Chapter 4

Curve Fitting

We consider two commonly used methods for curve fitting, interpolation and least squares. For interpolation, we use first polynomials and later splines. Polynomial interpolation may be implemented using a variety of bases. We consider power series, bases associated with the names of Newton and Lagrange, and finally Chebyshev series. We discuss the error arising directly from polynomial interpolation and the (linear algebra) error arising in the solution of the interpolation equations. This error analysis leads us to consider interpolating using piecewise polynomials, specifically splines. Finally, we introduce least squares curve fitting using simple polynomials and later generalize this approach sufficiently to permit other choices of least squares fitting functions; for example, splines, Chebyshev series, or trigonometric series. In a final section we show how to use the MATLAB functions to compute curve fits using many of the ideas introduced here and, indeed, we also consider for the first time fitting trigonometric series for periodic data; this approach uses the fast Fourier transform.

4.1 Polynomial Interpolation

We can approximate a function $f(x)$ by interpolating to the data $\{(x_i, f_i)\}_{i=0}^{N}$ by another (computable) function $p(x)$. (Here, implicitly we assume that the data is obtained by evaluating the function $f(x)$; that is, we assume that $f_i = f(x_i), i = 0, 1, \cdots, N$.)

Definition 4.1.1. *The function $p(x)$ **interpolates** to the data $\{(x_i, f_i)\}_{i=0}^{N}$ if the equations*

$$p(x_i) = f_i, \quad i = 0, 1, \cdots, N$$

are satisfied.

This system of $N + 1$ equations comprise the *interpolating conditions*. Note, the function $f(x)$ that has been evaluated to compute the data *automatically interpolates to its own data.*

Example 4.1.1. Consider the data: $\left(-\frac{\pi}{2}, -1\right)$, $(\pi, 0)$, $\left(\frac{5\pi}{2}, 1\right)$. This data was obtained by evaluating the function $f(x) = \sin(x)$ at $x_0 = -\frac{\pi}{2}$, $x_1 = \pi$ and $x_2 = \frac{5\pi}{2}$. Thus, $f(x) = \sin(x)$ interpolates to the given data. But the linear polynomial

$$p(x) = -\frac{2}{3} + \frac{2}{3\pi}x$$

also interpolates to the given data because $p(-\frac{\pi}{2}) = -1,\ p(\pi) = 0,$ and $p(\frac{5\pi}{2}) = 1$. Similarly the cubic polynomial

$$q(x) = \frac{16}{15\pi}x - \frac{8}{5\pi^2}x^2 + \frac{8}{15\pi^3}x^3$$

interpolates to the given data because $q(-\frac{\pi}{2}) = -1,\ q(\pi) = 0,$ and $q(\frac{5\pi}{2}) = 1$. The graphs of $f(x)$, $p(x)$ and $q(x)$, along with the interpolating data, are shown in Fig. 4.1. The interpolating conditions imply that these curves pass through the data points.

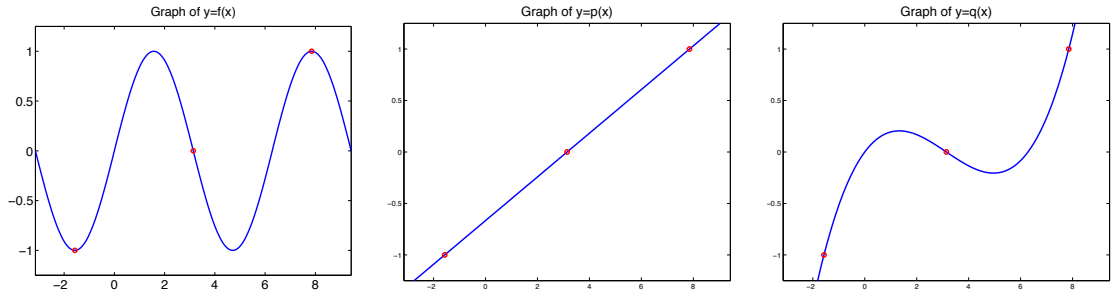

Figure 4.1: Graphs of $f(x) = \sin(x)$, $p(x) = -\frac{2}{3} + \frac{2}{3\pi}x$ and $q(x) = \frac{16}{15\pi}x - \frac{8}{5\pi^2}x^2 + \frac{8}{15\pi^3}x^3$. Each of these functions interpolates to the data $\left(-\frac{\pi}{2}, -1\right)$, $(\pi, 0)$, $\left(\frac{5\pi}{2}, 1\right)$. The data points are indicated by circles on each of the plots.

A common choice for the interpolating function $p(x)$ is a polynomial. Polynomials are chosen because there are efficient methods both for determining and for evaluating them, see, for example, Horner's rule described in Section 6.5. Indeed, they may be evaluated using only adds and multiplies, the most basic computer operations. A polynomial that interpolates to the data is **an interpolating polynomial**. A simple, familiar example of an interpolating polynomial is a straight line, that is a polynomial of degree one, joining two points.

Before we can construct an interpolating polynomial, we need to establish a standard form representation of a polynomial. In addition, we need to define what is meant by the "degree" of a polynomial.

Definition 4.1.2. *A polynomial $p_K(x)$ is of **degree** K if there are constants $c_0, c_1, \cdots, c_K$ for which*

$$p_K(x) = c_0 + c_1 x + \cdots + c_K x^K.$$

*The polynomial $p_K(x)$ is of **exact degree** K if it is of degree K and $c_K \neq 0$.*

Example 4.1.2. $p(x) \equiv 1 + x - 4x^3$ is a polynomial of exact degree 3. It is also a polynomial of degree 4 because we can write $p(x) = 1 + x + 0x^2 - 4x^3 + 0x^4$. Similarly, $p(x)$ is a polynomial of degree $5, 6, 7, \cdots$.

4.1.1 The Power Series Form of The Interpolating Polynomial

Consider first the problem of linear interpolation, that is straight line interpolation. If we have two data points (x_0, f_0) and (x_1, f_1) we can interpolate to this data using the linear polynomial $p_1(x) \equiv a_0 + a_1 x$ by satisfying the pair of linear equations

$$\begin{array}{rcccl} p_1(x_0) & \equiv & a_0 + a_1 x_0 & = & f_0 \\ p_1(x_1) & \equiv & a_0 + a_1 x_1 & = & f_1 \end{array}$$

Solving these equations for the coefficients a_0 and a_1 gives the straight line passing through the two points (x_0, f_0) and (x_1, f_1). These equations have a solution if $x_0 \neq x_1$. If $f_0 = f_1$, the solution is a constant ($a_0 = f_0$ and $a_1 = 0$); that is, it is a polynomial of degree zero,.

Generally, there are an infinite number of polynomials that interpolate to a given set of data. To explain the possibilities we consider the power series form of the complete polynomial (that is, a polynomial where all the powers of x appear)

$$p_M(x) = a_0 + a_1 x + \cdots + a_M x^M$$

of degree M. If the polynomial $p_M(x)$ interpolates to the given data $\{(x_i, f_i)\}_{i=0}^N$, then the interpolating conditions form a linear system of $N+1$ equations

$$\begin{array}{rcccl} p_M(x_0) & \equiv & a_0 + a_1 x_0 + \cdots + a_M x_0^M & = & f_0 \\ p_M(x_1) & \equiv & a_0 + a_1 x_1 + \cdots + a_M x_1^M & = & f_1 \\ & & \vdots & & \\ p_M(x_N) & \equiv & a_0 + a_1 x_N + \cdots + a_M x_N^M & = & f_N \end{array}$$

for the $M+1$ unknown coefficients $a_0, a_1, \ldots, a_M$.

From linear algebra (see Chapter 3), this linear system of equations has a *unique solution* for *every choice* of data values $\{f_i\}_{i=0}^N$ if and only if the system is square (that is, if $M = N$) and it is nonsingular. If $M < N$, there exist choices of the data values $\{f_i\}_{i=0}^N$ for which this linear system has no solution, while for $M > N$ if a solution exists it cannot be unique.

Think of it this way:

1. The complete polynomial $p_M(x)$ of degree M has $M+1$ unknown coefficients,
2. The interpolating conditions comprise $N+1$ equations.
3. So,
   - If $M = N$, there are as many coefficients in the complete polynomial $p_M(x)$ as there are equations obtained from the interpolating conditions,
   - If $M < N$, the number of coefficients is less than the number of data values and we wouldn't have enough coefficients so that we would be able to fit to all the data.
   - If $M > N$, the number of coefficients exceeds the number of data values and we would expect to be able to choose the coefficients in many ways to fit the data.

The square, $M = N$, coefficient matrix of the linear system of interpolating conditions is the Vandermonde matrix

$$V_N \equiv \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^M \\ 1 & x_1 & x_1^2 & \cdots & x_1^M \\ & & \vdots & & \\ 1 & x_N & x_N^2 & \cdots & x_N^M \end{bmatrix}$$

and it can be shown that the determinant of V_N is

$$\det(V_N) = \prod_{i>j} (x_i - x_j)$$

Recall from Problems 3.1.4, 3.2.21, and 3.3.4 that the determinant is a test for singularity, and this is an ideal situation in which to use it. In particular, observe that $\det(V_N) \neq 0$ (that is the matrix V_N is nonsingular) if and only if the nodes $\{x_i\}_{i=0}^N$ are distinct; that is, if and only if $x_i \neq x_j$ whenever $i \neq j$. So, for every choice of data $\{(x_i, f_i)\}_{i=0}^N$, there *exists* a *unique* solution satisfying the interpolation conditions if and only if $M = N$ and the nodes $\{x_i\}_{i=0}^N$ are distinct.

Theorem 4.1.1. (Polynomial Interpolation Uniqueness Theorem) When the nodes $\{x_i\}_{i=0}^N$ are distinct there is a unique polynomial, the *interpolating polynomial* $p_N(x)$, of degree N that interpolates to the data $\{(x_i, f_i)\}_{i=0}^N$.

Though the degree of the interpolating polynomial is N corresponding to the number of data values, its exact degree may be less than N. For example, this happens when the three data points (x_0, f_0), (x_1, f_1) and (x_2, f_2) are collinear; the interpolating polynomial for this data is of degree $N = 2$ but it is of exact degree 1; that is, the coefficient of x^2 turns out to be zero. (In this case, the interpolating polynomial is the straight line on which the data lies.)

Example 4.1.3. Determine the power series form of the quadratic interpolating polynomial $p_2(x) = a_0 + a_1 x + a_2 x^2$ to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$. The interpolating conditions, in the order of the data, are

$$\begin{array}{rcl} a_0 + (-1)a_1 + (1)a_2 & = & 0 \\ a_0 & = & 1 \\ a_0 + (1)a_1 + (1)a_2 & = & 3 \end{array}$$

We may solve this linear system of equations for $a_0 = 1$, $a_1 = \frac{3}{2}$ and $a_2 = \frac{1}{2}$. So, in power series form, the interpolating polynomial is $p_2(x) = 1 + \frac{3}{2}x + \frac{1}{2}x^2$.

Of course, polynomials may be written in many different ways, some more appropriate to a given task than others. For example, when interpolating to the data $\{(x_i, f_i)\}_{i=0}^N$, the Vandermonde system determines the values of the coefficients $a_0, a_1, \cdots, a_N$ in the **power series form**. The number of floating–point computations needed by GEPP to solve the interpolating condition equations grows like N^3 with the degree N of the polynomial (see Chapter 3). This cost can be significantly reduced by exploiting properties of the Vandermonde coefficient matrix. Another way to reduce the cost of determining the interpolating polynomial is to change the way the interpolating polynomial is represented. This is explored in the next two subsections.

Problem 4.1.1. *Show that, when $N = 2$,*

$$\det(V_2) = \det\left(\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix}\right) = \prod_{i>j}(x_i - x_j) = \overbrace{(x_1 - x_0)(x_2 - x_0)}^{j=0;\, i=1,2}\overbrace{(x_2 - x_1)}^{j=1;\, i=2}$$

Problem 4.1.2. *$\omega_{N+1}(x) \equiv (x - x_0)(x - x_1)\cdots(x - x_N)$ is a polynomial of exact degree $N+1$. If $p_N(x)$ interpolates the data $\{(x_i, f_i)\}_{i=0}^N$, verify that, for any choice of polynomial $q(x)$, the polynomial*

$$p(x) = p_N(x) + \omega_{N+1}(x)q(x)$$

interpolates the same data as $p_N(x)$. Hint: Verify that $p(x)$ satisfies the same interpolating conditions as does $p_N(x)$.

Problem 4.1.3. *Find the power series form of the interpolating polynomial to the data $(1, 2)$, $(3, 3)$ and $(5, 4)$. Check that your computed polynomial does interpolate the data. Hint: For these three data points you will need a complete polynomial that has three coefficients; that is, you will need to interpolate with a quadratic polynomial $p_2(x)$.*

Problem 4.1.4. *Let $p_N(x)$ be the unique interpolating polynomial of degree N for the data $\{(x_i, f_i)\}_{i=0}^N$. Estimate the cost of determining the coefficients of the power series form of $p_N(x)$, assuming that you can set up the coefficient matrix at no cost. Just estimate the cost of solving the linear system via Gaussian Elimination.*

4.1.2 The Newton Form of The Interpolating Polynomial

Consider the problem of determining the interpolating quadratic polynomial for the data (x_0, f_0), (x_1, f_1) and (x_2, f_2). Using the data written in this order, the **Newton form** of the quadratic interpolating polynomial is

$$p_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1)$$

To determine these coefficients b_i simply write down the interpolating conditions:

$$\begin{array}{rcll} p_2(x_0) & \equiv & b_0 & = & f_0 \\ p_2(x_1) & \equiv & b_0 + b_1(x_1 - x_0) & = & f_1 \\ p_2(x_2) & \equiv & b_0 + b_1(x_2 - x_0) + b_2(x_2 - x_0)(x_2 - x_1) & = & f_2 \end{array}$$

The coefficient matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & (x_1 - x_0) & 0 \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) \end{bmatrix}$$

for this linear system is lower triangular. When the nodes $\{x_i\}_{i=0}^{2}$ are distinct, the diagonal entries of this lower triangular matrix are nonzero. Consequently, the linear system has a unique solution that may be determined by forward substitution.

Example 4.1.4. Determine the Newton form of the quadratic interpolating polynomial $p_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1)$ to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$. Taking the data points in their order in the data we have $x_0 = -1$, $x_1 = 0$ and $x_2 = 1$. The interpolating conditions are

$$\begin{array}{lcl} b_0 & = & 0 \\ b_0 + (0 - (-1))b_1 & = & 1 \\ b_0 + (1 - (-1))b_1 + (1 - (-1))(1 - 0)b_2 & = & 3 \end{array}$$

and this lower triangular system may be solved to give $b_0 = 0$, $b_1 = 1$ and $b_2 = \frac{1}{2}$. So, the Newton form of the quadratic interpolating polynomial is $p_2(x) = 0 + 1(x + 1) + \frac{1}{2}(x + 1)(x - 0)$. After rearrangement we observe that this is the same polynomial $p_2(x) = 1 + \frac{3}{2}x + \frac{1}{2}x^2$ as the power series form in Example 4.1.3.

Note, the Polynomial Interpolation Uniqueness theorem tells us that the Newton form of the polynomial in Example 4.1.4 and the power series form in Example 4.1.3 must be the same polynomial. However, we do not need to convert between forms to show that two polynomials are the same. We can use the Polynomial Interpolation Uniqueness theorem as follows. If we have two polynomials of degree N and we want to check if they are different representations of the same polynomial we can evaluate each polynomial at any choice of $(N + 1)$ distinct points. If the values of the two polynomials are the same at all $(N+1)$ points, then the two polynomials interpolate each other and so must be the same polynomial.

Example 4.1.5. We show that the polynomials $p_2(x) = 1 + \frac{3}{2}x + \frac{1}{2}x^2$ and $q_2(x) = 0 + 1(x + 1) + \frac{1}{2}(x + 1)(x - 0)$ are the same. The Polynomial Interpolation Uniqueness theorem tells us that since these polynomials are both of degree two we should check their values at three distinct points. We find that $p_2(1) = 3 = q_2(1)$, $p_2(0) = 1 = q_2(0)$ and $p_2(-1) = 0 = q_2(-1)$, and so $p_2(x) \equiv q_2(x)$.

Example 4.1.6. For the data $(1, 2)$, $(3, 3)$ and $(5, 4)$, using the data points in the order given, the Newton form of the interpolating polynomial is

$$p_2(x) = b_0 + b_1(x - 1) + b_2(x - 1)(x - 3)$$

and the interpolating conditions are

$$\begin{array}{lcll} p_2(1) & \equiv & b_0 & = & 2 \\ p_2(3) & \equiv & b_0 + b_1(3 - 1) & = & 3 \\ p_2(5) & \equiv & b_0 + b_1(5 - 1) + b_2(5 - 1)(5 - 3) & = & 4 \end{array}$$

This lower triangular linear system may be solved by forward substitution giving

$$\begin{array}{lcll} b_0 & & & = & 2 \\ b_1 & = & \dfrac{3 - b_0}{(3 - 1)} & = & \dfrac{1}{2} \\ b_2 & = & \dfrac{4 - b_0 - (5 - 1)b_1}{(5 - 1)(5 - 3)} & = & 0 \end{array}$$

Consequently, the Newton form of the interpolating polynomial is

$$p_2(x) = 2 + \frac{1}{2}(x-1) + 0(x-1)(x-3)$$

Generally, this interpolating polynomial is of degree 2, but its exact degree may be less. Here, rearranging into power series form we have $p_2(x) = \frac{3}{2} + \frac{1}{2}x$ and the exact degree is 1; this happens because the data $(1,2)$, $(3,3)$ and $(5,4)$ are collinear.

The Newton form of the interpolating polynomial is easy to evaluate using *nested multiplication* (which is a generalization of Horner's rule, see Chapter 6). Start with the observation that

$$\begin{aligned} p_2(x) &= b_0 + b_1(x-x_0) + b_2(x-x_0)(x-x_1) \\ &= b_0 + (x-x_0)\{b_1 + b_2(x-x_1)\} \end{aligned}$$

To develop an algorithm to evaluate $p_2(x)$ at a value $x = z$ using nested multiplication, it is helpful to visualize the sequence of nested operations as:

$$\begin{aligned} t_2(z) &= b_2 \\ t_1(z) &= b_1 + (z-x_1)t_2(z) \\ p_2(z) = t_0(z) &= b_0 + (z-x_0)t_1(z) \end{aligned}$$

An extension to degree N of this nested multiplication scheme leads to the pseudocode given in Fig. 4.2. This pseudocode evaluates the Newton form of a polynomial of degree N at a point $x = z$ given the values of the nodes x_i and the coefficients b_i.

Evaluating the Newton Form

Input: nodes x_i
coefficients b_i
scalar z

Output: $p = p_N(z)$

$p := b_N$
for $i = N-1$ downto 0 do
  $p := b_i + (z - x_i) * p$
next i

Figure 4.2: Pseudocode to use nested multiplication to evaluate the Newton form of the interpolating polynomial, $p_N(x) = b_0 + b_1(x-x_0) + \cdots + b_{N-1}(x-x_0)\cdots(x-x_{N-1})$, at $x = z$.

How can the triangular system for the coefficients in the Newton form of the interpolating polynomial be systematically formed and solved? First, note that we may form a sequence of interpolating polynomials as in Table 4.1.

Polynomial	The Data
$p_0(x) = b_0$	(x_0, f_0)
$p_1(x) = b_0 + b_1(x-x_0)$	(x_0, f_0), (x_1, f_1)
$p_2(x) = b_0 + b_1(x-x_0) + b_2(x-x_0)(x-x_1)$	(x_0, f_0), (x_1, f_1), (x_2, f_2)

Table 4.1: Building the Newton form of the interpolating polynomial

So, the Newton equations that determine the coefficients b_0, b_1 and b_2 may be written

$$\begin{array}{lcl} b_0 & = & f_0 \\ p_0(x_1) + b_1(x_1 - x_0) & = & f_1 \\ p_1(x_2) + b_2(x_2 - x_0)(x_2 - x_1) & = & f_2 \end{array}$$

and we conclude that we may solve for the coefficients b_i efficiently as follows

$$\begin{aligned} b_0 &= f_0 \\ b_1 &= \frac{f_1 - p_0(x_1)}{(x_1 - x_0)} \\ b_2 &= \frac{f_2 - p_1(x_2)}{(x_2 - x_0)(x_2 - x_1)} \end{aligned}$$

This process clearly deals with any number of data points. So, if we have data $\{(x_i, f_i)\}_{i=0}^{N}$, we may compute the coefficients in the corresponding Newton polynomial

$$p_N(x) = b_0 + b_1(x - x_0) + \cdots + b_N(x - x_0)(x - x_1)\cdots(x - x_{N-1})$$

from evaluating $b_0 = f_0$ followed by

$$b_k = \frac{f_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})}, \quad k = 1, 2, \ldots, N$$

Indeed, we can easily incorporate additional data values. Suppose we have fitted to the data $\{(x_i, f_i)\}_{i=0}^{N}$ using the above formulas and we wish to add the data (x_{N+1}, f_{N+1}). Then we may simply use the formula

$$b_{N+1} = \frac{f_{N+1} - p_N(x_{N+1})}{(x_{N+1} - x_0)(x_{N+1} - x_1)\cdots(x_{N+1} - x_N)}$$

Fig. 4.3 shows pseudocode that can be used to implement this computational sequence for the Newton form of the interpolating polynomial of degree N given the data $\{(x_i, f_i)\}_{i=0}^{N}$.

Constructing the Newton Form

Input: data (x_k, f_k)
Output: coefficients b_k

```
b_0 := f_0
for k = 1 to N
    num := f_k − p_{k−1}(x_k)
    den := 1
    for j = 0 to k − 1
        den := den ∗ (x_k − x_j)
    next j
    b_k := num/den
next k
```

Figure 4.3: Pseudocode to compute the coefficients, b_i, of the Newton form of the interpolating polynomial, $p_N(x) = b_0 + b_1(x - x_0) + \cdots + b_{N-1}(x - x_0)\cdots(x - x_{N-1})$. This pseudocode requires evaluating $p_{i-1}(x_i)$, which can be done by implementing the pseudocode given in Fig. 4.2.

If the interpolating polynomial is to be evaluated at many points, generally it is best first to determine its Newton form and then to use the nested multiplication scheme to evaluate the interpolating polynomial at each desired point.

Problem 4.1.5. *Use the data in Example 4.1.4 in reverse order (that is, use $x_0 = 1$, $x_1 = 0$ and $x_2 = -1$) to build an alternative quadratic Newton interpolating polynomial. Is this the same polynomial that was derived in Example 4.1.4? Why or why not?*

Problem 4.1.6. *Let $p_N(x)$ be the interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^{N}$, and let $p_{N+1}(x)$ be the interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^{N+1}$. Show that*

$$p_{N+1}(x) = p_N(x) + b_{N+1}(x - x_0)(x - x_1)\cdots(x - x_N)$$

What is the value of b_{N+1}?

Problem 4.1.7. *In Example 4.1.4 we showed how to form the quadratic Newton polynomial $p_2(x) = 0 + 1(x + 1) + \frac{1}{2}(x + 1)(x - 0)$ that interpolates to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$. Starting from this quadratic Newton interpolating polynomial build the cubic Newton interpolating polynomial to the data $(-1, 0)$, $(0, 1)$, $(1, 3)$ and $(2, 4)$.*

Problem 4.1.8. *Let*

$$p_N(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots + b_N(x - x_0)(x - x_1)\cdots(x - x_{N-1})$$

be the Newton interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^{N}$. Write down the coefficient matrix for the linear system of equations for interpolating to the data with the polynomial $p_N(x)$.

Problem 4.1.9. *Let the nodes $\{x_i\}_{i=0}^{2}$ be given. A complete quadratic $p_2(x)$ can be written in power series form or Newton form:*

$$p_2(x) = \left\{ \begin{array}{ll} a_0 + a_1 x + a_2 x^2 & \textit{power series form} \\ b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) & \textit{Newton form} \end{array} \right\}$$

By expanding the Newton form into a power series in x, verify that the coefficients b_0, b_1 and b_2 are related to the coefficients a_0, a_1 and a_2 via the equations

$$\begin{bmatrix} 1 & -x_0 & x_0 x_1 \\ 0 & 1 & -(x_0 + x_1) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

Describe how you could use these equations to determine the a_i's given the values of the b_i's? Describe how you could use these equations to determine the b_i's given the values of the a_i's?

Problem 4.1.10. *Write a code segment to determine the value of the derivative $p_2'(x)$ at a point x of the Newton form of the quadratic interpolating polynomial $p_2(x)$. (Hint: $p_2(x)$ can be evaluated via nested multiplication just as a general polynomial can be evaluated by Horner's rule. Review how $p_2'(x)$ is computed via Horner's rule in Chapter 6.)*

Problem 4.1.11. *Determine the Newton form of the interpolating (cubic) polynomial to the data $(0, 1)$, $(-1, 0)$, $(1, 2)$ and $(2, 0)$. Determine the Newton form of the interpolating (cubic) polynomial to the same data written in reverse order. By converting both interpolating polynomials to power series form show that they are the same. Alternatively, show they are the same by evaluating the two polynomials at four distinct points of your choice. Finally, explain why the Polynomial Interpolation Uniqueness Theorem tells you directly that these two polynomials must be the same.*

Problem 4.1.12. *Determine the Newton form of the (quartic) interpolating polynomial to the data* $(0,1)$, $(-1,2)$, $(1,0)$, $(2,-1)$ *and* $(-2,3)$.

Problem 4.1.13. *Let* $p_N(x)$ *be the interpolating polynomial for the data* $\{(x_i,f_i)\}_{i=0}^N$. *Determine the number of adds (subtracts), multiplies, and divides required to determine the coefficients of the Newton form of* $p_N(x)$. *Hint: The coefficient* $b_0 = f_0$, *so it costs nothing to determine* b_0. *Now, recall that*

$$b_k = \frac{f_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})}$$

Problem 4.1.14. *Let* $p_N(x)$ *be the unique polynomial interpolating to the data* $\{(x_i,f_i)\}_{i=0}^N$. *Given its coefficients, determine the number of adds (or, equivalently, subtracts) and multiplies required to evaluate the Newton form of* $p_N(x)$ *at one value of* x *by nested multiplication. Hint: The nested multiplication scheme for evaluating the polynomial* $p_N(x)$ *has the form*

$$\begin{array}{lcl} t_N & = & b_N \\ t_{N-1} & = & b_{N-1} + (x - x_{N-1})t_N \\ t_{N-2} & = & b_{N-2} + (x - x_{N-2})t_{N-1} \\ & \vdots & \\ t_0 & = & b_0 + (x - x_0)t_1 \end{array}$$

4.1.3 The Lagrange Form of The Interpolating Polynomial

Consider the data $\{(x_i,f_i)\}_{i=0}^2$. The *Lagrange form* of the quadratic polynomial interpolating to this data may be written:

$$p_2(x) \equiv f_0 \cdot \ell_0(x) + f_1 \cdot \ell_1(x) + f_2 \cdot \ell_2(x)$$

We construct each *basis polynomial* $\ell_i(x)$ so that it is quadratic and so that it satisfies

$$\ell_i(x_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{4.1}$$

then clearly

$$\begin{array}{lcl} p_2(x_0) & = & 1 \cdot f_0 + 0 \cdot f_1 + 0 \cdot f_2 \quad = \quad f_0 \\ p_2(x_1) & = & 0 \cdot f_0 + 1 \cdot f_1 + 0 \cdot f_2 \quad = \quad f_1 \\ p_2(x_2) & = & 0 \cdot f_0 + 0 \cdot f_1 + 1 \cdot f_2 \quad = \quad f_2 \end{array}$$

This property of the basis functions may be achieved using the following construction:

$$\ell_0(x) = \frac{\text{product of linear factors for each node except } x_0}{\text{numerator evaluated at } x = x_0} = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}$$

$$\ell_1(x) = \frac{\text{product of linear factors for each node except } x_1}{\text{numerator evaluated at } x = x_1} = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}$$

$$\ell_2(x) = \frac{\text{product of linear factors for each node except } x_2}{\text{numerator evaluated at } x = x_2} = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Notice that these basis polynomials, $\ell_i(x)$, satisfy the condition given in equation (4.1), namely:

$$\begin{array}{lll} \ell_0(x_0) = 1, & \ell_0(x_1) = 0, & \ell_0(x_2) = 0 \\ \ell_1(x_0) = 0, & \ell_1(x_1) = 1, & \ell_1(x_2) = 0 \\ \ell_2(x_0) = 0, & \ell_2(x_1) = 0, & \ell_2(x_2) = 1 \end{array}$$

Example 4.1.7. For the data $(1,2)$, $(3,3)$ and $(5,4)$ the Lagrange form of the interpolating polynomial is

$$\begin{aligned} p_2(x) &= (2)\frac{(x-3)(x-5)}{(\text{numerator at } x=1)} + (3)\frac{(x-1)(x-5)}{(\text{numerator at } x=3)} + (4)\frac{(x-1)(x-3)}{(\text{numerator at } x=5)} \\ &= (2)\frac{(x-3)(x-5)}{(1-3)(1-5)} + (3)\frac{(x-1)(x-5)}{(3-1)(3-5)} + (4)\frac{(x-1)(x-3)}{(5-1)(5-3)} \end{aligned}$$

The polynomials multiplying the data values (2), (3) and (4), respectively, are *quadratic basis functions.*

More generally, consider the polynomial $p_N(x)$ of degree N interpolating to the data $\{(x_i, f_i)\}_{i=0}^N$:

$$p_N(x) = \sum_{i=0}^{N} f_i \cdot \ell_i(x)$$

where the **Lagrange basis polynomials** $\ell_k(x)$ are polynomials of degree N and have the property

$$\ell_k(x_j) = \begin{cases} 1, & k = j \\ 0, & k \neq j \end{cases}$$

so that clearly

$$p_N(x_i) = f_i, \ i = 0, 1, \cdots, N$$

The basis polynomials may be defined by

$$\ell_k(x) \equiv \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_N)}{\text{numerator evaluated at } x=x_k} = \prod_{j=0, j\neq k}^{N} \frac{(x-x_j)}{(x_k-x_j)}$$

Algebraically, the basis function $\ell_k(x)$ is a fraction whose numerator is the product of the linear factors $(x - x_i)$ associated with each of the nodes x_i except x_k, and whose denominator is the value of its numerator at the node x_k.

Pseudocode that can be used to evaluate the Lagrange form of the interpolating polynomial is given in Fig. 4.4. That is, given the data $\{(x_i, f_i)\}_{i=0}^N$ and the value z, the pseudocode in Fig. 4.4 returns the value of the interpolating polynomial $p_N(z)$.

Unlike when using the Newton form of the interpolating polynomial, the Lagrange form has no coefficients whose values must be determined. In one sense, Lagrange form provides an explicit solution of the interpolating conditions. However, the Lagrange form of the interpolating polynomial can be more expensive to evaluate than either the power form or the Newton form (see Problem 4.1.20).

In summary, the Newton form of the interpolating polynomial is attractive because it is easy to

- Determine the coefficients.
- Evaluate the polynomial at specified values via nested multiplication.
- Extend the polynomial to incorporate additional interpolation points and data.

The Lagrange form of the interpolating polynomial

- is useful theoretically because it does not require solving a linear system, and
- explicitly shows how each data value f_i affects the overall interpolating polynomial.

Evaluating the Lagrange Form

Input: data (x_i, f_i)
scalar z

Output: $p = p_N(z)$

```
p := 0
for k = 0 to N
    ℓ_num := 1
    ℓ_den := 1
    for j = 0 to k − 1
        ℓ_num := ℓ_num ∗ (z − x_j)
        ℓ_den := ℓ_den ∗ (x_k − x_j)
    next j
    for j = k + 1 to N
        ℓ_num := ℓ_num ∗ (z − x_j)
        ℓ_den := ℓ_den ∗ (x_k − x_j)
    next j
    ℓ := ℓ_num/ℓ_den
    p = p + f_k ∗ ℓ
next k
```

Figure 4.4: Pseudocode to evaluate the Lagrange form of the interpolating polynomial, $p_N(x) = f_0 \cdot \ell_0(x) + f_1 \cdot \ell_1(x) + \cdots + f_N \cdot \ell_N(x)$, at $x = z$.

Problem 4.1.15. *Determine the Lagrange form of the interpolating polynomial to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$. Is it the same polynomial as in Example 4.1.4? Why or why not?*

Problem 4.1.16. *Show that $p_2(x) = f_0 \cdot l_0(x) + f_1 \cdot l_1(x) + f_2 \cdot l_2(x)$ interpolates to the data $\{(x_i, f_i)\}_{i=0}^{2}$.*

Problem 4.1.17. *For the data $(1, 2)$, $(3, 3)$ and $(5, 4)$ in Example 4.1.7, check that the Lagrange form of the interpolating polynomial agrees precisely with the power series form $\frac{3}{2} + \frac{1}{2}x$ fitting to the same data, as for the Newton form of the interpolating polynomial.*

Problem 4.1.18. *Determine the Lagrange form of the interpolating polynomial for the data $(0, 1)$, $(-1, 0)$, $(1, 2)$ and $(2, 0)$. Check that you have determined the same polynomial as the Newton form of the interpolating polynomial for the same data, see Problem 4.1.11.*

Problem 4.1.19. *Determine the Lagrange form of the interpolating polynomial for the data $(0, 1)$, $(-1, 2)$, $(1, 0)$, $(2, -1)$ and $(-2, 3)$. Check that you have determined the same polynomial as the Newton form of the interpolating polynomial for the same data, see Problem 4.1.12.*

Problem 4.1.20. *Let $p_N(x)$ be the Lagrange form of the interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^{N}$. Determine the number of additions (subtractions), multiplications, and divisions that the code segments in Fig. 4.4 use when evaluating the Lagrange form of $p_N(x)$ at one value of $x = z$.*
How does the cost of using the Lagrange form of the interpolating polynomial for m different evaluation points x compare with the cost of using the Newton form for the same task? (See Problem 4.1.14.)

Problem 4.1.21. *In this problem we consider "cubic Hermite interpolation" where the function and its first derivative are interpolated at the points $x = 0$ and $x = 1$:*

1. *For the function $\phi(x) \equiv (1+2x)(1-x)^2$ show that $\phi(0) = 1, \phi(1) = 0, \phi'(0) = 0, \phi'(1) = 0$*
2. *For the function $\psi(x) \equiv x(1-x)^2$ show that $\psi(0) = 0, \psi(1) = 0, \psi'(0) = 1, \psi'(1) = 0$*
3. *For the cubic Hermite interpolating polynomial $P(x) \equiv f(0)\phi(x) + f'(0)\psi(x) + f(1)\phi(1-x) - f'(1)\psi(1-x)$ show that $P(0) = f(0), P'(0) = f'(0), P(1) = f(1), P'(1) = f'(1)$*

4.1.4 Chebyshev Polynomials and Series

Next, we consider the Chebyshev polynomials, $T_j(x)$, and we explain why Chebyshev series provide a better way to represent polynomials $p_N(x)$ than do power series (that is, why a Chebyshev polynomial basis often provides a better representation for the polynomials than a power series basis). This representation only works if all the data are in the interval $[-1, 1]$ but it is easy to transform any finite interval $[a, b]$ onto the interval $[-1, 1]$ as we see later. Rather than discuss transformations to put all the data in the interval $[-1, 1]$, for simplicity we assume the data is already in place.

The j^{th} Chebyshev polynomial $T_j(x)$ is defined by

$$T_j(x) \equiv \cos(j \arccos(x)), \quad j = 0, 1, 2, \cdots .$$

Note, this definition only works on the interval $x \in [-1, 1]$, because that is the domain of the function $\arccos(x)$. Now, let us compute the first few Chebyshev polynomials (and convince ourselves that they are polynomials):

$$\begin{aligned} T_0(x) &= \cos(0 \arccos(x)) = 1 \\ T_1(x) &= \cos(1 \arccos(x)) = x \\ T_2(x) &= \cos(2 \arccos(x)) = 2[\cos(\arccos(x))]^2 - 1 = 2x^2 - 1 \end{aligned}$$

We have used the double angle formula, $\cos 2\theta = 2 \cdot \cos^2 \theta - 1$, to derive $T_2(x)$, and we use its extension, the addition formula $\cos(\alpha) + \cos(\beta) = 2 \cos\left(\frac{\alpha+\beta}{2}\right) \cos\left(\frac{\alpha-\beta}{2}\right)$, to derive the higher degree Chebyshev polynomials. We have

$$\begin{aligned} T_{j+1}(x) + T_{j-1}(x) &= \cos[(j+1) \arccos(x)] + \cos[(j-1) \arccos(x)] \\ &= \cos[j \cdot \arccos(x) + \arccos(x)] + \cos[j \cdot \arccos(x) - \arccos(x)] \\ &= 2 \cos[j \cdot \arccos(x)] \cos[\arccos(x)] \\ &= 2x\, T_j(x) \end{aligned}$$

So, starting from $T_0(x) = 1$ and $T_1(x) = x$, the Chebyshev polynomials may be defined by the recurrence relation

$$T_{j+1}(x) = 2xT_j(x) - T_{j-1}(x), \; j = 1, 2, \ldots$$

By this construction we see that the Chebyshev polynomial $T_N(x)$ has exact degree N and that if N is even (odd) then $T_N(x)$ involves only even (odd) powers of x; see Problems 4.1.24 and 4.1.25.

A **Chebyshev series** of order N has the form:

$$b_0 T_0(x) + b_1 T_1(x) + \cdots + b_N T_N(x)$$

for a given set of coefficients $b_0, b_1, \cdots, b_N$. It is a polynomial of degree N; see Problem 4.1.26. The first eight Chebyshev polynomials written in power series form, and the first eight powers of x written in Chebyshev series form, are given in Table 4.2. This table illustrates the fact that every polynomial can be written either in power series form or in Chebyshev series form.

$T_0(x)$	=	1	$T_0(x)$	=	1
$T_1(x)$	=	x	$T_1(x)$	=	x
$T_2(x)$	=	$2x^2-1$	$\frac{1}{2}\{T_2(x)+T_0(x)\}$	=	x^2
$T_3(x)$	=	$4x^3-3x$	$\frac{1}{4}\{T_3(x)+3T_1(x)\}$	=	x^3
$T_4(x)$	=	$8x^4-8x^2+1$	$\frac{1}{8}\{T_4(x)+4T_2(x)+3T_0(x)\}$	=	x^4
$T_5(x)$	=	$16x^5-20x^3+5x$	$\frac{1}{16}\{T_5(x)+5T_3(x)+10T_1(x)\}$	=	x^5
$T_6(x)$	=	$32x^6-48x^4+18x^2-1$	$\frac{1}{32}\{T_6(x)+6T_4(x)+15T_2(x)+10T_0(x)\}$	=	x^6
$T_7(x)$	=	$64x^7-112x^5+56x^3-7x$	$\frac{1}{64}\{T_7(x)+7T_5(x)+21T_3(x)+35T_1(x)\}$	=	x^7

Table 4.2: Chebyshev polynomial conversion table

It is easy to compute the zeros of the first few Chebyshev polynomials manually. You'll find that $T_1(x)$ has a zero at $x=0$, $T_2(x)$ has zeros at $x=\pm\frac{1}{\sqrt{2}}$, $T_3(x)$ has zeros at $x=0,\pm\frac{\sqrt{3}}{2}$ etc. What you'll observe is that all the zeros are real and are in the interval $(-1,1)$, and that the zeros of $T_n(x)$ interlace those of $T_{n+1}(x)$; that is, between each pair of zeros of $T_{n+1}(x)$ is zero of $T_n(x)$. In fact, all these properties are seen easily using the general formula for the zeros of the Chebyshev polynomial $T_n(x)$:

$$x_i=\cos\left(\frac{2i-1}{2n}\pi\right),\ i=1,2,\ldots,n$$

which are so-called Chebyshev points. These points are important when discussing error on polynomial interpolation; see Section 4.2.

One advantage of using a Chebyshev polynomial $T_j(x)=\cos(j\arccos(x))$ over using the corresponding power term x^j of the same degree to represent data for values $x\in[-1,+1]$ is that $T_j(x)$ oscillates j times between $x=-1$ and $x=+1$, see Fig. 4.5. Also, as j increases the first and last zero crossings for $T_j(x)$ get progressively closer to, but never reach, the interval endpoints $x=-1$ and $x=+1$. This is illustrated in Fig. 4.6 and Fig. 4.7, where we plot some of the Chebyshev and power series bases on the interval $[0,1]$, with transformed variable $\bar{x}=2x-1$. That is, Fig. 4.6 shows $T_3(\bar{x})$, $T_4(\bar{x})$ and $T_5(\bar{x})$ for $x\in[0,1]$, and Fig. 4.7 shows the corresponding powers $\bar{x}^3$, $\bar{x}^4$ and $\bar{x}^5$. Observe that the graphs of the Chebyshev polynomials are quite distinct while those of the powers of x are quite closely grouped. When we can barely discern the difference of two functions graphically it is likely that the computer will have difficulty discerning the difference numerically.

Suppose we want to construct an interpolating polynomial of degree N using a Chebyshev series $p_N(x)=\sum_{j=0}^{N}b_jT_j(x)$. Given the data $(x_i,f_i)_{i=0}^N$, where all the $x_i\in[-1,1]$, we need to build and solve the linear system

$$p_N(x_i)\equiv\sum_{j=0}^{N}b_jT_j(x_i)=f_i,\quad i=0,1,\ldots,N$$

This is simply a set of $(N+1)$ equations in the $(N+1)$ unknowns, the coefficients b_j, $j=0,1,\ldots,N$. To evaluate $T_j(x_i)$, $j=0,1,\ldots,N$ in the i^{th} equation we simply use the recurrence relation $T_{j+1}(x_i)=2x_iT_j(x_i)-T_{j-1}(x_i)$, $j=1,2,\ldots,N-1$ with $T_0(x_i)=1, T_1(x_i)=x_i$. The resulting linear system can then be solved using, for example, the techniques discussed in Chapter 3. By the polynomial interpolation uniqueness theorem, we compute the same polynomial (written in a different form) as when using each of the previous interpolation methods on the same data.

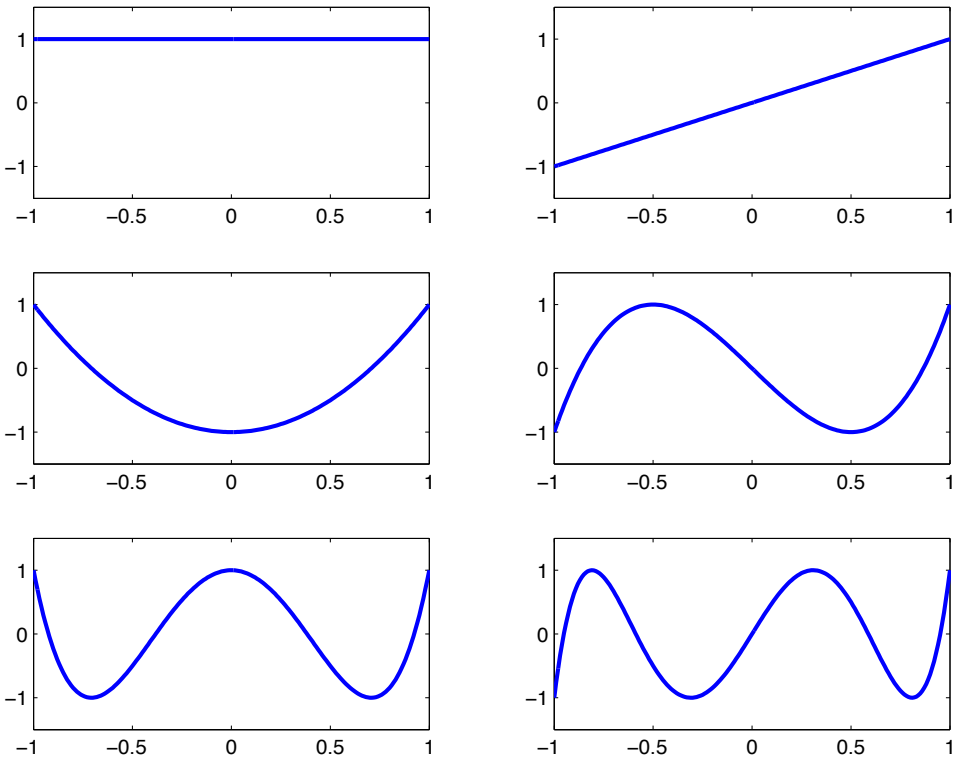

Figure 4.5: The Chebyshev polynomials $T_0(x)$ top left; $T_1(x)$ top right; $T_2(x)$ middle left; $T_3(x)$ middle right; $T_4(x)$ bottom left; $T_5(x)$ bottom right.

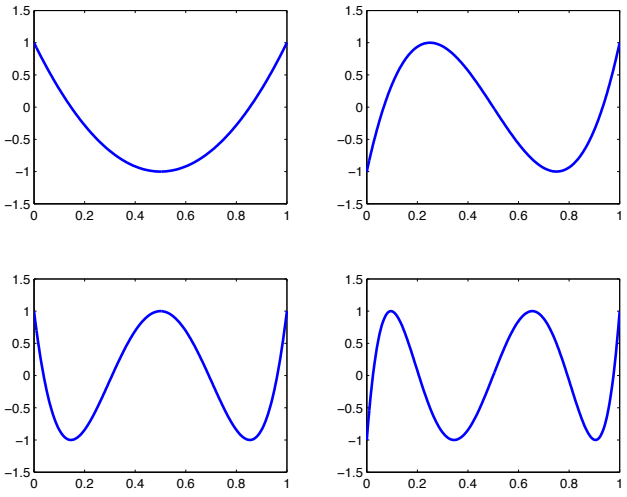

Figure 4.6: The shifted Chebyshev polynomials $T_2(\bar{x})$ top left; $T_3(\bar{x})$ top right; $T_4(\bar{x})$ bottom left, $T_5(\bar{x})$ bottom right.

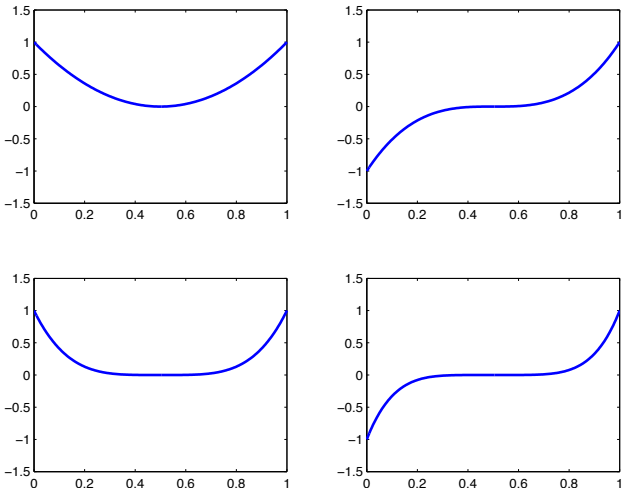

Figure 4.7: The powers $\bar{x}^2$ top left; $\bar{x}^3$ top right; $\bar{x}^4$ bottom left, $\bar{x}^5$ bottom right.

The algorithm most often used to evaluate a Chebyshev series is a recurrence that is similar to Horner's scheme for power series (see Chapter 6). Indeed, the power series form and the corresponding Chebyshev series form of a given polynomial can be evaluated at essentially the same arithmetic cost using these recurrences.

Problem 4.1.22. *In this problem you are to exploit the Chebyshev recurrence relation.*

1. *Use the Chebyshev recurrence relation to reproduce Table 4.2.*
2. *Use the Chebyshev recurrence relation and Table 4.2 to compute $T_8(x)$ as a power series in x.*
3. *Use Table 4.2 and the formula for $T_8(x)$ to compute x^8 as a Chebyshev series.*

Problem 4.1.23. *Use Table 4.2 to*

1. *Write $3T_0(x) - 2T_1(x) + 5T_2(x)$ as a polynomial in power series form.*
2. *Write the polynomial $4 - 3x + 5x^2$ as a Chebyshev series involving $T_0(x)$, $T_1(x)$ and $T_2(x)$.*

Problem 4.1.24. *Prove that the n^{th} Chebyshev polynomial $T_N(x)$ is a polynomial of degree N.*

Problem 4.1.25. *Prove the following results that are easy to observe in Table 4.2.*

1. *The even indexed Chebyshev polynomials $T_{2n}(x)$ all have power series representations in terms of even powers of x only.*
2. *The odd powers x^{2n+1} have Chebyshev series representations involving only odd indexed Chebyshev polynomials $T_{2s+1}(x)$.*

Problem 4.1.26. *Prove that the Chebyshev series*

$$b_0T_0(x) + b_1T_1(x) + \cdots + b_NT_N(x)$$

is a polynomial of degree N.

Problem 4.1.27. *Show that the* Chebyshev points $x_i = \cos\left(\frac{2i-1}{2N}\pi\right)$, $i = 1, 2, \ldots, N$ *are the complete set of zeros of* $T_N(x)$

Problem 4.1.28. *Show that the points* $x_i = \cos\left(\frac{i}{N}\pi\right)$, $i = 1, 2, \ldots, N-1$ *are the complete set of extrema of* $T_N(x)$.

4.2 The Error in Polynomial Interpolation

Let $p_N(x)$ be the polynomial of degree N interpolating to the data $\{(x_i, f_i = f(x_i))\}_{i=0}^N$. How accurately does the polynomial $p_N(x)$ approximate the data function $f(x)$ at any point x? Mathematically the answer to this question is the same for all forms of the polynomial of a given degree interpolating the same data. However, as we see later, there are differences in accuracy in practice, but these are related not to the accuracy of polynomial interpolation but to the accuracy of the implementation.

Let the evaluation point x and all the interpolation points $\{x_i\}_{i=0}^N$ lie in a closed interval $[a, b]$. An advanced course in numerical analysis shows, using Rolle's Theorem (see Problem 4.2.1) repeatedly, that the error expression may be written as

$$f(x) - p_N(x) = \frac{\omega_{N+1}(x)}{(N+1)!} f^{(N+1)}(\xi_x)$$

where $\omega_{N+1}(x) \equiv (x - x_0)(x - x_1)\cdots(x - x_N)$ and ξ_x is some (unknown) point in the interval $[a, b]$. The precise location of the point ξ_x depends on the function $f(x)$, the point x at which the interpolating polynomial is to be evaluated, and the data points $\{x_i\}_{i=0}^N$. The term $f^{(N+1)}(\xi_x)$ is the $(N+1)^{\text{st}}$ derivative of $f(x)$ evaluated at the point $x = \xi_x$. Some of the intrinsic properties of the interpolation error are:

1. For any value of i, the error is zero when $x = x_i$ because $w_{N+1}(x_i) = 0$ (the interpolating conditions).

2. The error is zero when the data f_i are measurements of a polynomial $f(x)$ of exact degree N because then the $(N+1)^{\text{st}}$ derivative $f^{(N+1)}(x)$ is identically zero. This is simply a statement of the uniqueness theorem of polynomial interpolation.

Taking absolute values in the interpolation error expression and maximizing both sides of the resulting inequality over $x \in [a, b]$, we obtain the **polynomial interpolation error bound**

$$\max_{x\in[a,b]} |f(x) - p_N(x)| \leq \max_{x\in[a,b]} |\omega_{N+1}(x)| \cdot \frac{\max\limits_{z\in[a,b]} |f^{(N+1)}(z)|}{(N+1)!} \tag{4.2}$$

To make this error bound small we must make either the term $\max\limits_{x\in[a,b]} \frac{|f^{(N+1)}(x)|}{(N+1)!}$ or the term $\max\limits_{x\in[a,b]} |\omega_{N+1}(x)|$, or both, small. Generally, we have little information about the function $f(x)$ or its derivatives, even about their sizes, and in any case we cannot change them to minimize the bound.

In the absence of information about $f(x)$ or its derivatives, we might aim to choose the nodes $\{x_i\}_{i=0}^N$ so that $|\omega_{N+1}(x)|$ is small throughout $[a, b]$. The plots in Fig. 4.8 illustrate that generally equally spaced nodes defined by $x_i = a + \frac{i}{N}(b-a)$, $i = 0, 1, \ldots, N$ are not a good choice, as for them the polynomial $\omega_{N+1}(x)$ oscillates with zeros at the nodes as anticipated, but with the amplitudes of the oscillations growing as the evaluation point x approaches either endpoint of the interval $[a, b]$. Of course, data is usually measured at equally spaced points and this is a matter over which you

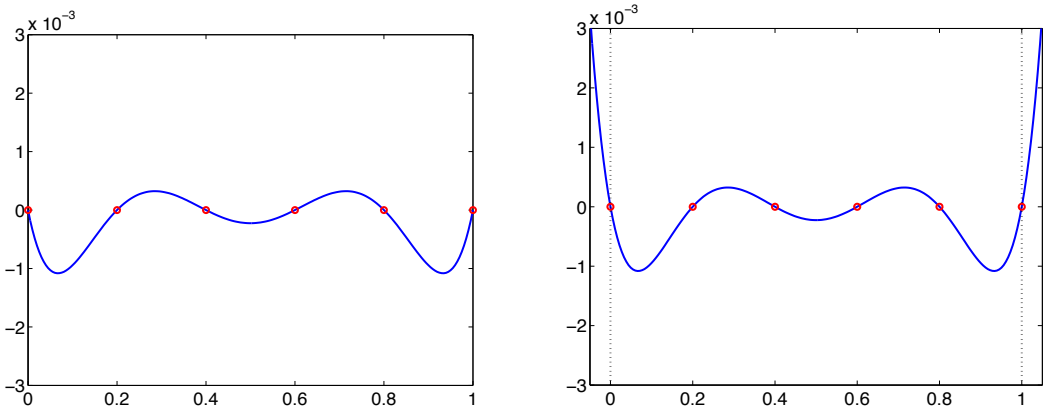

Figure 4.8: Plot of the polynomial $\omega_{N+1}(x)$ for equally spaced nodes $x_i \in [0,1]$, and $N = 5$. The equally spaced nodes are denoted by circles. The left plot shows $\omega_{N+1}(x)$ on the interval $[0,1]$, and the right plot shows $\omega_{N+1}(x)$ on the interval $[-0.05, 1.05]$.

will have little or no control. So, though equally spaced points are a bad choice in theory, they may be our only choice!

The right plot in Fig. 4.8 depicts the polynomial $\omega_{N+1}(x)$ for points x slightly outside the interval $[a,b]$ where $\omega_{N+1}(x)$ grows like x^{N+1}. Evaluating the polynomial $p_N(x)$ for points x outside the interval $[a,b]$ is *extrapolation*. We can see from this plot that if we evaluate the polynomial $p_N(x)$ outside the interval $[a,b]$, then the error component $\omega_{N+1}(x)$ can be very large, indicating that extrapolation should be avoided whenever possible; this is true for all choices of interpolation points. This observation implies that we need to know the application intended for the interpolating function before we measure the data, a fact frequently ignored when designing an experiment. We may not be able to avoid extrapolation but we certainly need to be aware of its dangers.

For the data in Fig. 4.9, we use the Chebyshev points for the interval $[a,b] = [0,1]$:

$$x_i = \frac{b+a}{2} - \frac{b-a}{2}\cos\left(\frac{2i+1}{2N+2}\pi\right) \quad i = 0, 1, \cdots, N.$$

Note that the Chebyshev points do not include the endpoints a or b and that all the Chebyshev points lie inside the open interval (a,b). For the interval $[a,b] = [-1,+1]$, the Chebyshev points are the zeros of the Chebyshev polynomial $T_{N+1}(x)$, hence the function $\omega_{N+1}(x)$ is a scaled version of $T_{N+1}(x)$ (see Section 4.1.4 for the definition of the Chebyshev polynomials and the Chebyshev points).

With the Chebyshev points as nodes, the maximum value of the polynomial $|\omega_{N+1}(x)|$ on $[a,b] = [0,1]$ is less than half its value in Fig. 4.8. As N increases, the improvement in the size of $|\omega_{N+1}(x)|$ when using the Chebyshev points rather than equally spaced points is even greater. Indeed, for polynomials of degree 20 or less, interpolation to the data measured at the Chebyshev points gives a maximum error not greater than twice the smallest possible maximum error (known as the minimax error) taken over all polynomial fits to the data (not just using interpolation). However, there are penalties:

1. As with equally space points, extrapolation should be avoided because the error component $\omega_{N+1}(x)$ can be very large for points outside the interval $[a,b]$ (see the right plot in Fig. 4.9). Indeed, extrapolation using the interpolating function based on data measured at Chebyshev points can be even more disastrous than extrapolation based on the same number of equally spaced data points.

2. It may be difficult to obtain the data f_i measured at the Chebyshev points.

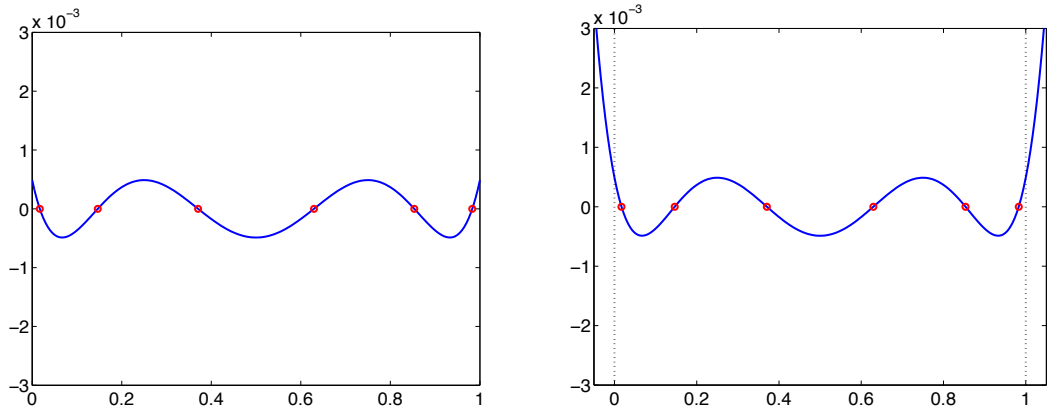

Figure 4.9: Plot of the polynomial $\omega_{N+1}(x)$ for Chebyshev nodes $x_i \in [0,1]$, and $N = 5$. The Chebyshev nodes are denoted by circles. The left plot shows $\omega_{N+1}(x)$ on the interval $[0,1]$, and the right plot shows $\omega_{N+1}(x)$ on the interval $[-0.05, 1.05]$

Figs. 4.8 — 4.9 suggest that to choose $p_N(x)$ so that it will accurately approximate all reasonable choices of $f(x)$ on an interval $[a,b]$, we should

1. Choose the nodes $\{x_i\}_{i=0}^N$ so that, for all likely evaluation points x, $\min(x_i) \le x \le \max(x_i)$.
2. If possible, choose the nodes as the Chebyshev points. If this is not possible, attempt to choose them so they are denser close to the endpoints of the interval $[a,b]$.

Problem 4.2.1. *Prove* Rolle's Theorem*: Let $f(x)$ be continuous and differentiable on the closed interval $[a,b]$ and let $f(a) = f(b) = 0$, then there exists a point $c \in (a,b)$ such that $f'(c) = 0$.*

Problem 4.2.2. *Let $x = x_0 + sh$ and $x_i = x_0 + ih$, show that $w_{n+1}(x) = h^{N+1} s(s-1)\cdots(s-N)$.*

Problem 4.2.3. *Determine the following bound for straight line interpolation (that is with a polynomial of degree one, $p_1(x)$) to the data $(a, f(a))$ and $(b, f(b))$*

$$\max_{x\in[a,b]} |f(x) - p_1(x)| \le \frac{1}{8}|b-a|^2 \max_{z\in[a,b]} |f^{(2)}(z)|$$

[Hint: You need to use the bound (4.2) for this special case, and use standard calculus techniques to find $\max_{x\in[a,b]} |\omega_{N+1}(x)|$.]

Problem 4.2.4. *Let $f(x)$ be a polynomial of degree $N+1$ for which $f(x_i) = 0$, $i = 0, 1, \cdots, N$. Show that the interpolation error expression reduces to $f(x) = A\, w_{N+1}(x)$ for some constant A. Explain why this result is to be expected.*

Problem 4.2.5. *For $N = 5$, $N = 10$ and $N = 15$ plot, on one graph, the three polynomials $\omega_{N+1}(x)$ for $x \in [0,1]$ defined using equally spaced nodes. For the same values of N plot, on one graph, the three polynomials $\omega_{N+1}(x)$ for $x \in [0,1]$ defined using Chebyshev nodes. Compare the three maximum values of $|\omega_{N+1}(x)|$ on the interval $[0,1]$ for each of these two graphs. Also, compare the corresponding maximum values in the two graphs.*

Runge's Example

To use polynomial interpolation to obtain an accurate estimate of $f(x)$ on a fixed interval $[a, b]$, it is natural to think that increasing the number of interpolating points in $[a, b]$, and hence increasing the degree of the polynomial, will reduce the error in the polynomial interpolation to $f(x)$. In fact, for some functions $f(x)$ this approach may worsen the accuracy of the interpolating polynomial. When this is the case, the effect is usually greatest for equally spaced interpolation points. At least a part of the problem arises from the term $\max_{x\in[a,b]} \dfrac{|f^{(N+1)}(x)|}{(N+1)!}$ in the expression for interpolation error. This term may grow, or at least not decay, as N increases, even though the denominator $(N+1)!$ grows very quickly as N increases.

Consider the function $f(x) = \dfrac{1}{1+x^2}$ on the interval $[-5, 5]$. Fig. 4.10 shows a plot of interpolating polynomials of degree $N = 10$ for this function using 11 (i.e., $N + 1$) equally spaced nodes and 11 Chebyshev nodes. Also shown in the figure is the error $f(x) - p_N(x)$ for each interpolating polynomial. Note that the behavior of the error for the equally spaced points clearly mimics the behavior of $\omega_{N+1}(x)$ in Fig. 4.8. For the same number of Chebyshev points the error is much smaller and more evenly distributed on the interval $[-5, 5]$ but it still mimics $\omega_{N+1}(x)$ for this placement of points.

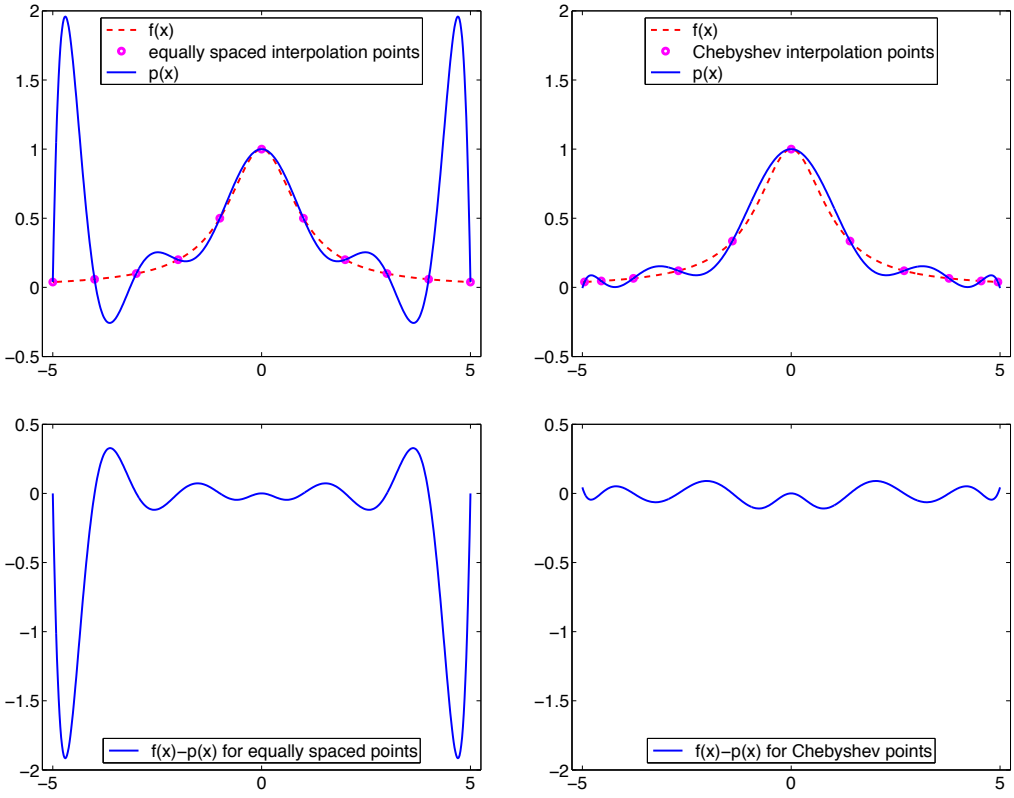

Figure 4.10: Interpolating polynomials for $f(x) = \dfrac{1}{1+x^2}$ using 11 equally spaced points (top left) and 11 Chebyshev points (top right) on the interval $[-5, 5]$. The associated error functions $f(x) - p_{10}(x)$ are shown in the bottom two plots.

Problem 4.2.6. Runge's example. *Consider the function* $f(x) = \dfrac{1}{1+x^2}$ *on the interval* $[-5, +5]$.

1. *For* $N = 5$, $N = 10$ *and* $N = 15$ *plot the error* $e(x) = p_N(x) - f(x)$ *where the polynomial* $p_N(x)$ *is computed by interpolating at the* $N + 1$ *equally spaced nodes,* $x_i = a + \dfrac{i}{N}(b-a), i = 0, 1, \cdots, N$.
2. *Repeat Part 1 but interpolate at the* $N + 1$ *Chebyshev nodes shifted to the interval* $[-5, 5]$.

Create a table listing the approximate maximum absolute error and its location in $[a, b]$ *as a function of* N*; there will be two columns one for each of Parts 1 and 2. [Hint: For simplicity, compute the interpolating polynomial* $p_N(x)$ *using the Lagrange form. To compute the approximate maximum error you will need to sample the error between each pair of nodes (recall, the error is zero at the nodes) — sampling at the midpoints of the intervals between the nodes will give you a sufficiently accurate estimate.]*

Problem 4.2.7. *Repeat the previous problem with data taken from* $f(x) = e^x$ *instead of* $f(x) = \dfrac{1}{1+x^2}$. *What major differences do you observe in the behavior of the error?*

The Effect of the Method of Representation of the Interpolating Polynomial

A second source of error in polynomial interpolation is that from the computation of the polynomial interpolating function and its evaluation. It is "well-known" that large Vandermonde systems tend to be ill–conditioned, but we see errors of different sizes for all our four fitting techniques (Vandermonde, Newton, Lagrange and Chebyshev) and the Newton approach (in our implementation) turns out to be even more ill–conditioned than the Vandermonde as the number of interpolation points increases. To illustrate this fact we present results from interpolating to e^x with $n = 10, 20, \ldots, 100$ equally spaced points in the interval $[-1, 1]$ so we are interpolating with polynomials of degrees $N = 9, 19, \ldots, 99$, respectively. In Table 4.3, we show the maximum absolute errors (over 201 equally spaced points in $[-1, 1]$) of this experiment. We don't know the size of the interpolation error as the error shown is a combination of interpolation error, computation of fit error and evaluation error, though we can be fairly confident that the errors for $n = 10$ are dominated by interpolation error since the errors are all approximately the same. The interpolation errors then decrease and at $n = 20$ the error is clearly dominated by calculation errors (since all the errors are different). For $n > 20$ it is our supposition that calculation errors dominate, and in any case, for each value of n the interpolation error cannot be larger than the smallest of the errors shown. Clearly, for large enough n using the Chebyshev series is best, and for small n it doesn't matter much which method is used. Surprisingly, the Lagrange interpolating function for which we do not solve a system of linear equations but which involves an expensive evaluation has very large errors when n is large, but Newton is always worst. A plot of the errors shows that in all cases the errors are largest close to the ends of the interval of interpolation.

We repeated the experiment using Chebyshev points; the results are shown in Table 4.4. Here, the interpolation error is smaller, as is to be anticipated using Chebyshev points, and the computation error does not show until $n = 50$ for Newton and hardly at all for the Lagrange and Chebyshev fits. When the errors are large they are largest near the ends of the interval of interpolation but overall they are far closer to being evenly distributed across the interval than in the equally spaced points case.

The size of the errors reported above depends, to some extent, on the way that the interpolation function is coded. In each case, we coded the function in the "obvious" way (using the formulas presented in this text). But, using these formulas is not necessarily the best way to reduce the errors. It is clear that in the case of the Newton form the choice of the order of the interpolation

n	Vandermonde	Newton	Lagrange	Chebyshev
10	3.837398e-009	3.837397e-009	3.837398e-009	3.837399e-009
20	3.783640e-013	1.100786e-013	1.179390e-012	1.857403e-013
30	6.497576e-010	7.048900e-011	7.481358e-010	2.436179e-011
40	5.791111e-008	1.385682e-008	8.337930e-007	3.987289e-008
50	3.855115e-005	2.178791e-005	2.021784e-004	9.303255e-005
60	1.603913e-001	4.659020e-002	9.780899e-002	2.822994e-003
70	1.326015e+003	2.905506e+004	2.923504e+002	1.934900e+000
80	8.751649e+005	3.160393e+010	3.784625e+004	4.930093e+001
90	2.280809e+010	7.175811e+015	6.391371e+007	1.914712e+000
100	2.133532e+012	4.297183e+022	1.122058e+011	1.904017e+000

Table 4.3: Errors fitting e^x using n equally spaced data points on the interval $[-1, 1]$.

n	Vandermonde	Newton	Lagrange	Chebyshev
10	6.027099e-010	6.027103e-010	6.027103e-010	6.027103e-010
20	8.881784e-016	1.332268e-015	3.552714e-015	1.554312e-015
30	8.881784e-016	9.992007e-016	2.664535e-015	1.332268e-015
40	1.110223e-015	1.110223e-015	3.552714e-015	1.332268e-015
50	7.105427e-015	1.532131e-009	3.996803e-015	1.776357e-015
60	5.494938e-012	1.224753e-004	5.773160e-015	1.332268e-015
70	1.460477e-009	1.502797e+000	4.884981e-015	2.164935e-015
80	2.291664e-006	3.292080e+005	4.440892e-015	1.332268e-015
90	2.758752e-004	3.140739e+011	5.773160e-015	1.332268e-015
100	1.422977e+000	1.389857e+018	8.881784e-015	1.776357e-015

Table 4.4: Errors fitting e^x using n Chebyshev data points in $[-1, 1]$.

points determines the diagonal values in the lower triangular matrix, and hence its degree of ill–conditioning. In our implementation of all the forms, and specifically the Newton form, we ordered the points in increasing order of value, that is starting from the left. It is less apparent what, if any, substantial effect changing the order of the points would have in the Vandermonde and Lagrange cases. It is however possible to compute these polynomials in many different ways. For example, the Lagrange form can be computed using "barycentric" formulas. Consider the standard Lagrange form $p_N(x) = \sum_{i=0}^{N} l_i(x) f_i$ where $l_i(x) = \prod_{j=0, j\neq i}^{N} \left(\frac{x - x_j}{x_i - x_j} \right)$. It is easy to see that we can rewrite this in the barycentric form $p_N(x) = l(x) \sum_{i=0}^{N} \frac{\omega_i}{x - x_i} f_i$ where $l(x) = \prod_{i=0}^{N} (x - x_i)$ and $\omega_j = \frac{1}{\prod_{k\neq j}(x_j - x_k)}$ which is less expensive for evaluation. It is also somewhat more accurate but not by enough to be even close to competitive with using the Chebyshev polynomial basis.

Problem 4.2.8. *Produce the equivalent of Tables 4.3 and 4.4 for Runge's function defined on the interval $[-1, 1]$. You should expect relatively similar behavior to that exhibited in the two tables.*

Problem 4.2.9. *Derive the barycentric form of the Lagrange interpolation formula from the standard form. For what values of x is the barycentric form indeterminate (requires computing a "zero over zero"), and how would you avoid this problem in practice*

4.3 Polynomial Splines

Now, we consider techniques designed to reduce the problems that arise when data are *interpolated* by a *single polynomial*. The first technique *interpolates* the data by a *collection of low degree polynomials* rather than by a single high degree polynomial. Another technique outlined in Section 4.4, *approximates* but not necessarily interpolates, the data by least squares fitting a *single low degree polynomial*.

Generally, by reducing the size of the interpolation error bound we reduce the actual error. Since the term $|\omega_{N+1}(x)|$ in the bound is the product of $N + 1$ linear factors $|x - x_i|$, each the distance between two points that both lie in $[a, b]$, we have $|x - x_i| \leq |b - a|$ and so

$$\max_{x\in[a,b]} |f(x) - p_N(x)| \leq |b - a|^{N+1} \cdot \frac{\max_{x\in[a,b]} |f^{(N+1)}(x)|}{(N + 1)!}$$

This (larger) bound suggests that we can make the error as small as we wish by freezing the value of N and then reducing the size of $|b-a|$. We still need an approximation over the original interval, so we use a *piecewise polynomial approximation*: the original interval is divided into non-overlapping subintervals and a different polynomial fit of the data is used on each subinterval.

4.3.1 Linear Polynomial Splines

A simple piecewise polynomial fit is the linear interpolating spline. For data $\{(x_i, f_i)\}_{i=0}^{N}$, where

$$a = x_0 < x_1 \cdots < x_N = b, \quad h \equiv \max_i |x_i - x_{i-1}|,$$

the **linear spline** $S_{1,N}(x)$ is a continuous function that interpolates to the data and is constructed from linear functions that we identify as two–point interpolating polynomials:

$$S_{1,N}(x) = \begin{cases} f_0 \dfrac{x-x_1}{x_0-x_1} & + & f_1 \dfrac{x-x_0}{x_1-x_0} & \text{when } x \in [x_0, x_1] \\ f_1 \dfrac{x-x_2}{x_1-x_2} & + & f_2 \dfrac{x-x_1}{x_2-x_1} & \text{when } x \in [x_1, x_2] \\ & \vdots & & \\ f_{N-1} \dfrac{x-x_N}{x_{N-1}-x_N} & + & f_N \dfrac{x-x_{N-1}}{x_N-x_{N-1}} & \text{when } x \in [x_{N-1}, x_N] \end{cases}$$

From the bound on the error for polynomial interpolation, in the case of an interpolating polynomial of degree one,

$$\begin{aligned} \max_{z \in [x_{i-1}, x_i]} |f(z) - S_{1,N}(z)| &\leq \frac{|x_i - x_{i-1}|^2}{8} \cdot \max_{x \in [x_{i-1}, x_i]} |f^{(2)}(x)| \\ &\leq \frac{h^2}{8} \cdot \max_{x \in [a,b]} |f^{(2)}(x)| \end{aligned}$$

(See Problem 4.2.3.) That is, the bound on the maximum absolute error behaves like h^2 as the maximum interval length $h \to 0$. Suppose that the nodes are chosen to be equally spaced in $[a, b]$, so that $x_i = a + ih$, $i = 0, 1, \cdots, N$, where $h \equiv \dfrac{b-a}{N}$. As the number of points N increases, the error in using $S_{1,N}(z)$ as an approximation to $f(z)$ tends to zero like $\dfrac{1}{N^2}$.

Example 4.3.1. We construct the linear spline to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$:

$$S_{1,2}(x) = \begin{cases} 0 \cdot \dfrac{x-0}{(-1)-0} & + & 1 \cdot \dfrac{x-(-1)}{0-(-1)} & \text{when } x \in [-1, 0] \\ 1 \cdot \dfrac{x-1}{0-1} & + & 3 \cdot \dfrac{x-0}{1-0} & \text{when } x \in [0, 1] \end{cases}$$

An alternative (but equivalent) method for representing a linear spline uses a **linear B–spline basis**. A B-spline basis is one made up of functions with minimal (compact) support; that is, the functions making up the basis should be defined to be nonzero on the minimum number of contiguous intervals. In the case of linear splines each B-spline basis function is nonzero on at most two contiguous intervals and may be represented as follows. The linear B–spline basis, $L_i(x), i = 0, 1, \cdots, N$, is chosen so that $L_i(x_j) = 0$ for all $j \neq i$ and $L_i(x_i) = 1$. Here, each $L_i(x)$ is a "roof" shaped function with the apex of the roof at $(x_i, 1)$ and the span on the interval $[x_{i-1}, x_{i+1}]$, and with $L_i(x) \equiv 0$ outside $[x_{i-1}, x_{i+1}]$. That is,

$$L_0(x) = \begin{cases} \dfrac{x-x_1}{x_0-x_1} & \text{when} \quad x \in [x_0, x_1] \\ 0 & \text{for all other } x, \end{cases}$$

$$L_i(x) = \begin{cases} \dfrac{x-x_{i-1}}{x_i-x_{i-1}} & \text{when} \quad x \in [x_{i-1}, x_i] \\ \dfrac{x-x_{i+1}}{x_i-x_{i+1}} & \text{when} \quad x \in [x_i, x_{i+1}] \\ 0 & \text{for all other } x \end{cases} \qquad i = 1, 2, \ldots, N-1,$$

and

$$L_N(x) = \begin{cases} \dfrac{x - x_{N-1}}{x_N - x_{N-1}} & \text{when} \quad x \in [x_{N-1}, x_N] \\ 0 & \text{for all other } x \end{cases}$$

In terms of the linear B–spline basis we can write

$$S_{1,N}(x) = \sum_{i=0}^{N} L_i(x) \cdot f_i$$

Example 4.3.2. We construct the linear spline to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$ using the linear B–spline basis. First we construct the basis:

$$\begin{aligned} L_0(x) &= \begin{cases} \dfrac{x-0}{(-1)-0}, & x \in [-1, 0] \\ 0, & x \in [0, 1] \end{cases} \\ L_1(x) &= \begin{cases} \dfrac{x-(-1)}{0-(-1)}, & x \in [-1, 0] \\ \dfrac{x-1}{0-1}, & x \in [0, 1] \end{cases} \\ L_2(x) &= \begin{cases} 0, & x \in [-1, 0] \\ \dfrac{x-0}{1-0}, & x \in [0, 1] \end{cases} \end{aligned}$$

which are shown in Fig. 4.11. Notice that each $L_i(x)$ is a "roof" shaped function. The linear spline interpolating to the data is then given by

$$S_{1,2}(x) = 0 \cdot L_0(x) + 1 \cdot L_1(x) + 3 \cdot L_2(x)$$

Problem 4.3.1. *Let $S_{1,N}(x)$ be a linear spline.*

1. *Show that $S_{1,N}(x)$ is continuous and that it interpolates the data $\{(x_i, f_i)\}_{i=0}^{N}$.*
2. *At the interior nodes x_i, $i = 1, 2, \cdots, N-1$, show that $S'_{1,N}(x_i^-) = \dfrac{f_i - f_{i-1}}{x_i - x_{i-1}}$ and $S'_{1,N}(x_i^+) = \dfrac{f_{i+1} - f_i}{x_{i+1} - x_i}$*
3. *Show that, in general, $S'_{1,N}(x)$ is discontinuous at the interior nodes.*
4. *Under what circumstances would $S_{1,N}(x)$ have a continuous derivative at $x = x_i$?*
5. *Determine the linear spline $S_{1,3}(x)$ that interpolates to the data $(0, 1)$, $(1, 2)$, $(3, 3)$ and $(5, 4)$. Is $S'_{1,3}(x)$ discontinuous at $x = 1$? At $x = 2$? At $x = 3$?*

Problem 4.3.2. *Given the data $(0, 1)$, $(1, 2)$, $(3, 3)$ and $(5, 4)$, write down the linear B–spline basis functions $L_i(x)$, $i = 0, 1, 2, 3$ and the sum representing $S_{1,3}(x)$. Show that $S_{1,3}(x)$ is the same linear spline that was described in Problem 4.3.1. Using this basis function representation of the linear spline, evaluate the linear spline at $x = 1$ and at $x = 2$.*

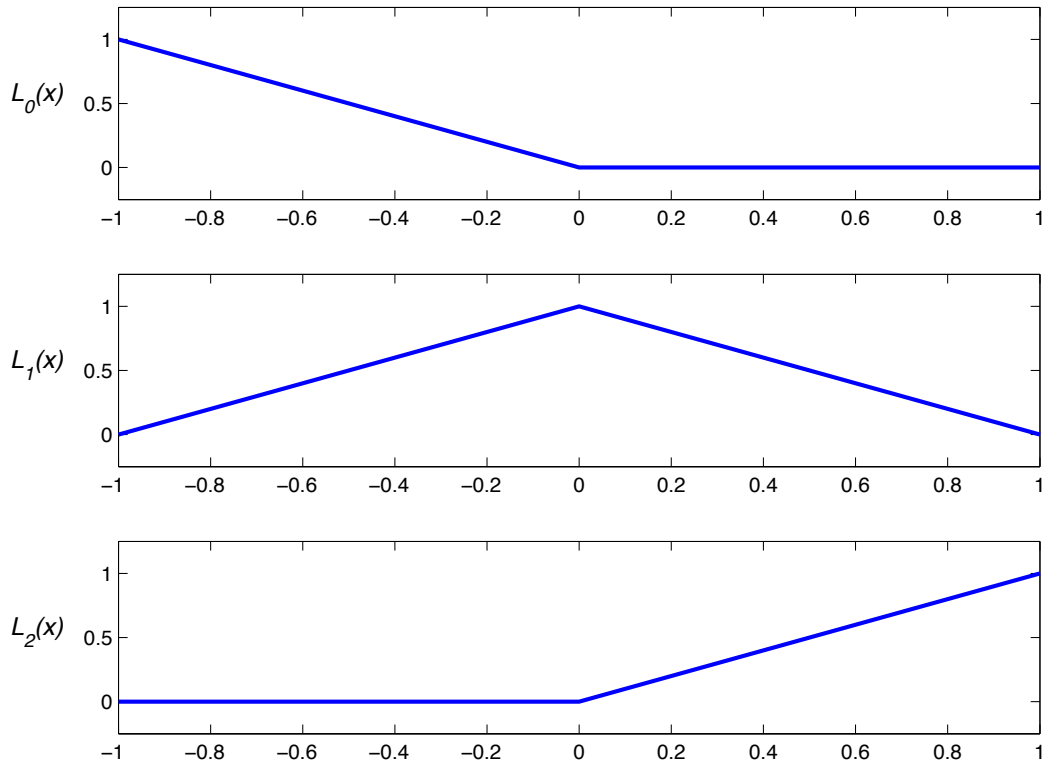

Figure 4.11: Linear B–spline basis functions for the data $(-1, 0), (0, 1)$ and $(1, 3)$. Each $L_i(x)$ is a "roof" shaped function with the apex of the roof at $(x_i, 1)$.

4.3.2 Cubic Polynomial Splines

Linear splines suffer from a major limitation: the derivative of a linear spline is generally discontinuous at each interior node, x_i. To derive a piecewise polynomial approximation with a continuous derivative requires that we use polynomial pieces of higher degree and constrain the pieces to make the curve smoother.

Before the days of Computer Aided Design, a (mechanical) spline, for example a flexible piece of wood, hard rubber, or metal, was used to help draw curves. To use a mechanical spline, pins were placed at a judicious selection of points along a curve in a design, then the spline was bent so that it touched each of these pins. Clearly, with this construction the spline *interpolates* the curve at these pins and could be used to reproduce the curve in other drawings[1]. The locations of the pins are called *knots*. We can change the shape of the curve defined by the spline by adjusting the location of the knots. For example, to interpolate to the data $\{(x_i, f_i)\}$ we can place knots at each of the nodes x_i. This produces a special interpolating cubic spline.

To derive a mathematical model of this mechanical spline, suppose the data is $\{(x_i, f_i)\}_{i=0}^{N}$ where, as for linear splines, $x_0 < x_1 < \cdots < x_N$. The shape of a mechanical spline suggests the curve between the pins is approximately a cubic polynomial. So, we model the mechanical spline by a mathematical cubic spline — a special piecewise cubic approximation. Mathematically, a cubic spline $S_{3,N}(x)$ is a C^2 (that is, twice continuously differentiable) piecewise cubic polynomial. This means that

- $S_{3,N}(x)$ is piecewise cubic; that is, between consecutive knots x_i

$$S_{3,N}(x) = \begin{cases} p_1(x) &= a_{1,0} + a_{1,1}x + a_{1,2}x^2 + a_{1,3}x^3 & x \in [x_0, x_1] \\ p_2(x) &= a_{2,0} + a_{2,1}x + a_{2,2}x^2 + a_{2,3}x^3 & x \in [x_1, x_2] \\ p_3(x) &= a_{3,0} + a_{3,1}x + a_{3,2}x^2 + a_{3,3}x^3 & x \in [x_2, x_3] \\ & \vdots & \\ p_N(x) &= a_{N,0} + a_{N,1}x + a_{N,2}x^2 + a_{N,3}x^3 & x \in [x_{N-1}, x_N] \end{cases}$$

  where $a_{i,0}$, $a_{i,1}$, $a_{i,2}$ and $a_{i,3}$ are the coefficients in the power series representation of the i^{th} cubic piece of $S_{3,N}(x)$. (Note: The approximation changes from one cubic polynomial piece to the next at the **knots** x_i.)

- $S_{3,N}(x)$ is C^2 (read C two); that is, $S_{3,N}(x)$ *is continuous and has continuous first and second derivatives* everywhere in the interval $[x_0, x_N]$ (and particularly at the knots).

To be an **interpolating cubic spline** we must have, in addition,

- $S_{3,N}(x)$ interpolates the data; that is,

$$S_{3,N}(x_i) = f_i, \;\; i = 0, 1, \cdots, N$$

(Note: the points of interpolation $\{x_i\}_{i=0}^{N}$ are called **nodes** (pins), and we have *chosen* them to coincide with the knots.) For the mechanical spline, the knots where $S_{3,N}(x)$ changes shape and the nodes where $S_{3,N}(x)$ interpolates are the same. For the mathematical spline, it is traditional to place the knots at the nodes, as in the definition of $S_{3,N}(x)$. However, this placement is a choice and not a necessity.

Within each interval (x_{i-1}, x_i) the corresponding cubic polynomial $p_i(x)$ is continuous and has continuous derivatives of all orders. Therefore, $S_{3,N}(x)$ or one of its derivatives can be discontinuous

[1]Splines were used frequently to trace the plan of an airplane wing. A master template was chosen, placed on the material forming the rib of the wing, and critical points on the template were transferred to the material. After removing the template, the curve defining the shape of the wing was "filled-in" using a mechanical spline passing through the critical points.

only at a knot. For example, consider the following illustration of what happens at the knot x_i.

For $x_{i-1} < x < x_i$, $S_{3,N}(x)$ has value $p_i(x) = a_{i,0} + a_{i,1}x + a_{i,2}x^2 + a_{i,3}x^3$	For $x_i < x < x_{i+1}$, $S_{3,N}(x)$ has value $p_{i+1}(x) = a_{i+1,0} + a_{i+1,1}x + a_{i+1,2}x^2 + a_{i+1,3}x^3$
Value of $S_{3,N}(x)$ as $x \to x_i^-$	Value of $S_{3,N}(x)$ as $x \to x_i^+$
$p_i(x_i)$	$p_{i+1}(x_i)$
$p_i'(x_i)$	$p_{i+1}'(x_i)$
$p_i''(x_i)$	$p_{i+1}''(x_i)$
$p_i'''(x_i)$	$p_{i+1}'''(x_i)$

(Here, $x \to x_i^+$ means take the limit as $x \to x_i$ from above.) Observe that the function $S_{3,N}(x)$ has two cubic pieces incident to the interior knot x_i; to the left of x_i it is the cubic $p_i(x)$ while to the right it is the cubic $p_{i+1}(x)$. Thus, a necessary and sufficient condition for $S_{3,N}(x)$ to be continuous and have continuous first and second derivatives is for these two cubic polynomials incident at the interior knot to match in value, and in first and second derivative values. So, we have a set of Smoothness Conditions; that is, at each interior knot:

$$p_i'(x_i) = p_{i+1}'(x_i), \;\; p_i''(x_i) = p_{i+1}''(x_i), \quad i = 1, 2, \cdots, N-1$$

In addition, to interpolate the data we have a set of Interpolation Conditions; that is, on the i^{th} interval:

$$p_i(x_{i-1}) = f_{i-1}, \;\; p_i(x_i) = f_i, \quad i = 1, 2, \cdots, N$$

This way of writing the interpolation conditions also forces $S_{3,N}(x)$ to be continuous at the knots.

Each of the N cubic pieces has four unknown coefficients, so our description of the function $S_{3,N}(x)$ involves $4N$ unknown coefficients. Interpolation imposes $2N$ linear constraints on the coefficients, and assuring continuous first and second derivatives imposes $2(N-1)$ additional linear constraints. (A linear constraint is a linear equation that must be satisfied by the coefficients of the polynomial pieces.) Therefore, there are a total of $4N - 2 = 2N + 2(N-1)$ linear constraints on the $4N$ unknown coefficients. In order to have the same number of equations as unknowns, we need 2 more (linear) constraints and the whole set of constraints must be linearly independent.

Natural Boundary Conditions

A little thought about the mechanical spline as it is forced to touch the pins indicates why two constraints are missing. What happens to the spline before it touches the first pin and after it touches the last? If you twist the spline at its ends you find that its shape changes. A natural condition is to let the spline rest freely without stress or tension at the first and last knot, that is don't twist it at the ends. Such a spline has "minimal energy". Mathematically, this condition is expressed as the Natural Spline Condition:

$$p_1''(x_0) = 0, \;\; p_N''(x_N) = 0$$

The so-called **natural spline** results when these conditions are used as the 2 missing linear constraints.

Despite its comforting name and easily understood physical origin, the natural spline is seldom used since it does not deliver an accurate approximation $S_{3,N}(x)$ near the ends of the interval $[x_0, x_N]$. This may be anticipated from the fact that we are forcing a zero value on the second derivative when this is not normally the value of the second derivative of the function that the data measures. A natural cubic spline is built up from cubic polynomials, so it is reasonable to expect that if the data is measured from a cubic polynomial then the natural cubic spline will reproduce the cubic polynomial. However, for example, if the data are measured from the function $f(x) = x^2$ then the natural spline $S_{3,N}(x) \neq f(x)$; the function $f(x) = x^2$ has nonzero second derivatives at the

nodes x_0 and x_N where value of the second derivative of the natural cubic spline $S_{3,N}(x)$ is zero by definition. In fact the error behaves like $O(h^2)$ where h is the largest distance between interpolation points. Since it can be shown that the best possible error behavior is $O(h^4)$, the accuracy of the natural cubic spline leaves something to be desired.

Second Derivative Conditions

To clear up the inaccuracy problem associated with the natural spline conditions we could replace them with the correct second derivative values

$$p_1''(x_0) = f''(x_0), \;\; p_N''(x_N) = f''(x_N).$$

These second derivatives of the data are not usually available but they can be replaced by sufficiently accurate approximations. If exact values or sufficiently accurate approximations are used then the resulting spline will be as accurate as possible for a cubic spline; that is the error in the spline will behave like $O(h^4)$ where h is the largest distance between interpolation points. (Approximations to the second derivative may be obtained by using polynomial interpolation. That is, two separate sets of data values near each of the endpoints of the interval $[x_0, x_N]$ are used to construct two interpolating polynomials. Then the two interpolating polynomials are each twice differentiated and the resulting twice differentiated polynomials are evaluated at the corresponding endpoints to approximate the values of $f''(x_0)$ and $f''(x_N)$.)

First Derivative (Slope) Conditions

Another choice of boundary conditions which delivers the full $O(h^4)$ accuracy of cubic splines is to use the correct first derivative values

$$p_1'(x_0) = f'(x_0), \;\; p_N'(x_N) = f'(x_N).$$

If we do not have access to the derivative of f, we can approximate it in a similar way to that described above for the second derivative.

Not-a-knot Conditions

A simpler, and accurate, spline may be determined by replacing the boundary conditions with the so-called **not-a-knot** conditions. Recall, at each knot, the spline $S_{3,N}(x)$ changes from one cubic to the next. The idea of the not-a-knot conditions is *not to change* cubic polynomials as one crosses both the first and the last *interior* nodes, x_1 and x_{N-1}. [Then, x_1 and x_{N-1} are no longer knots!] These conditions are expressed mathematically as the Not-a-Knot Conditions

$$p_1'''(x_1) = p_2'''(x_1), \;\; p_{N-1}'''(x_{N-1}) = p_N'''(x_{N-1}).$$

By construction, the first two pieces, $p_1(x)$ and $p_2(x)$, of the cubic spline $S_{3,N}(x)$ agree in value, as well as in first and second derivative at x_1. If $p_1(x)$ and $p_2(x)$ also satisfy the not-a-knot condition at x_1, it follows that $p_1(x) \equiv p_2(x)$; that is, x_1 is no longer a knot. The accuracy of this approach is also $O(h^4)$ but the error may still be quite large near the ends of the interval.

Cubic Spline Accuracy

For each way of supplying the additional linear constraints discussed above, the system of $4N$ linear constraints has a unique solution as long as the knots are distinct. So, the cubic spline interpolating function constructed using any one of the natural, the correct endpoint first or second derivative value, an approximated endpoint first or second derivative value, or the not-a-knot conditions is unique.

This uniqueness result permits an estimate of the error associated with approximations by cubic splines. From the error bound for polynomial interpolation, for a cubic polynomial $p_3(x)$ interpolating at data points in the interval $[a, b]$, we have

$$\max_{x\in[x_{i-1},x_i]} |f(x) - p_3(x)| \leq Ch^4 \cdot \max_{x\in[a,b]} |f^{(4)}(x)|$$

where C is a constant and $h = \max_i |x_i - x_{i-1}|$. We might anticipate that the error associated with approximation by a cubic spline behave like h^4 for h small, as for an interpolating cubic polynomial. However, the maximum absolute error associated with the natural cubic spline approximation behaves like h^2 as $h \to 0$. In contrast, the maximum absolute error for a cubic spline based on correct endpoint first or second derivative values or on the not-a-knot conditions behaves like h^4. Unlike the natural cubic spline, the correct first and second derivative value and not-a-knot cubic splines reproduce cubic polynomials. That is, in both these cases, $S_{3,N} \equiv f$ on the interval $[a, b]$ whenever the data values are measured from a cubic polynomial f. This reproducibility property is a necessary condition for the error in the cubic spline $S_{3,N}$ approximation to a general function f to behave like h^4.

B–splines

Codes that work with cubic splines do not use the power series representation of $S_{3,N}(x)$. Rather, often they represent the spline as a linear combination of cubic **B–splines**; this approach is similar to using a linear combination of the linear B–spline roof basis functions L_i to represent a linear spline. B–splines have **compact support**, that is they are non-zero only inside a set of contiguous subintervals just like the linear spline roof basis functions. So, the linear B–spline basis function, L_i, has support (is non-zero) over just the two contiguous intervals which combined make up the interval $[x_{i-1}, x_{i+1}]$, whereas the corresponding cubic B–spline basis function, B_i, has support (is non-zero) over four contiguous intervals which combined make up the interval $[x_{i-2}, x_{i+2}]$.

Construction of a B–spline

Assume the points x_i are equally spaced with spacing h. We'll construct $B_p(x)$ the cubic B–spline centered on x_p. We know already that $B_p(x)$ is a cubic spline that is identically zero outside the interval $[x_p-2h, x_p+2h]$ and has knots at x_p-2h, x_p-h, x_p, x_p+h, and x_p+2h. We'll normalize it at x_p by requiring $B_p(x_p) = 1$. So on the interval $[x_p-2h, x_p-h]$ we can choose $B_p(x) = A(x-[x_p-2h])^3$ where A is a constant to be determined later. This is continuous and has continuous first and second derivatives matching the zero function at the knot $x_p - 2h$. Similarly on the interval $[x_p+h, x_p+2h]$ we can choose $B_p(x) = -A(x - [x_p + 2h])^3$ where A will turn out to be the same constant by symmetry. Now, we need $B_p(x)$ to be continuous and have continuous first and second derivatives at the knot $x_p - h$. This is achieved by choosing $B_p(x) = A(x - [x_p - 2h])^3 + B(x - [x_p - h])^3$ on the interval $[x_p - h, x_p]$ and similarly $B_p(x) = -A(x - [x_p + 2h])^3 - B(x - [x_p + h])^3$ on the interval $[x_p, x_p + h]$ where again symmetry ensures the same constants. Now, all we need to do is to fix up the constants A and B to give the required properties at the knot $x = x_p$. Continuity and the requirement $B_p(x_p) = 1$ give

$$A(x_p - [x_p - 2h])^3 + B(x_p - [x_p - h])^3 = -A(x_p - [x_p + 2h])^3 - B(x_p - [x_p + h])^3 = 1$$

that is

$$8h^3A + h^3B = -8(-h)^3A - (-h)^3B = 1$$

which gives one equation, $8A + B = \frac{1}{h^3}$, for the two constants A and B. Now first derivative continuity at the knot $x = x_p$ gives

$$3A(x_p - [x_p - 2h])^2 + 3B(x_p - [x_p - h])^2 = -3A(x_p - [x_p + 2h])^2 - 3B(x_p - [x_p + h])^2$$

After cancelation, this reduces to $4A + B = -4A - B$. The second derivative continuity condition gives an automatically satisfied identity. So, solving we have $B = -4A$. Hence $A = \frac{1}{4h^3}$ and

$B = -\frac{1}{h^3}$. So,

$$B_p(x) = \begin{cases} 0, & x < x_p - 2h \\ \frac{1}{4h^3}(x - [x_p - 2h])^3, & x_p - 2h \le x < x_p - h \\ \frac{1}{4h^3}(x - [x_p - 2h])^3 - \frac{1}{h^3}(x - [x_p - h])^3, & x_p - h \le x < x_p \\ -\frac{1}{4h^3}(x - [x_p + 2h])^3 + \frac{1}{h^3}(x - [x_p + h])^3, & x_p \le x < x_p + h \\ -\frac{1}{4h^3}(x - [x_p + 2h])^3, & x_p + h \le x < x_p + 2h \\ 0, & x \ge x_p + 2h \end{cases}$$

Interpolation using Cubic B–splines

Suppose we have data $\{(x_i, f_i)\}_{i=0}^n$ and the points x_i are equally spaced so that $x_i = x_0 + ih$. Define the "exterior" equally spaced points x_{-i}, x_{n+i}, $i = 1, 2, 3$ then these are all the points we need to define the B–splines $B_i(x)$, $i = -1, 0, \ldots, n+1$. This is the B–spline basis; that is, the set of all B–splines which are nonzero in the interval $[x_0, x_n]$. We seek a B–spline interpolating function of the form $S_n(x) = \sum_{i=-1}^{n+1} a_i B_i(x)$. The interpolation conditions give

$$\sum_{i=-1}^{n+1} a_i B_i(x_j) = f_j, \ j = 0, 1, \ldots, n$$

which simplifies to

$$a_{j-1}B_{j-1}(x_j) + a_j B_j(x_j) + a_{j+1}B_{j+1}(x_j) = f_j, \ j = 0, 1, \ldots, n$$

as all other terms in the sum are zero at $x = x_j$. Now, by definition $B_j(x_j) = 1$, and we compute $B_{j-1}(x_j) = B_{j+1}(x_j) = \frac{1}{4}$ by evaluating the above expression for the B–spline, giving the equations

$$\begin{aligned} \tfrac{1}{4}a_{-1} + a_0 + \tfrac{1}{4}a_1 &= f_0 \\ \tfrac{1}{4}a_0 + a_1 + \tfrac{1}{4}a_2 &= f_1 \\ &\vdots \\ \tfrac{1}{4}a_{n-1} + a_n + \tfrac{1}{4}a_{n+1} &= f_n \end{aligned}$$

These are $n+1$ equations in the $n+3$ unknowns a_j, $j = -1, 0, \ldots, n+1$. The additional equations come from applying the boundary conditions. For example, if we apply the natural spline conditions $S_n''(x_0) = S_n''(x_n) = 0$ we get the two additional equations

$$\begin{aligned} \tfrac{3}{2h^2}a_{-1} - \tfrac{3}{h^2}a_0 + \tfrac{3}{2h^2}a_1 &= 0 \\ \tfrac{3}{2h^2}a_{n-1} - \tfrac{3}{h^2}a_n + \tfrac{3}{2h^2}a_{n+1} &= 0 \end{aligned}$$

The full set of $n+3$ linear equations may be solved by Gaussian elimination but we can simplify the equations. Taking the first of the additional equations and the first of the previous set together we get $a_0 = \frac{2}{3}f_0$; similarly, from the last equations we find $a_n = \frac{2}{3}f_n$. So the set of linear equations reduces to

$$\begin{aligned} a_1 + \tfrac{1}{4}a_2 &= f_1 - \tfrac{1}{6}f_0 \\ \tfrac{1}{4}a_1 + a_2 + \tfrac{1}{4}a_3 &= f_2 \\ \tfrac{1}{4}a_2 + a_3 + \tfrac{1}{4}a_4 &= f_3 \\ &\vdots \\ \tfrac{1}{4}a_{n-3} + a_{n-2} + \tfrac{1}{4}a_{n-1} &= f_{n-2} \\ \tfrac{1}{4}a_{n-2} + a_{n-1} &= f_{n-1} - \tfrac{1}{6}f_n \end{aligned}$$

The coefficient matrix of this linear system is

$$\begin{bmatrix} 1 & \frac{1}{4} & & & \\ \frac{1}{4} & 1 & \frac{1}{4} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{1}{4} & 1 & \frac{1}{4} \\ & & & \frac{1}{4} & 1 \end{bmatrix}$$

Matrices of this structure are called tridiagonal and this particular tridiagonal matrix is of a special type known as positive definite. For this type of matrix interchanges are not needed for the stability of Gaussian elimination. When interchanges are not needed for a tridiagonal system, Gaussian elimination reduces to a particularly simple algorithm. After we have solved the linear equations for $a_1, a_2, \ldots, a_{n-1}$ we can compute the values of a_{-1} and a_{n+1} from the "additional" equations above.

Problem 4.3.3. *Let $r(x) = r_0 + r_1x + r_2x^2 + r_3x^3$ and $s(x) = s_0 + s_1x + s_2x^2 + s_3x^3$ be cubic polynomials in x. Suppose that the value, first, second, and third derivatives of $r(x)$ and $s(x)$ agree at some point $x = a$. Show that $r_0 = s_0$, $r_1 = s_1$, $r_2 = s_2$, and $r_3 = s_3$, i.e., $r(x)$ and $s(x)$ are the same cubic. [Note: This is another form of the polynomial uniqueness theorem.]*

Problem 4.3.4. *Write down the equations determining the coefficients of the not-a-knot cubic spline interpolating to the data $(0, 1)$, $(1, 0)$, $(2, 3)$ and $(3, 2)$. Just four equations are sufficient. Why?*

Problem 4.3.5. *Let the knots be at the integers, i.e. $x_i = i$, so $B_0(x)$ has support on the interval $[-2, +2]$. Construct $B_0(x)$ so that it is a cubic spline normalized so that $B_0(0) = 1$. [Hint: Since $B_0(x)$ is a cubic spline it must be piecewise cubic and it must be continuous and have continuous first and second derivatives at all the knots, and particularly at the knots $-2, -1, 0, +1, +2$.]*

Problem 4.3.6. *In the derivation of the linear system for B–spline interpolation replace the equations corresponding to the natural boundary conditions by equations corresponding to (a) exact second derivative conditions and (b) knot-a-knot conditions. In both cases use these equations to eliminate the coefficients a_{-1} and a_{n+1} and write down the structure of the resulting linear system.*

Problem 4.3.7. *Is the following function $S(x)$ a cubic spline? Why or why not?*

$$S(x) = \begin{cases} 0, & x < 0 \\ x^3, & 0 \le x < 1 \\ x^3 + (x-1)^3, & 1 \le x < 2 \\ -(x-3)^3 - (x-4)^3, & 2 \le x < 3 \\ -(x-4)^3, & 3 \le x < 4 \\ 0, & 4 \le x \end{cases}$$

4.3.3 Monotonic Piecewise Cubic Polynomials

Another approach to interpolation with piecewise cubic polynomials is, in addition to interpolating the data, to attempt to preserve a qualitative property exhibited by the data. For example, the piecewise cubic polynomial could be chosen in such a way that it is monotonic in the intervals between successive monotonic data values. In the simplest case where the data values are nondecreasing (or non-increasing) throughout the interval then the fitted function is forced to have the same property at all points in the interval. The *piecewise cubic Hermite interpolating polynomial* (PCHIP) achieves

monotonicity by choosing the values of the derivative at the data points so that the function is non-decreasing, that is with derivative nonnegative (or is non-increasing, that is derivative non-positive) throughout the interval. The resulting piecewise cubic polynomial will usually have discontinuous second derivatives at the data points and will hence be less smooth and a little less accurate for smooth functions than the cubic spline interpolating the same data. However, forcing monotonicity when it is present in the data has the effect of preventing oscillations and overshoot which can occur with a spline. For an illustration of this, see Examples 4.6.10 and 4.6.11 in Section 4.6.3.

It would be wrong, though, to think that because we don't enforce monotonicity directly when constructing a cubic spline that we don't get it in practice. When the data points are sufficiently close (that is, h is sufficiently small) and the function represented is smooth, the cubic spline is a very accurate approximation to the function that the data represents and any properties, such as monotonicity, are preserved.

4.4 Least Squares Fitting

In previous sections we determined an approximation of $f(x)$ by interpolating to the data $\{(x_i, f_i)\}_{i=0}^{N}$. An alternative to approximation via interpolation is approximation via a least squares fit.

Let $q_M(x)$ be a polynomial of degree M. Observe that $q_M(x_r) - f_r$ is the error in accepting $q_M(x_r)$ as an approximation to f_r. So, the sum of the squares of these errors

$$\sigma(q_M) \equiv \sum_{r=0}^{N} \{q_M(x_r) - f_r\}^2$$

gives a measure of how well $q_M(x)$ fits $f(x)$. The idea is that the smaller the value of $\sigma(q_M)$, the closer the polynomial $q_M(x)$ fits the data.

We say $p_M(x)$ is a least squares polynomial of degree M if $p_M(x)$ is a polynomial of degree M with the property that

$$\sigma(p_M) \leq \sigma(q_M)$$

for all polynomials $q_M(x)$ of degree M; usually we only have equality if $q_M(x) \equiv p_M(x)$. For simplicity, we write σ_M in place of $\sigma(p_M)$. As shown in an advanced course in numerical analysis, if the points $\{x_r\}$ are distinct and if $N \geq M$ there is one and only one least squares polynomial of degree M for this data, so we say $p_M(x)$ is the least squares polynomial of degree M. So, the polynomial $p_M(x)$ that produces the smallest value σ_M yields the least squares fit of the data.

While $p_M(x)$ produces the closest fit of the data in the least squares sense, it may not produce a very useful fit. For example, consider the case $M = N$ then the least squares fit $p_N(x)$ is the same as the interpolating polynomial; see Problem 4.4.1. We have seen already that the interpolating polynomial can be a poor fit in the sense of having a large and highly oscillatory error. So, a close fit in a least squares sense does not necessarily imply a very good fit and, in some cases, the closer the fit is to an interpolating function the less useful it might be.

Since the least squares criterion relaxes the fitting condition from interpolation to a weaker condition on the coefficients of the polynomial, we need fewer coefficients (that is, a lower degree polynomial) in the representation. For the problem to be well–posed, it is sufficient that all the data points be distinct and that $M \leq N$.

Example 4.4.1. How is $p_M(x)$ determined for polynomials of each degree M? Consider the example:

- data:

i	x_i	f_i
0	1	2
1	3	4
2	4	3
3	5	1

- least squares fit to this data by a straight line: $p_1(x) = a_0 + a_1 x$; that is, the coefficients a_0 and a_1 are to be determined.

We have

$$\sigma_1 = \sum_{r=0}^{3} \{p_1(x_r) - f_r\}^2 = \sum_{r=0}^{3} \{a_0 + a_1 x_r - f_r\}^2$$

Multivariate calculus provides a technique to identify the values of a_0 and a_1 that make σ_1 smallest; for a minimum of σ_1, the unknowns a_0 and a_1 must satisfy the linear equations

$$\begin{aligned}
\frac{\partial}{\partial a_0}\sigma_1 &\equiv \sum_{r=0}^{3} 2\{a_0 + a_1 x_r - f_r\} &= 2(4a_0 + 13a_1 - 10) &= 0 \\
\frac{\partial}{\partial a_1}\sigma_1 &\equiv \sum_{r=0}^{3} 2x_r\{a_0 + a_1 x_r - f_r\} &= 2(13a_0 + 51a_1 - 31) &= 0
\end{aligned}$$

that is, in matrix form after canceling the 2's throughout,

$$\begin{bmatrix} 4 & 13 \\ 13 & 51 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 10 \\ 31 \end{bmatrix}$$

Gaussian elimination followed by backward substitution computes the solution

$$a_0 = \frac{107}{35} \simeq 3.057$$

$$a_1 = -\frac{6}{35} \simeq -0.171$$

Substituting a_0 and a_1 gives the minimum value of $\sigma_1 = \dfrac{166}{35}$.

Generally, if we consider fitting data using a polynomial written in power series form

$$p_M(x) = a_0 + a_1 x + \cdots + a_M x^M$$

then σ_M is quadratic in the unknown coefficients $a_0, a_1, \cdots, a_M$. For data $\{(x_i, f_i)\}_{i=0}^{N}$ we have

$$\begin{aligned}
\sigma_M &= \sum_{r=0}^{N} \{p_M(x_r) - f_r\}^2 \\
&= \{p_M(x_0) - f_0\}^2 + \{p_M(x_1) - f_1\}^2 + \cdots + \{p_M(x_N) - f_N\}^2
\end{aligned}$$

The coefficients $a_0, a_1, \cdots, a_M$ are determined by solving the linear system

$$\begin{aligned}
\frac{\partial}{\partial a_0}\sigma_M &= 0 \\
\frac{\partial}{\partial a_1}\sigma_M &= 0 \\
&\vdots \\
\frac{\partial}{\partial a_M}\sigma_M &= 0
\end{aligned}$$

For each value $j = 0, 1, \cdots, M$, the linear equation $\dfrac{\partial}{\partial a_j}\sigma_M = 0$ is formed as follows. Observe that

$\dfrac{\partial}{\partial a_j} p_M(x_r) = x_r^j$ so, by the chain rule,

$$\begin{aligned}
\frac{\partial}{\partial a_j}\sigma_M &= \frac{\partial}{\partial a_j}\left[\{f_0 - p_M(x_0)\}^2 + \{f_1 - p_M(x_1)\}^2 + \cdots + \{f_N - p_M(x_N)\}^2\right] \\
&= 2\left[\{p_M(x_0) - f_0\}\frac{\partial p_M(x_0)}{\partial a_j} + \{p_M(x_1) - f_1\}\frac{\partial p_M(x_1)}{\partial a_j} + \cdots + \{p_M(x_N) - f_N\}\frac{\partial p_M(x_N)}{\partial a_j}\right] \\
&= 2\left[\{p_M(x_0) - f_0\}x_0^j + \{p_M(x_1) - f_1\}x_1^j + \cdots + \{p_M(x_N) - f_N\}x_N^j\right] \\
&= 2\sum_{r=0}^{N}\{p_M(x_r) - f_r\}x_r^j \\
&= 2\sum_{r=0}^{N} x_r^j\, p_M(x_r) - 2\sum_{r=0}^{N} f_r\, x_r^j.
\end{aligned}$$

Substituting for the polynomial $p_M(x_r)$ the power series form leads to

$$\begin{aligned}
\frac{\partial}{\partial a_j}\sigma_M &= 2\left(\textstyle\sum_{r=0}^{N} x_r^j\left\{a_0 + a_1 x_r + \cdots + a_M x_r^M\right\} - \sum_{r=0}^{N} f_r\, x_r^j\right) \\
&= 2\left(a_0\textstyle\sum_{r=0}^{N} x_r^j + a_1\sum_{r=0}^{N} x_r^{j+1} + \cdots + a_M\sum_{r=0}^{N} x_r^{j+M} - \sum_{r=0}^{N} f_r\, x_r^j\right)
\end{aligned}$$

Therefore, $\dfrac{\partial}{\partial a_j}\sigma_M = 0$ may be rewritten as the **Normal equations**:

$$a_0\sum_{r=0}^{N} x_r^j + a_1\sum_{r=0}^{N} x_r^{j+1} + \cdots + a_M\sum_{r=0}^{N} x_r^{j+M} = \sum_{r=0}^{N} f_r\, x_r^j, \quad j = 0, 1, \cdots, M$$

In matrix form the Normal equations may be written

$$\begin{bmatrix} \sum_{r=0}^{N} 1 & \sum_{r=0}^{N} x_r & \cdots & \sum_{r=0}^{N} x_r^M \\ \sum_{r=0}^{N} x_r & \sum_{r=0}^{N} x_r^2 & \cdots & \sum_{r=0}^{N} x_r^{M+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{r=0}^{N} x_r^M & \sum_{r=0}^{N} x_r^{M+1} & \cdots & \sum_{r=0}^{N} x_r^{2M} \end{bmatrix}\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{bmatrix} = \begin{bmatrix} \sum_{r=0}^{N} f_r \\ \sum_{r=0}^{N} f_r x_r \\ \vdots \\ \sum_{r=0}^{N} f_r x_r^M \end{bmatrix}$$

The coefficient matrix of the Normal equations has special properties (it is both symmetric and positive definite). These properties permit the use of an accurate, efficient version of Gaussian elimination which exploits these properties, without the need for partial pivoting by rows for size.

Example 4.4.2. To compute a straight line fit $a_0 + a_1 x$ to the data $\{(x_i, f_i)\}_{i=0}^{N}$ we set $M = 1$ in the Normal equations to give

$$\begin{aligned}
a_0\sum_{r=0}^{N} 1 + a_1\sum_{r=0}^{N} x_r &= \sum_{r=0}^{N} f_r \\
a_0\sum_{r=0}^{N} x_r + a_1\sum_{r=0}^{N} x_r^2 &= \sum_{r=0}^{N} f_r x_r
\end{aligned}$$

Substituting the data

i	x_i	f_i
0	1	2
1	3	4
2	4	3
3	5	1

from Example 4.4.1 we have the Normal equations

$$\begin{aligned} 4a_0 + 13a_1 &= 10 \\ 13a_0 + 51a_1 &= 31 \end{aligned}$$

which gives the same result as in Example 4.4.1.

Example 4.4.3. To compute a quadratic fit $a_0 + a_1 x + a_2 x^2$ to the data $\{(x_i, f_i)\}_{i=0}^{N}$ we set $M = 2$ in the Normal equations to give

$$\begin{aligned} a_0 \sum_{r=0}^{N} 1 + a_1 \sum_{r=0}^{N} x_r + a_2 \sum_{r=0}^{N} x_r^2 &= \sum_{r=0}^{N} f_r \\ a_0 \sum_{r=0}^{N} x_r + a_1 \sum_{r=0}^{N} x_r^2 + a_2 \sum_{r=0}^{N} x_r^3 &= \sum_{r=0}^{N} f_r x_r \\ a_0 \sum_{r=0}^{N} x_r^2 + a_1 \sum_{r=0}^{N} x_r^3 + a_2 \sum_{r=0}^{N} x_r^4 &= \sum_{r=0}^{N} f_r x_r^2 \end{aligned}$$

The least squares formulation permits more general functions $f_M(x)$ than simply polynomials, but the unknown coefficients in $f_M(x)$ must still occur linearly. The most general form is

$$f_M(x) = a_0\phi_0(x) + a_1\phi_1(x) + \cdots + a_M\phi_M(x) = \sum_{r=0}^{M} a_r\phi_r(x)$$

with a linearly independent basis $\{\phi_r(x)\}_{r=0}^{M}$. By analogy with the power series case, the linear system of Normal equations is

$$a_0 \sum_{r=0}^{N} \phi_0(x_r)\phi_j(x_r) + a_1 \sum_{r=0}^{N} \phi_1(x_r)\phi_j(x_r) + \cdots + a_M \sum_{r=0}^{N} \phi_M(x_r)\phi_j(x_r) = \sum_{r=0}^{N} f_r\phi_j(x_r)$$

for $j = 0, 1, \cdots, M$. The Normal equations are

$$\begin{bmatrix} \sum_{r=0}^{N} \phi_0(x_r)^2 & \sum_{r=0}^{N} \phi_0(x_r)\phi_1(x_r) & \cdots & \sum_{r=0}^{N} \phi_0(x_r)\phi_M(x_r) \\ \sum_{r=0}^{N} \phi_1(x_r)\phi_0(x_r) & \sum_{r=0}^{N} \phi_1(x_r)^2 & \cdots & \sum_{r=0}^{N} \phi_1(x_r)\phi_M(x_r) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{r=0}^{N} \phi_M(x_r)\phi_0(x_r) & \sum_{r=0}^{N} \phi_M(x_r)\phi_1(x_r) & \cdots & \sum_{r=0}^{N} \phi_M(x_r)^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{bmatrix} = \begin{bmatrix} \sum_{r=0}^{N} f_r\phi_0(x_r) \\ \sum_{r=0}^{N} f_r\phi_1(x_r) \\ \vdots \\ \sum_{r=0}^{N} f_r\phi_M(x_r) \end{bmatrix}$$

The coefficient matrix of this linear system is symmetric and potentially ill–conditioned. In particular, the basis functions ϕ_j could be, for example, a linear polynomial spline basis, a cubic polynomial B–spline basis, Chebyshev polynomials in a Chebyshev series fit, or a set of linearly independent trigonometric functions.

The coefficient matrix of the Normal equations is usually reasonably well-conditioned when the number, M, of functions being fitted is small. The ill–conditioning grows with the number of functions. To avoid the possibility of ill–conditioning the Normal equations are not usually formed but instead a stable QR factorization of a related matrix is employed to compute the least squares

solution directly (the approach taken by MATLAB's `polyfit` function), which is discussed in the next section.

Problem 4.4.1. *Consider the data $\{(x_i, f_i)\}_{i=0}^{N}$. Argue why the interpolating polynomial of degree N is also the least squares polynomial of degree N. Hint: What is the value of $\sigma(q_N)$ when $q_N(x)$ is the interpolating polynomial?*

Problem 4.4.2. *Show that $\sigma_0 \geq \sigma_1 \geq \cdots \geq \sigma_N = 0$ for any set of data $\{(x_i, f_i)\}_{i=0}^{N}$. Hint: The proof follows from the concept of minimization.*

Problem 4.4.3. *Find the least squares constant fit $p_0(x) = a_0$ to the data in Example 4.4.1. Plot the data, and both the constant and the linear least squares fits on one graph.*

Problem 4.4.4. *Find the least squares linear fit $p_1(x) = a_0 + a_1 x$ to the following data. Explain why you believe your answer is correct?*

i	x_i	f_i
0	1	1
1	3	1
2	4	1
3	5	1
4	7	1

Problem 4.4.5. *Find the least squares quadratic polynomial fits $p_2(x) = a_0 + a_1 x + a_2 x^2$ to each of the data sets:*

i	x_i	f_i
0	-2	6
1	-1	3
2	0	1
3	1	3
4	2	6

and

i	x_i	f_i
0	-2	-5
1	-1	-3
2	0	0
3	1	3
4	2	5

Problem 4.4.6. *Use the chain rule to derive the Normal equations for the general basis functions $\{\phi_j\}_{j=0}^{M}$.*

Problem 4.4.7. *Write down the Normal equations for the following choice of basis functions: $\phi_0(x) = 1$, $\phi_1(x) = \sin(\pi x)$ and $\phi_2(x) = \cos(\pi x)$. Find the coefficients a_0, a_1 and a_2 for a least squares fit to the data*

i	x_i	f_i
0	-1	-5
1	-0.5	-3
2	0	0
3	0.5	3
4	1	5

Problem 4.4.8. *Write down the Normal equations for the following choice of basis functions: $\phi_0(x) = T_0(x)$, $\phi_1(x) = T_1(x)$ and $\phi_2(x) = T_2(x)$. Find the coefficients a_0, a_1 and a_2 for a least squares fit to the data in the Problem 4.4.7.*

4.5 Least Squares and Matrix Factorizations

The previous section described how to use calculus to derive the normal equations for a least squares data fitting problem. In this section, we connect the process to the language of linear algebra. To begin, we start from the basic polynomial least squares data fitting problem:

- Given data ponts $\{(x_i, f_i)\}_{i=0}^N$,
- find a polynomial $p_M(x)$ of degree $M \leq N$ so that $p_M(x_i) \approx f_i$.

The ideal situation would be that the polynomial exactly fits all of the data; that is,

$$p_M(x_i) = f_i\,, \quad i = 0, 1, 2, \ldots, N\,. \tag{4.3}$$

If we use the standard monomial basis for $p_M(x)$,

$$p_M(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_M x^M\,,$$

then the equations (4.3) can be written in matrix-vector form as $Va = f$:

$$\underbrace{\begin{bmatrix} 1 & x_0 & \cdots & x_0^M \\ 1 & x_1 & \cdots & x_1^M \\ 1 & x_2 & \cdots & x_2^M \\ \vdots & \vdots & & \vdots \\ 1 & x_N & \cdots & x_N^M \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix}}_{f}\,. \tag{4.4}$$

If $M = N$, then we have the polynomial interpolation problem, where we saw that if the x_i are distinct, then the matrix V is square and nonsingular, and therefore there is a unique solution to the linear system.

However, if $M < N$, then there are more rows than columns in the matrix, and we should recall from linear algebra that in this case, it is very unlikely that there is a vector a that exactly solves $Va = f$. However, we can consider trying to find a vector a such that

$$Va \approx f\,.$$

We need to define what "approximate" means in the language of linear algebra. A natural way to define approximation is to use norms, and to consider finding a vector a that minimizes the norm of the residual vector; that is, we consider

$$\min_a \|Va - f\|$$

where $\|\cdot\|$ is a specified vector norm. As we have seen in section (3.6.1), there are many vector norms we could use, such as $\|\cdot\|_1$, $\|\cdot\|_2$, $\|\cdot\|_\infty$, etc.. The choice of norm will (very likely) give different approximation vectors a, and the choice of norm will have an effect on the computational cost. The most commonly used (and computationally easiest) is to use $\|\cdot\|_2$. Thus, we consider

$$\min_a \|Va - f\|_2 \quad \text{or, equivalently} \quad \min_a \|Va - f\|_2^2\,. \tag{4.5}$$

Note that for the polynomial least squares data fitting problem, it is not difficult to show that

$$\|Va - f\|_2^2 = \sum_{i=0}^{N} \left(p_M(x_i) - f_i\right)^2\,,$$

and thus the polynomial least squares data fitting problem is equivalent to the linear algebra approximation problem (4.5). We will refer to (4.5) as a linear algebra least squares problem.

What is even better is that this linear algebra formulation can be used for any general function $g(x)$, provided that $g(x)$ can be written as a linear combination of $M+1$ given basis functions. That is, consider the problem:

- Given data ponts $\{(x_i, f_i)\}_{i=0}^{N}$, a set of basis functions $\{\phi_0(x), \phi_1(x), \ldots, \phi_M(x)\}$, and

$$g(x) = a_0\phi_0(x) + a_1\phi_1(x) + \cdots + a_M\phi_M(x),$$

- find coefficients $a_0, a_1, \ldots, a_M$ so that $g(x_i) \approx f_i$.

To find the best a_0, a_1, ..., a_M using the least squares data fitting approach, we again consider the ideal situation where we can exactly fit the data; that is,

$$g(x_i) = f_i\,, \quad i = 0, 1, 2, \ldots, N\,.$$

These equations can be written in matrix-vector form as $Wa = f$:

$$\underbrace{\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_M(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_M(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_M(x_2) \\ \vdots & \vdots & & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_M(x_N) \end{bmatrix}}_{W} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix}}_{f}.$$

Thus, to find the best least squares fit of the data to the function $g(x)$, we need only solve the linear algebra least squares problem

$$\min_a \|Wa - f\|_2\,.$$

Example 4.5.1. Suppose we are given the basis functions,

$$\phi_0(x) = 1, \quad \phi_1(x) = \sin(\pi x), \quad \phi_2(x) = \cos(\pi x),$$

and suppose we want to find the best least squares fit of the function

$$g(x) = a_0\phi_0(x) + a_1\phi_1(x) + a_2\phi_2(x)$$

to the given the data:

i	x_i	f_i
0	-1	-5
1	-0.5	-3
2	0	0
3	0.5	3
4	1	5

To set up the equivalent linear algebra least squares problem, we start by considering the ideal case where we can get equality for all data:

$$\begin{aligned} g(x_i) = f_i \quad &\Rightarrow \quad a_0\phi_0(x_i) + a_1\phi_1(x_i) + a_2\phi_2(x_i) = f_i \\ &\Rightarrow \quad a_0 + a_1\sin(\pi x_i) + a_2\cos(\pi x_i) = f_i\,. \end{aligned}$$

That is,

$$\left.\begin{aligned} g(-1) &= -5 \\ g(-0.5) &= -3 \\ g(0) &= 0 \\ g(0.5) &= 3 \\ g(1) &= 5 \end{aligned}\right\} \Rightarrow \left\{\begin{aligned} a_0 + a_1\sin(-\pi) + a_2\cos(-\pi) &= -5 \\ a_0 + a_1\sin(-\pi/2) + a_2\cos(-\pi/2) &= -3 \\ a_0 + a_1\sin(0) + a_2\cos(0) &= 0 \\ a_0 + a_1\sin(\pi/2) + a_2\cos(\pi/2) &= 3 \\ a_0 + a_1\sin(\pi) + a_2\cos(\pi) &= 5 \end{aligned}\right.$$

or equivalently in matrix-vector form, $Wa = f$:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & -1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} -5 \\ -3 \\ 0 \\ 3 \\ 5 \end{bmatrix}.$$

Thus, to find a_0, a_1, a_2, we need to solve the linear algebra least squares problem

$$\min_a \|Wa - f\|_2 .$$

There are many ways to solve linear algebra least squares problems, which are discussed in the following subsections.

4.5.1 Linear Algebra Normal Equations

In the previous section we used calculus to derive the normal equations for polynomial and general function least squares data fitting. In this subsection we describe how the normal equations can be found directly from the linear algebra least squares formulation.

To illustrate, consider the polynomial least squares data fitting problem, which we saw has the equivalent linear algebra least squares formulation

$$\min_a \|Va - f\|_2 ,$$

where

$$V = \begin{bmatrix} 1 & x_0 & \cdots & x_0^M \\ 1 & x_1 & \cdots & x_1^M \\ 1 & x_2 & \cdots & x_2^M \\ \vdots & \vdots & & \vdots \\ 1 & x_N & \cdots & x_N^M \end{bmatrix}, \quad a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{bmatrix}, \quad f = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix}.$$

Using the above defined matrix and vectors, it is not difficult to verify (we leave it as an exercise) that the normal equations can be obtained by the simple linear algebra computation

$$V^T V a = V^T f . \tag{4.6}$$

In fact, this simple approach can be used to construct the normal equations for any least squares problem (not just those associated with polynomial data fitting).

It is important to note some properties about the matrix $V^T V$. If V is an $n \times m$ matrix (where, for example, $n = N + 1$ and $m = M + 1$ for polynomial least squares problems), then

- $V^T V$ is a square $m \times m$ matrix,
- if the columns of V are linearly independent (i.e. V has full column rank), then
  - $V^T V$ is nonsingular, and also
  - $V^T V$ is symmetric and positive definite (SPD).

Thus, if V has full column rank, there is always a unique solution to the least squares problem. Moreover, because $V^T V$ is SPD, we can use the Cholesky factorization to efficiently solve (4.6).

4.5.2 Least Squares and QR Factorization

While the normal equations formulation is a convenient way to solve least squares problems (especially when doing hand calculations for small problems), it is not the best approach when V is ill-conditioned. Instead, least squares problems are generally solved using the QR factorization. Recall that if $V \in \mathcal{R}^{n\times m}$, $n > m$, and $\text{rank}(V) = m$, then we can compute the factorization

$$V = QR$$

where $Q \in \mathcal{R}^{n\times n}$ is an orthogonal matrix (that is, $Q^TQ = I$) and $R \in \mathcal{R}^{n\times m}$ is an upper triangular matrix. Also recall that if $Q \in \mathcal{R}^{n\times n}$ is an orthogonal matrix and $r \in \mathcal{R}^n$ then

$$\|Q^T r\|_2 = \|r\|_2 .$$

Using this property in the linear algebra least squares problem, observe that

$$\begin{aligned}
\|Va - f\|_2^2 &= \|QRa - f\|_2^2 \\
&= \|Q^T(QRa - f)\|_2^2 \\
&= \|Q^TQRa - Q^Tf\|_2^2 \\
&= \|Ra - Q^Tf\|_2^2 \\
&= \left\| \begin{bmatrix} R_m \\ 0 \end{bmatrix} a - \begin{bmatrix} b \\ c \end{bmatrix} \right\|_2^2 \\
&= \|R_m a - b\|_2^2 + \|c\|_2^2 ,
\end{aligned}$$

where we have used the notation

- $R = \begin{bmatrix} R_m \\ 0 \end{bmatrix}$, $R_m \in \mathcal{R}^{m\times m}$ is upper triangular,
- b is a vector containing the first m entries of the vector Q^Tf,
- c is a vector containing the bottom $n - m$ entries of the vector A^Tf.

Thus, in order to solve

$$\min_a \|Va - f\|_2$$

we need only solve

$$\min_a \|R_m a - b\|_2 .$$

Since R_m is a square, upper triangular matrix, this latter minimization problem can easily be done by using backward substitution to compute the unique solution of

$$R_m a = b .$$

The singular value decomposition (SVD) can also be used to solve linear algebra least squares problem; the approach is similar to that done with the QR factorization, but because the SVD is much more expensive to compute, the QR factorization is the preferred approach in most applications.

4.6 Matlab Notes

MATLAB has several functions designed specifically for manipulating polynomials, and for curve fitting. Some functions that are relevant to the topics discussed in this chapter include:

`polyfit`	used to create the coefficients of an interpolating or least squares polynomial
`polyval`	used to evaluate a polynomial
`spline`	used to create, and evaluate, an interpolatory cubic spline
`pchip`	used to create, and evaluate, a piecewise cubic Hermite interpolating polynomial
`interp1`	used to create, and evaluate, a variety of piecewise interpolating polynomials, including linear and cubic splines, and piecewise cubic Hermite polynomials
`ppval`	used to evaluate a piecewise polynomial, such as given by `spline`, `pchip` or `interp1`

In general, these functions assume a canonical representation of the power series form of a polynomial to be

$$p(x) = a_1 x^N + a_2 x^{N-1} + \cdots + a_N x + a_{N+1}.$$

Note that this is slightly different than the notation used in Section 4.1, but in either case, all that is needed to represent the polynomial is a vector of coefficients. Using MATLAB's canonical form, the vector representing $p(x)$ is:

$$a = \left[\begin{array}{cccccc} a_1; & a_2; & \cdots & a_N; & a_{N+1} \end{array} \right].$$

Note that although a could be a row or a column vector, we will generally use column vectors.

Example 4.6.1. Consider the polynomial $p(x) = 7x^3 - x^2 + 1.5x - 3$. Then the vector of coefficients that represents this polynomial is given by:

```
>> a = [7; -1; 1.5; -3];
```

Similarly, the vector of coefficients that represents the polynomial $p(x) = 7x^5 - x^4 + 1.5x^2 - 3x$ is given by:

```
>> a = [7; -1; 0; 1.5; -3; 0];
```

Notice that it is important to explicitly include any zero coefficients when constructing the vector a.

4.6.1 Polynomial Interpolation

In Section 4.1, we constructed interpolating polynomials using three different forms: power series, Newton and Lagrange forms. MATLAB's main tool for polynomial interpolation, `polyfit`, uses the power series form. To understand how this function is implemented, suppose we are given data points

$$(x_1, f_1),\ (x_2, f_2),\ \ldots, (x_{N+1}, f_{N+1}).$$

Recall that to find the power series form of the (degree N) polynomial that interpolates this data, we need to find the coefficients, a_i, of the polynomial

$$p(x) = a_1 x^N + a_2 x^{N-1} + \cdots + a_N x + a_{N+1}$$

such that $p(x_i) = f_i$. That is,

$$\begin{aligned}
p(x_1) = f_1 \quad &\Rightarrow \quad a_1 x_1^N + a_2 x_1^{N-1} + \cdots + a_N x_1 + a_{N+1} = f_1 \\
p(x_2) = f_2 \quad &\Rightarrow \quad a_1 x_2^N + a_2 x_2^{N-1} + \cdots + a_N x_2 + a_{N+1} = f_2 \\
&\vdots \\
p(x_N) = f_N \quad &\Rightarrow \quad a_1 x_N^N + a_2 x_N^{N-1} + \cdots + a_N x_N + a_{N+1} = f_N \\
p(x_{N+1}) = f_{N+1} \quad &\Rightarrow \quad a_1 x_{N+1}^N + a_2 x_{N+1}^{N-1} + \cdots + a_N x_{N+1} + a_{N+1} = f_{N+1}
\end{aligned}$$

or, more precisely, we need to solve the linear system $Va = f$:

$$\begin{bmatrix} x_1^N & x_1^{N-1} & \cdots & x_1 & 1 \\ x_2^N & x_2^{N-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_N^N & x_N^{N-1} & \cdots & x_N & 1 \\ x_{N+1}^N & x_{N+1}^{N-1} & \cdots & x_{N+1} & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \\ a_{N+1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \\ f_{N+1} \end{bmatrix}.$$

In order to write a MATLAB function implementing this approach, we need to:

- Define vectors containing the given data:

  $\texttt{x} = \left[\; x_1; \quad x_2; \quad \cdots \quad x_{N+1} \; \right]$
  $\texttt{f} = \left[\; f_1; \quad f_2; \quad \cdots \quad f_{N+1} \; \right]$

- Let $n = \text{length}(x) = N + 1$.

- Construct the $n \times n$ matrix, V, which can be done one column at a time using the vector `x`:

  ```
  jth column of V = V(:, j) = x .^ (n-j)
  ```

- Solve the linear system, $Va = f$, using MATLAB's backslash operator:

  ```
  a = V \ f
  ```

Putting these steps together, we obtain the following function:

```
function a = InterpPow1(x, f)
%
%      a = InterpPow1(x, f);
%
% Construct the coefficients of a power series representation of the
% polynomial that interpolates the data points (x_i, f_i):
%
%   p = a(1)*x^N + a(2)*x^(N-1) + ... + a(N)*x + a(N+1)
%
n = length(x);
V = zeros(n, n);
for j = 1:n
  V(:, j) = x .^ (n-j);
end
a = V \ f;
```

We remark that MATLAB provides a function, `vander`, that can be used to construct the matrix V from a given vector x. Using `vander` in place of the first five lines of code in `InterpPow1`, we obtain the following function:

```
function a = InterpPow(x, f)
%
%      a = InterpPow(x, f);
%
% Construct the coefficients of a power series representation of the
% polynomial that interpolates the data points (x_i, f_i):
%
%   p = a(1)*x^N + a(2)*x^(N-1) + ... + a(N)*x + a(N+1)
%
V = vander(x);
a = V \ f;
```

Problem 4.6.1. *Implement* `InterpPow`*, and use it to find the power series form of the polynomial that interpolates the data* $(-1, 0)$, $(0, 1)$, $(1, 3)$*. Compare the results with that found in Example 4.1.3.*

Problem 4.6.2. *Implement* `InterpPow`*, and use it to find the power series form of the polynomial that interpolates the data* $(1, 2)$, $(3, 3)$, $(5, 4)$*. Compare the results with what you computed by hand in Problem 4.1.3.*

The built-in MATLAB function, `polyfit`, essentially uses the approach outlined above to construct an interpolating polynomial. The basic usage of `polyfit` is:

```
a = polyfit(x, f, N)
```

where `N` is the degree of the interpolating polynomial. In general, provided the x_i values are distinct, `N = length(x) - 1 = length(f) - 1`. As we see later, `polyfit` can be used for polynomial least squares data fitting by choosing a different (usually smaller) value for `N`.

Once the coefficients are computed, we may want to plot the resulting interpolating polynomial. Recall that to plot any function, including a polynomial, we must first evaluate it at many (e.g., 200) points. MATLAB provides a built-in function, `polyval`, that can be used to evaluate polynomials:

```
y = polyval(a, x);
```

Note that `polyval` requires the first input to be a vector containing the coefficients of the polynomial, and the second input a vector containing the values at which the polynomial is to be evaluated. At this point, though, we should be sure to distinguish between the (relatively few) "data points" used to construct the interpolating polynomial, and the (relatively many) "evaluation points" used for plotting. An example will help to clarify the procedure.

Example 4.6.2. Consider the data points $(-1, 0)$, $(0, 1)$, $(1, 3)$. First, plot the data using (red) circles, and set the axis to an appropriate scale:

```
x_data = [-1 0 1];
f_data = [0 1 3];
plot(x_data, f_data, 'ro')
axis([-2, 2, -1, 6])
```

Now we can construct and plot the polynomial that interpolates this data using the following set of MATLAB statements:

```
hold on
a = polyfit(x_data, f_data, length(x_data)-1);
x = linspace(-2,2,200);
y = polyval(a, x);
plot(x, y)
```

By including labels on the axes, and a legend (see Section 1.3.2):

```
xlabel('x')
ylabel('y')
legend('Data points','Interpolating polynomial', 'Location', 'NW')
```

we obtain the plot shown in Fig. 4.12. Note that we use different vectors to distinguish between the given data (`x_data` and `f_data`) and the set of points `x` and values `y` used to evaluate and plot the resulting interpolating polynomial.

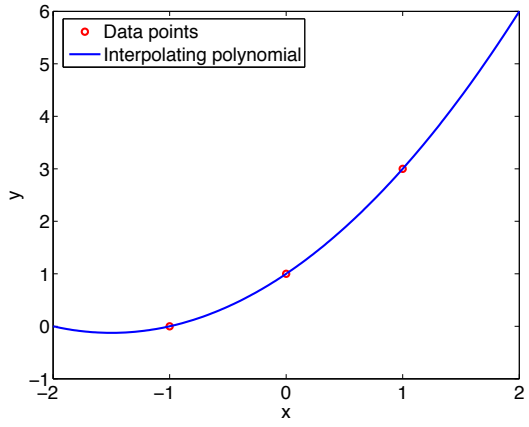

Figure 4.12: Plot generated by the MATLAB code in Example 4.6.2.

Although the power series approach usually works well for a small set of data points, one difficulty that can arise, especially when attempting to interpolate a large set of data, is that the matrix V may be very ill–conditioned. Recall that an ill–conditioned matrix is close to singular, in which case large errors can occur when solving $Va = f$. Thus, if V is ill–conditioned, the computed polynomial coefficients may be inaccurate. MATLAB's `polyfit` function checks V and displays a warning message if it detects it is ill–conditioned. In this case, we can try the alternative calling sequence of `polyfit` and `polyval`:

```
[a, s, mu] = polyfit(x_data, f_data, length(x_data)-1);
y = polyval(a, x, s, mu);
```

This forces `polyfit` to first scale `x_data` (using its mean and standard deviation) before constructing the matrix V and solving the corresponding linear system. This scaling usually results in a matrix that is better-conditioned.

Example 4.6.3. Consider the following set of data, obtained from the National Weather Service, `http://www.srh.noaa.gov/fwd`, which shows average high and low temperatures, total precipitation, and the number of clear days for each month in 2003 for Dallas-Fort Worth, Texas.

Monthly Weather Data, 2003
Dallas - Fort Worth, Texas

Month	1	2	3	4	5	6	7	8	9	10	11	12
Avg. High	54.4	54.6	67.1	78.3	85.3	88.7	96.9	97.6	84.1	80.1	68.8	61.1
Avg. Low	33.0	36.6	45.2	55.6	65.6	69.3	75.7	75.8	64.9	57.4	50.0	38.2
Precip.	0.22	3.07	0.85	1.90	2.53	5.17	0.08	1.85	3.99	0.78	3.15	0.96
Clear Days	15	6	10	11	4	9	13	10	11	13	7	18

Suppose we attempt to fit an interpolating polynomial to the average high temperatures. Our first attempt might use the following set of MATLAB commands:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
a = polyfit(x_data, f_data, 11);
x = linspace(1, 12, 200);
y = polyval(a, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

If we run these commands in MATLAB, then a warning message is printed in the command window indicating that V is ill–conditioned. If we replace the two lines containing `polyfit` and `polyval` with:

```
[a, s, mu] = polyfit(x_data, f_data, 11);
y = polyval(a, x, s, mu);
```

the warning no longer occurs. The resulting plot is shown in Fig. 4.13 (we also made use of the MATLAB commands `axis`, `legend`, `xlabel` and `ylabel`).

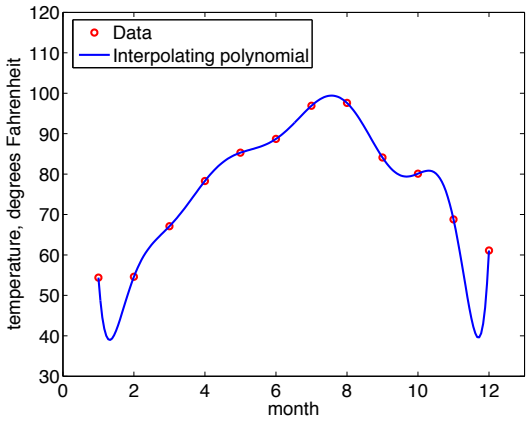

Figure 4.13: Interpolating polynomial from Example 4.6.3

Notice that the polynomial does not appear to provide a good model of the monthly temperature changes between months 1 and 2, and between months 11 and 12. This is a mild example of the more serious problem, discussed in Section 4.2, of excessive oscillation of the interpolating polynomial. A more extreme illustration of this is the following example.

Example 4.6.4. Suppose we wish to construct an interpolating polynomial approximation of the function $f(x) = \sin(x + \sin 2x)$ on the interval $[-\frac{\pi}{2}, \frac{3\pi}{2}]$. The following MATLAB code can be used to construct an interpolating polynomial approximation of $f(x)$ using 11 equally spaced points:

```
f = @(x) sin(x+sin(2*x));
x_data = linspace(-pi/2, 3*pi/2, 11);
f_data = f(x_data);
a = polyfit(x_data, f_data, 10);
```

A plot of the resulting polynomial is shown on the left of Fig. 4.14. Notice that, as with Runge's example (Example 4.2), the interpolating polynomial has severe oscillations near the end points of the interval. However, because we have an explicit function that can be evaluated, instead of using equally spaced points, we can choose to use the Chebyshev points. In MATLAB these points can be generated as follows:

```
c = -pi/2; d = 3*pi/2;
N = 10; I = 0:N;
x_data = (c+d)/2 - (d-c)*cos((2*I+1)*pi/(2*N+2))/2;
```

Notice that `I` is a vector containing the integers $0, 1, \ldots, 9$, and that we avoid using a loop to create `x_data` by making use of MATLAB's ability to operate on vectors. Using this `x_data`, and the corresponding `f_data = f(x_data)` to construct the interpolating polynomial, we can create the plot shown on the right of Fig. 4.14. We observe from this example that much better approximations can be obtained by using the Chebyshev points instead of equally spaced points.

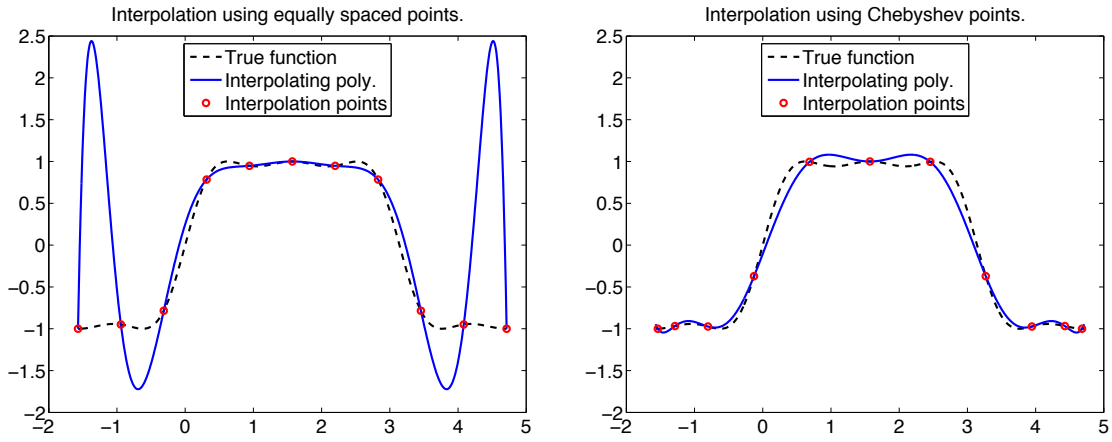

Figure 4.14: Interpolating polynomials for $f(x) = \sin(x + \sin 2x)$ on the interval $[-\frac{\pi}{2}, \frac{3\pi}{2}]$. The plot on the left uses 11 equally spaced points, and the plot on the right uses 11 Chebyshev points to generate the interpolating polynomial.

Problem 4.6.3. *Consider the data $(0, 1)$, $(1, 2)$, $(3, 3)$ and $(5, 4)$. Construct a plot that contains the data (as circles) and the polynomial of degree 3 that interpolates this data. You should use the* `axis` *command to make the plot look sufficiently nice.*

Problem 4.6.4. *Consider the data $(1, 1)$, $(3, 1)$, $(4, 1)$, $(5, 1)$ and $(7, 1)$. Construct a plot that contains the data (as circles) and the polynomial of degree 4 that interpolates this data. You should use the* `axis` *command to make the plot look sufficiently nice. Do you believe the curve is a good representation of the data?*

Problem 4.6.5. *Construct plots that contain the data (as circles) and interpolating polynomials for the following sets of data:*

x_i	f_i
-2	6
-1	3
0	1
1	3
2	6

and

x_i	f_i
-2	-5
-1	-3
0	0
1	3
2	5

Problem 4.6.6. *Construct interpolating polynomials through all four sets of weather data given in Example 4.6.3. Use subplot to show all four plots in the same figure, and use the* `title, xlabel` *and* `ylabel` *commands to document the plots. Each plot should show the data points as circles on the corresponding curves.*

Problem 4.6.7. *Consider the function given in Example 4.6.4. Write a* MATLAB *script M–file to create the plots shown in Fig. 4.14. Experiment with using more points to construct the interpolating polynomial, starting with 11, 12, ... At what point does* MATLAB *print a warning that the polynomial is badly conditioned (both for equally spaced and Chebyshev points)? Does centering and scaling improve the results?*

4.6.2 Chebyshev Polynomials and Series

MATLAB does not provide specific functions to construct a Chebyshev representation of the interpolating polynomial. However, it is not difficult to write our own MATLAB functions similar to `polyfit`, `spline` and `pchip`. Recall that if $x \in [-1, 1]$, the jth Chebyshev polynomial, $T_j(x)$, is defined as

$$T_j(x) = \cos(j \arccos(x)),\ j = 0, 1, 2, ...$$

We can easily plot Chebyshev polynomials with just a few MATLAB commands.

Example 4.6.5. The following set of MATLAB commands can be used to plot the 5th Chebyshev polynomial.

```
x = linspace(-1, 1, 200);
y = cos( 5*acos(x) );
plot(x, y)
```

Recall that the Chebyshev form of the interpolating polynomial (for $x \in [-1, 1]$) is

$$p(x) = b_0 T_0(x) + b_1 T_1(x) + \cdots + b_N T_N(x).$$

Suppose we are given data (x_i, f_i), $i = 0, 1, \ldots, N$, and that $-1 \le x_i \le 1$. To construct Chebyshev form of the interpolating polynomial, we need to determine the coefficients $b_0, b_1, \ldots, b_N$ such that $p(x_i) = f_i$. That is,

$$\begin{aligned}
p(x_0) = f_0 \quad &\Rightarrow \quad b_0 T_0(x_0) + b_1 T_1(x_0) + \cdots + b_N T_N(x_0) = f_1 \\
p(x_1) = f_1 \quad &\Rightarrow \quad b_0 T_0(x_1) + b_1 T_1(x_1) + \cdots + b_N T_N(x_1) = f_2 \\
&\ \ \vdots \\
p(x_N) = f_N \quad &\Rightarrow \quad b_0 T_0(x_N) + b_1 T_1(x_N) + \cdots + b_N T_N(x_N) = f_N
\end{aligned}$$

or, more precisely, we need to solve the linear system $Tb = f$:

$$\begin{bmatrix} T_0(x_0) & T_1(x_0) & \cdots & T_N(x_0) \\ T_0(x_1) & T_1(x_1) & \cdots & T_N(x_1) \\ \vdots & \vdots & & \vdots \\ T_0(x_N) & T_1(x_N) & \cdots & T_N(x_N) \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_N \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_N \end{bmatrix}.$$

If the data x_i are not all contained in the interval $[-1, 1]$, then we must perform a variable transformation. For example,

$$\bar{x}_j = -\frac{x_j - x_{mx}}{x_{mn} - x_{mx}} + \frac{x_j - x_{mn}}{x_{mx} - x_{mn}} = \frac{2x_j - x_{mx} - x_{mn}}{x_{mx} - x_{mn}},$$

where $x_{mx} = \max(x_i)$ and $x_{mn} = \min(x_i)$. Notice that when $x_j = x_{mn}$, $\bar{x} = -1$ and when $x_j = x_{mx}$, $\bar{x} = 1$, and therefore $-1 \le \bar{x}_j \le 1$. The matrix T should then be constructed using $\bar{x}_j$. In this case, we need to use the same variable transformation when we evaluate the resulting Chebyshev form of the interpolating polynomial. For this reason, we write a function (such as `spline` and `pchip`) that can be used to construct **and** evaluate the interpolating polynomial.

The basic steps of our function can be outlined as follows:

- The input should be three vectors,

  `x_data` = vector containing given data, x_j.

  `f_data` = vector containing given data, f_j.

  `x` = vector containing points at which the polynomial is to be evaluated. Note that the values in this vector should be contained in the interval $[x_{mn}, x_{mx}]$.

- Perform a variable transformation on the entries in `x_data` to obtain $\bar{x}_j$, $-1 \le \bar{x}_j \le 1$, and define a new vector containing the transformed data:

  `x_data` = $\begin{bmatrix} \bar{x}_0; & \bar{x}_1; & \cdots & \bar{x}_N \end{bmatrix}$

- Let $n =$ `length(x_data)` $= N + 1$.

- Construct the $n \times n$ matrix, T one column at a time using the recurrence relation for generating the Chebyshev polynomials and the (transformed) vector `x_data`:

```
column 1 of T = T(:,1) = ones(n,1)
column 2 of T = T(:,2) = x_data
```

  and for $j = 3, 4, \ldots, n$,

```
column j of T = T(:, j) = 2*x_data .* T(:,j-1) - T(:,j-2)
```

- Compute the coefficients using MATLAB's backslash operator:

```
b = T \ f_data
```

- Perform the variable transformation of the entries in the vector `x`.

- Evaluate the polynomial at the transformed points.

Putting these steps together, we obtain the following function:

```
function y = chebfit(x_data, f_data, x)
%
%      y = chebfit(x_data, f_data, x);
%
% Construct and evaluate a Chebyshev representation of the
% polynomial that interpolates the data points (x_i, f_i):
%
%         p = b(1)*T_0(x) + b(1)*T_1(x) + ... + b(n)T_N(x)
%
% where n = N+1, and T_j(x) = cos(j*acos(x)) is the jth Chebyshev
% polynomial.
%
n = length(x_data);
xmax = max(x_data);
xmin = min(x_data);
xx_data = (2*x_data - xmax - xmin)/(xmax - xmin);
T = zeros(n, n);
T(:,1) = ones(n,1);
T(:,2) = xx_data;
for j = 3:n
  T(:,j) = 2*xx_data.*T(:,j-1) - T(:,j-2);
end
b = T \ f_data;
xx = (2*x - xmax - xmin)/(xmax - xmin);
y = zeros(size(x));
for j = 1:n
  y = y + b(j)*cos( (j-1)*acos(xx) );
end
```

Example 4.6.6. Consider again the average high temperatures from Example 4.6.3. Using `chebfit`, for example with the following MATLAB commands,

```
x_data = (1:12)';
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1]';
x = linspace(1, 12, 200);
y = chebfit(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

we obtain the plot shown in Fig. 4.15. Observe the placement of *transpose* operations on the statements to construct the vectors `x_data` and `f_data`, which ensure these are *column* vectors. It is important to make sure we use consistent (and legal) linear algebra operations, and the code in `chebfit` expects `x_data` and `f_data` to be *column* vectors. Although it is not shown in the above code, as with previous examples, we also used the MATLAB commands `axis, legend, xlabel` and `ylabel` to make the plot look a bit nicer. Because the polynomial that interpolates this data is unique, it should not be surprising that the plot obtained using `chebfit` looks identical to that obtained using `polyfit`. (When there is a large amount of data, the plots may differ slightly due to the effects of computational error and the possible difference in conditioning of the linear systems.)

Problem 4.6.8. *Write a* MATLAB *script M–file that will generate a figure containing the jth Chebyshev polynomials, $j = 1, 3, 5, 7$. Use* `subplot` *to put all plots in one figure, and* `title` *to document the various plots.*

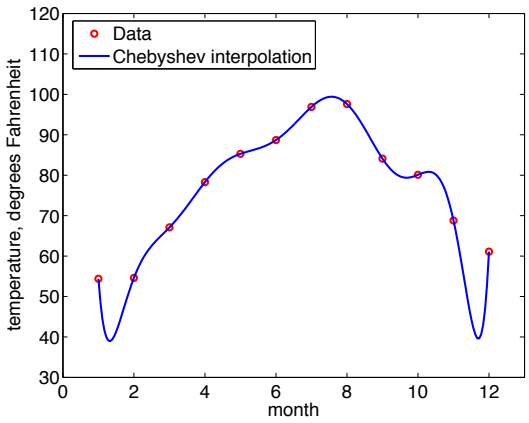

Figure 4.15: Interpolating polynomial or average high temperature data given in Example 4.6.3 using the Chebyshev form.

Problem 4.6.9. *Consider the data* $(0,1)$, $(1,2)$, $(3,3)$ *and* $(5,4)$. *Construct a plot that contains the data (as circles) and the polynomial of degree 3 that interpolates this data using the function* `chebfit`. *You should use the* `axis` *command to make the plot look sufficiently nice.*

Problem 4.6.10. *Consider the data* $(1,1)$, $(3,1)$, $(4,1)$, $(5,1)$ *and* $(7,1)$. *Construct a plot that contains the data (as circles) and the polynomial of degree 4 that interpolates this data using the function* `chebfit` *You should use the* `axis` *command to make the plot look sufficiently nice. Do you believe the curve is a good representation of the data?*

Problem 4.6.11. *Construct plots that contain the data (as circles) and interpolating polynomials, using the function* `chebfit`, *for the following sets of data:*

x_i	f_i
-2	6
-1	3
0	1
1	3
2	6

and

x_i	f_i
-2	-5
-1	-3
0	0
1	3
2	5

Problem 4.6.12. *Construct interpolating polynomials, using the function* `chebfit`, *through all four sets of weather data given in Example 4.6.3. Use subplot to show all four plots in the same figure, and use the* `title, xlabel` *and* `ylabel` *commands to document the plots. Each plot should show the data points as circles on the corresponding curves.*

4.6.3 Polynomial Splines

Polynomial splines help to avoid excessive oscillations by fitting the data using a collection of low degree polynomials. We've actually already used *linear polynomial splines* to connect data via MATLAB's plot command. But we can create this linear spline more explicitly using the `interp1` function. The basic calling syntax is given by:

```
y = interp1(x_data, f_data, x);
```

where

- `x_data` and `f_data` are vectors containing the given data points,
- `x` is a vector containing values at which the linear spline is to be evaluated (e.g., for plotting the spline), and
- `y` is a vector containing $S(x)$ values.

The following example illustrates how to use `interp1` to construct, evaluate, and plot a linear spline interpolating function.

Example 4.6.7. Consider the average high temperatures from Example 4.6.3. The following set of MATLAB commands can be used to construct a linear spline interpolating function of this data:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = interp1(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

The resulting plot is shown in Fig. 4.16. Note that we also used the MATLAB commands `axis`, `legend`, `xlabel` and `ylabel` to make the plot look a bit nicer.

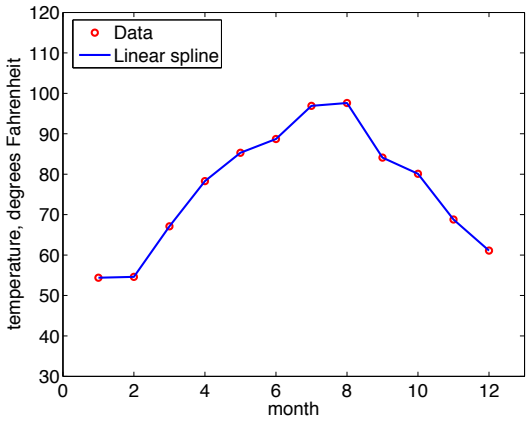

Figure 4.16: Linear polynomial spline interpolating function for average high temperature data given in Example 4.6.3

We can obtain a smoother curve by using a higher degree spline. The primary MATLAB function for this purpose, `spline`, constructs a cubic spline interpolating function. The basic calling syntax, which uses, by default, not-a-knot end conditions, is as follows:

```
y = spline(x_data, f_data, x);
```

where `x_data`, `f_data`, `x` and `y` are defined as for `interp1`. The following example illustrates how to use `spline` to construct, evaluate, and plot a cubic spline interpolating function.

Example 4.6.8. Consider again the average high temperatures from Example 4.6.3. The following set of MATLAB commands can be used to construct a cubic spline interpolating function of this data:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = spline(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

The resulting plot is shown in Fig. 4.17. Note that we also used the MATLAB commands `axis, legend, xlabel, ylabel` and `title` to make the plot look a bit nicer.

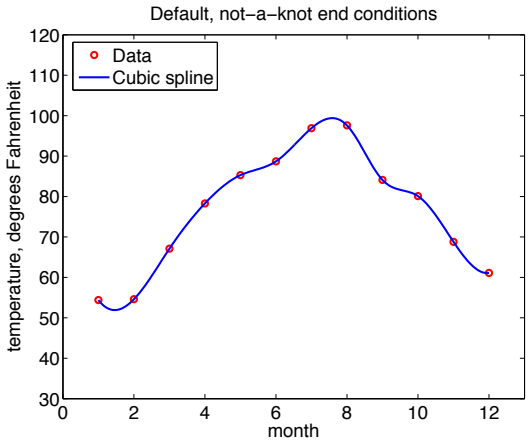

Figure 4.17: Cubic polynomial spline interpolating function, using not-a-knot end conditions, for average high temperature data given in Example 4.6.3

It should be noted that other end conditions can be used, if they can be defined by the slope of the spline at the end points. In this case, the desired slope values at the end points are attached to the beginning and end of the vector `f_data`. This is illustrated in the next example.

Example 4.6.9. Consider again the average high temperatures from Example 4.6.3. The so-called *clamped* end conditions assume the slope of the spline is 0 at the end points. The following set of MATLAB commands can be used to construct a cubic spline interpolating function with *clamped* end conditions:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = spline(x_data, [0, f_data, 0], x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

Observe that, when calling spline, we replaced `f_data` with the vector `[0, f_data, 0]`. The first and last values in this augmented vector specify the desired slope of the spline (in this case zero) at the end points. The resulting plot is shown in Fig. 4.18 Note that we also used the MATLAB commands `axis, legend, xlabel, ylabel` and `title` to make the plot look a bit nicer.

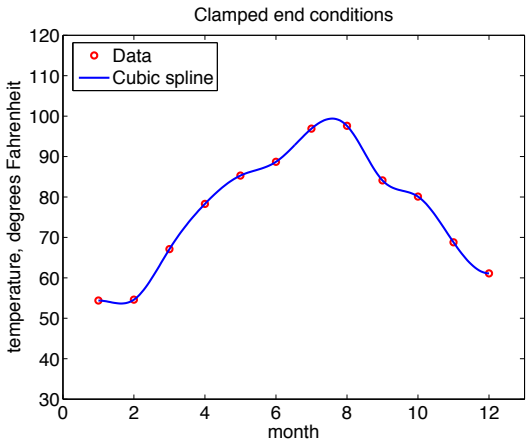

Figure 4.18: Cubic polynomial spline interpolating function, using clamped end conditions, for average high temperature data given in Example 4.6.3

In all of the previous examples using `interp1` and `spline`, we construct **and** evaluate the spline in one command. Recall that for polynomial interpolation we used `polyfit` and `polyval` in a two step process. It is possible to use a two step process with both linear and cubic splines as well. For example, when using `spline`, we can execute the following commands:

```
S = spline(x_data, f_data);
y = ppval(S, x);
```

where `x` is a set of values at which the spline is to be evaluated. If we specify only two input arguments to `spline`, it returns a *structure array* that defines the piecewise cubic polynomial, $S(x)$, and the function `ppval` is then used to evaluate $S(x)$.

In general, if all we want to do is evaluate and/or plot $S(x)$, it is not necessary to use this two step process. However, there may be situations for which we must access explicitly the polynomial pieces. For example, we could find the maximum and minimum (i.e., critical points) of $S(x)$ by explicitly computing $S'(x)$. Another example is given in Section 5.3 where the cubic spline is used to integrate tabular data.

We end this subsection by mentioning that MATLAB has another function, `pchip`, that can be used to construct a piecewise *Hermite* cubic interpolating polynomial, $H(x)$. In interpolation, the name *Hermite* is usually attached to techniques that use specific slope information in the construction of the polynomial pieces. Although `pchip` constructs a piecewise cubic polynomial, strictly speaking, it is not a spline because the second derivative may be discontinuous at the knots. However, the first derivative is continuous. The slope information used to construct $H(x)$ is determined from the data. In particular, if there are intervals where the data are monotonic, then so is $H(x)$, and at points where the data has a local extremum, so does $H(x)$. We can use `pchip` exactly as we use `spline`; that is, either one call to construct and evaluate:

```
y = pchip(x_data, f_data, x);
```

or via a two step approach to construct and then evaluate:

```
H = pchip(x_data, f_data);
y = ppval(H, x);
```

where `x` is a set of values at which $H(x)$ is to be evaluated.

Example 4.6.10. Because the piecewise cubic polynomial constructed by `pchip` will usually have discontinuous second derivatives at the data points, it will be less smooth and a little less accurate for

smooth functions than the cubic spline interpolating the same data. However, forcing monotonicity when it is present in the data has the effect of preventing oscillations and overshoot which can occur with a spline. To illustrate the point we consider a somewhat pathological example of non-smooth data from the MATLAB Help pages. The following MATLAB code fits first a cubic spline then a piecewise cubic monotonic polynomial to the data, which is monotonic.

```
x_data = -3:3;
f_data = [-1 -1 -1 0 1 1 1];
x = linspace(-3, 3, 200);
s = spline(x,y,t);
p = pchip(x,y,t);
plot(x_data, f_data, 'ro')
hold on
plot(x, s, 'k--')
plot(x, p, 'b-')
legend('Data', 'Cubic spline', 'PCHIP interpolation', 'Location', 'NW')
```

The resulting plot is given in Fig. 4.19. We observe both oscillations and overshoot in the cubic spline fit and monotonic behavior in the piecewise cubic monotonic polynomial fit.

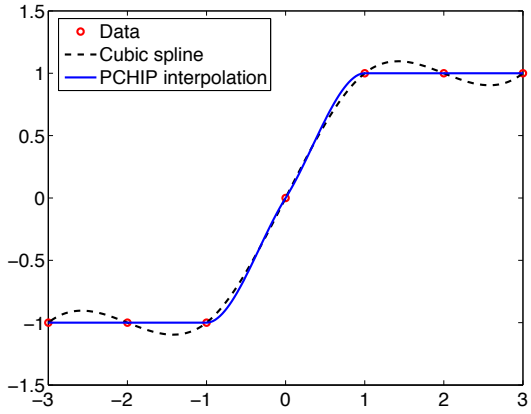

Figure 4.19: Comparison of fitting a monotonic piecewise cubic polynomial to monotonic data (using MATLAB's `pchip` function) and to a cubic spline (using MATLAB's `spline` function).

Example 4.6.11. Consider again the average high temperatures from Example 4.6.3. The following set of MATLAB commands can be used to construct a piecewise cubic Hermite interpolating polynomial through the given data:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = pchip(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

The resulting plot is shown in Fig. 4.20 As with previous examples, we also used the MATLAB commands `axis, legend, xlabel` and `ylabel` to make the plot more understandable. Observe that fitting the data using `pchip` avoids oscillations and overshoot that can occur with the cubic spline fit, and that the monotonic behavior of the data is preserved.

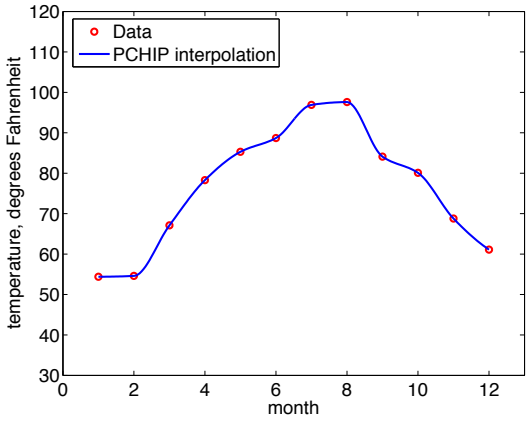

Figure 4.20: Piecewise cubic Hermite interpolating polynomial, for average high temperature data given in Example 4.6.3

Notice that the curve generated by `pchip` is smoother than the piecewise linear spline, but because `pchip` does not guarantee continuity of the second derivative, it is not as smooth as the cubic spline interpolating function. In addition, the flat part on the left, between the months January and February, is probably not an accurate representation of the actual temperatures for this time period. Similarly, the maximum temperature at the August data point (which is enforced by the `pchip` construction) may not be accurate; more likely is that the maximum temperature occurs some time between July and August. We mention these things to emphasize that it is often extremely difficult to produce an accurate, physically reasonable fit to data.

Problem 4.6.13. *Consider the data* $(0,1)$, $(1,2)$, $(3,3)$ *and* $(5,4)$. *Use subplot to make a figure with 4 plots: a linear spline, a cubic spline with not-a-knot end conditions, a cubic spline with clamped end conditions, and a piecewise cubic Hermite interpolating polynomial fit of the data. Each plot should also show the data points as circles on the curve, and should have a title indicating which of the four methods was used.*

Problem 4.6.14. *Consider the following sets of data:*

x_i	f_i
-2	6
-1	3
0	1
1	3
2	6

and

x_i	f_i
-2	-5
-1	-3
0	0
1	3
2	5

Make two figures, each with four plots: a linear spline, a cubic spline with not-a-knot end conditions, a cubic spline with clamped end conditions, and a piecewise cubic Hermite interpolating polynomial fit of the data. Each plot should also show the data points as circles on the curve, and should have a title indicating which of the four methods was used.

Problem 4.6.15. *Consider the four sets of weather data given in Example 4.6.3. Make four figures, each with four plots: a linear spline, a cubic spline with not-a-knot end conditions, a cubic spline with clamped end conditions, and a piecewise cubic Hermite interpolating polynomial fit of the data. Each plot should also show the data points as circles on the curve, and should have a title indicating which of the four methods was used.*

Problem 4.6.16. *Notice that there is an obvious "wobble" at the left end of the plot given in Fig. 4.17. Repeat the previous problem, but this time shift the data so that the year begins with April. Does this help eliminate the wobble? Does shifting the data have an effect on any of the other plots?*

4.6.4 Interpolating to Periodic Data

The methods considered thus far and those to be considered later are designed to fit to general data. When the data has a specific property it behoves us to exploit that property. So, in the frequently occurring case where the data is thought to represent a periodic function we should fit it using a periodic function, particularly with a trigonometric series where the frequency of the terms is designed to fit the frequency of the data.

Given a set of data that is supposed to represent a periodic function sampled at equal spacing across the period of the function, MATLAB's function `interpft` for periodic data produces interpolated values to this data on a finer user specified equally spaced mesh. First, `interpft` uses a discrete Fourier transform internally to produce a trigonometric series interpolating the data. Then, it uses an inverse discrete Fourier transform to evaluate the trigonometric series on the finer mesh, over the same interval. (At no stage does the user see the trigonometric series.)

Example 4.6.12. Consider sampling $f(x) = \sin(\pi x)$ at $x = i/10$, $i = 0, 1, \ldots, 9$. The data is assumed periodic so we do not evaluate $f(x)$ for $i = 10$. We use `interpft` to interpolate to this data at half the mesh size and plot the results using the commands:

```
x_data = (0:1:9)/10;
f_data = sin(pi*x_data);
x = (0:1:19)/20;
y = interpft(f_data, 20);
plot(x_data, f_data, 'ro')
hold on
plot(x, y, '+')
```

The plot is given on the left in Figure 4.21 where the interpolated and data values coincide at the original mesh points.

If we use instead the nonperiodic function $f(x) = \sin(\pi x^2)$ at the same points as follows

```
x_data = (0:1:9)/10;
f_data = sin(pi*(x.^2));
x = (0:1:19)/20;
y = interpft(f_data, 20);
plot(x_data, f_data, 'ro')
hold on
plot(x, y, '+')
```

we obtain the plot on the right in Figure 4.21. Note the behavior near $x = 0$. The plotted function is periodic. The negative value arises from fitting a periodic function to data arising from a nonperiodic function.

4.6.5 Least Squares

Using least squares to compute a polynomial that approximately fits given data is very similar to the interpolation problem. MATLAB implementations can be developed by mimicking what was done in Section 4.6.1, except we replace the interpolation condition $p(x_i) = f_i$ with $p(x_i) \approx f_i$. For example, suppose we want to find a polynomial of degree M that approximates the $N+1$ data points (x_i, f_i),

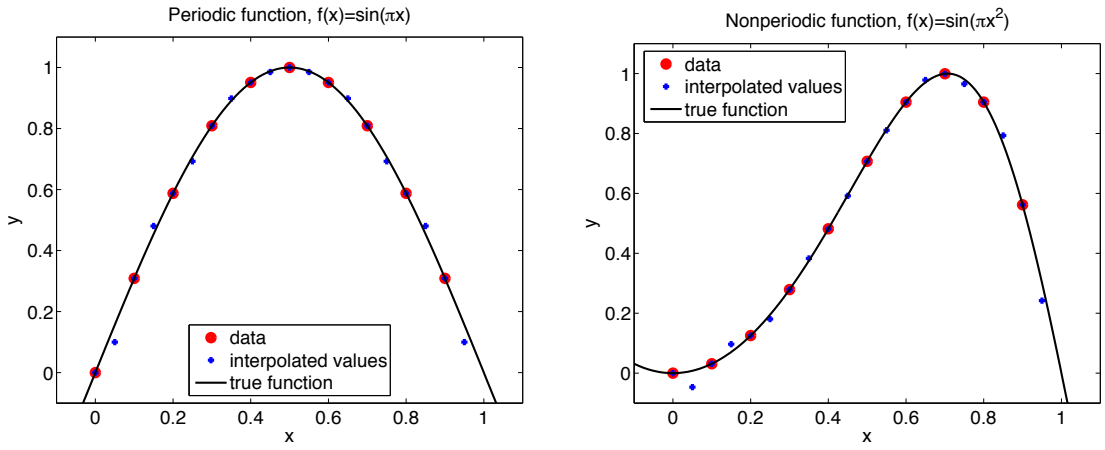

Figure 4.21: Trigonometric fit to data using MATLAB's `interpft` function. The left plot fits to periodic data, and the right plot fits to non-periodic data.

$i = 1, 2, \ldots N+1$. If we use the power series form, then we end up with an $(N+1) \times (M+1)$ linear system

$$Va \approx f.$$

The Normal equations approach to construct the least squares fit of the data is simply the $(M+1) \times (M+1)$ linear system

$$V^T V a = V^T f.$$

Example 4.6.13. Consider the data

x_i	1	3	4	5
f_i	2	4	3	1

Suppose we want to find a least squares fit to this data by a line: $p(x) = a_0 + a_1 x$. Then using the conditions $p(x_i) \approx f_i$ we obtain

$$\begin{array}{lll} p(x_0) \approx f_0 & \Rightarrow & a_0 + a_1 \approx 2 \\ p(x_1) \approx f_1 & \Rightarrow & a_0 + 3a_1 \approx 4 \\ p(x_2) \approx f_2 & \Rightarrow & a_0 + 4a_1 \approx 3 \\ p(x_3) \approx f_3 & \Rightarrow & a_0 + 5a_1 \approx 1 \end{array} \quad \Rightarrow \quad \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \approx \begin{bmatrix} 2 \\ 4 \\ 3 \\ 1 \end{bmatrix}.$$

To find a least squares solution, we solve the Normal equations

$$V^T V = V^T f \quad \Rightarrow \quad \begin{bmatrix} 4 & 23 \\ 13 & 51 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 10 \\ 31 \end{bmatrix}$$

Notice that this linear system is equivalent to what was obtained in Example 4.4.1.

MATLAB's backslash operator can be used to compute least squares approximations of linear systems $Va \approx f$ using the simple command

```
a = V \ f
```

In general, when the backslash operator is used, MATLAB checks first to see if the matrix is square (number of rows = number of columns). If the matrix is square, it will use Gaussian elimination

with partial pivoting by rows to solve $Va = f$. If the matrix is not square, it will compute a least squares solution, which is equivalent to solving the Normal equations. MATLAB does not use the Normal equations, but instead a generally more accurate approach based on a QR decomposition of the rectangular matrix V. The details are beyond the scope of this book, but the important point is that methods used for polynomial interpolation can be used to construct polynomial least squares fit to data. Thus, the main built-in MATLAB functions for polynomial least squares data fitting are `polyfit` and `polyval`. In particular, we can construct the coefficients of the polynomial using

```
a = polyfit(x_data, f_data, M)
```

where M is the degree of the polynomial. In the case of interpolation, `M = N = length(x_data)-1`, but in least squares, `M` is generally less than `N`.

Example 4.6.14. Consider the data

x_i	1	3	4	5
f_i	2	4	3	1

To find a least squares fit of this data by a line (i.e., polynomial of degree $M = 1$), we use the MATLAB commands:

```
x_data = [1 3 4 5];
f_data = [2 4 3 1];
a = polyfit(x_data, f_data, 1);
```

As with interpolating polynomials, we can use `polyfit` to evaluate the polynomial, and `plot` to generate a figure:

```
x = linspace(1, 5, 200);
y = polyval(a, x);
plot(x_data, f_data, 'ro');
hold on
plot(x, y)
```

If we wanted to construct a least squares fit of the data by a quadratic polynomial, we simply replace the `polyfit` statement with:

```
a = polyfit(x_data, f_data, 2);
```

Fig. 4.22 shows the data, and the linear and quadratic polynomial least squares fits.

Note that it is possible to modify `chebfit` so that it can compute a least squares fit of the data; see Problem 4.6.17.

It may be the case that functions other than polynomials are more appropriate to use in data fitting applications. The next two examples illustrate how to use least squares to fit data to more general functions.

Example 4.6.15. Suppose it is known that data collected from an experiment, (x_i, f_i), can be represented well by a sinusoidal function of the form

$$g(x) = a_1 + a_2 \sin(x) + a_3 \cos(x).$$

To find the coefficients, a_1, a_2, a_3, so that $g(x)$ is a least squares fit of the data, we use the criteria $g(x_i) \approx f_i$. That is,

$$\begin{array}{rcl} g(x_1) \approx f_1 & \Rightarrow & a_1 + a_2 \sin(x_1) + a_3 \cos(x_1) \approx f_1 \\ g(x_2) \approx f_2 & \Rightarrow & a_1 + a_2 \sin(x_2) + a_3 \cos(x_2) \approx f_2 \\ & \vdots & \\ g(x_n) \approx f_n & \Rightarrow & a_1 + a_2 \sin(x_n) + a_3 \cos(x_n) \approx f_n \end{array}$$

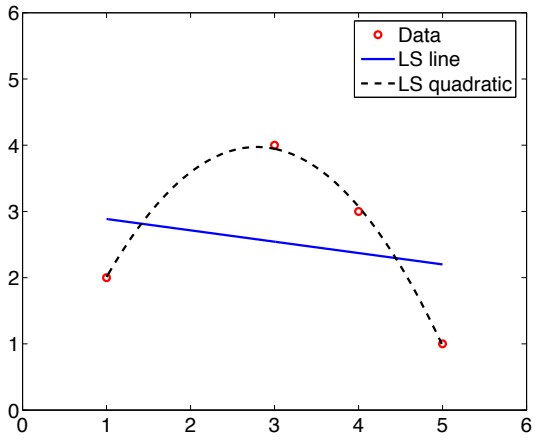

Figure 4.22: Linear and quadratic polynomial least squares fit of data given in Example 4.6.14

which can be written in matrix-vector form as

$$\begin{bmatrix} 1 & \sin(x_1) & \cos(x_1) \\ 1 & \sin(x_2) & \cos(x_2) \\ \vdots & \vdots & \vdots \\ 1 & \sin(x_n) & \cos(x_n) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \approx \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}.$$

In MATLAB, the least squares solution can then be found using the backslash operator which, assuming more than 3 data points, uses a QR decomposition of the matrix. A MATLAB function to compute the coefficients could be written as follows:

```
function a = sinfit(x_data, f_data)
%
%        a = sinfit(x_data, f_data);
%
%  Given a set of data, (x_i, f_i), this function computes the
%  coefficients of
%       g(x) = a(1) + a(2)*sin(x) + a(3)*cos(x)
%  that best fits the data using least squares.
%
n = length(x_data);
W = [ones(n, 1), sin(x_data), cos(x_data)];
a = W \ f_data;
```

Example 4.6.16. Suppose it is known that data collected from an experiment, (x_i, f_i), can be represented well by an exponential function of the form

$$g(x) = a_1 e^{a_2 x}.$$

To find the coefficients, a_1, a_2, so that $g(x)$ is a least squares fit of the data, we use the criteria $g(x_i) \approx f_i$. The difficulty in this problem is that $g(x)$ is not linear in its coefficients. But we can

use logarithms, and their properties, to rewrite the problem as follows:

$$\begin{aligned} g(x) &= a_1 e^{a_2 x} \\ \ln(g(x)) &= \ln\left(a_1 e^{a_2 x}\right) \\ &= \ln(a_1) + a_2 x \\ &= \hat{a}_1 + a_2 x, \quad \text{where } \hat{a}_1 = \ln(a_1), \text{ or } a_1 = e^{\hat{a}_1}. \end{aligned}$$

With this transformation, we would like $\ln(g(x_i)) \approx \ln(f_i)$, or

$$\begin{aligned} \ln(g(x_1)) \approx \ln(f_1) &\Rightarrow \hat{a}_1 + a_2 x_1 \approx \ln(f_1) \\ \ln(g(x_2)) \approx \ln(f_2) &\Rightarrow \hat{a}_1 + a_2 x_2 \approx \ln(f_2) \\ &\vdots \\ \ln(g(x_n)) \approx \ln(f_n) &\Rightarrow \hat{a}_1 + a_2 x_n \approx \ln(f_n) \end{aligned}$$

which can be written in matrix-vector form as

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} \hat{a}_1 \\ a_2 \end{bmatrix} \approx \begin{bmatrix} \ln(f_1) \\ \ln(f_2) \\ \vdots \\ \ln(f_n) \end{bmatrix}.$$

In MATLAB, the least squares solution can then be found using the backslash operator. A MATLAB function to compute the coefficients could be written as follows:

```
function a = expfit(x_data, f_data)
%
%        a = expfit(x_data, f_data);
%
%  Given a set of data, (x_i, f_i), this function computes the
%  coefficients of
%       g(x) = a(1)*exp(a(2)*x)
%  that best fits the data using least squares.
%
n = length(x_data);
W = [ones(n, 1), x_data];
a = W \ log(f_data);
a(1) = exp(a(1));
```

Here we use the built-in MATLAB functions `log` and `exp` to compute the natural logarithm and the natural exponential, respectively.

Problem 4.6.17. *Modify the function* `chebfit` *so that it computes a least squares fit to the data. The modified function should have an additional input value specifying the degree of the polynomial. Use the data in Example 4.6.14 to test your function. In particular, produce a plot similar to the one given in Fig. 4.22.*