EFFICIENT ways of representing data are fundamental to advancing the understanding of scientific problems. This chapter introduces polynomial interpolation and spline interpolation as convenient tools for finding functions that pass through given data points.

3.1 Data and interpolating functions

A mathematical function is said to **interpolate** a set of data points if it passes through those points. Suppose you have collected a set of \((x, y)\) data points, such as \((0, 1)\), \((2, 2)\), and \((3, 4)\). There is a parabola that passes through the three points, shown in Figure 3.1. This parabola is called the degree 2 interpolating polynomial passing through these three points.

**Definition 3.1** The function \(y = P(x)\) **interpolates** the data points \((x_1, y_1), \ldots, (x_n, y_n)\) if \(P(x_i) = y_i\) for each \(1 \leq i \leq n\).

![Figure 3.1: Interpolation by parabola.](image)

The points \((0, 1)\), \((2, 2)\), and \((3, 4)\) are interpolated by the function \(P(x) = \frac{1}{2}x^2 - \frac{1}{2}x + 1\).

Note that \(P\) is required to be a function, that is, each value \(x\) corresponds to a single \(y\). This puts a restriction on the set of data points \(\{(x_i, y_i)\}\) that can be interpolated - the \(x_i\)’s must be all distinct, in order for a function to pass through them. There is no such restriction on the \(y_i\)’s.

Now suppose we are given a set of points \((x_1, y_1), \ldots, (x_n, y_n)\) with all distinct \(x_i\)’s. Does an interpolating function always exist? Of course the answer is yes - one could simply “connect the dots”, draw line segments between the points, and that would qualify as an interpolating function.
Why polynomials? Polynomials are very often used for interpolation because of their straightforward mathematical properties. There is simple theory about when an interpolating polynomial of a given degree exists for a given set of points. More important, in a real sense polynomials are the most fundamental of functions for digital computers. Central processing units usually have fast methods in hardware for adding and multiplying floating points numbers, which are the operations needed to evaluate a polynomial. Complicated functions can be approximated by interpolating polynomials in order to make them computable with these two hardware operations.

More useful would be a single polynomial through the data points. In the remainder of this section, we will focus on the problem of finding interpolating polynomials.

Interpolation is the reverse of evaluation. In polynomial evaluation (such as Horner’s method) we are given a polynomial and asked to evaluate a \( y \)-value for a given \( x \)-value - that is, compute points lying on the curve. Polynomial interpolation asks for the opposite process: Given these points, compute a polynomial that can generate them.

Does such a polynomial always exist? Again, we must assume that the \( x \)-coordinates of the points are distinct. It would be impossible, by definition of function, for a polynomial or any function to go through two points with the same \( x \) coordinate and different \( y \) coordinates. With that proviso, the answer is yes. No matter how many points are given, there is some polynomial \( y = P(x) \) that runs through all the points. We will prove this and several other facts about the existence of interpolating polynomials. Later, we will consider how to interpolate with other, non-polynomial, functions, such as splines.

### 3.1.1 Lagrange interpolation

Assume we have \( n \) data points \((x_1, y_1), \ldots, (x_n, y_n)\) and would like to find an interpolating polynomial. With any problem, it is good to know before the computing starts how many solutions there are. Let’s fix the degree \( d \) of the polynomial. How many degree \( d \) polynomials pass through the \( n \) points? The answer depends on how the points are situated with respect to one another.

The case \( d = 1 \) is simple but illuminating. Either the points are collinear, in which case there is a degree 1 interpolating polynomial, or they are not collinear, and none exists. Again, we ignore the case of the points lying along a vertical line, because we will require in this chapter that all \( x_i \) are distinct.

For polynomials of degree greater than one, we will have to work a little harder, but by the time we are done we will have a computational method for determining exactly what degrees of polynomials go through a given set of points, as well as for computing the polynomials explicitly when they exist.

If we are given \( n \) data points, there is an explicit formula for writing down a polynomial of degree \( d = n - 1 \) which interpolates the points. This is called the Lagrange interpolating formula.
For example, suppose we are given three points \((x_1, y_1), (x_2, y_2), (x_3, y_3)\). Then the polynomial

\[
P_2(x) = y_1 \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} \tag{3.1}
\]

is the Lagrange interpolating polynomial for these points. First notice why the points each lie on the polynomial curve. When \(x_1\) is substituted for \(x\), the terms evaluate to \(y_1 + 0 + 0 = y_1\). The second and third numerators are chosen to disappear when \(x_1\) is substituted, and the first denominator is chosen just so to balance the first denominator so that \(y_1\) pops out. Similar things happen when \(x_2\) and \(x_3\) are substituted. When any other number is substituted for \(x\), we have little control over the result. But then, the job was only to interpolate at the three points - that’s the extent of our concern. Second, notice that the polynomial (3.1) is of degree two in the variable \(x\).

**Example 3.1** Find an interpolating polynomial for the data points \((0, 1), (2, 2)\) and \((3, 4)\) in Figure 3.1.

Substituting into Lagrange’s formula (3.1) yields

\[
P_2(x) = 1 \frac{(x - 2)(x - 3)}{(0 - 2)(0 - 3)} + 2 \frac{(x - 0)(x - 3)}{(2 - 0)(2 - 3)} + 4 \frac{(x - 0)(x - 2)}{(3 - 0)(3 - 2)}
\]

\[
= \frac{1}{6}(x^2 - 5x + 6) + 2(-\frac{1}{2})(x^2 - 3x) + 4\frac{1}{3}(x^2 - 2x)
\]

\[
= \frac{1}{2}x^2 - \frac{1}{2}x + 1.
\]

Finally, check that \(P_2(0) = 1, P_2(2) = 2, \) and \(P_2(3) = 4\).

In general, suppose we are presented with \(n\) points \((x_1, y_1), \ldots, (x_n, y_n)\). For each \(k\) between 1 and \(n\), define the term

\[
L_k(x) = \frac{(x - x_1) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_1) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}.
\]

The interesting property of \(L_k\) is that \(L_k(x_j) = 1\) while \(L_k(x_j) = 0\), where \(x_j\) is any of the other data points. Then define the degree \(n - 1\) polynomial

\[
P_{n-1}(x) = y_1L_1(x) + \cdots + y_nL_n(x)
\]

This is a straightforward generalization of the polynomial in (3.1), and works the same way. Substituting \(x_k\) for \(x\) yields

\[
P(x_k) = y_1L_1(x_k) + \cdots + y_nL_n(x_k) = 0 + \cdots + 0 + y_kL_k(x_k) + 0 + \cdots + 0 = y_k,
\]

so it works as designed.

We have constructed a polynomial of degree at most \(n - 1\) that passes through any set of \(n\) points with distinct \(x_i\)’s. Next it is natural to ask how many of these there are. Assume for the sake of discussion that there are two, say \(P(x)\) and \(Q(x)\), that have degree at most \(n - 1\) and that both interpolate all \(n\) points. That is, we are assuming \(P(x_1) = Q(x_1) = y_1, P(x_2) = Q(x_2) = y_2, \ldots, P(x_n) = Q(x_n) = y_n\). Now define the new polynomial \(H(x) = P(x) - Q(x)\). Clearly
the degree of $H$ is also at most $n - 1$, and note that $0 = H(x_1) = H(x_2) = \cdots = H(x_n)$, that is $H$ has $n$ distinct zeros. According to the Fundamental Theorem of Algebra, a degree $d$ polynomial can have at most $d$ zeros, unless it is the identically zero polynomial. Therefore $H$ is the identically zero polynomial, and $P(x) \equiv Q(x)$. We conclude that there is a unique $P(x)$ of degree $\leq n - 1$ interpolating the $n$ points $(x_i, y_i)$. These facts are summarized in the following theorem.

**Theorem 3.2** (Main Theorem of Polynomial Interpolation) Let $(x_1, y_1), \ldots, (x_n, y_n)$ be $n$ points in the plane with distinct $x_i$. Then there exists one and only one polynomial $P$ of degree $n - 1$ or less that satisfies $P(x_i) = y_i$ for $i = 1, \ldots, n$.

**Example 3.2** Find the polynomial of degree 3 or less that interpolates the points $(0, 2), (1, 1), (2, 0),$ and $(3, -1)$.

The Lagrange form is

\[
P_2(x) = \frac{2(x - 1)(x - 2)(x - 3)}{(0 - 1)(0 - 2)(0 - 3)} + \frac{1(x - 0)(x - 2)(x - 3)}{(1 - 0)(1 - 2)(1 - 3)} + \frac{0(x - 0)(x - 1)(x - 3)}{(2 - 0)(2 - 1)(2 - 3)} - \frac{1(x - 0)(x - 1)(x - 2)}{(3 - 0)(3 - 1)(3 - 2)}
\]

\[
= -\frac{1}{3}(x^3 - 6x^2 + 11x - 6) + \frac{1}{2}(x^3 - 5x^2 + 6x) - \frac{1}{6}(x^3 - 3x^2 + 2x)
\]

\[
= -x + 2
\]

Theorem 3.2 says that there exists exactly one interpolating polynomial of degree 3 or less - it may or may not be exactly degree 3. In this case, the data points are collinear, so the interpolating polynomial has degree 1. The theorem implies that there are no interpolating polynomials of degree two or three. It may be already intuitively obvious to you that no parabola can pass through these four collinear points, but here is the reason. Theorem 3.2 also forbids any four collinear points to lie on a cubic polynomial.

\[
\]

### 3.1.2 Newton’s divided differences

The Lagrange interpolation method, as described in the previous section, never fails. It is a constructive way to write the unique polynomial promised by Theorem 3.2. It is also intuitive; one glance explains why it works. However, it is seldom used for calculation due to the existence of alternative methods that result in more manageable and less computationally complex forms.

Newton divided differences give a particularly simple way to write the interpolating polynomial. Given $n$ data points, the result will be a degree at most $n - 1$ polynomial, just as Lagrange form does. Theorem 3.2 says that it can be none other than the same as the lagrange interpolating polynomial, written in a disguised form.

The idea of divided differences is fairly simple, but some notation needs to be mastered first. View the data points as given by some function $f$, and list the data points in a table:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$f(x_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_2$</td>
<td>$f(x_2)$</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>$x_n$</td>
<td>$f(x_n)$</td>
</tr>
</tbody>
</table>
Now define the divided differences, which are the real numbers

\[ f[x_k] = f(x_k) \]

\[ f[x_k \ x_{k+1}] = \frac{f[x_{k+1}] - f[x_k]}{x_{k+1} - x_k} \]

\[ f[x_k \ x_{k+1} \ x_{k+2}] = \frac{f[x_{k+1} \ x_{k+2}] - f[x_k \ x_{k+1}]}{x_{k+2} - x_k} \]

\[ f[x_k \ x_{k+1} \ x_{k+2} \ x_{k+3}] = \frac{f[x_{k+1} \ x_{k+2} \ x_{k+3}] - f[x_k \ x_{k+1} \ x_{k+2}]}{x_{k+3} - x_k} \]  \hspace{1cm} (3.2)

and so on. These numbers are the coefficients of the interpolating polynomial for the data points, which is given by the Newton’s divided difference formula

\[ P(x) = f[x_1] + f[x_1 \ x_2](x - x_1) \]

\[ + f[x_1 \ x_2 \ x_3](x - x_1)(x - x_2) \]

\[ + f[x_1 \ x_2 \ x_3 \ x_4](x - x_1)(x - x_2)(x - x_3) \]

\[ + \ldots \]

\[ + f[x_1 \ \cdots \ x_n](x - x_1)\cdots(x - x_{n-1}). \]  \hspace{1cm} (3.3)

A proof of this fact is given in Section 3.2.2. Notice that the divided difference formula gives the interpolating polynomial as a nested polynomial in the sense of Horner’s method. It is automatically ready to be evaluated in an efficient way.

Finally, the recursive definition of the Newton’s divided differences allows arrangement into a convenient table. For three points the table has form

<table>
<thead>
<tr>
<th>x_1</th>
<th>f[x_1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_2</td>
<td>f[x_2]</td>
</tr>
<tr>
<td></td>
<td>f[x_1 \ x_2]</td>
</tr>
</tbody>
</table>

and the coefficients of the polynomial (3.3) can be read from the top edge of the triangle.

**Example 3.3** Use Newton’s divided difference to find the interpolating polynomial passing through the points \((0, 1), (2, 2), (3, 4)\).

Applying the definitions of divided differences leads to the table

\[
\begin{array}{c|ccc}
0 & 1 & 1/2 & 1/2 \\
2 & 2 & 1 & 1 \\
3 & 4 & & \\
\end{array}
\]
This table is computed as follows. After writing down the \( x \) and \( y \) coordinates down in separate columns, calculate the next columns, left to right, as divided differences as in (3.2). For example,

\[
\frac{2 - 1}{2 - 0} = \frac{1}{2} \\
\frac{2 - \frac{1}{2}}{3 - 0} = \frac{1}{2} \\
\frac{4 - 2}{3 - 2} = 2
\]

After completing the divided difference triangle, the coefficients of the polynomial \( 1, 1/2, 1/2 \) can be read from the top edge of the table. The interpolating polynomial can be written as

\[
P(x) = 1 + \frac{1}{2}(x - 0) + \frac{1}{2}(x - 0)(x - 2),
\]

or in nested form

\[
P(x) = 1 + (x - 0) \left( \frac{1}{2} + (x - 2) \cdot \frac{1}{2} \right).
\]

The base points for the nested form (see Horner’s Method in Chapter 0) are \( r_1 = 0 \) and \( r_2 = 2 \). Alternatively, we could do more algebra and write the interpolating polynomial as

\[
P(x) = 1 + \frac{1}{2}x + \frac{1}{2}x(x - 2) = \frac{1}{2}x^2 - \frac{1}{2}x + 1,
\]

matching the version Lagrange interpolation gave above.

Using the Newton divided difference approach, new data points that arrive after computing the original interpolating polynomial can be easily added.

**Example 3.4** *Add the fourth data point \((1, 0)\) to the list in Example 3.3.*

We can keep the calculations that were already done and just add a new bottom row to the triangle:

\[
\begin{array}{ccc}
0 & 1 & \\
2 & 2 & \frac{1}{2} \\
3 & 4 & 0 \\
1 & 0 & \end{array}
\]

The result is one new term to add to the original polynomial \( P_2(x) \). Reading from the top edge of the triangle, we see that the new degree 3 interpolating polynomial is

\[
P_3(x) = 1 + \frac{1}{2}(x - 0) + \frac{1}{2}(x - 0)(x - 2) - \frac{1}{2}(x - 0)(x - 2)(x - 3).
\]

Note that \( P_3(x) = P_2(x) - \frac{1}{2}(x - 0)(x - 2)(x - 3) \), so the previous polynomial can be reused as part of the new one.
It is interesting to compare the extra work necessary to add a new point using the Lagrange formulation versus the divided difference formulation. The Lagrange polynomial must be restarted from the beginning when a new point is added; none of the previous calculation can be used. On the other hand, in divided difference form, we keep the earlier work and add one new term to the polynomial. Therefore the divided difference approach has a “real-time updating” property that the Lagrange form lacks.

**Example 3.5** Use Newton’s divided differences to find the interpolating polynomial passing through \((0, 2), (1, 1), (2, 0), (3, -1)\).

The Newton divided difference triangle is

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

Reading off the coefficients, we find that the interpolating polynomials of degree 3 or less is

\[
P(x) = 2 + (-1)(x - 0) = 2 - x,
\]

agreeing with Example 3.2 but with much less work.

### 3.1.3 How many degree \(d\) polynomials pass through \(n\) points?

Theorem 3.2, the Main Theorem of polynomial interpolation, answers this question if \(0 \leq d \leq n - 1\). Given \(n = 3\) points \((0, 1), (2, 2), (3, 4)\), there is one interpolating polynomial of degree 2 or less. Example 3.1 shows that it is degree two, so there are no degree zero or one interpolating polynomials through the three data points.

How many degree 3 polynomials interpolate the above three points? One way to construct one such polynomial is clear from the previous discussion - add a fourth point. Extending the Newton divided difference triangle gives a new top coefficient. In Example 3.4 the point \((1, 0)\) was added. The resulting polynomial

\[
P_3(x) = P_2(x) + \frac{1}{2}(x - 0)(x - 2)(x - 3)
\]

passes through the three points in question, in addition to the new point \((1, 0)\). So there is at least one degree 3 polynomial passing through our three original points \((0, 1), (2, 2), (3, 4)\).

Of course, there are many different ways we could have chosen the fourth point. For example, if we keep the same \(x_4 = 1\) and simply change \(y_4\) from 0, we must get a different degree 3 interpolating polynomial, since a function can only go through one \(y\) value at \(x_4\). Now we know there
are infinitely many polynomials that interpolate the three points \((x_1, y_1), (x_2, y_2), (x_3, y_3)\), since for any fixed \(x_4\) there are infinitely many ways \(y_4\) can be chosen, each giving a different polynomial. This line of thinking shows that given \(n\) data points \((x_i, y_i)\) with distinct \(x_i\), there are infinitely many degree \(n\) polynomials passing through them.

A second look at (3.4) suggests a more direct way to produce interpolating polynomials of degree 3 through 3 points. Instead of adding a fourth point to generate a new degree 3 coefficient, why not cut out the middleman and just pencil in a new coefficient? Does the result interpolate the original 3 points? Yes, because \(P_2(x)\) does, and the new term evaluates to zero at \(x_1, x_2\) and \(x_3\). So there is really no need to construct the extra Newton divided differences for this purpose. This technique will also easily construct (infinitely many) polynomials of degree \(\geq n\) for \(n\) given data points, as illustrated in the next example.

**Example 3.6** How many degree 0, 1, 2, 3, 4 and 5 polynomials pass through the points \((-1, -5), (0, -1), (2, 1)\) and \((3, 11)\)?

The Newton’s divided difference triangle is

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

so there are no interpolating polynomials of degree 0, 1 or 2, and the single degree 3 is

\[ P_3(x) = -5 + 4(x + 1) - (x + 1)x + (x + 1)x(x - 2). \]

There are infinitely many degree 4 interpolating polynomials

\[ P_4(x) = P_3(x) + c_1(x + 1)x(x - 2)(x - 3) \]

for arbitrary \(c_1 \neq 0\), and infinitely many degree 5 interpolating polynomials

\[ P_5(x) = P_3(x) + c_2(x + 1)x^2(x - 2)(x - 3) \]

for arbitrary \(c_2 \neq 0\).

3.1.4 Code for interpolation

The Matlab program for computing the coefficients follows:

```matlab
% Program 3.1 Newton Divided Difference Interpolation Method
% Computes coefficients of interpolating polynomial
% Input: x and y are vectors containing the x and y coordinates
% of the n data points
```
% Output: coefficients c of interpolating polynomial in nested form
% Use with horner.m to evaluate interpolating polynomial
function c=newtdd(x,y,n)
for j=1:n 
    v(j,1)=y(j); % Fill in y column of Newton triangle
end
for i=2:n % For column i, 
    for j=1:n+1-i % fill in column from top to bottom 
        v(j,i)=(v(j+1,i-1)-v(j,i-1))/(x(j+i-1)-x(j));
    end
end
for i=1:n 
    c(i)=v(1,i); % Read along top of triangle
end % for output coefficients

This program can be applied to the data points of Example 3.3 to return the coefficients $1, 1/2, 1/2$ found above. These coefficients can be used in the Horner's method program to evaluate the interpolating polynomial at various $x$-values.

For example, the Matlab code segment

```matlab
x0=[0 2 3];
y0=[1 2 4];
c=newtdd(x0,y0,3);
x=0:.01:4;
y=horner(2,c,x,x0);
plot(x0,y0,'o',x,y)
```

will result in the plot shown in Figure 3.1.

Just for fun, now that we have Matlab code for finding the coefficients of the interpolating polynomial (newtdd.m) and for evaluating the polynomial (horner.m), we can put them together to build a polynomial interpolation machine. We'll use Matlab's graphics capability to plot the interpolating polynomial as it is being created. Matlab's mouse input command `ginput` is used to facilitate data entry.

```matlab
% Program 3.2. clickinterp.m Polynomial Interpolation Program
% Left click in Matlab figure window to locate data point.
% Continue, to add more points.
% Right-click to terminate program.
xl=-3;xr=3;yb=-3;yt=3;
plot([xl xr],[0 0],'k',[0 0],[yb yt],'k'); grid on;
xlist=[];ylist=[];
button=1;k=0; % initialize counter k
while(button ˜= 3) % if right click, terminate
    [xnew,ynew,button] = ginput(1); % get one mouse click
    if button == 1 % current click is a left click
        k=k+1; % k counts clicks
        xlist(k)=xnew; ylist(k)=ynew; % add new point to the list
    end
    c=newtdd(xlist,ylist,k); % get interpolation coeffs
    x=xl:.01:xr; % define x coordinates of curve
    y=horner(k-1,c,x,xlist); % get y coordinates of curve
    plot(xlist,ylist,'o',x,y,[xl xr],[0,0],'k',[0 0],[yb yt],'k');
end
```

% Program 3.2. clickinterp.m Polynomial Interpolation Program
% Left click in Matlab figure window to locate data point.
% Continue, to add more points.
% Right-click to terminate program.
xl=-3;xr=3;yb=-3;yt=3;
plot([xl xr],[0 0],'k',[0 0],[yb yt],'k'); grid on;
xlist=[];ylist=[];
button=1;k=0; % initialize counter k
while(button ˜= 3) % if right click, terminate
    [xnew,ynew,button] = ginput(1); % get one mouse click
    if button == 1 % current click is a left click
        k=k+1; % k counts clicks
        xlist(k)=xnew; ylist(k)=ynew; % add new point to the list
    end
    c=newtdd(xlist,ylist,k); % get interpolation coeffs
    x=xl:.01:xr; % define x coordinates of curve
    y=horner(k-1,c,x,xlist); % get y coordinates of curve
    plot(xlist,ylist,'o',x,y,[xl xr],[0,0],'k',[0 0],[yb yt],'k');
3.1.5 Representing functions by approximating polynomials.

A major use of polynomial interpolation is to replace evaluation of a complicated function by evaluation of a polynomial, which of course involves only elementary computer operations like addition, subtraction and multiplication. Think of this as a form of compression: something complex is replaced with something simpler, and computable, with perhaps some loss in accuracy that we will have to analyze. We begin with an example from trigonometry.

**Example 3.7** Interpolate the function $\sin x$ at 4 equally-spaced points on $[0, \pi/2]$.

Let’s compress the sine function on the interval $[0, \pi/2]$. Take four data points at equally spaced points and form the divided difference triangle. We list the values to 4 correct places:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>$\pi/6$</th>
<th>$2\pi/6$</th>
<th>$3\pi/6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000</td>
<td>0.9549</td>
<td>0.6691</td>
<td>0.2559</td>
</tr>
<tr>
<td>$\pi/6$</td>
<td>0.5000</td>
<td>−0.2443</td>
<td>−0.1139</td>
<td></td>
</tr>
<tr>
<td>$2\pi/6$</td>
<td></td>
<td>0.6232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3\pi/6$</td>
<td></td>
<td>1.0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2: Interpolation Program 3.2 using mouse input. Screenshot of Matlab code `clickinterp.m` with 4 input data points.
This is our first encounter with the concept of compression in numerical analysis. At first, interpolation may not seem like compression. After all, we take \( n \) points as input and deliver \( n \) coefficients (of the interpolating polynomial) as output. What has been compressed?

Think of the data points as coming from somewhere, say as representatives chosen from the multitude of points on a curve \( y = f(x) \). The degree \( n - 1 \) polynomial, characterized by \( n \) coefficients, is a “compressed version” of \( f(x) \), and may in some cases be used as a fairly simple representative of \( f(x) \) for computational purposes.

For example, what happens when the \( \sin \) button is pushed on a calculator? The calculator has hardware to add and multiply, but how does it compute the \( \sin \) of a number? Somehow the operation must reduce to the evaluation of a polynomial, which requires exactly those operations. By choosing data points lying on the sine curve, an interpolating polynomial can be calculated and stored in the calculator as a compressed version of the sine function.

This type of compression is “lossy compression”, meaning there will be error involved, since the sine function is not actually a polynomial. How much error is made when a function \( f(x) \) is replaced by an interpolating polynomial is the subject of the next section.

The degree 3 interpolating polynomial is therefore

\[
P_3(x) = 0 + 0.9549x - 0.2443x(x - \pi/6) - 0.1139x(x - \pi/6)(x - \pi/3)
\]

\[
= 0 + x(0.9549 + (x - \pi/6)(-0.2443 + (x - \pi/3)(-0.1139))
\]

This polynomial is graphed together with the sine function in Figure 3.3. At this level of resolution, \( P_3(x) \) and \( \sin x \) are virtually indistinguishable on the interval \([0, \pi/2]\). We have compressed the infinite amount of information held by the sine curve into a few stored coefficients and the ability to perform the 3 adds and 3 multiplies in (3.5).

How close are we to designing the \( \sin \) button on a calculator? Certainly we need to handle inputs from the entire real line. But due to the symmetries of the sine function, we have done the hard part. The interval \([0, \pi/2]\) is a so-called fundamental domain for \( \sin \), and an input from any other interval can be referred back to it. Given an input \( x \) from \([\pi/2, \pi]\), say, we can compute \( \sin x \) as \( \sin(\pi - x) \), since \( \sin \) is symmetric about \( x = \pi/2 \). Given an input \( x \) from \([\pi, 2\pi]\), \( \sin x = -\sin(2\pi - x) \) due to antisymmetry about \( x = \pi \). Finally, because \( \sin \) repeats its behavior on the interval \([0, 2\pi]\) across the entire real line, we can calculate for any input by first reducing modulo \( 2\pi \). This leads to a straightforward design for the \( \sin \) button:
% at least until we have discussed accuracy.
%Input: x
%Output: approximation for sin(x)
function y=sin1(x)
%First calculate the interpolating polynomial and store coefficients
b=pi*(0:3)/6; yb=sin(b); % b holds base points
c=newtdd(b,yb,4);
%For each input x, move x to the fundamental domain and evaluate
% the interpolating polynomial
s=1; % Correct the sign of sin
x1=mod(x,2*pi);
if x1>pi
    x1 = 2*pi-x1;
    s = -1;
end
if x1 > pi/2
    x1 = pi-x1;
end
y = s*horner(3,c,x1,b);

Figure 3.3: Degree 3 interpolation of \( \sin x \). The interpolation polynomial is plotted along with
\( y = \sin x \). Equally-spaced interpolation nodes are at 0, \( \pi/6 \), 2\( \pi/6 \) and 3\( \pi/6 \). The approximation is
very close between 0 and \( \pi/2 \).

Most of the work in Program 3.3 is to place \( x \) into the fundamental domain. Then we evaluate
the degree 3 polynomial by Horner’s method. Here is some typical output from Program 3.3.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \sin x )</th>
<th>( P_3(x) )</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8415</td>
<td>0.8411</td>
<td>0.0004</td>
</tr>
<tr>
<td>2</td>
<td>0.9093</td>
<td>0.9102</td>
<td>0.0009</td>
</tr>
<tr>
<td>3</td>
<td>0.1411</td>
<td>0.1428</td>
<td>0.0017</td>
</tr>
<tr>
<td>4</td>
<td>-0.7568</td>
<td>-0.7557</td>
<td>0.0011</td>
</tr>
<tr>
<td>14</td>
<td>0.9906</td>
<td>0.9928</td>
<td>0.0022</td>
</tr>
<tr>
<td>1000</td>
<td>0.8269</td>
<td>0.8263</td>
<td>0.0006</td>
</tr>
</tbody>
</table>
Not bad for the first try. The error is usually under 1%. In order to get enough correct digits to fill the calculator readout we’ll need to know a little more about interpolation error, the topic of the next section.

### Exercises 3.1

3.1.1. Use Lagrange interpolation to find a polynomial that passes through the points.

(a) \((0, 1), (2, 3), (3, 0)\)
(b) \((-1, 0), (2, 1), (3, 1), (5, 2)\)
(c) \((0, -2), (2, 1), (4, 4)\)

3.1.2. Repeat the previous problem, but use Newton’s divided differences.

3.1.3. Count the number of multiplications and additions necessary to evaluate the interpolating polynomial through \(n\) data points in (a) Lagrange form (b) Newton divided difference form (using nested multiplication to evaluate).

3.1.4. Let \(P_1, P_2, P_3,\) and \(P_4\) be four different points lying on a parabola \(y = ax^2 + bx + c.\) How many cubic (degree 3) polynomials pass through those four points? Explain your answer.

3.1.5. (a) How many degree \(n = 2\) polynomials pass through the four points \((-1, 3), (1, 1), (2, 3), (3, 7)\)? Write one down if possible.
(b) How many degree \(n = 3\) polynomials pass through the four points? Write one down if possible.
(c) How many degree \(n = 6\) polynomials pass through the four points? Write one down if possible.

3.1.6. (a) Find a polynomial \(P(x)\) of degree 3 or less whose graph passes through the points \((0, 0), (1, 1), (2, 2), (3, 7)\).
(b) Find 2 other polynomials (of any degree) which pass through these 4 points.
(c) Decide whether there exists a polynomial \(P(x)\) of degree 3 or less whose graph passes through the points \((0, 0), (1, 1), (2, 2), (3, 7),\) and \((4, 2)\). Be sure to justify your answer.

3.1.7. (a) Find a polynomial \(P(x)\) of degree 3 or less whose graph passes through the four data points \((-2, 8), (0, 4), (1, 2), (3, -2)\).
(b) Are there any other polynomials of degree 4 or less which pass through the 4 points in part (a)? If so, find as many as you can.

3.1.8. Decide whether there is a polynomial \(P(x)\) of degree 3 or less that passes through the five data points \((0, 2), (1, 0), (3, 2), (5, 4), (6, 2)\).

3.1.9. Write down a polynomial of degree exactly 5 that interpolates the four points \((1, 1), (2, 3), (3, 3), (4, 4)\).

3.1.10. Find \(P(0)\), where \(P(x)\) is the degree 10 polynomial which is zero at \(x = 1, \ldots, 10\) and satisfies \(P(12) = 44\).

3.1.11. Let \(P(x)\) be the degree 9 polynomial that takes the value 112 at \(x = 1\), takes the value 2 at \(x = 10\), and equals zero for \(x = 2, \ldots, 9\). Calculate \(P(0)\).

3.1.12. Give an example of the following or explain convincingly why no such example exists.

(a) A degree 6 polynomial \(L(x)\) which is zero at \(x = 1, 2, 3, 4, 5, 6\) and equal to 10 at \(x = 7\).
(b) A degree 6 polynomial \(L(x)\) which is zero at \(x = 1, 2, 3, 4, 5, 6\), equal to 10 at \(x = 7\), and equal to 70 at \(x = 8\).

3.1.13. Can a parabola (degree 2 polynomial) intersects a cubic (degree 3 polynomial) in exactly 4 points? Explain.

3.1.14. A degree \(m\) and \(n\) polynomial can intersect in at most how many points?

3.1.15. Write down a polynomial \(y = P(x)\) of degree exactly 31 that passes through the 31 data points \((-15, 225), (-14, 196), (-13, 169), (-12, 144), (-11, 121), \ldots, (15, 225)\).
3.1.1. Write a version of Program 3.2 that is a Matlab function, whose inputs x and y are equal length vectors of data points, and whose output is a plot of the interpolating polynomial. In this way, the points can be entered more accurately than by mouse input. Check your program by replicating Figure 3.4.

3.1.2. Write a Matlab function polyinterp.m that takes as input a set of \((x, y)\) interpolating points, and another \(x_0\), and outputs \(y_0\), the value of the interpolating polynomial at \(x_0\). The first line of the file should be \(function\ y_0 = polyinterp(x, y, x_0)\) where \(x\) and \(y\) are input vectors of data points. Your function may call newtdd from Program 3.1, and horner from Chapter 0, and may be structured similarly to Program 3.2, but without the graphics. Demonstrate that your function works.

3.1.3. Remodel the sin1 calculator button in Program 3.3 to build cos1, a cos button that follows the same principles. First decide on the fundamental domain for cos.

3.1.4. (a) Use the addition formulas for sin and cos to prove that \(\tan(\pi/2 - x) = \tan x\). (b) Show that \([0, \pi/4]\) can be used as a fundamental domain for \(\tan x\). (c) Design a tan button, following the principles of Program 3.3, using degree 3 polynomial interpolation on this fundamental domain.

3.2 Interpolation error

The accuracy of our sin calculator key depends on the approximation in Figure 3.3. How close is it? We showed a table indicating that for a few examples, the first two digits are fairly reliable, but after that the digits are not necessarily correct. This may be good enough for some purposes but not all. In this section we investigate ways to measure this error and determine how to make it smaller.

3.2.1 Interpolation error formula

Assume we start with a function \(y = f(x)\) and take data points from it to build an interpolating polynomial \(P(x)\), as we did with sin in Example 3.7. The interpolation error at \(x\) is \(f(x) - P(x)\), the difference between the original function that provided the data points and the interpolating polynomial, evaluated at \(x\). The interpolation error is the vertical distance between the curves in Figure 3.3. The next theorem gives a formula for the interpolation error that is usually impossible to evaluate exactly, but often can at least lead to an error bound.

**Theorem 3.3** Assume that \(P(x)\) is the (degree \(n - 1\) or less) interpolating polynomial fitting the \(n\) points \((x_1, y_1), \ldots, (x_n, y_n)\). The formula for interpolation error is

\[
 f(x) - P(x) = \frac{(x - x_1)(x - x_2) \cdots (x - x_n)}{n!} f^{(n)}(c),
\]

where \(c\) lies between the smallest and largest of the numbers \(x, x_1, \ldots, x_n\).

See Section 3.2.2 for a proof of Theorem 3.3. We can use the theorem to assess the accuracy of the sin key we built in Example 3.7. Equation (3.6) yields

\[
 \sin x - P(x) = \frac{(x - 0)(x - \pi)(x - \pi/3)(x - \pi/2)}{4!} f'''(c)
\]

where \(0 < c < \pi/2\). The limitation on the location of \(c\) doesn’t help much, since \(f'''(c) = \sin c\) varies from 0 to 1 in this range. But at worst, \(|\sin c|\) is no more than 1, so we can be assured of an upper bound on interpolation error:

\[
 |\sin x - P(x)| \leq \frac{|(x - 0)(x - \pi)(x - \pi/3)(x - \pi/2)|}{24} |1|.
\]
At $x = 1$, the worst-case error is
\[
|\sin 1 - P(1)| \leq \frac{|(1 - 0)(1 - \frac{\pi}{6})(1 - \frac{\pi}{3})(1 - \frac{\pi}{2})|}{24} \approx 0.0005348. \tag{3.7}
\]

This is an upper bound for the error, since we used a "worst case" bound for the fourth derivative. Note that the actual error at $x = 1$ was .0004, which is within the error bound given by (3.7). We can make some conclusions based on the form of the interpolation error formula. We expect smaller errors when $x$ is closer to the middle of the interval of $x_i$’s than near one of the ends, because there will be more small terms in that case. For example, we compare the above error bound to the case $x = 0.2$, which is near the left end of the range of data points. In this case the error formula is
\[
|\sin 0.2 - P(0.2)| \leq \frac{|(.2 - 0)(.2 - \frac{\pi}{6})(.2 - \frac{\pi}{3})(.2 - \frac{\pi}{2})|}{24} \approx 0.00313,
\]
about six times larger. Correspondingly, the actual error is larger, specifically
\[
\sin 0.2 - P(0.2) = 0.19867 - 0.20056 = 0.00189
\]

**Example 3.8** Find an upper bound for the difference at $x = 0.25$ and $x = 0.75$ between $f(x) = e^x$ and the polynomial that interpolates it at base points $-1, -0.5, 0, 0.5, 1$.

Note that we don’t need to construct the interpolating polynomial to find the bound. The interpolation error formula (3.6) gives
\[
f(x) - P_4(x) = \frac{(x + 1)(x + \frac{1}{2})x(x - \frac{1}{2})(x - 1)}{5!} f^{(5)}(c),
\]
where \(-1 < c < 1\). The fifth derivative is \(f^{(5)}(c) = e^c\). Since \(e^x\) is increasing with \(x\), its maximum is at the right-hand end of the interval and so \(|f^{(5)}(c)| \leq e^1\) on \([-1, 1]\). For \(-1 \leq x \leq 1\), the error formula boils down to

\[|e^x - P_4(x)| \leq \frac{(x + 1)(x + 1/2)x(x - 1/2)(x - 1)}{5!} e.\]

At \(x = 0.25\), the interpolation error has the upper bound

\[|e^{0.25} - P_4(0.25)| \leq \frac{(1.25)(0.75)(0.25)(-0.25)(-0.75)}{120} e = .000995\]

At \(x = 0.75\), the interpolation error is potentially larger:

\[|e^{0.75} - P_4(0.75)| \leq \frac{(1.75)(1.25)(0.75)(0.25)(-0.25)}{120} e = .002323\]

Note again that the interpolation error will tend to be smaller close to the center of the interpolation interval.

### 3.2.2 Proof of Newton form and error formula

In this section we explain the reasoning behind two important facts used earlier. First we establish the Newton divided difference form of the interpolating polynomial, and then we prove the interpolation error formula.

Recall what we know so far. If \(x_1, \ldots, x_n\) are \(n\) distinct points on the real line, we know there is exactly one (degree at most \(n - 1\)) interpolating polynomial \(P_{n-1}(x)\) for these points. We also know that the Lagrange interpolating formula gives such a polynomial.

The only thing we don’t know is whether the Newton divided difference formula also gives us an interpolating polynomial. Once we prove that it does in Theorem 3.7, we will know it must agree with the Lagrange version. We start by finding a formula for the degree \(n - 1\) coefficient of \(P_{n-1}\), and then specialize to the Newton form.

Consider two other interpolating polynomials: \(P_{n-2}\), which interpolates \(x_1, \ldots, x_{n-1}\), and \(Q_{n-2}\), which interpolates \(x_2, \ldots, x_n\). The following relation holds between them.

**Lemma 3.4** \((P_{n-1}(x) - P_{n-2}(x))(x_n - x_1) = (x - x_1)(Q_{n-2}(x) - P_{n-2}(x))\)

**Proof.** Since the difference between the two sides is a polynomial of degree at most \(n - 1\), it suffices to prove the two sides are equal at any \(n\) distinct points. Then by the Fundamental Theorem of Algebra, the difference must be the identically zero polynomial. The \(n\) points we choose are the \(n\) interpolation points. At \(x = x_1\), the left side is zero because both \(P_{n-1}\) and \(P_{n-2}\) interpolate the point \((x_1, y_1)\). The right side is also zero. At \(x = x_2, \ldots, x_{n-1}\), both sides are zero because all three polynomials, \(P_{n-1}, P_{n-2}, Q_{n-2}\) interpolate the points. For \(x = x_n\), both sides are identical because \(P_{n-1}(x_n) = Q_{n-2}(x_n)\). \(\square\)
Lemma 3.6 If \( f[x_1 \ldots x_n] \) denotes the degree \( n - 1 \) coefficient of the degree \( \leq n - 1 \) polynomial interpolating \( x_1, \ldots, x_n \), then

\[
f[x_1 \ldots x_n] = \frac{f[x_2 \ldots x_n] - f[x_1 \ldots x_{n-1}]}{x_n - x_1}
\]

(3.8)

Now we know how to find the degree \( n - 1 \) coefficient. Using the recursive nature of (3.8), we can evaluate it using a Newton divided difference triangle.

We therefore specialize our discussion to the Newton form:

\[
P_{n-1}(x) = a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2) + \ldots + a_n(x - x_1) \cdots (x - x_{n-1}).
\]

(3.9)

First, is it always possible to write the interpolating polynomial in this form? We show that it is possible, by solving for the coefficients \( a_i \) in terms of the data points \( y_i \). For example, substituting \( x_1 \) into the Newton form (3.9) yields

\[
a_1 = P_n(x_1) = y_1.
\]

(3.10)

Substituting \( x_2 \) yields

\[
a_1 + a_2(x_2 - x_1) = P_{n-1}(x_2) = y_2,
\]

(3.11)

so we may also solve for \( a_2 \), because \( a_1 \) is known from (3.10) and the coefficient of \( a_2 \) is nonzero. Substituting \( x_3 \), we see the same thing:

\[
a_1 + a_2(x_3 - x_1) + a_3(x_3 - x_1)(x_3 - x_2) = P_{n-1}(x_3) = y_3,
\]

(3.12)

and in general

\[
a_1 + a_2(x_m - x_1) + a_3(x_m - x_1)(x_m - x_2) + \ldots + a_m(x_m - x_1) \cdots (x_m - x_{m-1}) = P_{n-1}(x_m) = y_m.
\]

(3.13)

The important fact is that the previous \( a_i \) are known, and that the coefficient of \( a_m \) in the equation is nonzero because the points \( x_i \) are distinct.

Now that we know the Newton form always exists, we show how to compute the \( a_m \). Of course, we already know the degree \( n - 1 \) coefficient from Lemma 3.6, \( a_n = f[x_1 \ldots x_n] \). For a polynomial written in Newton form, denote by \((P_{n-1})_{m-1}\) the degree \( \leq m - 1 \) polynomial comprised of the first \( m \) terms of the Newton form of \( P_{n-1} \). Note that \((P_{n-1})_{m-1}\) interpolates \((x_1, y_1), \ldots, (x_m, y_m)\) because \( P_{n-1} \) does and because the higher terms are all zero at those points. By uniqueness of interpolating polynomial, \((P_{n-1})_{m-1}\) is the degree \( \leq m - 1 \) interpolating polynomial of \( x_1, \ldots, x_m \), and so by Lemma 3.6, \( a_m = f[x_1 \ldots x_m] \). We have shown
Theorem 3.7 Given $n$ points $(x_1, y_1), \ldots, (x_n, y_n)$ such that the $x_i$ are distinct, the polynomial $P_{n-1}(x)$ written in Newton form (3.9) has coefficients given by

$$a_m = f[x_1 \ldots x_m]$$

for $1 \leq m \leq n$. In other words, the Newton divided difference formula gives the unique degree $\leq n - 1$ interpolating polynomial through the $n$ points.

Next we prove the Interpolation Error Theorem 3.3. Consider adding one more point $x$ to the set of interpolation points. The new interpolation polynomial would be

$$P_n(t) = P_{n-1}(t) + f[x_1 \ldots x_n x](t - x_1) \cdots (t - x_n).$$

Evaluated at the extra point $x$, $P_n(x) = f(x)$, so

$$f(x) = P_{n-1}(x) + f[x_1 \ldots x_n x](x - x_1) \cdots (x - x_n). \tag{3.14}$$

This formula is true for all $x$. Now define

$$h(t) = f(t) - P_{n-1}(t) - f[x_1 \ldots x_n x](t - x_1) \cdots (t - x_n).$$

Note that $h(x) = 0$ by (3.14) and $0 = h(x_1) = \cdots = h(x_n)$ because $P_{n-1}$ interpolates $f$ at these points. Between each neighboring pair of the $n + 1$ points $x, x_1, \ldots, x_n$, there must be a new point where $h' = 0$, by Rolle’s Theorem (see Chapter 0). There are $n$ of these points. Between each pair of these, there must be a new point where $h'' = 0$; there are $n - 1$ of these. Continuing in this way, there must be one point $c$ for which $h^{(n)}(c) = 0$, where $c$ lies between the smallest and largest of $x, x_1, \ldots, x_n$. Note that

$$h^{(n)}(t) = f^{(n)}(t) - n! f[x_1 \ldots x_n x]$$

because the $n$th derivative of the polynomial $P_{n-1}(t)$ is zero. Substituting $c$ gives

$$f[x_1 \ldots x_n x] = \frac{f^{(n)}(c)}{n!},$$

which leads to

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

using (3.14).

3.2.3 Runge phenomenon

Polynomials can fit any set of data points, as Theorem 3.2 shows. However, there are some shapes that polynomials prefer above others. You can achieve a better understanding of this point by playing with Program 3.3. Try data points that cause the function to be zero at equally spaced points $x = -3, -2.5, -2, -1.5, \ldots, 2.5, 3$ except for $x = 0$, where we set a value of 1. The data points are flat along the $x$-axis except for a triangular “bump” at $x = 0$, as shown in Figure 3.5.

The polynomial that goes through points situated like this refuses to stay between 0 and 1 vertically, unlike the data points. This is an illustration of the so-called Runge phenomenon. It is usually used to describe extreme “polynomial wiggle” associated with high-degree polynomial interpolation at evenly-space points.
Figure 3.5: Interpolation of triangular bump function. The interpolating polynomial wiggles much more than the input data points.

Example 3.9 An example of large polynomial interpolation error.

An often-discussed example of this type is \( f(x) = 1/(1+12x^2) \), which we call the Runge example. This function has the same general shape as the triangular bump in Figure 3.5. Figure 3.6 shows behavior that is characteristic of the Runge phenomenon - polynomial wiggle near the ends of the interpolation interval.

As we have seen, examples with the Runge phenomenon characteristically have large error near the outside of the interval of data points. The cure for this problem is intuitive: move some of the interpolation points toward the outside of the interval, where the function producing the data can be better fit. We will see how to accomplish this in the next section.

Exercises 3.2

3.2.1. (a) Find the degree 2 interpolating polynomial through the points \((0, 0), (\pi/2, 1), \) and \((\pi, 0)\). Draw a picture. You are approximating one hump of a sine curve with a parabola. Should you draw the parabola "inside" or "outside" the sine bump?

(b) Use the result of (a) to approximate \( \sin(\pi/4) \).

(c) Give an error bound for the approximation in part (b).

(d) Using a calculator or Matlab, compare the true error to your error bound.

3.2.2. (a) Given the data points \((1, 0), (2, \ln 2), (4, \ln 4)\), find the degree 2 interpolating polynomial.

(b) Use the result of (a) to approximate \( \ln 3 \).

(c) Give an error bound for the approximation in part (b).

(d) Compare the true error to your error bound.

3.2.3. Assume that the polynomial \( P_9(x) \) interpolates the function \( f(x) = e^{2x} \) at the 10 evenly-spaced points \( x = 0, 1/9, 2/9, 3/9, \ldots, 8/9, 1 \). (a) Find an upper bound for the interpolation error \( |f(1/2) - P_9(1/2)| \). (b) How many decimal places can you guarantee to be correct if \( P_9(1/2) \) is used to approximate \( e \)?
3.2.4. You plan to find an interpolating polynomial for \( f(x) = 1/(x + 5) \) using interpolation nodes \( x = 0, 2, 4, 6, 8, 10 \). Find an upper bound (worst case) for the interpolation error at \( x = 1 \) and \( x = 5 \).

3.2.5. Assume a function \( f(x) \) has been approximated by the degree 5 interpolating polynomial \( P(x) \) using the data points \( (x_i, f(x_i)) \) where \( x_1 = .1, x_2 = .2, x_3 = .3, x_4 = .4, x_5 = .5, x_6 = .6 \). Do you expect the interpolation error \( |f(x) - P(x)| \) to be smaller for \( x = .35 \) or for \( x = .55 \)? Explain why.

3.2.6. Assume that the polynomial \( P_5(x) \) interpolates a function \( f(x) \) at the six data points \( (x_i, f(x_i)) \) with \( x \)-coordinates \( x_0 = 0, x_1 = .2, x_2 = .4, x_3 = .6, x_4 = .8 \) and \( x_5 = 1 \). Assume that the interpolation error at \( x = .3 \) is \( |f(.3) - P_5(.3)| = .01 \). Estimate (as well as you can) the new interpolation error \( |f(.3) - P_7(.3)| \) that would result if two additional interpolation points \( (x_6, y_6) = (.1, f(.1)) \) and \( (x_7, y_7) = (.5, f(.5)) \) are added. What assumptions have you made to make this estimate?

**Computer Problems 3.2**

3.2.1. (a) Use the method of divided differences to find the degree 4 interpolating polynomial \( P_4(x) \) for the data \( (0.6, 1.433329), (0.7, 1.633316), (0.8, 1.896481), (0.9, 2.247908) \) and \( (1.0, 2.718282) \). (b) Calculate \( P_4(0.82) \) and \( P_4(0.98) \). (c) The above data comes from the function \( f(x) = e^{x^2} \). Use the interpolation error formula to find upper bounds for the error at \( x = 0.82 \) and \( x = 0.98 \), and compare the bounds with the actual error. (d) Plot the actual interpolation error \( P(x) - e^{x^2} \) on the intervals \([.5, 1]\) and \([0, 2]\).

3.2.2. Plot the interpolation error of the \( \sin x \) button from Program 3.3 on the interval \([-2\pi, 2\pi]\).

### 3.3 Chebyshev interpolation

It is common to choose the base points \( x_i \) for interpolation to be evenly-spaced. In many cases, the data to be interpolated are only available in that form, for example when the data consists of instrument readings separated by a constant time interval. In other cases, for example the sin button, we are free to choose the base points as we see fit. It turns out that the choice of base point spacing can have a significant effect on the interpolation error. Chebyshev interpolation refers to a particular optimal way of spacing.
3.3.1 Chebyshev’s Theorem

The motivation for Chebyshev interpolation is to better control the maximum value of interpolation error

\[
\frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{n!}f^{(n)}(c)
\]
	on the interpolation interval. Let’s fix the interval to be \([-1, 1]\) for now.

The numerator

\[
(x-x_1)(x-x_2)\cdots(x-x_n)
\]

(3.15)
of the interpolation error formula is itself a degree \(n\) polynomial in \(x\), and has some maximum value on \([-1, 1]\). Is it possible to find particular \(x_1, \ldots, x_n\) in \([-1, 1]\) that cause the maximum value of (3.15) to be as small as possible? This is the so-called “minimax” problem of interpolation.

For example, Figure 3.7(a) shows a plot of the degree 9 polynomial (3.15) when \(x_1, \ldots, x_9\) are evenly spaced. The tendency for this polynomial to be large near the ends of the interval \([-1, 1]\) is related to the Runge phenomenon of the previous section. Figure 3.7(b) shows the same polynomial (3.15), but where the points \(x_1, \ldots, x_9\) have been chosen in a way that equalizes the size of the polynomial throughout \([-1, 1]\). These points have been chosen according to Chebyshev’s prescription.

![Figure 3.7: Part of the interpolation error formula. Plots of \((x-x_1)\cdots(x-x_9)\) for (a) 9 evenly-spaced base points \(x_i\) (b) 9 Chebyshev roots \(x_i\)](image)

These ideas are relevant to the Interpolation Error Theorem 3.3. There are three parts to the error formula (3.6), the numerator, the denominator, and the derivative. Once the degree is fixed, the only part we can optimize is the numerator

\[
(x-x_1)(x-x_2)\cdots(x-x_n).
\]

(3.16)

In Figure 3.7 we see that the maximum contribution of the term (3.16) depends on the \(x\) coordinates of the interpolation points.

For evenly-spaced points, as in Figure 3.7(a), there is relatively large error near the ends. When points are moved toward the ends, as in Figure 3.7(b), the maximum error on the interval is decreased. In fact, this precise positioning, where the base points \(x_i\) are chosen to be
Theorem 3.8  The choice of real numbers $-1 \leq x_1, \ldots, x_n \leq 1$ that makes the value of

$$\max_{-1 \leq x \leq 1} |(x - x_1) \cdots (x - x_n)|$$

as small as possible is

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

for $i = 1, \ldots, n$

and the minimum value is $1/2^{n-1}$. In fact, the minimum is achieved by

$$(x - x_1) \cdots (x - x_n) = \frac{1}{2^{n-1}} T_n(x),$$

where $T_n(x)$ denotes the degree $n$ Chebyshev polynomial.

The proof of this theorem is given below, after we establish a few properties of Chebyshev polynomials. We conclude from the theorem that interpolation error can be minimized if the interpolation base points in $[-1, 1]$ are chosen to be the roots of the degree $n$ Chebyshev interpolating polynomial $T_n(x)$. These roots are

$$x_i = \cos \left(\frac{\text{odd } \pi}{2n}\right)$$  \hspace{1cm} (3.17)

where “odd” stands for the odd numbers from 1 to $2n - 1$. Then we are guaranteed that (3.15) is less than $1/2^{n-1}$ for all $x$ in $[-1, 1]$.

Choosing the Chebyshev roots as the base points for interpolation distributes the interpolation error as equally as possible across the interval $[-1, 1]$. We will call the interpolating polynomial that uses the Chebyshev roots as base points the Chebyshev interpolating polynomial.

Example 3.10  Find a worst-case error bound for the difference on $[-1, 1]$ between $f(x) = e^x$ and the Chebyshev interpolating polynomial.

The interpolation error formula (3.6) gives

$$f(x) - P_4(x) = \frac{(x - x_1)(x - x_2)(x - x_3)(x - x_4)(x - x_5)}{5!} f^{(5)}(c),$$

where $x_1, \ldots, x_5$ are the Chebyshev zeros

$$x_1 = \cos \frac{\pi}{10}, x_2 = \cos \frac{3\pi}{10}, \ldots, x_5 = \cos \frac{9\pi}{10}$$

and where $-1 < c < 1$. According to the Chebyshev Theorem 3.8, for $-1 \leq x \leq 1$,

$$|(x - x_1) \cdots (x - x_5)| \leq \frac{1}{24}.$$

Also $|f^{(5)}| \leq e^1$ on $[-1, 1]$. Therefore the interpolation error, the difference between $e^x$ and $P_4(x)$, is

$$|e^x - P_4(x)| \leq \frac{e}{24!} = 0.00141.$$
for all \( x \) in the interval \([-1, 1]\).

Compare this result with Example 3.8. The error bound for Chebyshev interpolation for the entire interval is only slightly larger than the bound for a point near the center of the interval, when evenly-spaced interpolation is used. Near the ends of the interval, the Chebyshev error is much smaller.

Returning to the Runge Example 3.9, we can try to eliminate the Runge phenomenon by choosing the interpolation points according to Chebyshev’s idea. Figure 3.8 shows that the interpolation error is made small throughout the interval \([-1, 1]\).

![Figure 3.8: Interpolation of Runge example with Chebyshev nodes.](image)

**Figure 3.8:** Interpolation of Runge example with Chebyshev nodes. The Runge function \( f(x) = 1/(1 + 12x^2) \) is graphed along with its Chebyshev interpolation polynomial for (a) 15 points (b) 25 points. The error on \([-1, 1]\) is negligible at this resolution. Polynomial wiggle has mysteriously vanished, at least between \(-1\) and 1.

### 3.3.2 Chebyshev polynomials.

Define the \( n \)th Chebyshev polynomial by \( T_n(x) = \cos(n \arccos x) \). Despite not looking like the formula for a polynomial, it is a polynomial in the variable \( x \) for each \( n \). For example, for \( n = 0 \) it gives the degree 0 polynomial 1, and for \( n = 1 \) we get \( T_1(x) = \cos(\arccos x) = x \). For \( n = 2 \), recall the cosine addition formula \( \cos(a + b) = \cos a \cos b - \sin a \sin b \). Set \( y = \arccos x \), so that \( \cos y = x \). Then \( T_2(x) = \cos 2y = \cos^2 y - \sin^2 y = 2 \cos^2 y - 1 = 2x^2 - 1 \), a degree 2 polynomial. In general, note that

\[
\begin{align*}
T_{n+1}(x) & = \cos(n + 1)y = \cos(ny + y) = \cos ny \cos y - \sin ny \sin y \\
T_{n-1}(x) & = \cos(n - 1)y = \cos(ny - y) = \cos ny \cos y - \sin ny \sin(-y)
\end{align*}
\]  

(3.18)

Because \( \sin(-y) = \sin y \) we can add the above equations to get

\[
T_{n+1}(x) + T_{n-1}(x) = 2 \cos ny \cos y = 2xT_n(x).
\]  

(3.19)
The resulting relation

\[ T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \]  \hspace{1cm} (3.20)

is called the recursion relation for the Chebyshev polynomials. Several facts follow from (3.20):

**Figure 3.9:** Plot of the degree 1 through 5 Chebyshev polynomials. Note that \( T_n(1) = 1 \) and the maximum absolute value taken on by \( T_n(x) \) inside \([-1, 1]\) is 1.

**Fact 1.** The \( T_n \)'s are polynomials. We showed this explicitly for \( T_1 \) and \( T_2 \). Since \( T_3 \) is a polynomial combination of \( T_1 \) and \( T_2 \), \( T_3 \) is also a polynomial. The same argument goes for all \( T_n \). The first few Chebyshev polynomials are:

\[
\begin{align*}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2x^2 - 1 \\
T_3(x) &= 4x^3 - 3x
\end{align*}
\]

**Figure 3.10:** Location of zeros of the Chebyshev polynomial. The roots are the \( x \)-coordinates of evenly-spaced points around the circle. (a) degree 5 (b) degree 15 (c) degree 25.

**Fact 2.** \( \deg(T_n) = n \), and the leading coefficient is \( 2^{n-1} \). This is clear for \( n = 1 \) and \( 2 \), and the recursion relation extends the fact to all \( n \).

**Fact 3.** \( T_n(1) = 1 \) and \( T_n(-1) = (-1)^n \). Both are clear for \( n = 1 \) and \( 2 \). In general,

\[
T_{n+1}(1) = 2(1)T_n(1) - T_{n-1}(1) = 2(1) - 1 = 1,
\]
and

\[ T_{n+1}(-1) = 2(-1)T_n(-1) - T_{n-1}(-1) \]
\[ = -2(-1)^n - (-1)^{n-1} \]
\[ = (-1)^{n-1}(2 - 1) = (-1)^{n-1} = (-1)^{n+1}. \]

**Fact 4.** The maximum absolute value of \( T_n(x) \) for \(-1 \leq x \leq 1\) is 1. This follows immediately from the fact that \( T_n(x) = \cos y \) for some \( y \).

**Fact 5.** All zeros of \( T_n(x) \) are located between \(-1\) and 1. In fact, the zeros are the solution of \( 0 = \cos(n \arccos x) \). Since \( \cos y = 0 \) if and only if \( y = \text{odd integer} \cdot (\pi/2) \), we find that

\[ n \arccos x = \text{odd} \cdot \pi/2, \]
\[ x = \cos \frac{\text{odd} \cdot \pi}{2n}. \]

**Fact 6.** \( T_n(x) \) alternates between \(-1\) and 1 a total of \( n + 1 \) times. One check that this happens at \( \cos 0, \cos \pi/n, \ldots, \cos(n - 1)\pi/n, \cos \pi \).

It follows from Fact 2 above that the polynomial \( T_n(x)/2^{n-1} \) is monic (has leading coefficient 1). Since all of its roots are real, according to Fact 5, we can write \( T_n(x)/2^{n-1} \) in factored form as \( (x - x_1) \cdots (x - x_n) \) where the \( x_i \) are the Chebyshev nodes as described in Theorem 3.8.

Given these facts, we are prepared to prove Chebyshev’s theorem.

**Proof of Theorem 3.8.** Let \( P_n(x) \) be a monic polynomial with an even smaller absolute maximum on \([-1, 1]\), or in other words, \( |P_n(x)| < 1/2^{n-1} \) for \(-1 \leq x \leq 1\). This assumption leads to a contradiction. Since \( T_n(x) \) alternates between \(-1\) and 1 a total of \( n + 1 \) times (Fact 6), at these \( n + 1 \) points, the difference \( P_n - T_n/2^{n-1} \) is alternately positive and negative. Therefore \( P_n - T_n/2^{n-1} \) must cross zero at least \( n \) times, that is, must have at least \( n \) roots. This contradicts the fact that because \( P_n \) and \( T_n/2^{n-1} \) are monic, their difference is of degree \( \leq n - 1 \).

\[ \square \]

### 3.3.3 Change of interval

So far our discussion of Chebyshev interpolation has been restricted to the interval \([-1, 1]\), because Theorem 3.8 is most easily stated for this interval. Next we will move the whole methodology to a general interval \([a, b]\).

The base points are moved so that they have the same relative positions in \([a, b]\) that they had in \([-1, 1]\). It is best to think of doing this in two steps: (1) **stretch** the points by the factor \((b - a)/2\) (the ratio of the two interval lengths), and (2) **translate** the points by \((b + a)/2\) to move the center of mass from 0 to the midpoint of \([a, b]\). In other words, move from the original points

\[ \cos \frac{\text{odd} \pi}{2n} \]

to

\[ \frac{b - a}{2} \cos \frac{\text{odd} \pi}{2n} + \frac{b + a}{2}. \]
With the new Chebyshev base points \( x_1, \ldots, x_n \) in \([a, b]\), the corresponding upper bound on the numerator of the interpolation error formula is

\[
|(x-x_1)\cdots(x-x_n)| \leq \frac{(b-a)^n}{2^n-1}
\]  

(3.21)

The next example illustrates the usage of Chebyshev interpolation in a general interval.

Example 3.11 Find the 4 Chebyshev base points for interpolation on the interval \([0, \pi/2]\), and find an upper bound for the Chebyshev interpolation error for \( f(x) = \sin x \) on the interval.

This is a second attempt, as we attempted to do this with evenly-spaced base points in Example 3.7. The Chebyshev base points are

\[
x_1 = \frac{\pi}{4} + \frac{\pi}{4} \cos \left( \frac{\text{odd } \pi}{2(4)} \right), \quad x_2 = \frac{\pi}{4} + \frac{\pi}{4} \cos \frac{3\pi}{8}, \quad x_3 = \frac{\pi}{4} + \frac{\pi}{4} \cos \frac{5\pi}{8}, \quad x_4 = \frac{\pi}{4} + \frac{\pi}{4} \cos \frac{7\pi}{8}
\]

The worst-case interpolation error for \( 0 \leq x \leq \pi/2 \) is

\[
|\sin x - P_3(x)| = \frac{|(x-x_1)(x-x_2)(x-x_3)(x-x_4)|}{4!} f^{(4)}(c) \leq \frac{\left(\frac{\pi}{2} - 0\right)^4}{4!} \approx 0.00198
\]

The Chebyshev interpolating polynomial for this example is evaluated at several points in the following table:
As shown in this section, Chebyshev interpolation is a good way to turn general functions into a small number of floating point operations, for ease of computation. An upper bound for the error made is easily available, is usually smaller than for evenly-spaced interpolation, and can be made as small as desired.

Although we have used the sine function to demonstrate this process for a general function, a different approach is taken to construct the actual "sine button" on most calculators and canned software. Special properties of the sine function allow it to be approximated by a simple Taylor expansion slightly altered to take rounding effects into account. Because sine is an odd function, the even numbered terms in its Taylor series around zero are missing, making it especially efficient to calculate.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\sin x$</th>
<th>$P_3(x)$</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8415</td>
<td>0.8408</td>
<td>0.0007</td>
</tr>
<tr>
<td>2</td>
<td>0.9093</td>
<td>0.9097</td>
<td>0.0004</td>
</tr>
<tr>
<td>3</td>
<td>0.1411</td>
<td>0.1420</td>
<td>0.0009</td>
</tr>
<tr>
<td>4</td>
<td>−0.7568</td>
<td>−0.7555</td>
<td>0.0013</td>
</tr>
<tr>
<td>14</td>
<td>0.9906</td>
<td>0.9917</td>
<td>0.0011</td>
</tr>
<tr>
<td>1000</td>
<td>0.8269</td>
<td>0.8261</td>
<td>0.0008</td>
</tr>
</tbody>
</table>

The interpolation errors are well below the worst-case estimate above. Figure 3.11 plots the interpolation error as a function of $x$ on the interval $[0, \pi/2]$, compared with the same for evenly-spaced interpolation. The Chebyshev error (lower curve) is a bit smaller, and distributed more evenly throughout the interpolation interval.

Example 3.12  Design a sin button that will give output correct to 10 decimal places.

Thanks to our work earlier on setting up a fundamental domain for sine, we can continue to concentrate on the interval $[0, \pi/2]$. Repeat the previous calculation, but leave $n$, the number of base points, as an unknown to be determined. The maximum interpolation error for the polynomial $P_{n-1}(x)$ on the interval $[0, \pi/2]$ is

$$\left| \sin x - P_{n-1}(x) \right| = \left| \frac{(x-x_1)\cdots(x-x_n)}{n!} \right| f^{(n)}(c) \leq \frac{\left( \frac{\pi}{2} - 0 \right)^n}{n!2^{n-1}}$$

This equation is not simple to solve for $n$, but a little trial and error finds that for $n = 9$, the error bound is $\approx 0.1224 \times 10^{-8}$, and for $n = 10$ it is $\approx 0.4807 \times 10^{-10}$. The latter meets our criterion for 10 correct decimal places. Figure 3.11(b) compares the actual error of the Chebyshev interpolation polynomial with the error of the evenly-spaced interpolation polynomial.
The 10 Chebyshev base points on \([0, \pi/2]\) are \(\pi/4 + (\pi/4) \cos(\text{odd}\pi/20)\). The button can be designed by storing the 10 \(y\)-values for sine at the base points, and do a Horner method evaluation for each button press.

The following Matlab code, \texttt{sin2.m}, carries this out. The code is a bit awkward as written - we have to do 10 \(\sin\) evaluations, at the 10 Chebyshev nodes, in order to set up the interpolating polynomial to approximate \(\sin\) at one point. Of course, in a real implementation, these numbers would be computed once and stored.

%Program 3.4 Building a sin calculator button, attempt #2
%Approximates sin curve with degree 9 polynomial
%Input: x
%Output: approximation for sin(x), correct to 10 decimal places
function y=sin2(x)

%First calculate the interpolating polynomial and store coefficients
n=10; b=1:2:2*n-1; b=pi/4+(pi/4)*cos(b*pi/(2*n));
yb=sin(b); % b holds Chebyshev base points
% b holds Chebyshev base points
if x1 > pi/2
    x1 = pi-x1;
end

y = s*horner(n-1,c,x1,b);

Exercises 3.3

3.3.1. Write down the 8 interpolation nodes (base points) needed to use Chebyshev minimax interpolation with a degree 7 interpolating polynomial on the interval \([-10, 10]\).

3.3.2. List the interpolation nodes necessary to construct the degree 10 Chebyshev interpolation on the interval \([4, 12]\).

3.3.3. Assume Chebyshev interpolation is used to find a fourth-degree interpolating polynomial \(Q_4(x)\) on the interval \([.6, 1.0]\) for the function \(f(x) = e^x\). Use the interpolation error formula to find a worst-case estimate for the error \(|e^x - Q_4(x)|\) which is valid for \(x\) throughout the interval \([.6, 1.0]\). How many digits after the decimal point will be correct when using \(Q_4(x)\) to approximate \(e^x\)?

3.3.4. Suppose you are designing the \texttt{ln} key for a calculator whose display shows (a maximum of six) digits to the right of the decimal point.

(a) Begin by designing a method for evaluating \(\ln x\) for \(1 \leq x \leq e\). Using Chebyshev minimax interpolation, how many numbers will need to be stored in memory, and how many additions and multiplications will be required for each press of the \texttt{ln} key?

(b) Discuss possible approaches for extending the allowable range of inputs to all \(x\). Goals are to minimize storage and calculation time as in part (a).

3.3.5. Assume you are to use Chebyshev minimax interpolation to find a degree 3 interpolating polynomial \(Q_3(x)\) that approximates the function \(f(x) = x^{-3}\) on the interval \([3, 4]\).
(a) Write down the \((x, y)\) points that will serve as interpolation nodes for \(Q_3\).
(b) Find a worst-case estimate for the error \(|x^{-3} - Q_3(x)|\) which is valid for all \(x\) in the interval \([3, 4]\). How many digits after the decimal point will be correct when using \(Q_3(x)\) to approximate \(x^{-3}\)?

3.3.6. The recursion relation for Chebyshev polynomials is \(T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)\).

(a) What is the degree of the polynomial \(T_k(x)\)?
(b) Find a formula for the leading coefficient of \(T_k(x)\).
(c) Find the number \(T_{999}(1)\).
(d) Find the numbers \(T_{999}(-1), T_{1000}(-1), T_{999}(0),\) and \(T_{1000}(0)\).
(e) Find the numbers \(T_{999}(-1/2)\) and \(T_{1000}(-1/2)\).

3.3.7. Find the maximum error on \([0, 2]\) when the degree 3 Chebyshev interpolating polynomial is used to approximate \(f(x) = \sin \pi x\).

3.3.8. Assume you want to use Chebyshev interpolation to approximate the function \(f(x) = x^{-1}\) to 6 correct decimal places on the entire interval \([1, 2]\) with a polynomial. Find the degree of such a polynomial. How many additions and multiplications will be necessary to do the approximation for each input \(x\)? How will your answers change if the interval \([1, 10]\) is used instead?

\section*{Computer Problems 3.3}

3.3.1. Rebuild Program 3.3 to implement the Chebyshev interpolating polynomial with 4 base points on the interval \([0, \pi/2]\). (Only one line of code needs to be changed.) Then plot the polynomial and compare it with the sine function on the interval \([-2, 2]\).

3.3.2. Build a Matlab program to evaluate \(\cos\) correct to 10 decimal places using Chebyshev interpolation. It should be possible to achieve this goal for inputs between \(-10^4\) and \(10^4\). You may want to use some of the Matlab code written in this chapter.

3.3.3. Repeat the problem 3.3.2 for \(\ln x\), for inputs \(x\) between \(10^{-4}\) and \(10^4\). Use \([1, e]\) as fundamental domain. Use Chebyshev interpolation. What is the degree of the interpolation polynomial that guarantees 10 correct digits? Your Matlab program should begin by finding the integer \(k\) such that \(e^k < x < e^{k+1}\). Then \(xe^{-k}\) lies in the fundamental domain. Demonstrate the accuracy of your program by comparing to Matlab’s \(\log\) command.

3.3.4. Let \(f(x) = e^{\|x\|}\). Compare evenly-spaced interpolation to Chebyshev interpolation by plotting degree \(n\) polynomials of both types on the interval \([-1, 1]\), for \(n = 10\) and 20. For evenly-spaced interpolation, the left and right interpolation base points should be \(-1\) and 1. By sampling at 0.01 stepsize, create the empirical interpolation errors for each type, and plot a comparison. Can the Runge phenomenon be observed in this problem?

3.3.5. Repeat the previous problem for \(e^{-x^2}\).

\section*{3.4 Cubic splines}

\textbf{SPLINES} represent an alternative approach to data interpolation. In polynomial interpolation, a single formula, given by a polynomial, is used to meet all data points. The idea of splines is to use several formulas, each of a low degree polynomial, to pass through the data points.

The simplest example of a spline is a linear spline, in which one “connects the dots” with straight line segments. Assume we are given a set of data points \((x_1, y_1), \ldots, (x_n, y_n)\) with \(x_1 < \ldots < x_n\). A linear spline consists of the \(n - 1\) line segments that are drawn between neighboring pairs of points. Figure 3.12(a) shows a linear spline where between each neighboring pair of points \((x_i, y_i), (x_{i+1}, y_{i+1})\), the linear function \(y = a_i + b_ix\) is drawn through the two points. The given data points in Figure 3.12 are \((1, 2), (2, 1), (4, 4),\) and \((5, 3),\) and the linear spline is given by

\begin{align*}
S_1(x) &= 2 - (x - 1) \text{ on } [1, 2] \\
S_2(x) &= 1 + \frac{3}{2}(x - 2) \text{ on } [2, 4] \\
S_3(x) &= 4 - (x - 4) \text{ on } [4, 5]
\end{align*}
Figure 3.12: Splines through four data points. (a) Linear spline through \((1, 2), (2, 1), (4, 4)\) and \((5, 3)\) consists of three linear polynomials given by (3.22). (b) Cubic spline through the same points, given by (3.23)

The linear spline successfully interpolates an arbitrary set of \(n\) data points. However, linear splines lack smoothness. Cubic splines are meant to address this shortcoming of linear splines. A cubic spline replaces linear functions between the data points by degree 3 (cubic) polynomials.

An example of a cubic spline which interpolates the same points \((1, 2), (2, 1), (4, 4)\) and \((5, 3)\) is shown in Figure 3.12(b). The equations defining the spline are

\[
S_1(x) = 2 - \frac{13}{8}(x - 1) + 0(x - 1)^2 + \frac{5}{8}(x - 1)^3 \quad \text{on} \quad [1, 2]
\]

\[
S_2(x) = 1 + \frac{1}{4}(x - 2) + \frac{15}{8}(x - 2)^2 - \frac{5}{8}(x - 2)^3 \quad \text{on} \quad [2, 4]
\]

\[
S_3(x) = 4 + \frac{1}{4}(x - 4) - \frac{15}{8}(x - 4)^2 + \frac{5}{8}(x - 4)^3 \quad \text{on} \quad [4, 5].
\]  

(3.23)

Note in particular the smooth transition from one \(S_i\) to the next at the base points or “knots” at \(x = 2\) and \(x = 4\). This is achieved by arranging for the neighboring pieces \(S_i\) and \(S_{i+1}\) of the spline to have the same 0th, 1st, and 2nd derivatives when evaluated at the knots. Just how to do this is the topic of the next section.

Given \(n\) points \((x_1, y_1), \ldots, (x_n, y_n)\), there is obviously one and only one linear spline through the data points. This will not be true for cubic splines. We will find there are infinitely many through any set of data points. Because of this, we will add extra conditions when it is necessary to nail down a particular spline of interest.

### 3.4.1 Properties of splines

To be a little more precise about the properties of a cubic spline we make the following definition. Assume that we are given the \(n\) data points \((x_1, y_1), \ldots, (x_n, y_n)\) to interpolate. A **cubic spline**
Chapter 3 \textbf{INTERPOLATION}

$S(x)$ through the data points $(x_1, y_1), \ldots, (x_n, y_n)$ is a set of cubic polynomials

\begin{align*}
S_1(x) &= y_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 \text{ on } [x_1, x_2] \\
S_2(x) &= y_2 + b_2(x - x_2) + c_2(x - x_2)^2 + d_2(x - x_2)^3 \text{ on } [x_2, x_3] \\
&\vdots \\
S_{n-1}(x) &= y_{n-1} + b_{n-1}(x - x_{n-1}) + c_{n-1}(x - x_{n-1})^2 + d_{n-1}(x - x_{n-1})^3 \text{ on } [x_{n-1}, x_n]
\end{align*}

with the following properties:

\textbf{Property 1.} $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, \ldots, n - 1$.

\textbf{Property 2.} $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, \ldots, n - 1$.

\textbf{Property 3.} $S''_{i-1}(x_i) = S''_i(x_i)$ for $i = 2, \ldots, n - 1$.

Property 1 guarantees the spline $S(x)$ interpolates the data points. Property 2 forces the slopes of neighboring parts of the spline to agree where they meet, and Property 3 does the same for the curvature, represented by the second derivative.

\begin{example}
\textbf{Example 3.13} Check that $\{S_1, S_2, S_3\}$ in (3.23) satisfies all spline properties for the data points $(1, 2)$, $(2, 1)$, $(4, 4)$, $(5, 3)$.
\end{example}

We will check all three properties.

\textit{Property 1.} There are $n = 4$ data points. We must check

\begin{align*}
S_1(1) &= 2 \quad \text{and} \quad S_1(2) = 1 \\
S_2(2) &= 1 \quad \text{and} \quad S_2(4) = 4 \\
S_3(4) &= 4 \quad \text{and} \quad S_3(5) = 3
\end{align*}

These follow easily from the defining equations (3.23).

\textit{Property 2.} The first derivatives of the spline functions are

\begin{align*}
S'_1(x) &= -\frac{13}{8} + \frac{15}{8}(x - 1)^2 \\
S'_2(x) &= \frac{1}{4} + \frac{15}{4}(x - 2) - \frac{15}{8}(x - 2)^2 \\
S'_3(x) &= \frac{1}{4} - \frac{15}{4}(x - 4) + \frac{15}{8}(x - 4)^2
\end{align*}

We must check $S'_1(2) = S'_2(2)$ and $S'_2(4) = S'_3(4)$. The first is

\[-\frac{13}{8} + \frac{15}{8} = \frac{1}{4},\]

and the second is

\[\frac{1}{4} + \frac{15}{4}(4 - 2) - \frac{15}{8}(4 - 2)^2 = \frac{1}{4},\]

both of which check.
Property 3. The second derivatives are

\[
S''_1(x) = \frac{15}{4}(x - 1)
\]
\[
S''_2(x) = \frac{15}{4} - \frac{15}{4}(x - 2)
\]
\[
S''_3(x) = -\frac{15}{4} + \frac{15}{4}(x - 4)
\] (3.25)

We must check \(S''_1(2) = S''_2(2)\) and \(S''_2(4) = S''_3(4)\), both of which are true. Therefore (3.23) is a cubic spline.

Constructing a spline from a set of data points means finding the coefficients \(b_i, c_i, d_i\) that make Properties 1 - 3 hold. Before we discuss how to determine the unknown coefficients \(b_i, c_i, d_i\) of the spline, let us count the number of conditions imposed by the definition. The first half of Property 1 is already reflected in the form (3.24); it says that the constant term of the cubic \(S_i\) must be \(y_i\). The second half of Property 1 consists of \(n - 1\) separate equations that must be satisfied by the coefficients, which we consider as unknowns. Each of Properties 2 and 3 add \(n - 2\) additional equations, for a total of \(n - 1 + 2(n - 2) = 3n - 5\) independent equations to be satisfied.

How many unknown coefficients are there? For each part \(S_i\) of the spline, three coefficients \(b_i, c_i, d_i\) are needed, for a total of \(3(n - 1) = 3n - 3\). Therefore solving for the coefficients is a problem of solving \(3n - 5\) linear equations in \(3n - 3\) unknowns. Unless there are inconsistent equations in the system (and there are not), the system of equations is underdetermined, and so has infinitely many solutions. In other words, there are infinitely many cubic splines passing through the arbitrary set of data points \((x_1, y_1), \ldots, (x_n, y_n)\).

Users of splines normally exploit this embarrassment of riches by adding two extra constraints to the \(3n - 5\) equations, to arrive at a system of \(m\) equations in \(m\) unknowns, where \(m = 3n - 3\). Aside from allowing the user to constrain the spline to given specifications, narrowing the field to a single solution simplifies computing and describing the result.

The simplest way of adding 2 more constraints is to require, in addition to the previous \(3n - 5\) constraints, that the spline \(S(x)\) has an inflection point at each end of the defining interval \([x_1, x_n]\).

The constraints added to Properties 1 - 3 are

**Property 4a. (Natural spline)** \(S''_1(x_1) = 0\) and \(S''_{n-1}(x_n) = 0\).

A cubic spline that satisfies these two additional conditions is called a natural cubic spline. Note that (3.23) is a natural cubic spline, since it is easily verified from (3.25) that \(S''_1(1) = 0\) and \(S''_2(5) = 0\).

There are several other ways to add two additional conditions. Usually, as in the natural spline case, they determine extra properties of the left and right ends of the spline, so they are called end conditions. We will take up this topic in the next section, but for now we concentrate on natural cubic splines.

Now that we have the right number of equations, \(3n - 3\) equations in \(3n - 3\) unknowns, we can write a Matlab function to solve them for the spline coefficients. First we write out the equations in
the unknowns \( b_i, c_i, d_i \). Part 2 of Property 1 implies the \( n - 1 \) equations:

\[
y_2 = S_1(x_2) = y_1 + b_1(x_2 - x_1) + c_1(x_2 - x_1)^2 + d_1(x_2 - x_1)^3 \quad (3.26)
\]

\[
\vdots
\]

\[
y_n = S_{n-1}(x_n) = y_{n-1} + b_{n-1}(x_n - x_{n-1}) + c_{n-1}(x_n - x_{n-1})^2 + d_{n-1}(x_n - x_{n-1})^3
\]

Property 2 generates the \( n - 2 \) equations

\[
0 = S'_1(x_2) - S'_2(x_2) = b_1 + 2c_1(x_2 - x_1) + 3d_1(x_2 - x_1)^2 - b_2 \quad (3.27)
\]

\[
\vdots
\]

\[
0 = S'_{n-2}(x_{n-1}) - S'_{n-1}(x_{n-1}) = b_{n-2} + 2c_{n-2}(x_{n-1} - x_{n-2}) + 3d_{n-2}(x_{n-1} - x_{n-2})^2 - b_{n-1},
\]

and Property 3 implies the \( n - 2 \) equations

\[
0 = S''_1(x_2) - S''_2(x_2) = 2c_1 + 6d_1(x_2 - x_1) - 2c_2 \quad (3.28)
\]

\[
\vdots
\]

\[
0 = S''_{n-2}(x_{n-1}) - S''_{n-1}(x_{n-1}) = 2c_{n-2} + 6d_{n-2}(x_{n-1} - x_{n-2}) - 2c_{n-1}
\]

Instead of solving the equations in this form, the system can be simplified drastically by decoupling the equations. With a little algebra, a much smaller system of equations in the \( c_i \) can be solved first, followed by explicit formulas for the \( b_i \) and \( d_i \) in terms of the known \( c_i \).

It is conceptually simpler if an extra unknown \( c_n = S''_{n-1}(x_n)/2 \) is introduced. In addition, we introduce the shorthand notation \( \delta_i = x_{i+1} - x_i \) and \( \Delta_i = y_{i+1} - y_i \). Then (3.28) can be solved for the coefficients

\[
d_i = \frac{c_{i+1} - c_i}{3\delta_i} \quad \text{for } i = 1, \ldots, n - 1.
\quad (3.29)
\]

Solving (3.26) for \( b_i \) yields

\[
b_i = \frac{\Delta i}{\delta i} - c_i\delta_i - d_i\delta_i^2
\]

\[
= \frac{\Delta i}{\delta i} - c_i\delta_i - \frac{\delta_i}{3}(c_{i+1} - c_i)
\]

\[
= \frac{\Delta i}{\delta i} - \frac{\delta_i}{3}(2c_i + c_{i+1})
\quad (3.30)
\]

for \( i = 1, \ldots, n - 1 \).

Substituting (3.29) and (3.30) into (3.27) results in the following \( n - 2 \) equations in \( c_1, \ldots, c_n \):

\[
\delta_1 c_1 + 2(\delta_1 + \delta_2)c_2 + \delta_2 c_3 = 3 \left( \frac{\Delta_2}{\delta_2} - \frac{\Delta_1}{\delta_1} \right)
\]

\[
\vdots
\]

\[
\delta_{n-2} c_{n-2} + 2(\delta_{n-2} + \delta_{n-1})c_{n-1} + \delta_{n-1} c_n = 3 \left( \frac{\Delta_{n-1}}{\delta_{n-1}} - \frac{\Delta_{n-2}}{\delta_{n-2}} \right)
\]

Two more equations are given by the natural spline conditions (Property 4a):

\[
S''_1(x_1) = 0 \rightarrow 2c_1 = 0
\]

\[
S''_{n-1}(x_n) = 0 \rightarrow 2c_n = 0
\]
This gives a total of \( n \) equations in \( n \) unknowns \( c_i \), which can be written in the matrix form

\[
\begin{pmatrix}
1 & 0 & 0 \\
\delta_1 & 2\delta_1 + 2\delta_2 & \delta_2 \\
0 & \delta_2 & 2\delta_2 + 2\delta_3 \\
& \ddots & \ddots \\
0 & \delta_{n-2} & 2\delta_{n-2} + 2\delta_{n-1} & \delta_{n-1} \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
c_1 \\
c_2 \\
\vdots \\
c_n
\end{pmatrix}
= \begin{pmatrix}
0 \\
3 \left( \frac{\Delta_2}{\delta_1} - \frac{\Delta_1}{\delta_2} \right) \\
\vdots \\
3 \left( \frac{\Delta_{n-1}}{\delta_{n-2}} - \frac{\Delta_{n-2}}{\delta_{n-3}} \right) \\
0
\end{pmatrix}
\]  
(3.31)

After finding \( c_1, \ldots, c_n \) from (3.31), \( b_1, \ldots, b_{n-1} \) and \( d_1, \ldots, d_{n-1} \) are found from (3.29) and (3.30).

Note that (3.31) is always solvable for the \( c_i \). The coefficient matrix is strictly diagonally dominant, so by Theorem 2.9 of Chapter 2, there is a unique solution for the \( c_i \), and therefore also for the \( b_i \) and \( d_i \). We have proved:

**Theorem 3.9** Let \( n \geq 2 \). For a set of data points \((x_1, y_1), \ldots, (x_n, y_n)\) with distinct \( x_i \), there is a unique natural cubic spline fitting the points.

### 3.4.2 Calculation of the cubic spline.

We summarize the calculations above and implement them in Matlab code. Assume we are given \( n \) data points \((x_1, y_1), \ldots, (x_n, y_n)\) with distinct \( x_i \). The coefficients of the spline functions (3.24)

\[
S_1(x) = y_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 \quad \text{on } [x_1, x_2] \\
S_2(x) = y_2 + b_2(x - x_2) + c_2(x - x_2)^2 + d_2(x - x_2)^3 \quad \text{on } [x_2, x_3] \\
\vdots \\
S_{n-1}(x) = y_{n-1} + b_{n-1}(x - x_{n-1}) + c_{n-1}(x - x_{n-1})^2 + d_{n-1}(x - x_{n-1})^3 \quad \text{on } [x_{n-1}, x_n]
\]

are found as follows.

For \( i = 1, \ldots, n - 1 \), set

\[
a_i = y_i.
\]  
(3.32)

Set \( \delta_i = x_{i+1} - x_i \) and \( \Delta_i = y_{i+1} - y_i \) for \( i = 1, \ldots, n - 1 \) and solve the \( n \times n \) linear system (3.31) for \( c_1, \ldots, c_n \). Then (3.29) and (3.30) give

\[
d_i = \frac{c_{i+1} - c_i}{3\delta_i}
\]  
(3.33)

and

\[
b_i = \frac{\Delta_i}{\delta_i} - \frac{\delta_i}{3} (2c_i + c_{i+1})
\]  
(3.34)

for \( i = 1, \ldots, n - 1 \). This gives the coefficients of the natural cubic spline.

**Example 3.14** Find the natural cubic spline through \((0, 3), (1, -2), (2, 1)\).
The $x$-coordinates are $x_1 = 0, x_2 = 1, x_2 = 2$. The $y$-coordinates are $a_1 = y_1 = 3, a_2 = y_2 = 2$, and the differences are $\delta_1 = \delta_2 = 1, \Delta_1 = -5, \Delta_2 = 3$. The tridiagonal matrix equation (3.31) is

$$
\begin{bmatrix}
1 & 0 & 0 \\
1 & 4 & 1 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
c_1 \\
c_2 \\
c_3
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
24 \\
0
\end{bmatrix}.
$$

The solution is $[c_1, c_2, c_3] = [0, 6, 0]$. Now (3.33) and (3.34) yield

\begin{align*}
d_1 &= \frac{c_2 - c_1}{3\delta_1} = \frac{\frac{2}{3}}{3} = 2 \\
d_2 &= \frac{c_3 - c_2}{3\delta_2} = \frac{-\frac{6}{3}}{3} = -2 \\
b_1 &= \frac{\Delta_1}{\delta_1} - \frac{\delta_1}{3}(2c_1 + c_2) = -5 - \frac{1}{3}(6) = -7 \\
b_2 &= \frac{\Delta_2}{\delta_2} - \frac{\delta_2}{3}(2c_2 + c_3) = 3 - \frac{1}{3}(12) = -1
\end{align*}

Therefore the cubic spline is

\begin{align*}
S_1(x) &= 3 - 7x + 0x^2 + 2x^3 \text{ on } [0, 1] \\
S_2(x) &= -2 - 1(x - 1) + 6(x - 1)^2 - 2(x - 1)^3 \text{ on } [1, 2].
\end{align*}

Matlab code for this calculation follows. For different (not natural) endpoint conditions, discussed in the next section, the top and bottom rows of (3.31) are replaced by other appropriate rows.

```matlab
%Program 3.5 Calculation of spline coefficients
function coeff=splinecoeff(x,y)
%Calculates coefficients of cubic spline
%Input: x,y vectors of data points
% plus two optional extra data v1, vn
%Output: matrix of coefficients b1,c1,d1;b2,c2,d2;...
 n=length(x);v1=0;vn=0;
 A=zeros(n,n); % matrix A is nxn
 r=zeros(n,1);
 for i=1:n-1 % define the deltas
 dx(i) = x(i+1)-x(i); dy(i)=y(i+1)-y(i);
 end
 for i=2:n-1 % load the A matrix
 A(i,i-1:i+1)=[dx(i-1) 2*(dx(i-1)+dx(i)) dx(i)];
 r(i)=3*(dy(i)/dx(i)-dy(i-1)/dx(i-1)); % right-hand side
 end
 % Set endpoint conditions
 % Use only one of following 5 pairs:
 A(1,1) = 1; % natural spline conditions
 A(n,n) = 1;
 A(1,1)=2;r(1)=v1; % curvature-adj conditions
```

% $A(n,n)=2; r(n)=v_n$
% $A(1,1:2)=[2* \text{dx}(1) \ \text{dx}(1)]; r(1)=3* \left( \frac{\text{dy}(1)}{\text{dx}(1)}-v_1 \right)$; % clamped conditions
% $A(n,n-1:n)=[\text{dx}(n-1) \ 2* \text{dx}(n-1)]; r(n)=3*(v_n-\text{dy}(n-1)/\text{dx}(n-1))$
% $A(1,1:2)=[1 \ -1]$; % parabol-term conditions, for $n>=3$
% $A(n,n-1:n)=[1 \ -1]$; % not-a-knot conditions, for $n>=4$
% $A(n,n-2:n)=[\text{dx}(n-1) \ -\text{dx}(n-2)+\text{dx}(n-1) \ \text{dx}(n-2)]$
coeff=zeros(n,3);
coeff(:,2)=A\r; % solve for $c$ coefficients
for i=1:n-1 % solve for $b$ and $d$
    coeff(i,3)=(coeff(i+1,2)-coeff(i,2))/(3*\text{dx}(i));
    coeff(i,1)=\text{dy}(i)/\text{dx}(i)-\text{dx}(i)*(2*coeff(i,2)+coeff(i+1,2))/3;
end
coeff=coeff(1:n-1,1:3);

We have taken the liberty to list other choices for end conditions, although they are commented
out for now. The alternate conditions will be discussed in the next section. Another Matlab function,
titled splineplot.m, calls splinecoeff.m to get the coefficients, and then plots the cubic spline.

Program 3.6 Cubic spline plot
function splineplot(x,y)
    %Plots spline from data points
    %Input: x, y vectors of data points
    %Output: none
    n=length(x);
    coeff=splinecoeff(x,y);
    cla;hold on;       %clear figure window and turn hold on
    for i=1:n-1
        x0=linspace(x(i),x(i+1),100);
        dx=x0-x(i);
        y0=coeff(i,3)*dx; % evaluate using horner’s method
        y0=(y0+coeff(i,2)).*dx;
        y0=(y0+coeff(i,1)).*dx+y(i);
        plot([x(i) x(i+1)],[y(i) y(i+1)],’o’,x0,y0)
    end
    hold off

Figure 3.13(a) shows a natural cubic spline generated by splinecoeff.m.

3.4.3 Endpoint conditions

The two extra conditions specified in Property 4a above are called the “endpoint conditions” for a
natural spline. Requiring that these be satisfied along with Properties 1 through 3 narrows the field
to exactly one cubic spline, according to Theorem 3.9. It turns out that there are many different
versions of Property 4, meaning many other pairs of endpoint conditions, for which an analogous
theorem holds. In this section we present a few of the more popular ones.

Property 4b. (Curvature-adjusted cubic spline) The first alternative to a natural cubic spline is a
simple generalization. The values of $S''(x_1)$ and $S_n''(x_n)$ are set to arbitrary values, chosen
by the user, instead of zero. This choice corresponds to setting the desired curvatures at
the left and right endpoints of the spline. In terms of (3.24) this translates to the two extra
Chapter 3  INTERPOLATION

Figure 3.13: Cubic splines through 6 points. The plots are generated by `splinecoeff.m` with input vectors $x = [0, 1, 2, 3, 4, 5]$ and $y = [3, 1, 6, 1, 2, 0]$. (a) Natural cubic spline (notice inflection points at ends) (b) Not-a-knot cubic spline (single cubic equation on [0, 2], and on [3, 5]) (c) Parabolically terminated spline (d) Clamped cubic spline (clamped at slope 0 at both ends)

conditions

\[ \begin{align*}
2c_1 &= v_1 \\
2c_n &= v_n,
\end{align*} \]

where $v_1, v_n$ denote the desired values. The equations turn into the two tableau rows

\[
\begin{pmatrix}
2 & 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 & | & v_1 \\
0 & 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 2 & | & v_n
\end{pmatrix}
\]

to replace the top and bottom rows of (3.31), which were added for the natural spline. Notice that the new coefficient matrix is again strictly diagonally dominant, so that a generalized form of Theorem 3.9 holds for curvature-adjusted splines. See Theorem 3.10 below.

In `splinecoeff.m`, the two lines

```plaintext
A(1,1)=2; r(1)=v1; % curvature-adj conditions
A(n,n)=2; r(n)=v_n;
```
must be substituted in place of the two existing lines for the natural spline.

The next alternative set of end conditions is

**Property 4c. (Clamped cubic spline)** Similar to the above, but it is the values of the first derivatives $S'_1(x_1)$ and $S'_{n-1}(x_n)$ that are set to user-specified values $v_1$ and $v_n$, respectively. Thus the slope at the beginning and end of the spline are under the user’s control.

Using (3.29) and (3.30), the extra condition $S'_1(x_1) = v_1$ can be rewritten

$$2\delta_1 c_1 + \delta_1 c_2 = 3 \left( \frac{\Delta_1}{\delta_1} - v_1 \right),$$

and $S'_{n-1}(x_n) = v_n$ can be written as

$$\delta_{n-1} c_{n-1} + 2\delta_{n-1} c_n = 3 \left( v_n - \frac{\Delta_{n-1}}{\delta_{n-1}} \right).$$

The two corresponding tableau rows are

$$
\begin{pmatrix}
2\delta_1 & \delta_1 & 0 & 0 & \cdots & \cdots & 0 & 0 & 0 & 0 & 3(\Delta_1/\delta_1 - v_1) \\
0 & 0 & 0 & 0 & \cdots & \cdots & 0 & \delta_{n-1} & 2\delta_{n-1} & 3(v_n - \Delta_{n-1}/\delta_{n-1})
\end{pmatrix}.
$$

Note the strict diagonal dominance holds also for the revised coefficient matrix in (3.31), so Theorem 3.9 also holds with natural replaced with clamped. In `splinecoeff.m`, the two lines

```matlab
A(1,1:2)=[2*dx(1) dx(1)];r(1)=3*(dy(1)/dx(1)-v1); % clamped conditions
A(n,n-1:n)=[dx(n-1) 2*dx(n-1)];r(n)=3*(vn-dy(n-1)/dx(n-1));
```

must be substituted.

**Property 4d. (Parabolically-terminated cubic spline)** The first and last parts of the spline, $S_1$ and $S_{n-1}$, are forced to be at most degree 2, by specifying $d_1 = 0 = d_{n-1}$. Equivalently, according to (3.29), we can require $c_1 = c_2$ and $c_{n-1} = c_n$. The equations form the two tableau rows

$$
\begin{pmatrix}
1 & -1 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 1 & -1 & 0
\end{pmatrix}
$$

to be used as the top and bottom rows of (3.31). Assume that the number $n$ of data points satisfies $n \geq 3$ (see Exercise 3.4.16 for the case $n = 2$). In this case, upon replacing $c_1$ by $c_2$ and $c_n$ by $c_{n-1}$, the matrix equation reduces to a strictly diagonally dominant $n - 2 \times n - 2$ matrix equation in $c_2, \ldots, c_{n-1}$. Therefore a version of Theorem 3.9 holds for parametrically-terminated splines, assuming $n \geq 3$.

In `splinecoeff.m`, the two lines

```matlab
A(1,1:2)=[1 -1]; % parabol-term conditions
A(n,n-1:n)=[1 -1];
```

must be substituted.
Property 4e. (Not-a-knot cubic spline) The two added equations are \( d_1 = d_2 \) and \( d_{n-2} = d_{n-1} \), or equivalently \( S''_1(x_2) = S''_2(x_2) \) and \( S''_{n-2}(x_{n-1}) = S''_{n-1}(x_{n-1}) \). Since \( S_1 \) and \( S_2 \) are degree 3 or less polynomials, requiring their third derivatives to agree at \( x_1 \), while their zero, first, and second derivatives already agree there, causes \( S_1 \) and \( S_2 \) to be identical cubic polynomials (cubics are defined by 4 coefficients, and 4 conditions are specified). Therefore the result of this condition is that \( x_2 \) is not needed as a base point: the spline is given by the same formula \( S_1 = S_2 \) on the entire interval \([x_1, x_3]\). The same reasoning shows that \( S_{n-2} = S_{n-1} \), and so not only \( x_2 \) but also \( x_{n-1} \) is “no longer a knot”.

Note that \( d_1 = d_2 \) implies that \((c_2 - c_1)/\delta_1 = (c_3 - c_2)/\delta_2\), or
\[
\delta_2 c_1 - (\delta_1 + \delta_2) c_2 + \delta_1 c_3 = 0,
\]
and similarly \( d_{n-2} = d_{n-1} \) implies
\[
\delta_{n-1} c_{n-2} - (\delta_{n-2} + \delta_{n-1}) c_{n-1} + \delta_{n-2} c_n = 0.
\]

The two tableau rows are
\[
\begin{pmatrix}
\delta_2 & - (\delta_1 + \delta_2) & \delta_1 & 0 & \cdots & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & | & 0 \\
0 & 0 & 0 & 0 & \cdots & \cdots & \delta_{n-1} & - (\delta_{n-2} + \delta_{n-1}) & \delta_{n-2} & | & 0
\end{pmatrix}
\]

In `splinecoeff.m`, the two lines
\[
\text{A(1,1:3)=}\left[\text{dx}(2) - (\text{dx}(1)+\text{dx}(2)) \ \text{dx}(1)\right]; % \text{not-a-knot conditions}
\text{A(n,n-2:n)=}\left[\text{dx}(n-1) - (\text{dx}(n-2)+\text{dx}(n-1)) \ \text{dx}(n-2)\right];
\]

are used. Figure 3.13(b) shows an example of a not-a-knot cubic spline, compared with the natural spline through the same data points in part (a) of the figure.

As mentioned above, a theorem analogous to Theorem 3.9 exists for each of the above choices of end conditions. That is:

**Theorem 3.10** Assume \( n \geq 2 \). For a set of data points \((x_1, y_1), \ldots, (x_n, y_n)\) and for any one of the end conditions Properties 4a - 4c, there is a unique cubic spline satisfying the end conditions and fitting the points. The same is true assuming \( n \geq 3 \) for Property 4d and \( n \geq 4 \) for Property 4e.

Matlab’s default `spline` command constructs a not-a-knot spline when given 4 or more points. Let \( x \) and \( y \) be vectors containing the \( x_i \) and \( y_i \) data values, respectively. Then the \( y \)-coordinates of the not-a-knot spline at another input \( x_0 \) is calculated by the Matlab command
\[
\text{>> y0 = spline}(x,y,x0);
\]

If \( x_0 \) is a vector of \( x \)-coordinates, then the output \( y_0 \) will be a corresponding vector of \( y \)-coordinates, suitable for plotting, etc. Alternatively, if the vector input \( y \) has exactly two more inputs than \( x \), the clamped cubic spline is calculated, with \( v_1 \) and \( v_n \) equal to the first and last entries of \( y \).

**Exercises 3.4**
3.4.1. Decide whether
\[
S(x) = \begin{cases} 
  x^3 + x - 1 & \text{on } [0, 1] \\
  -(x - 1)^3 + 3(x - 1)^2 + 3(x - 1)^1 + 1 & \text{on } [1, 2]
\end{cases}
\]
is a natural cubic spline.

3.4.2. Assume the two leftmost parts of a potential cubic spline are given by
\[
\begin{align*}
S_1(x) &= 1 + 2x + 3x^2 + 4x^3 \text{ on } [0, 1] \\
S_2(x) &= 10 + 20(x - 1) + 15(x - 1)^2 + 4(x - 1)^3 \text{ on } [1, 2]
\end{align*}
\]
(a) Check the spline conditions. Decide whether this can or cannot be part of a spline.
(b) Regardless of your answer to (a), decide whether any of the following extra conditions are satisfied for this example: natural, parabolically terminated, not-a-knot.

3.4.3. How many natural cubic splines on \([0, 2]\) are there for the given data \((0, 0), (1, 1), (2, 2)\)? Exhibit one such spline.

3.4.4. Find the parabolically-terminated cubic spline through the data points \((0, 1), (1, 1), (2, 1), (3, 1), (4, 1)\). Is this spline also not-a-knot? natural?

3.4.5. Find \(S'(2)\) for the cubic spline
\[
\begin{align*}
S_1(x) &= 3 + b_1 x + x^3 \text{ on } [0, 1] \\
S_2(x) &= 1 + b_2(x - 1) + 3(x - 1)^2 - 2(x - 1)^3 \text{ on } [1, 2]
\end{align*}
\]

3.4.6. Can the following be two parts of a cubic spline? Explain why or why not.
\[
\begin{align*}
2x^3 + x^2 + 4x + 5 & \text{ on } [0, 1] \\
(x - 1)^3 - 5(x - 1)^2 + 13(x - 1) + 12 & \text{ on } [1, 2]
\end{align*}
\]

3.4.7. Find \(c\) in the following cubic splines. Also, which of the three end conditions, natural, parabolically terminated, or not-a-knot, if any, are satisfied?
(a)
\[
S(x) = \begin{cases} 
  4 - \frac{11}{2}x + \frac{3}{4}x^3 & \text{on } [0, 1] \\
  2 - \frac{7}{2}(x - 1) + c(x - 1)^2 - \frac{3}{4}(x - 1)^3 & \text{on } [1, 2]
\end{cases}
\]

(b)
\[
S(x) = \begin{cases} 
  3 - 9x + 4x^2 & \text{on } [0, 1] \\
  -2 - (x - 1) + c(x - 1)^2 & \text{on } [1, 2]
\end{cases}
\]

(c)
\[
S(x) = \begin{cases} 
  -2 - \frac{3}{2}x + \frac{7}{2}x^2 - x^3 & \text{on } [0, 1] \\
  -1 + c(x - 1) + \frac{5}{2}(x - 1)^2 - (x - 1)^3 & \text{on } [1, 2] \\
  1 + \frac{1}{2}(x - 1) - \frac{3}{2}(x - 1)^2 - (x - 2)^3 & \text{on } [2, 3]
\end{cases}
\]

3.4.8. Find \(k_1, k_2, k_3\) in the following cubic spline. Also, which of the three end conditions, natural, parabolically terminated, or not-a-knot, if any, are satisfied?
\[
S(x) = \begin{cases} 
  4 + k_1 x + 2x^2 - \frac{1}{3}x^3 & \text{on } [0, 1] \\
  1 - \frac{5}{3}(x - 1) + k_2(x - 1)^2 - \frac{1}{3}(x - 1)^3 & \text{on } [1, 2] \\
  1 + k_3(x - 2) + (x - 2)^2 - \frac{1}{6}(x - 2)^3 & \text{on } [2, 3]
\end{cases}
\]

3.4.9. How many not-a-knot cubic splines are there for the given data \((1, 3), (3, 3), (4, 2), (5, 0)\)? Exhibit one such spline.

3.4.10. True or false: Given \(n = 3\) data points, the parabolically-terminated cubic spline through the points must also be not-a-knot.

3.4.11. (a) How many parabolically-terminated cubic splines on \([0, 2]\) are there for the given data \((0, 2), (1, 0), (2, 2)\)? Exhibit one such spline.
(b) Same question for not-a-knot.

3.4.12. Can a cubic spline be both natural and parabolically-terminated? If so, what else can you say about such a spline?
3.4.13. Assume the leftmost piece of a given natural cubic spline is the constant function \( S_1(x) = 1 \), on the interval \([-1, 0]\). Find three different possibilities for the neighboring piece \( S_2(x) \) of the spline on \([0, 1]\).

3.4.14. Does there exist a (simultaneously) natural, parabolically-terminated, not-a-knot cubic spline through each set of data points \((x_1, y_1), \ldots, (x_{100}, y_{100})\) with distinct \( x_i \)? If so, give a reason. If not, explain what conditions must hold on the 100 points in order for such a spline to exist.

3.4.15. Assume a car travels along a straight road from one point to another, from a standing start at time \( t = 0 \) to a standing stop at time \( t = 1 \). The distance along the road is sampled at certain times between 0 and 1. Which cubic spline (in terms of end conditions) will be most appropriate for describing distance versus time?

3.4.16. (a) The case \( n = 2 \) for parabolically-terminated cubic splines is not covered by Theorem 3.10. Discuss existence and uniqueness for the cubic spline in this case. (b) Similarly, the cases \( n = 2 \) and \( 3 \) are not covered for not-a-knot cubic splines. Discuss existence and uniqueness.

### Computer Problems 3.4

3.4.1. Find the equations and plot the natural cubic spline that interpolates the data points (a) \((0, 3), (1, 5), (2, 4), (3, 1)\) (b) \((-1, 3), (0, 5), (3, 1), (4, 1), (5, 1)\).

3.4.2. Repeat the previous problem, but find the not-a-knot cubic spline.

3.4.3. Find a cubic spline \( S \) satisfying \( S(0) = 1, S(1) = 3, S(2) = 3, S(3) = 4, S(4) = 2 \), and with \( S''(0) = S''(4) = 0 \).

### 3.5 Bezier splines

Bezier splines are a type of spline that allows the user to control the slopes at the knots. In return for the extra freedom, the smoothness of the first and second derivatives across the knot, which are automatic features of the cubic splines of the previous section, are no longer guaranteed. Bezier splines are appropriate for cases where corners (discontinuous first derivatives) and abrupt curvature changes (discontinuous second derivatives) are sometimes needed.

Pierre Bezier (1910-1999) developed this idea during his work for the Renault automobile company. The same idea was discovered independently and used by Paul de Casteljau, working for Citroen, a rival automobile company. Because the Bezier spline was considered an industrial secret for both companies, the fact that both had developed the idea came to light only after Bezier published his research. Today Bezier splines are a fundamental component of computer-aided design and manufacturing. In addition, Bezier splines are nearly ubiquitous due to their role in modern font design, as we will see later.

Each piece of a Bezier spline is determined by four points \((x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\). The first and last of the points are in the spline curve, the middle two are control points, as shown in Figure 3.15. The curve leaves \((x_1, y_1)\) along the tangent direction \((x_2 - x_1, y_2 - y_1)\), and ends at \((x_4, y_4)\) along the tangent direction \((x_4 - x_3, y_4 - y_3)\). The equations that accomplish this are expressed as a parametric curve \((x(t), y(t))\) for \(0 \leq t \leq 1\):

\[
\begin{align*}
x(t) &= x_1 + b_x t + c_x t^2 + d_x t^3 \\
y(t) &= y_1 + b_y t + c_y t^2 + d_y t^3
\end{align*}
\]

where

\[
\begin{align*}
b_x &= 3(x_2 - x_1) \\
b_y &= 3(y_2 - y_1) \\
c_x &= 3(x_3 - x_2) - b_x \\
c_y &= 3(y_3 - y_2) - b_y \\
d_x &= x_4 - x_1 - b_x - c_x \\
d_y &= y_4 - y_1 - b_y - c_y
\end{align*}
\]

(3.35)

One checks the facts of the previous paragraph from the equations. In fact,

\[
\begin{align*}
x(0) &= x_0 \\
x'(0) &= 3(x_2 - x_1) \\
x(1) &= x_4 \\
x'(1) &= 3(x_4 - x_3)
\end{align*}
\]

and the analogous facts hold for \(y(t)\).
Find the Bezier spline \((x(t), y(t))\) through the points \((x, y) = (1, 1)\) and \((2, 2)\) with control points \((1, 3)\) and \((3, 3)\).

The four points are \((x_1, y_1) = (1, 1), (x_3, y_3) = (1, 3), (x_4, y_4) = (2, 2)\). Formula (3.35) yields \(b_x = 0, c_x = 6, d_x = -5\) and \(b_y = 6, c_y = -6, d_y = 1\). The Bezier spline is

\[
\begin{align*}
x(t) &= 1 + 6t^2 - 5t^3 \\
y(t) &= 1 + 6t - 6t^2 + t^3
\end{align*}
\]

The spline is shown in Figure 3.14 along with the control points.

Bezier splines are building blocks that can be put side-by-side to fit arbitrary function values and slopes. They are an improvement over cubic splines, in the sense that the slopes at the nodes can be specified as the user wants them. However, this freedom comes at the expense of curvature smoothness. The second derivatives from the two different directions generally disagree at the nodes. Which type of spline is preferred depends on the application.

Bezier splines are also simple to program and are often used in drawing software. A freehand curve in the plane can be represented as a parametric curve \((x(t), y(t))\), which can be replaced by a Bezier spline. We have implemented the equations in the following Matlab freehand drawing program. The user clicks the mouse twice to fix a point in the plane; once to define the starting point \((x_0, y_0)\), and the second click, to mark the first control point, in the intended direction of the path. After another pair of clicks, to define the second control point and endpoint, a Bezier spline is drawn between the two points. Each subsequent triple of mouse clicks extends the curve further, using the previous spline point as the starting point for the next piece. The Matlab command \texttt{ginput} is used to read the mouse location, and which button is being activated.

```plaintext
% Program 3.7 Freehand Draw Program Using Bezier Splines
% Left click in Matlab figure window to locate point, and click three
% more times to specify 2 control points and the next spline point.
% Continue with groups of 3 points to add more to
% the curve. Right-click to terminate program.
hold off
plot([-1 1],[0,0],’k’,[0 0],[-1 1],’k’); hold on
xlist=[]; ylist=[]; t=0:.02:1;
button=1; k=0;
```
Figure 3.15: Draw Program 3.7 made with Bezier splines. Screenshot of Matlab code `draw.m` including direction vectors drawn at each control point.

```matlab
while(button ~= 3) % if right click, terminate
    [xnew,ynew,button] = ginput(1); % get one mouse click
    if button == 1 % current click is a left click
        k=k+1; % k counts clicks
        xlist(k)=xnew; ylist(k)=ynew; % add new point to the list
        if k>1 % After first input point:
            if mod(k,3) == 1 % Wait for three new input points,
                for i=1:4 % gather the previous four points
                    x(i)=xlist(k-i+1);
                    y(i)=ylist(k-i+1);
                end % Plot spline points and control pts
                plot([x(1) x(2)],[y(1) y(2)],'c:',x(2),y(2),'cs');
                plot([x(3) x(4)],[y(3) y(4)],'c:',x(3),y(3),'cs');
                plot(x(1),y(1),'bo',x(4),y(4),'bo');
                bx=3*(x(2)-x(1)); by=3*(y(2)-y(1)); % spline equations ...
                cx=3*(x(3)-x(2))-bx; cy=3*(y(3)-y(2))-by;
                dx=x(4)-x(1)-bx-cx; dy=y(4)-y(1)-by-cy;
                xp=x(1)+t.*(bx+t.*(cx+t*dx)); % Horner’s method
                yp=y(1)+t.*(by+t.*(cy+t*dy));
                plot(xp,yp) % Plot spline curve and wait for more pts
            end
        end
    end
end
end
```

REALITY CHECK: Constructing fonts from Bezier splines

Although developed for the design of automobile shapes for aerodynamic studies, Bezier splines
have been used for many applications. They found a natural use with the advent of computer typesetting in the 1980’s. Chuck Geschke and John Warnock were engineers for Xerox Corporation who quit to form a company called Adobe. They released the Postscript language in 1984, where it came to the attention of Steve Jobs at Apple Corporation, who was looking for a way to control their first laser printer. Postscript is a vector-based page description language, which revolutionized printing in the sense that it was device-independent. Formerly, fonts at different sizes had to be tuned to the dot patterns of printers of various resolutions. Using Bezier splines, it was simple to adapt the same mathematical curves to whatever resolution was required.

A bare-bones Postscript file follows:

```postscript
%!PS
newpath
100 100 moveto
100 300 300 300 200 200 curveto
stroke
```

This is a complete Postscript file. The `newpath` command initiates the curve. Postscript uses a stack and postix commands, similar to HP handheld calculators. The `moveto` command sets the current plot point to be the \((x, y)\) point specified by the last two numbers on the stack, in this case, the point \((100, 100)\). The `curveto` command accepts three \((x, y)\) points from the current stack, and constructs the Bezier spline starting at the current plot point, using the next three \((x, y)\) pairs as the two control points and the endpoint, respectively. If the above file is sent to a Postscript printer, the Bezier curve of Example 3.15 will be printed. We have multiplied the coordinates by 100 to match the conventions of Postscript, which are 72 units to the inch. A sheet of letter-sized paper is 612 Postscript units wide and 792 high.

Postscript fonts are drawn on computer screens and printers using the Bezier curves of the `curveto` command. The `fill` command, given at the end, will fill in the outline if the figure is closed.

The upper case T character in the Times-Roman font is constructed of the following 16 Bezier curves. Each line consists of the numbers \(x_1 \ y_1 \ x_2 \ y_2 \ x_3 \ y_3 \ x_4 \ y_4\) that define one piece of the Bezier spline.

\[
237 \hspace{1em} 620 \hspace{1em} 237 \hspace{1em} 620 \hspace{1em} 237 \hspace{1em} 120 \hspace{1em} 237 \hspace{1em} 120; \\
237 \hspace{1em} 120 \hspace{1em} 237 \hspace{1em} 35 \hspace{1em} 226 \hspace{1em} 24 \hspace{1em} 143 \hspace{1em} 19; \\
143 \hspace{1em} 19 \hspace{1em} 143 \hspace{1em} 19 \hspace{1em} 143 \hspace{1em} 0 \hspace{1em} 143 \hspace{1em} 0; \\
143 \hspace{1em} 0 \hspace{1em} 143 \hspace{1em} 0 \hspace{1em} 435 \hspace{1em} 0 \hspace{1em} 435 \hspace{1em} 0; \\
435 \hspace{1em} 0 \hspace{1em} 435 \hspace{1em} 0 \hspace{1em} 435 \hspace{1em} 19 \hspace{1em} 435 \hspace{1em} 19; \\
435 \hspace{1em} 19 \hspace{1em} 353 \hspace{1em} 23 \hspace{1em} 339 \hspace{1em} 36 \hspace{1em} 339 \hspace{1em} 109; \\
339 \hspace{1em} 109 \hspace{1em} 339 \hspace{1em} 108 \hspace{1em} 339 \hspace{1em} 620 \hspace{1em} 339 \hspace{1em} 620; \\
339 \hspace{1em} 620 \hspace{1em} 339 \hspace{1em} 620 \hspace{1em} 393 \hspace{1em} 620 \hspace{1em} 393 \hspace{1em} 620; \\
393 \hspace{1em} 620 \hspace{1em} 507 \hspace{1em} 620 \hspace{1em} 529 \hspace{1em} 602 \hspace{1em} 602 \hspace{1em} 552 \hspace{1em} 492; \\
552 \hspace{1em} 492 \hspace{1em} 552 \hspace{1em} 492 \hspace{1em} 576 \hspace{1em} 492 \hspace{1em} 576 \hspace{1em} 492; \\
576 \hspace{1em} 492 \hspace{1em} 576 \hspace{1em} 492 \hspace{1em} 570 \hspace{1em} 662 \hspace{1em} 570 \hspace{1em} 662; \\
570 \hspace{1em} 662 \hspace{1em} 570 \hspace{1em} 662 \hspace{1em} 6 \hspace{1em} 662 \hspace{1em} 6 \hspace{1em} 662; \\
6 \hspace{1em} 662 \hspace{1em} 6 \hspace{1em} 662 \hspace{1em} 0 \hspace{1em} 492 \hspace{1em} 0 \hspace{1em} 492; \\
0 \hspace{1em} 492 \hspace{1em} 0 \hspace{1em} 492 \hspace{1em} 24 \hspace{1em} 492 \hspace{1em} 24 \hspace{1em} 492; \\
24 \hspace{1em} 492 \hspace{1em} 48 \hspace{1em} 602 \hspace{1em} 71 \hspace{1em} 620 \hspace{1em} 183 \hspace{1em} 620; \\
183 \hspace{1em} 620 \hspace{1em} 183 \hspace{1em} 620 \hspace{1em} 237 \hspace{1em} 620 \hspace{1em} 237 \hspace{1em} 620;
```

The Times-Roman lower case f requires 21 Bezier curves:
Figure 3.16: Times-Roman 5 made with Bezier splines. Spline points are circles, control points are squares.

The number 5 also requires 21 curves:

289 452 289 452 166 452 166 452;
166 452 166 452 166 568 166 568;
166 568 166 627 185 657 223 657;
223 657 245 657 258 647 276 618;
276 618 292 589 304 580 321 580;
321 580 345 580 363 598 363 621;
363 621 363 657 319 683 259 683;
259 683 196 683 144 656 118 656;
118 656 92 565 84 530 83 450;
83 450 83 450 1 450 1 450;
1 450 1 450 1 450 1 450;
1 450 1 450 1 418 1 418;
1 418 1 418 83 418 83 418;
83 418 83 418 83 104 83 104;
83 104 83 83 31 72 19 0 15;
0 15 0 15 0 0 0 0;
0 0 0 0 260 0 260 0;
260 0 260 0 260 15 260 15;
260 15 178 18 167 29 167 104;
167 167 167 167 167 104 167 104;
167 167 167 167 167 104 167 104;
167 167 167 167 167 104 167 104;
289 418 289 418 289 452 289 452;
1. Use the `draw.m` program to sketch the upper case initial of your first name.
2. Revise the draw program to accept an $n \times 8$ matrix of numbers, each row representing a piece of a Bezier spline. Use the program to draw the letter $f$ in the Times-Roman font, using the above Bezier data.
3. Write a Postscript file to draw the letter $f$. The program should begin with a `moveto` to the first point, followed by 21 `curveto` commands. Test it by opening your file in a Ghostscript viewer or by sending to a Postscript printer.
4. Although font information was a closely-guarded secret for many years, much of it is now freely available on the web. Search for other fonts, and find the Bezier curve data to draw letters of your choice in Postscript or using `draw.m`.

Exercises 3.5

3.5.1.

Computer Problems 3.5

3.5.1.