume that the x_i are distinct and that $n \geq 2$. Let $p_L(x)$ and $p_R(x)$ be degree $n-2$ polynomials that satisfy

$$\begin{aligned} p_L(x_i) &= y_i \qquad i = 1{:}n-1 \\ p_R(x_i) &= y_i \qquad i = 2{:}n. \end{aligned}$$

Note that if

$$p(x) = \frac{(x-x_n)p_L(x) - (x-x_1)p_R(x)}{x_1 - x_n},$$

then $p(x_i) = y_i$, $i = 1{:}n$. In other words, $p(x)$ is the unique degree $n-1$ interpolant of (x_i, y_i), $i = 1{:}n$. Using this result, complete the following function:

```
  function pVal = RecurEval(x,y,z);
%
% x is column n-vector with distinct entries, y is a column n-vector, and z is
% a column m-vector.
%
% pVal is a column m-vector with the property that pVal(i) = p(z(i))
% where p(x) is the degree n-1 polynomial interpolant of (x(i),y(i)), i=1:n.
```

The implementation should be recursive and vectorized. No loops are necessary! Use `RecurEval` to produce an interpolant of $\sin(2\pi x)$ at $x = 0{:}.25{:}1$.

P2.2.3 Write a MATLAB script that solicits the name of a built-in function (as a string), the left and right limits of an interval $[L, R]$, and a positive integer n and then displays both the function and the $n-1$ degree interpolant of it at `linspace(L,R,n)`.

P2.2.4 Assume that `n`, `z(1:n)`, `L`, `R`, and `a(1:6)` are available. Write an efficient MATLAB script that assigns to `q(i)` the value of the polynomial

$$q(x) = a_1 + a_2(x-L) + a_3(x-L)^2 + a_4(x-L)^3 + a_5(x-L)^3(x-R) + a_6(x-L)^3(x-R)^2$$

at $x = z_i$, $i = 1{:}n$. It doesn't matter whether `q(1:n)` is a row vector or a column vector.

P2.2.5 Write a MATLAB script that plots a closed curve

$$(p_x(t), p_y(t)) \qquad 0 \leq t \leq 1$$

that passes through the points (0,0), (0,3), (4,0). The functions p_x and p_y should be cubic polynomials. Make effective use of `InterpN` and `HornerN`. The plot should be based on one hundred evaluations of p_x and p_y.

2.3 Properties

With two approaches to the polynomial interpolation problem, we have an occasion to assess their relative merits. Speed and accuracy are the main concerns.

2.3.1 Efficiency

One way to talk about the efficiency of a numerical method such as `InterpV` or `InterpN` is to relate the number of required flops to the "length" of the input. For `InterpV`, the amount of required arithmetic grows as the cube of n, the number of interpolation points. We say that `InterpV` is an $O(n^3)$ method meaning that work goes up by a factor of 8 if n is doubled. (An n-by-n linear equation solve requires about $2n^3/3$ flops.) On the other hand, `InterpN` is an $O(n^2)$ method. If we double n then work increases by an approximate factor of 4.

Generally speaking quadratic methods (like `InterpN`) are to be preferred to cubic methods (like `InterpV`) especially for large values of n. However, the "big-oh" predictions are typically not realized in practice for small values of n. Moreover, counting flops does not take into account overheads associated with function calls and memory access. Benchmarking these two methods using `tic` and `toc` would reveal that they are not terribly different for modest n.

So far we have just discussed execution efficiency. Memory efficiency is also important. `InterpV` requires an n-by-n array, while `InterpN` needs just a few n-vectors. In this case we say that `InterpV` is quadratic in memory while `InterpN` is linear in memory.

2.3.2 Accuracy

We know that the polynomial interpolant exists and is unique, but how well does it approximate? The answer to the question depends on the derivatives of the function that is being interpolated.

Theorem 2 *Suppose $p_{n-1}(x)$ interpolates the function $f(x)$ at the distinct points $x_1, \ldots, x_n$. If f is n times continuously differentiable on an interval I containing the x_i, then for any $x \in I$*

$$f(x) = p_{n-1}(x) \ + \ \frac{f^{(n)}(\eta)}{n!}(x - x_1)\cdots(x - x_n)$$

where $a \le \eta \le b$.

Proof For clarity and with not a tremendous loss of generality, we prove the theorem for the $n = 4$ case. Consider the function

$$F(t) = f(t) - p_3(t) - cL(t),$$

where

$$c = \frac{f(x) - p_3(x)}{(x - x_1)(x - x_2)(x - x_3)(x - x_4)}$$

and $L(t) = (t - x_1)(t - x_2)(t - x_3)(t - x_4)$. Note that $F(x) = 0$ and $F(x_i) = 0$ for $i = 1{:}4$. Thus, F has at least five zeros in I. In between these zeros F' has a zero and so F' has at least four zeros in I. Continuing in this way, we conclude that

$$F^{(4)}(t) = f^{(4)}(t) - p_3^{(4)}(t) - cL^{(4)}(t)$$

has at least one zero in I which we designate by η_x. Since p_3 has degree ≤ 3, $p_3^{(4)}(t) \equiv 0$. Since L is a monic polynomial with degree 4, $L_3^{(4)}(t) = 4!$. Thus,

$$0 = F^{(4)}(\eta_x) = f^{(4)}(\eta_x) - p_3^{(4)}(\eta_x) - cL^{(4)}(\eta_x) = f^{(4)}(\eta_x) - c \cdot 4!. \ \square$$

This result shows that the quality of $p_{n-1}(x)$ depends on the size of the nth derivative. If we have a bound on this derivative, then we can compute a bound on the error. To illustrate this point in a practical way, suppose $|f^{(n)}(x)| \le M_n$ for all $x \in [a, b]$. It follows that for any $z \in [a, b]$ we have

$$|f(z) - p_{n-1}(z)| \le \frac{M_n}{n!} \max_{a \le x \le b} |(x - x_1)(x - x_2)\cdots(x - x_n)|.$$

If we base the interpolant on the equally spaced points

$$x_i = a + \left(\frac{b-a}{n-1}\right)(i-1), \qquad i = 1{:}n$$

then, by a simple change of variable,

$$|f(z) - p_{n-1}(z)| \le M_n \left(\frac{b-a}{n-1}\right)^n \max_{0\le s\le n-1} \left|\frac{s(s-1)\cdots(s-n+1)}{n!}\right|.$$

It can be shown that the max is no bigger than $1/(4n)$, from which we conclude that

$$|f(z) - p_{n-1}(z)| \le \frac{M_n}{4n}\left(\frac{b-a}{n-1}\right)^n. \tag{2.2}$$

Thus, if a function has ill-behaved higher derivatives, then the quality of the polynomial interpolants may actually decrease as the degree increases.

A classic example of this is the problem of interpolating the function $f(x) = 1/(1+25x^2)$ across the interval $[-1,1]$. See Figure 2.3. While the interpolant "captures" the trend of the function in the middle part of the interval, it blows up near the endpoints. The script `RungeEg` explores the phenomenon in greater detail.

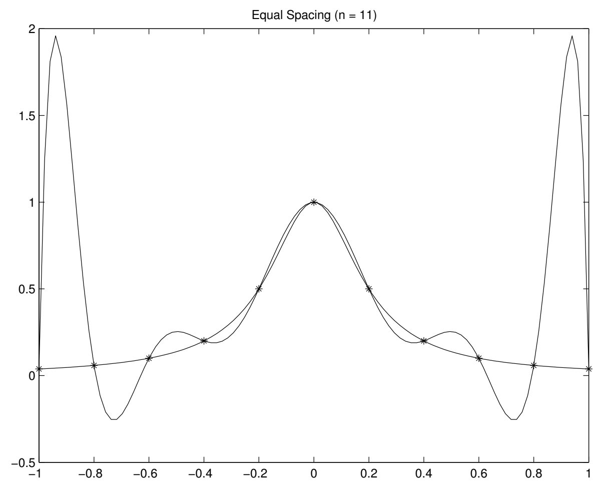


FIGURE 2.3 *The Runge phenomenon*

Problems

P2.3.1 Write a MATLAB script that compares `HornerN` and `HornerV` from the flop point of view.

P2.3.2 Write a MATLAB script that repeatedly solicits an integer n and produces a reasonable plot of the function $e(s) = |s(s-1)\cdots(s-n+1)/n!|$ on the interval $[0, n-1]$. Verify experimentally that this function is never bigger than 1, a fact that we used to establish (2.2).

P2.3.3 Write a MATLAB function `nBest(L,R,a,delta)` that returns an integer n such that if $p_{n-1}(x)$ interpolates e^{ax} at $L + (i-1)(R-L)/(n-1)$, $i = 1{:}n$, then $|p_{n-1}(z) - e^{az}| \le \delta$ for all $z \in [L, R]$. Try to make the value of n as small as you can.

2.4 Special Topics

As a follow-up to the preceding developments, we briefly discuss properties and algorithms associated with divided differences, inverse interpolation, and two-dimensional linear interpolation. We also introduce the important idea of trigonometric interpolation.

2.4.1 Divided Differences

Returning to the $n = 4$ example used in the previous section, we can express c_1, c_2, c_3, and c_4 in terms of the x_i and f:

$$\begin{aligned}
c_1 &= f(x_1) \\
c_2 &= \frac{f(x_2) - f(x_1)}{x_2 - x_1} \\
c_3 &= \frac{\dfrac{f(x_3) - f(x_1)}{x_3 - x_1} - \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_2} \\
c_4 &= \frac{\dfrac{\dfrac{f(x_4) - f(x_1)}{x_4 - x_1} - \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_4 - x_2} - \dfrac{\dfrac{f(x_3) - f(x_1)}{x_3 - x_1} - \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_2}}{x_4 - x_3}.
\end{aligned}$$

The coefficients are called *divided differences.* To stress the dependence of c_k on f and $x_1, \ldots, x_k$, we write

$$c_k = f[x_1, \ldots, x_k]$$

and refer to this quantity as the *$k - 1$st order divided difference.* Thus,

$$p_{n-1}(x) = \sum_{k=1}^{n} f[x_1, \ldots, x_k] \left(\prod_{j=1}^{k-1} (x - x_j) \right)$$

is the n-point polynomial interpolant of f at $x_1, \ldots, x_n$.

We now establish another recursive property that relates the divided differences of f on designated subsets of $\{x_1, \ldots, x_n\}$. Suppose $p_L(x)$ and $p_R(x)$ are the interpolants of f on $\{x_1, \ldots, x_{k-1}\}$ and $\{x_2, \ldots, x_k\}$, respectively. It is easy to confirm that if

$$p(x) = \frac{(x - x_k)p_L(x) - (x - x_1)p_R(x)}{x_1 - x_k}, \tag{2.3}$$

then $p(x_i) = f(x_i)$, $i = 1{:}k$. Thus $p(x)$ is the interpolant of f on $\{x_1, \ldots, x_k\}$ and so

$$p(x) = f[x_1] + f[x_1, x_2](x - x_1) + \cdots + f[x_1, \ldots, x_n](x - x_1) \cdots (x - x_{k-1}). \tag{2.4}$$

Note that since

$$p_L(x) = f[x_1] + f[x_1, x_2](x - x_1) + \cdots + f[x_1, \ldots, x_{k-1}](x - x_1) \cdots (x - x_{k-2}),$$

the coefficient of x^{k-2} is given by $f[x_1, \ldots, x_{k-1}]$. Likewise, since

$$p_R(x) = f[x_2] + f[x_2, x_3](x - x_2) + \cdots + f[x_2, \ldots, x_k](x - x_2) \cdots (x - x_{k-1}),$$

the coefficient of x^{k-2} is given by $f[x_2, \ldots, x_k]$. Comparing the coefficients of x^{k-1} in (2.3) and (2.4), we conclude that

$$f[x_1, \ldots, x_k] = \frac{f[x_2, \ldots, x_k] - f[x_1, \ldots, x_{k-1}]}{x_k - x_1}. \tag{2.5}$$

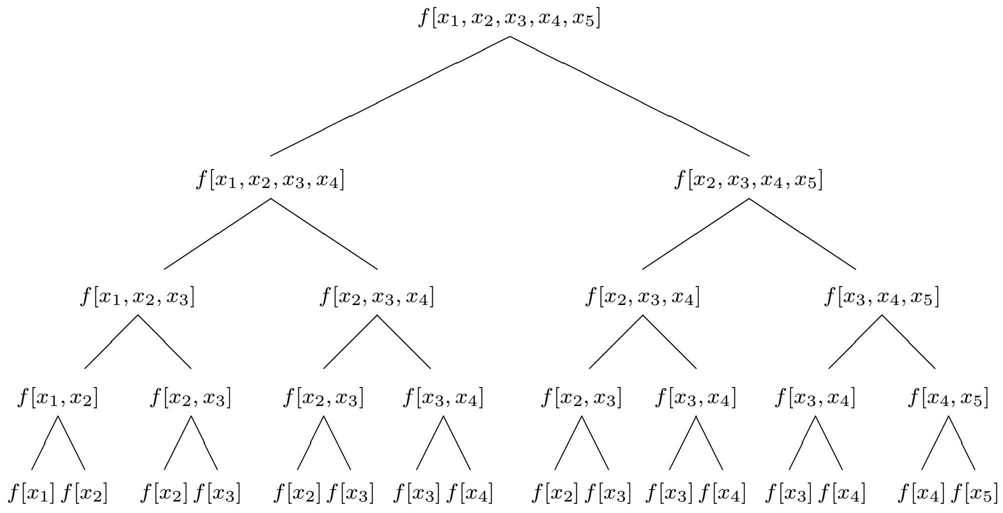


FIGURE 2.4 *Divided differences*

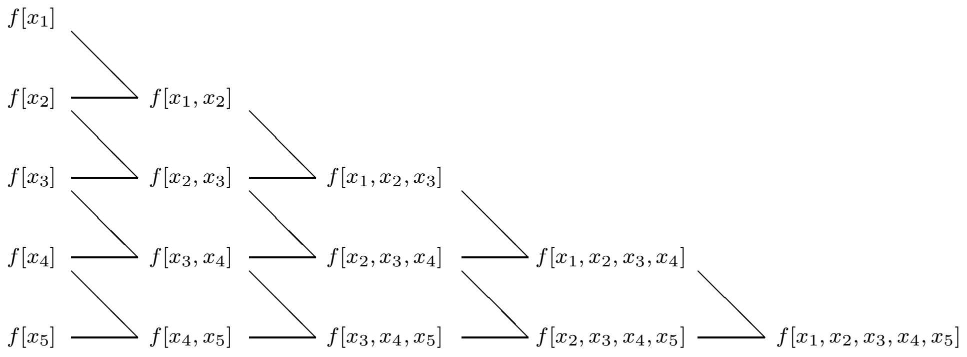


FIGURE 2.5 *Efficient computation of divided differences*

The development of higher-order divided differences from lower order divided differences is illustrated in Figure 2.4. Observe that the sought-after divided differences are along the left edge of the tree. Pruning the excess, we see that the required divided differences can be built up as shown in Figure 2.5. This enables us to rewrite `InterpN` as follows:

```
  function c = InterpN2(x,y)
% c = InterpN2(x,y)
% The Newton polynomial interpolant.
% x is a column n-vector with distinct components and y is
% a column n-vector. c is a column n-vector with the property that if
%      p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
% then
%      p(x(i)) = y(i), i=1:n.
n = length(x);
for k = 1:n-1
   y(k+1:n) = (y(k+1:n)-y(k:n-1)) ./ (x(k+1:n) - x(1:n-k));
end
c = y;
```

A number of simplifications result if the x_i are equally spaced. Assume that

$$x_i = x_1 + (i-1)h,$$

where $h > 0$ is the spacing. From (2.5) we see that

$$f[x_1,\ldots,x_k] = \frac{f[x_2,\ldots,x_k] - f[x_1,\ldots,x_{k-1}]}{h(k-1)}.$$

This makes divided difference a scaling of the *differences* $\Delta f[x_1,\ldots,x_k]$, which we define by

$$\Delta f[x_1,\ldots,x_k] \;=\; \begin{cases} f(x_1) & \text{if } k=1 \\ \\ \Delta f[x_2,\ldots,x_k] - \Delta f[x_1,\ldots,x_{k-1}] & \text{if } k>1 \end{cases}.$$

For example,

0th Order	1st Order	2nd Order	3rd Order	4th Order
f_1				
f_2	$f_2 - f_1$			
f_3	$f_3 - f_2$	$f_3 - 2f_2 + f_1$		
f_4	$f_4 - f_3$	$f_4 - 2f_3 + f_2$	$f_4 - 3f_3 + 3f_2 - f_1$	
f_5	$f_5 - f_4$	$f_5 - 2f_4 + f_3$	$f_5 - 3f_4 + 3f_3 - f_2$	$f_5 - 4f_4 + 6f_3 - 4f_2 + f_1$

It is not hard to show that

$$f[x_1,\ldots,x_k] = \frac{\Delta f[x_1,\ldots,x_k]}{h^{k-1}(k-1)!}.$$

The built-in function `diff` can be used to compute differences. In particular, if `y` is an n-vector, then

```
d = diff(y)
```

and

```
d = y(2:n) - y(1:n-1)
```

are equivalent. A second argument can be used to compute higher-order differences. For example,

```
d = diff(y,2)
```

computes the second-order differences:

```
d = y(3:n) - 2*y(2:n-1) + y(1:n-2)
```

Problems

P2.4.1 Compare the computed c_i produced by `InterpN` and `InterpN2`.

P2.4.2 Complete the following MATLAB function:

```
function [c,x,y] = InterpNEqual(fname,L,R,n)
```

Make effective use of the `diff` function.

2.4.2 Inverse Interpolation

Suppose the function $f(x)$ has an inverse on $[a,b]$. This means that there is a function g so that $g(f(x)) = x$ for all $x \in [a,b]$. Thus $g(x) = \sqrt{x}$ is the inverse of $f(x) = x^2$ on $[0,1]$. If

$$a = x_1 < x_2 < \cdots < x_n = b$$

and $y_i = f(x_i)$, then the polynomial that interpolates the data (y_i, x_i), $i = 1{:}n$ is an interpolate of f's inverse. Thus the script

```
x = linspace(0,1,6)';
y = x.*x;
a  = InterpV(y,x);
yvals = linspace(y(1),y(6));
xvals = HornerV(a,yvals);
plot(yvals,xvals);
```

plots a quintic interpolant of the square root function. This is called *inverse* interpolation, and it has an important application in zero finding. Suppose $f(x)$ is continuous and either monotone increasing or decreasing on $[a,b]$. If $f(a)f(b) < 0$, then f has a zero in $[a,b]$. If $q(y)$ is an inverse interpolant, then $q(0)$ can be thought of as an approximation to this root.

Problems

P2.4.3 Suppose we have three data points (x_1, y_1), (x_2, y_2), and (x_3, y_3) with the property that $x_1 < x_2 < x_3$ and that y_1 and y_3 are opposite in sign. Write a function `root = InverseQ(x,y)` that returns the value of the inverse quadratic interpolant at 0.

2.4.3 Interpolation in Two Dimensions

Suppose $(\tilde{x}, \tilde{y})$ is inside the rectangle

$$R = \{(x,y) : a \le x \le b, \qquad c \le y \le d\}.$$

Suppose $f(x,y)$ is defined on R and that we have its values on the four corners

$$\begin{aligned} f_{ac} &= f(a,c) \\ f_{bc} &= f(b,c) \\ f_{ad} &= f(a,d) \\ f_{bd} &= f(b,d). \end{aligned}$$

Our goal is to use linear interpolation to obtain an estimate of $f(\tilde{x}, \tilde{y})$. Suppose $\lambda \in [0,1]$ with the property that $\tilde{x} = (1-\lambda)a + \lambda b$. It follows that

$$\begin{aligned} f_{xc} &= (1-\lambda)f_{ac} + \lambda f_{bc} \\ f_{xd} &= (1-\lambda)f_{ad} + \lambda f_{bd} \end{aligned}$$

are linearly interpolated estimates of $f(\tilde{x}, c)$ and $f(\tilde{x}, d)$, respectively. Consequently, if $\mu \in [0,1]$ with $\tilde{y} = (1-\mu)c + \mu d$, then a second interpolation between f_1 and f_2 gives an estimate of $f(\tilde{x}, \tilde{y})$:

$$z = (1-\mu)f_{xc} + \mu f_{xd} \approx f(\tilde{x}, \tilde{y}).$$

Putting it all together, we see that

$$\begin{aligned} z &= (1-\mu)((1-\lambda)f_{ac} + \lambda f_{bc}) + \mu((1-\lambda)f_{ad} + \lambda f_{bd}) \\ &\approx f((1-\lambda)a + \lambda b, (1-\mu)c + \mu d). \end{aligned}$$

Figure 2.6 depicts the interpolation points. To interpolate the values in a matrix of $f(x,y)$ evaluations it is necessary to "locate" the point at which the interpolation is required. The four relevant values from the array must then be combined as described above:

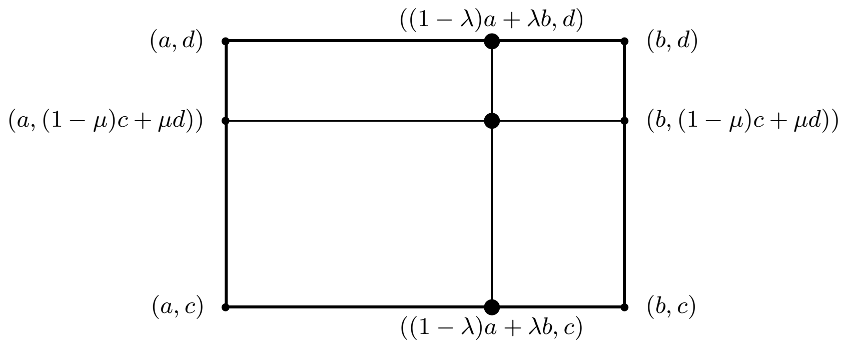


FIGURE 2.6 *Linear interpolation in two dimensions*

```
  function z = LinInterp2D(xc,yc,a,b,c,d,fA)
% z = LinInterp2D(xc,yc,a,b,n,c,d,m,fA)
% Linear interpolation on a grid of f(x,y) evaluations.
% xc, yc, a, b, c, and d  are scalars that satisfy a<=xc<=b and c<=yc<=d.
% fA is an n-by-m matrix with the property that
%
%     A(i,j) = f(a+(i-1)(b-a)/(n-1),c+(j-1)(d-c)/(m-1)) , i=1:n, j=1:m
%
% z is a linearly interpolated value of f(xc,yc).

[n,m] = size(fA);
% xc  = a+(i-1+dx)*hx    0<=dx<=1
hx = (b-a)/(n-1); i  = max([1 ceil((xc-a)/hx)]); dx = (xc - (a+(i-1)*hx))/hx;
% yc  = c+(j-1+dy)*hy    0<=dy<=1
hy = (d-c)/(m-1); j  = max([1 ceil((yc-c)/hy)]); dy = (yc - (c+(j-1)*hy))/hy;
z = (1-dy)*((1-dx)*fA(i,j)+dx*fA(i+1,j)) + dy*((1-dx)*fA(i,j+1)+dx*fA(i+1,j+1));
```

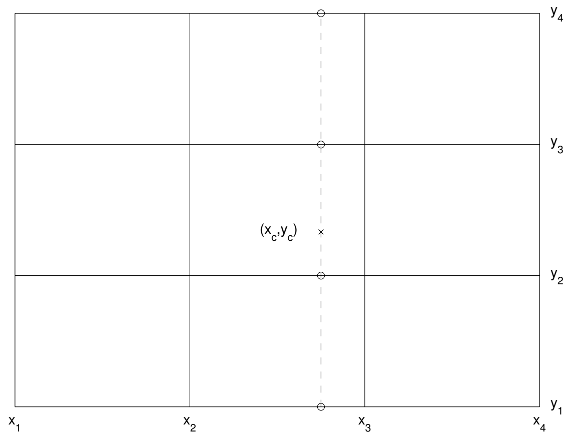
The following can be used for the table-generation across a uniform grid:

```
  function fA = SetUp(f,a,b,n,c,d,m)
% Sets up a matrix of f(x,y) evaluations.
% f is a handle to a function of the form f(x,y).
% a, b, c, and d  are scalars that satisfy a<=b and c<=d.
% n and m are integers >=2.
% fA is an n-by-m matrix with the property that
%
%     A(i,j) = f(a+(i-1)(b-a)/(n-1),c+(j-1)(d-c)/(m-1)) , i=1:n, j=1:m

x = linspace(a,b,n);
y = linspace(c,d,m);
fA = zeros(n,m);
for i=1:n
   for j=1:m
      fA(i,j) = f(x(i),y(j));
   end
end
```

Problems

P2.4.4 Analogous to `LinInterp2D`, write a function `CubicInterp2D(xc,yc,a,b,n,c,d,m,fA)` that does cubic interpolation from a matrix of $f(x,y)$ evaluations. Start by figuring out "where" (x_c, y_c) is in the grid with respect to `x = linspace(a,b,n)` and `y = linspace(c,d,m)`. Suppose this is the situation: as in `LinearInterp2D`.



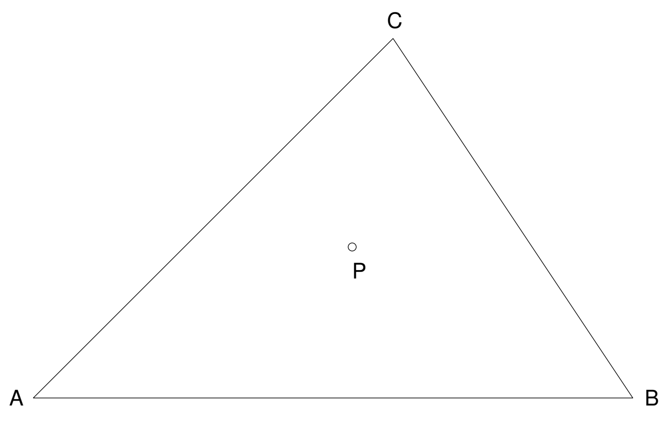
For $i = 1{:}4$, construct a cubic $p_i(x)$ that interpolates f at $(x_1, y_i), (x_2, y_i), (x_3, y_i)$ and (x_4, y_i). Then construct a cubic $q(y)$ that interpolates $(x_c, p_i(x_c))$, $i = 1{:}4$ and return the value of $q(y_c)$. The above picture/plan assumes that (x_c, y_c) is not in an "edge tile" so you'll have to work out something reasonable to do if that is the case.

P2.4.5 (a) Use `SetUp` to produce a matrix of function evaluations for

$$f(x,y) = \frac{1}{.2(x-3)^2 \; + \; .3 * (y-1)^2 + .2}.$$

Set $(a, b, n, c, d, m) = (0, 5, 300, 0, 3, 150)$. (b) Produce a plot that shows what f "looks like" along the line segment $\{(x, y) \,|\, x = 5 - 5t,\ y = 3t,\ 0 \le t \le 1\}$. Do this by interpolating f at a sufficiently large number of points along the line segment.

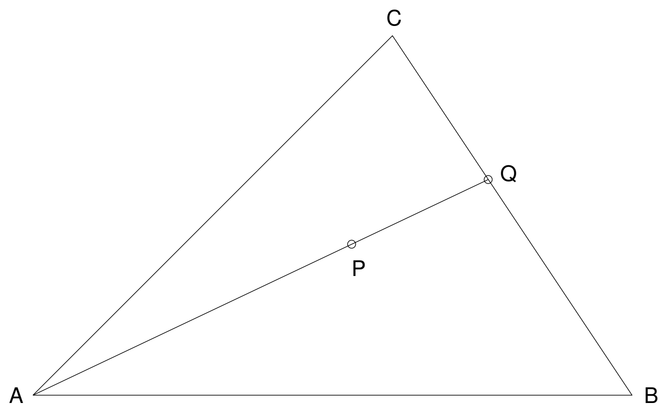
P2.4.6 This problem is about two-dimensional linear interpolation inside a triangle. Suppose that we know the value of a function $f(u, v)$ at the vertices of triangle ABC and that we wish to estimate its value at a point P inside the following triangle:



Consider the following method for doing this:

- Compute the intersection Q of line AP and line BC.
- Use linear interpolation to estimate f at Q from its value at B and C.
- Use linear interpolation to estimate f at P from its value at A and its estimate at Q.

Complete the following function so that it implements this method. Vectorization is not important.



```
  function fp = InterpTri(x,y,fvals,p)
% Suppose f(u,v) is a function of two variables defined everywhere in the plane.
% Assume that x, y, and fvals are column 3-vectors and p is a column 2-vector.
% Assume that (p(1),p(2)) is inside the triangle defined by (x(1),y(1)), (x(2),y(2)),
% and (x(3),y(3)) and that fvals(i) = f(x(i),y(i)) for i=1:3. fp is an estimate of
% f(p(1),p(2)) obtained by linear interpolation.
```

2.4.4 Trigonometric Interpolation

Suppose $f(t)$ is a periodic function with period T, $n = 2m$, and that we want to interpolate the data $(t_0, f_0), \ldots, (t_n, f_n)$ where $f_k = f(t_k)$ and

$$t_k = k\frac{T}{n}, \qquad k = 0{:}n.$$

Because the data is periodic it makes more sense to interpolate with a periodic function rather than with a polynomial. So let us pursue an interpolant that is a linear combination of cosines and sines rather than an interpolant that is a linear combination of 1, x, x^2, etc.

Assuming that j is an integer, the functions $\cos(2\pi jt/T)$ and $\sin(2\pi jt/T)$ have the same period as f prompting us to seek real scalars $a_0, \ldots, a_m$ and $b_0, \ldots, b_m$ so that if

$$F(t) = \sum_{j=0}^{m} \left[a_j \cos\left(\frac{2\pi j}{T}t\right) \; + \; b_j \sin\left(\frac{2\pi j}{T}t\right) \right],$$

then $F(t_k) = f_k$ for $k = 0{:}n$. This is a linear system that consists of $n+1$ equations in $2(m+1) = n+2$ unknowns. However, we note that b_0 and b_m are not involved in any equation since $\sin(2\pi jt/T) = 0$ if $t = t_0 = 0$ or $t = t_n = T$. Moreover, the $k = 0$ equation and the $k = n$ equation are identical because of periodicity. Thus, we really want to determine $a_0, \ldots, a_m$ and $b_1, \ldots, b_{m-1}$ so that if

$$F(t) = a_0 + \sum_{j=1}^{m-1} \left[a_j \cos\left(\frac{2\pi j}{T}t\right) \; + \; b_j \sin\left(\frac{2\pi j}{T}t\right) \right] \; + a_m \cos\left(\frac{2\pi m}{T}t\right),$$

then $F(t_k) = f(t_k) = f_k$ for $k = 0{:}n-1$. This is an n-by-n linear system in n unknowns:

$$f_k = a_0 + \sum_{j=1}^{m-1} \left(a_j \cos(kj\pi/m) \; + \; b_j \sin(kj\pi/m)\right) \; + (-1)^k a_m \qquad k = 0{:}n-1.$$

Here is what the system looks like for the case $n = 6$ with angles specified in degrees:

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{bmatrix} = \begin{bmatrix} \cos(0) & \cos(0) & \cos(0) & \cos(0) & \sin(0) & \sin(0) \\ \cos(0) & \cos(60) & \cos(120) & \cos(180) & \sin(60) & \sin(120) \\ \cos(0) & \cos(120) & \cos(240) & \cos(360) & \sin(120) & \sin(240) \\ \cos(0) & \cos(180) & \cos(360) & \cos(540) & \sin(180) & \sin(360) \\ \cos(0) & \cos(240) & \cos(480) & \cos(720) & \sin(240) & \sin(480) \\ \cos(0) & \cos(300) & \cos(600) & \cos(900) & \sin(300) & \sin(600) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1/2 & -1/2 & -1 & \sqrt{3}/2 & \sqrt{3}/2 \\ 1 & -1/2 & -1/2 & 1 & \sqrt{3}/2 & -\sqrt{3}/2 \\ 1 & -1 & 1 & -1 & 0 & 0 \\ 1 & -1/2 & -1/2 & 1 & -\sqrt{3}/2 & \sqrt{3}/2 \\ 1 & 1/2 & -1/2 & -1 & -\sqrt{3}/2 & -\sqrt{3}/2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \end{bmatrix}.$$

For general even n, `CSInterp` sets up the defining linear system and solves for a and b:

```
  function F = CSInterp(f)
% F = CSInterp(f)
% f is a column n vector and n = 2m.
% F.a is a column m+1 vector and F.b is a column m-1 vector so that if
% tau = (pi/m)*(0:n-1)', then
%        f = F.a(1)*cos(0*tau) +...+ F.a(m+1)*cos(m*tau) +
%            F.b(1)*sin(tau)   +...+ F.b(m-1)*sin((m-1)*tau)

  n = length(f); m = n/2;
  tau = (pi/m)*(0:n-1)';
  P = [];
  for j=0:m,   P = [P cos(j*tau)]; end
  for j=1:m-1, P = [P sin(j*tau)]; end
  y = P\f;
  F = struct('a',y(1:m+1),'b',y(m+2:n));
```

Note that the a and b vectors are returned in a structure. The matrix of coefficients can be shown to be nonsingular so the interpolation process that we have presented is well-defined. However, it involves $O(n^3)$ flops because of the linear system solve. In P2.4.7 we show how to reduce this to $O(n^2)$. The evaluation of the trigonmetric interpolant can be handled by

```
  function Fvals = CSeval(F,T,tvals)
% F.a is a length m+1 column vector, F.b is a length m-1 column vector,
% T is a positive scalar, and tvals is a column vector.
% If
%   F(t)  = F.a(1) + F.a(2)*cos((2*pi/T)*t) +...+ F.a(m+1)*cos((2*m*pi/T)*t) +
%                    F.b(1)*sin((2*pi/T)*t) +...+ F.b(m-1)*sin((2*m*pi/T)*t)
%
% then Fvals = F(tvals).
  Fvals = zeros(length(tvals),1);
  tau = (2*pi/T)*tvals;
  for j=0:length(F.a)-1, Fvals = Fvals + F.a(j+1)*cos(j*tau); end
  for j=1:length(F.b),   Fvals = Fvals + F.b(j)*sin(j*tau); end
```

We close this section by applying `CSinterp` and `CSeval` to a data fitting problem that confronted Gauss in connection with the asteroid Pallas. The problem is to interpolate the following ascension-declination data

α	0	30	60	90	120	150	180	210	240	270	300	330
d	408	89	-66	10	338	807	1238	1511	1583	1462	1183	804

with a function of the form

$$d(\alpha) = a_0 \; + \; \sum_{j=1}^{5} [a_j \cos(2\pi j\alpha/360) + b_j \sin(2\pi j\alpha/360)] \; + \; a_6 \cos(12\pi\alpha/360).$$

Here is a script that does this and plots the results shown in Figure 2.7:

```
% Script File: Pallas
% Plots the trigonometric interpolant of the Gauss Pallas data.
A = linspace(0,360,13)';
D = [ 408 89 -66 10 338 807 1238 1511 1583 1462 1183 804 408]';
Avals = linspace(0,360,200)';
F = CSInterp(D(1:12));
Fvals = CSeval(F,360,Avals);

plot(Avals,Fvals,A,D,'o')
axis([-10 370 -200 1700])
set(gca,'xTick',linspace(0,360,13))
xlabel('Ascension (Degrees)')
ylabel('Declination (minutes)')
```

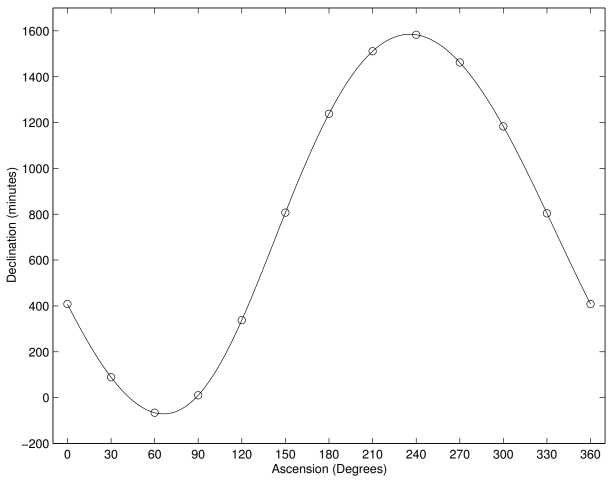


FIGURE 2.7 *Fitting the Pallas data*

Problems

P2.4.7 Observe that the matrix of coefficients P in `CSinterp` has the property that P^TP is diagonal. Use this fact to reduce the flop count in that function from $O(n^3)$ to $O(n^2)$. (With the *fast Fourier transform* it is possible to actually the flop count to an amazing $O(n \log n)$.)

M-Files and References

Script Files

`ShowV`	Illustrates `InterpV` and `HornerV`
`ShowN`	Illustrates `InterpN` and `HornerN`
`ShowRungePhenom`	Examines accuracy of interpolating polynomial.
`TableLookUp2D`	Illustrates `SetUp` and `LinInterp2D.`
`Pallas`	Fits periodic data with `CSInterp` and `CSEval`.

Function Files

`InterpV`	Construction of Vandermonde interpolating polynomial.
`HornerV`	Evaluates the Vandermonde interpolating polynomial.
`InterpNRecur`	Recursive construction of the Newton interpolating polynomial.
`InterpN`	Nonrecursive construction of the Newton interpolating polynomial.
`InterpN2`	Another nonrecursive construction of the Newton interpolating polynomial.
`HornerN`	Evaluates the Newton interpolating polynomial.
`SetUp`	Sets up matrix of f(x,y) evaluation.
`LinInterp2D`	2-Dimensional Linear Interpolation.
`Humps2D`	A sample function of two variables.
`CSInterp`	Fits periodic data with sines and cosines.
`CSEval`	Evaluates sums of sines and cosines.
`ShowMatPolyTools`	Illustrates `polyfit` and `polyval`.

References

W.L. Briggs and V.E. Henson (1995). *The DFT: An Owner's Manual for the Discrete Fourier Transform*, SIAM Publications, Philadelphia, PA.

S.D. Conte and C. de Boor (1980). *Elementary Numerical Analysis: An Algorithmic Approach, Third Edition*, McGraw-Hill, New York.

P. Davis (1963). *Interpolation and Approximation*, Blaisdell, New York.