Chapter 11

Piecewise Polynomial Interpolation

The previous chapter discusses polynomial interpolation, an essential ingredient in the construction of general approximation methods for a variety of problems including function approximation, differentiation, integration, and the solution of differential equations.

However, polynomial interpolation is often not sufficiently flexible to directly yield useful general-purpose procedures; rather, it typically provides an essential building block for other, more general techniques. In the present chapter we develop robust methods for the interpolation of functions which work even if the number of data points is large, or their abscissae locations are not under our control, or the interval over which the function is approximated is long.

Note: In many ways, this chapter is a direct continuation of the previous one. Section 10.1, in particular, is directly relevant here as well.

We start in Section 11.1 by making the case for piecewise polynomial interpolation as the cure for several polynomial interpolation ills. In Section 11.2 we introduce two favorite piecewise polynomial interpolants which are defined entirely locally: each polynomial piece can be constructed independently of and in parallel with the others.

The workhorse of robust interpolation, however, is the *cubic spline* which is derived and demonstrated in Section 11.3. Basis functions for piecewise polynomial interpolation are considered in Section 11.4.

In Section 11.5 we shift focus and introduce parametric curves, fundamental in **computer aided geometric design** (CAGD) and other applications. Despite the different setting, these generally turn out to be a surprisingly simple extension of the techniques of Section 11.2.

Interpolation in more than one independent variable is a much broader, more advanced topic that is briefly reviewed in Section 11.6.

11.1 The case for piecewise polynomial interpolation

We continue to consider interpolation of the $n+1$ data pairs

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n),$$

looking for a function $v(x)$ that satisfies

$$v(x_i) = y_i, \quad i = 0, 1, \ldots, n.$$

We also continue to assume an underlying function $f(x)$ that is to be approximated on an interval $[a,b]$ containing the abscissae x_i. The function $f(x)$ is unknown in general, except for its values $f(x_i) = y_i$, $i = 0,\ldots,n$. Occasionally we will interpolate function derivative values $f'(x_i)$ as well. Furthermore, although this will not always be obvious in what follows, we consider only interpolants in linear form that can be written as

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x),$$

where $\phi_j(x)$ are given basis functions and c_j are unknown coefficients to be determined.

Shortcomings of polynomial interpolation

An interpolant of the form discussed in Chapter 10 is not always suitable for the following reasons:

- The error term
$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i)$$
(see the Interpolation Error Theorem on page 314) may not be small if $\frac{\|f^{(n+1)}\|}{(n+1)!}$ isn't. In Example 10.6 we have already seen that it does not take much for this to happen.
- High order polynomials tend to oscillate "unreasonably."
- Data often are only piecewise smooth, whereas polynomials are infinitely differentiable. The high derivatives $f^{(n+1)}$ may blow up (or be very large) in such a case, which again yields a large error term.
- No locality: changing any one data value may drastically alter the entire interpolant.

Returning to Example 10.6, for that particular function polynomial interpolation at Chebyshev points works well as we have seen. However, polynomial interpolation for various functions at *any* set of fixed points does not always produce satisfactory results, and the quality of the resulting approximation is hard to control. For more examples of "unreasonable" polynomial interpolation see Exercise 15 in this chapter and Exercise 9 in Chapter 10. Such examples are abundant in practice.

Piecewise polynomials

We must find a way to reduce the error term without increasing the degree n. In what follows we do this by decreasing the size of the interval $b-a$. Note that simply rescaling the independent variable x will not help! (Can you see why?) We thus resort to using polynomial pieces only locally. Globally, we use a *piecewise polynomial interpolation*.

Thus, we divide the interval into a number of smaller subintervals (or *elements*) by the partition

$$a = t_0 < t_1 < \cdots < t_r = b$$

and use a (relatively low degree) polynomial interpolation in each of these subintervals $[t_i, t_{i+1}]$, $i = 0,\ldots,r-1$. These polynomial pieces, $s_i(x)$, are then patched together to form a continuous (or C^1, or C^2) global interpolating curve $v(x)$ which satisfies

$$v(x) = s_i(x), \quad t_i \le x \le t_{i+1},\ i = 0,\ldots,r-1.$$

The points $t_0, t_1, \ldots, t_r$ are called *break points*. See Figure 11.1.

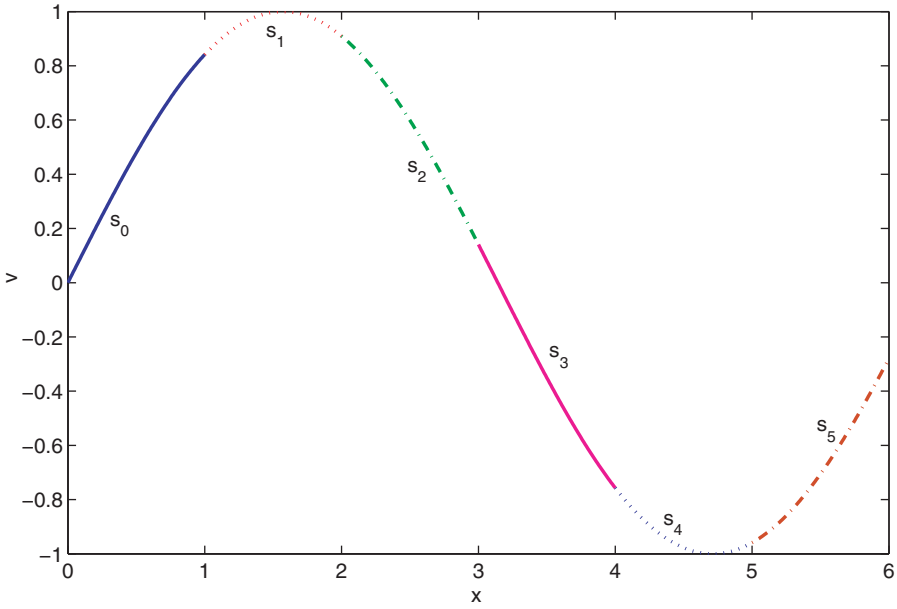


Figure 11.1. *A piecewise polynomial function with break points* $t_i = i,\ i = 0,1,\ldots,6.$

11.2 Broken line and piecewise Hermite interpolation

In this section we consider two important general-purpose interpolants that can be constructed entirely locally. That is to say, a polynomial interpolant is constructed on each subinterval $[t_i, t_{i+1}]$, and our interpolant on the entire interval $[a,b]$ is simply the assembly of the local pieces.

Broken line interpolation

The simplest instance of continuous piecewise polynomial interpolation is piecewise linear, or "broken line interpolation." Thus, the polynomial pieces are linear and the piecewise linear interpolant is continuous (but not continuously differentiable) everywhere.

Example 11.1. An instance of broken line interpolation is provided in Figure 11.2. It consists simply of connecting data values by straight lines. By Newton's formula for a linear polynomial interpolant, we can write

$$v(x) = s_i(x) = f(x_i) + f[x_i, x_{i+1}](x - x_i), \quad x_i \le x \le x_{i+1},\ \ 0 \le i \le 4.$$

You can roughly figure out (if you must) the values of $(x_i, f(x_i))$ that gave rise to Figure 11.2. ∎

A great advantage of piecewise linear interpolation, other than its obvious simplicity, is that the maximum and minimum values are at the data points: no new extremum point is "invented" by the interpolant, which is important for a general-purpose interpolation black box routine. MATLAB uses this interpolation as the default option for plotting. (More precisely, a parametric curve based on broken line interpolation is plotted by default—see Section 11.5.)

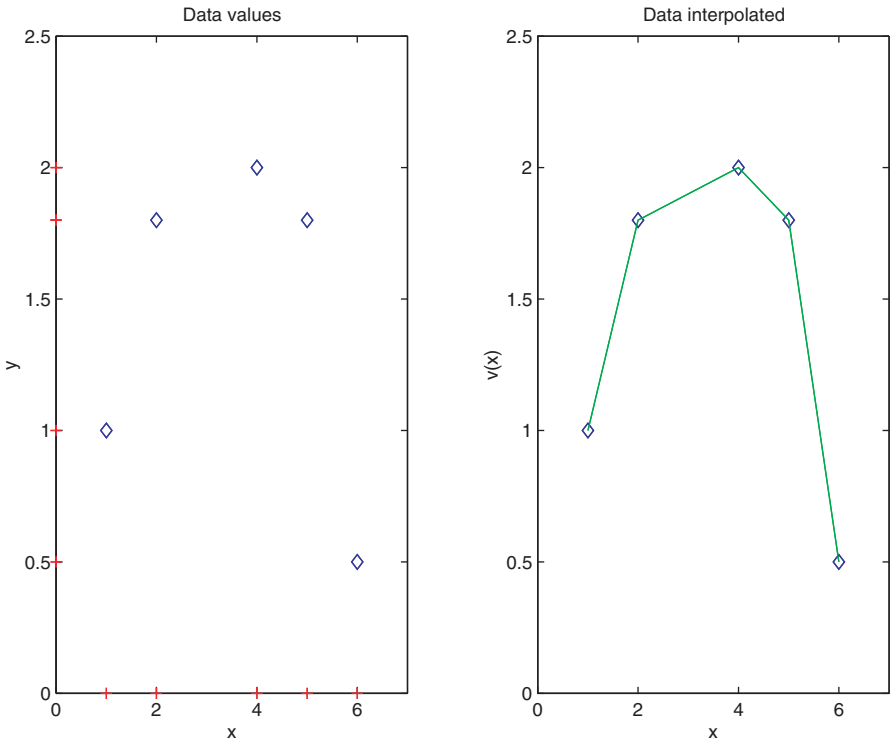


Figure 11.2. *Data and their broken line interpolation.*

Error bound for piecewise linear interpolation

Continuing with the notation of Example 11.1, let $n = r$, $t_i = x_i$, and

$$h = \max_{1 \le i \le r} (t_i - t_{i-1}).$$

It is not difficult to show that the error for piecewise linear interpolation is bounded by

$$|f(x) - v(x)| \le \frac{h^2}{8} \max_{a \le \xi \le b} |f''(\xi)|$$

for any x in the interval $[a,b]$. Indeed, for any $x \in [a,b]$ there is an index i, $1 \le i \le r$, such that $t_{i-1} \le x \le t_i$. The interpolant is linear on this subinterval, so applying the error formula from Section 10.5 for polynomial interpolation to this linear segment, we have

$$f(x) - v(x) = \frac{f''(\xi)}{2!}(x - t_{i-1})(x - t_i).$$

It can be shown (try!) that the maximum value of $|(x - t_{i-1})(x - t_i)|$ is attained at the midpoint $x = \frac{t_{i-1}+t_i}{2}$, and hence

$$|(x - t_{i-1})(x - t_i)| \le \left(\frac{t_i - t_{i-1}}{2}\right)^2.$$

The above error bound is obtained upon noting that $2^2 \cdot 2! = 8$ and applying bounds in an obvious way. ♦

So, at least for equally spaced data points, as n increases the error decreases, at rate $\mathcal{O}(h^2) = \mathcal{O}(n^{-2})$. Note that n is no longer the degree of the polynomial pieces; the latter is held here at 1.

Piecewise constant interpolation

Often the break points t_i are the given data abscissae x_i, sorted in ascending order, in which case we have $r = n$. Such is the case for the piecewise linear interpolation discussed above. One simple exception is *piecewise constant* interpolation: not knowing anything about the smoothness of f—not even if it is continuous—we may want to construct the simplest approximation, and one such is given by

$$v(x) = s_i(x) = f(x_i), \quad t_i \le x < t_{i+1},$$

where $t_i = \frac{1}{2}(x_i + x_{i-1})$, $i = 1,2,\ldots,n$. Set also $t_0 = a \le x_0$, $t_r = b \ge x_n$, $r = n+1$.

Exercise 1 asks you to find an error bound for such an interpolation scheme. (It clearly should be $\mathcal{O}(h)$ if f has a bounded derivative.)

Piecewise cubic interpolation

But piecewise linear interpolation is often not smooth enough. (Granted, what's "enough" depends on the application.) From Figure 11.2 it is clear that this interpolant has discontinuous first derivatives at data points. To have higher smoothness, say, C^1 or C^2, we must increase the degree of each polynomial piece.

The most popular piecewise polynomial interpolation is with *cubics*. Here we may write

$$v(x) = s_i(x) = a_i + b_i(x - t_i) + c_i(x - t_i)^2 + d_i(x - t_i)^3,$$
$$t_i \le x \le t_{i+1},\ i = 0,\ldots,r-1.$$

Clearly, we have $4r$ unknowns (coefficients). To fix these parameters we need $4r$ algebraic conditions. There are two types of conditions:

- interpolation conditions, and
- continuity conditions.

In the piecewise linear case these conditions read

$$s_i(t_i) = f(t_i),\ s_i(t_{i+1}) = f(t_{i+1}), \quad i = 0,1,\ldots,r-1,$$

where $t_i = x_i$, $i = 0,1,\ldots,r$. This gives $2r$ conditions, which exhausts all the freedom we have in the piecewise linear case because we have only $2r$ coefficients to determine. Continuity is implied by $s_i(t_{i+1}) = f(t_{i+1}) = s_{i+1}(t_{i+1})$. But in the piecewise cubic case we can impose $2r$ additional conditions. We now look at a way for doing this, which retains the local construction charm. Another important way for setting those conditions is considered in Section 11.3.

Piecewise cubic Hermite interpolation

The simplest, or cleanest, case is where also values of $f'(t_i)$ are provided. Thus, $n+1 = 2(r+1)$ and the abscissae are

$$(x_0, x_1, x_2, \ldots, x_{n-1}, x_n) = (t_0, t_0, t_1, t_1, \ldots, t_r, t_r).$$

Matching these gives precisely $2r$ more conditions, written as

$$s_i'(t_i) = f'(t_i),\ s_i'(t_{i+1}) = f'(t_{i+1}),\ \ i = 0,1,\ldots,r-1.$$

This is the *piecewise cubic Hermite interpolation*. Note that $v(x)$ is clearly in C^1, i.e., both v and v' are continuous everywhere.

This interpolant can be constructed locally, piece by piece, by applying the osculating interpolation algorithm from Section 10.7 for the cubic polynomial in each subinterval separately. Specifically, for any x in the subinterval $[t_i, t_{i+1}]$ the interpolant $v(x)$ coincides with the corresponding Hermite cubic polynomial $s_i(x)$ on this interval, given explicitly by

$$s_i(x) = f_i + \left(h_i f_i'\right)\tau + \Big(3(f_{i+1} - f_i) - h_i(f_{i+1}' + 2f_i')\Big)\tau^2 + \Big(h_i(f_{i+1}' + f_i') - 2(f_{i+1} - f_i)\Big)\tau^3,$$

where $h_i = t_{i+1} - t_i$, $f_i = f(t_i)$, $f_i' = f'(t_i)$, and $\tau = \frac{x - t_i}{h_i}$. See Exercise 4.

Note also that changing the data at one data point changes the value of the interpolant only in the two subintervals associated with that point.

Example 11.2. Let us return to Example 10.6 and consider interpolation at 20 equidistant abscissae. Using a polynomial of degree 20 for this purpose led to poor results; recall Figure 10.6 on page 317.

In contrast, interpolating at the same points $x_i = -1 + 2i/19$ using values of $f(x_i)$ and $f'(x_i)$, $i = 0, 1, \ldots, 19$, yields a perfect looking curve with a computed maximum error of .0042. ∎

Error bound for piecewise cubic Hermite interpolation

Theorem: Piecewise Polynomial Interpolation Error.
Let v interpolate f at the $n+1$ points $x_0 < x_1 < \cdots < x_n$, define $h = \max_{1 \le i \le n} x_i - x_{i-1}$, and assume that f has as many bounded derivatives as appear in the bounds below on an interval $[a, b]$ containing these points.

Then, using a local constant, linear or Hermite cubic interpolation, for each $x \in [a, b]$ the interpolation error is bounded by

$$|f(x) - v(x)| \le \frac{h}{2} \max_{a \le \xi \le b} |f'(\xi)| \quad \text{piecewise constant,}$$

$$|f(x) - v(x)| \le \frac{h^2}{8} \max_{a \le \xi \le b} |f''(\xi)| \quad \text{piecewise linear,}$$

$$|f(x) - v(x)| \le \frac{h^4}{384} \max_{a \le \xi \le b} |f''''(\xi)| \quad \text{piecewise cubic Hermite.}$$

The error in this interpolant can be bounded directly from the error expression for polynomial interpolation. The resulting expression is

$$|f(x) - v(x)| \le \frac{h^4}{384} \max_{a \le \xi \le b} |f''''(\xi)|.$$

Note that $384 = 2^4 \cdot 4!$, which plays the same role as $8 = 2^2 \cdot 2!$ plays in the error expression for the piecewise linear interpolant.

Example 11.3. Recall Example 10.9. For the function $f(x) = \ln(x)$ on the interval $[1,2]$ we have estimated $|f''''(\xi)| \le 6$. Therefore, with $h = .25$ the interpolation error upon applying the Hermite piecewise cubic interpolation on four equidistant subintervals will be bounded by

$$|f(x) - v(x)| \le \frac{6}{384} \times .25^4 \approx 6 \times 10^{-5}.$$

The actual maximum error turns out to be $\approx 3.9 \times 10^{-5}$. Of course, with this error level plotting, the resulting solution yields the same curve as that for $\ln(x)$ in Figure 10.10, as far as the eye can discern. ∎

The error bounds for the various local interpolants that we have seen in this section are gathered in the Piecewise Polynomial Interpolation Error Theorem on the facing page.

Specific exercises for this section: Exercises 1–4.

11.3 Cubic spline interpolation

The main disadvantage of working with Hermite piecewise cubics is that we need values for $f'(t_i)$, which is a lot to ask for, especially when there is no f, just discrete data values! Numerical differentiation comes to mind (see Exercise 9 and Section 14.1), but this does not always make sense and a more direct, robust procedure is desired. Another potential drawback is that there are many applications in which an overall C^1 requirement does not provide sufficient smoothness.

Suppose we are given only data values (x_i, y_i), $i = 0, \ldots, n$, where $x_0 < x_1 < \cdots < x_{n-1} < x_n$ are distinct and $y_i = f(x_i)$ for some function f that may not be explicitly available. Set also $a = x_0$ and $b = x_n$. We identify x_i with the break points t_i, and $n = r$. Having used $2n$ parameters to satisfy the interpolation conditions by a continuous interpolant, we now use the remaining $2n$ parameters to require that $v(x) \in C^2[a,b]$. The result is often referred to as a **cubic spline**.

Thus, the conditions to be satisfied by the cubic spline are

$$s_i(x_i) = f(x_i), \quad i = 0, \ldots, n-1, \tag{11.1a}$$
$$s_i(x_{i+1}) = f(x_{i+1}), \quad i = 0, \ldots, n-1, \tag{11.1b}$$
$$s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}), \quad i = 0, \ldots, n-2, \tag{11.1c}$$
$$s_i''(x_{i+1}) = s_{i+1}''(x_{i+1}), \quad i = 0, \ldots, n-2. \tag{11.1d}$$

See Figure 11.3.[44] Note that the matching conditions are applied only at interior abscissae, not including the first and last points. For this reason there are only $n-1$ such conditions for the first and second derivatives.

Example 11.4. Consider the data

i	0	1	2
x_i	0.0	1.0	2.0
$f(x_i)$	1.1	0.9	2.0

[44] It may appear to the naked eye as if the curve in Figure 11.3 is infinitely smooth near $x_i = 0$. But in fact it was constructed by matching at $x = 0$ two cubics that agree in value, first and second derivative, yet differ by ratio $7:1$ in the values of the third derivative. The apparent smoothness of the combined curve illustrates why the cubic spline is attractive in general.

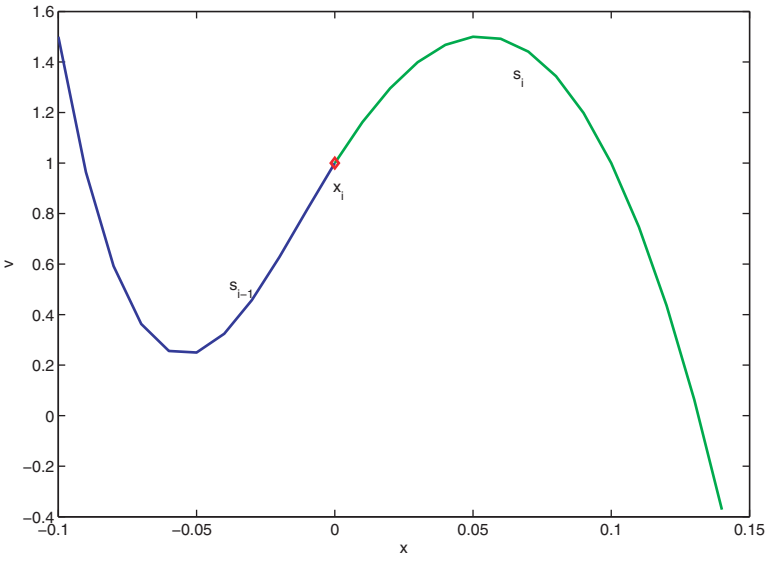


Figure 11.3. *Matching s_i, s_i', and s_i'' at $x = x_i$ with values of s_{i-1} and its derivatives at the same point. In this example, $x_i = 0$ and $y_i = 1$.*

Of course, for these three data points a quadratic polynomial interpolant would do just fine. But our purpose here is to demonstrate the construction of a cubic spline, not to ask if this is a sensible thing to do. For the latter purpose, note that $n = 2$. Thus, we have three data points and just one break point at $x_1 = 1.0$.

The interpolating cubic spline is written as

$$v(x) = \begin{cases} s_0(x) = a_0 + b_0(x - 0.0) + c_0(x - 0.0)^2 + d_0(x - 0.0)^3, & x < 1.0, \\ s_1(x) = a_1 + b_1(x - 1.0) + c_1(x - 1.0)^2 + d_1(x - 1.0)^3, & x \geq 1.0. \end{cases}$$

We now must determine the eight coefficients $a_0, b_0, c_0, d_0, a_1, b_1, c_1$, and d_1.

The interpolation conditions (11.1a) and (11.1b), evaluated at their left ends $i = 0$, give

$$\begin{aligned} 1.1 &= v(0.0) = s_0(0.0) = a_0, \\ 0.9 &= v(1.0) = s_1(1.0) = a_1, \end{aligned}$$

which determine $a_0 = 1.1$ and $a_1 = 0.9$.

Evaluating these two conditions at $i = 1$, we also have

$$\begin{aligned} 0.9 &= v(1.0) = s_0(1.0) = 1.1 + b_0 + c_0 + d_0, \\ 2.0 &= v(2.0) = s_1(2.0) = 0.9 + b_1 + c_1 + d_1. \end{aligned}$$

Thus, we have two relations between the remaining six coefficients. Note how equating $s_0(1.0)$ and $s_1(1.0)$ to the same value of $f(1.0)$ implies continuity of the constructed interpolant $v(x)$.

Next, we evaluate (11.1c) and (11.1d) at $i = 0$ (the only value of i for which they are defined here), i.e., we specify that also $s_0'(x_1) = s_1'(x_1)$ and $s_0''(x_1) = s_1''(x_1)$, where $x_1 = 1.0$. This gives

$$\begin{aligned} b_0 + 2c_0 + 3d_0 &= b_1, \\ 2c_0 + 6d_0 &= 2c_1. \end{aligned}$$

In total, then, we have four equations for the six coefficients b_0, b_1, c_0, c_1, d_0, and d_1. We require two more conditions to complete the specification of $v(x)$. ∎

Two additional conditions

In total, above there are $4n-2$ conditions for $4n$ unknowns, so we still have two conditions to specify. We specify one at each of the boundaries x_0 and x_n:

1. One popular choice is that of **free boundary**, giving a **natural spline**:

$$v''(x_0) = v''(x_n) = 0.$$

   This condition is somewhat arbitrary, though, and may cause general deterioration in approximation quality because there is no a priori reason to assume that f'' also vanishes at the endpoints.

2. If f' is available at the interval ends, then it is possible to administer **clamped boundary** conditions, specified by

$$v'(x_0) = f'(x_0), \quad v'(x_n) = f'(x_n).$$

   The resulting interpolant is also known as the **complete spline**.

3. A third alternative is called **not-a-knot**. In the absence of information on the derivatives of f at the endpoints we use the two remaining parameters to ensure third derivative continuity of the spline interpolant at the nearest interior break points, x_1 and x_{n-1}. Unlike the free boundary conditions, this does not require the interpolant to satisfy something that the interpolated function does not satisfy, so error quality does not deteriorate.

Example 11.5. Continuing Example 10.6, discussed on page 317, we use the MATLAB function `spline`, which implements the not-a-knot variant of cubic spline interpolation, to plot in Figure 11.4 the resulting approximation of $f(x) = 1/(1+25x^2)$ at 20 equidistant data points. The interpolant in Figure 11.4 marks a clear improvement over the high degree polynomial interpolant of Figure 10.6.

Computing the maximum absolute difference between this interpolant and $f(x)$ over the mesh `-1:.001:1` gives the error .0123, which is only three times larger than that calculated for piecewise Hermite interpolation in Example 11.2, even though twice as much data was used there! ∎

Constructing the cubic spline

The actual construction of the cubic spline is somewhat more technical than anything we have seen so far in this chapter.[45] Furthermore, the cubic spline is not entirely local, unlike the piecewise Hermite cubic. If you feel that you have a good grasp on the principle of the thing from the preceding discussion and really don't need the construction details, then skipping to (11.6) on page 343 is possible.

We begin by completing Example 11.4 and then continue to develop the construction algorithm for the general case.

[45] So much so that in the derivation there are a lot of numbered equations, an infrequent event in this book.

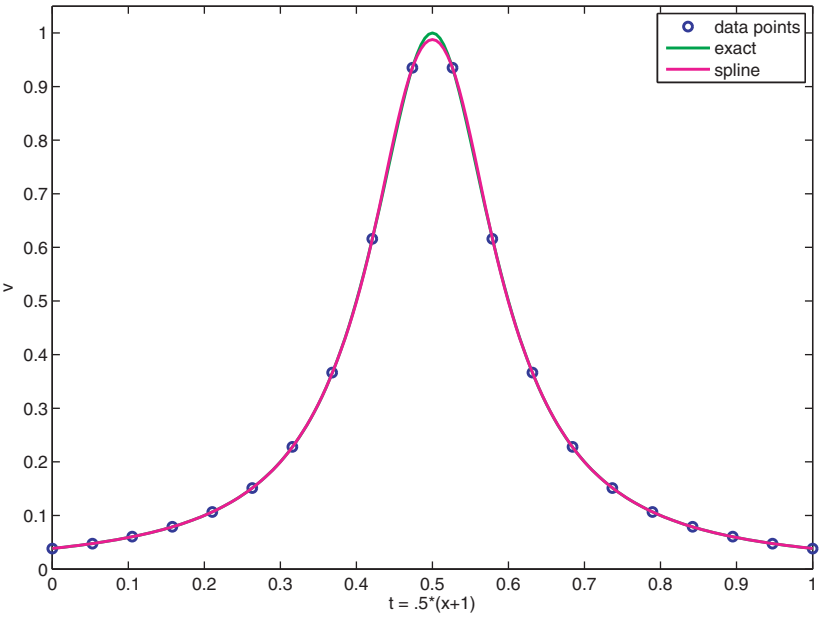


Figure 11.4. *Not-a-knot cubic spline interpolation for the Runge Example* 10.6 *at* 20 *equidistant points. The interval has been rescaled to be* $[0,1]$.

Example 11.6. Let us continue to consider the data of Example 11.4. Four equations have been constructed for the six coefficients b_0, b_1, c_0, c_1, d_0, and d_1. We require two more conditions to complete the specification of $v(x)$.

For instance, suppose we know that $f''(0.0) = f''(2.0) = 0$. Then the "natural" conditions $v''(0.0) = s_0''(0.0) = 0$, $v''(2.0) = s_1''(2.0) = 0$ are justified and yield

$$2c_0 = 0,$$
$$2c_1 + 6d_1 = 0.$$

Solving the resulting six equations yields

$$v(x) = \begin{cases} s_0(x) = 1.1 - 0.525x + 0.325x^3, & x < 1.0, \\ s_1(x) = 0.9 + 0.45(x-1.0) + 0.975(x-1.0)^2 - 0.325(x-1.0)^3, & x \geq 1.0. \end{cases}$$

Figure 11.5 depicts this interpolant. Note again the apparent smoothness at the break point depicted by a red circle, despite the jump in the third derivative. ∎

General construction

We now consider a general $n \geq 2$, so we are patching together n local cubics $s_i(x)$. Having identified the break points with the data points, the spline $v(x)$ satisfies $v(x) = s_i(x)$, $x_i \leq x \leq x_{i+1}$, $i = 0, 1, \ldots, n-1$, where

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \tag{11.2a}$$
$$s_i'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2, \tag{11.2b}$$
$$s_i''(x) = 2c_i + 6d_i(x - x_i). \tag{11.2c}$$

Our present task is therefore to determine the $4n$ coefficients a_i, b_i, c_i and d_i, $i = 0, 1, \ldots, n-1$.

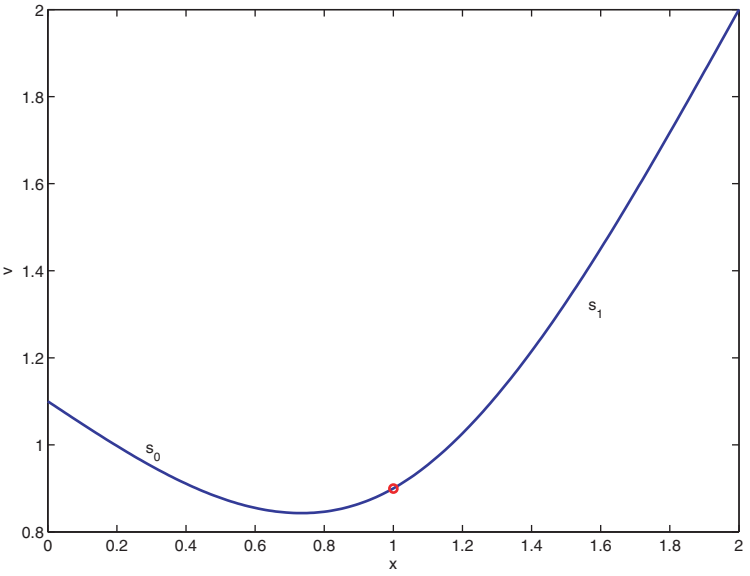


Figure 11.5. *The interpolant of Example* 11.6.

The interpolation conditions (11.1a) at the left end of each subinterval immediately determine

$$a_i = f(x_i), \quad i = 0, \ldots, n-1. \tag{11.3a}$$

Denote

$$h_i = x_{i+1} - x_i, \quad i = 0, 1, \ldots, n-1.$$

Then the interpolation conditions (11.1b) (implying continuity) give

$$a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = f(x_{i+1}).$$

Plugging the values of a_i, dividing by h_i, and rearranging gives

$$b_i + h_i c_i + h_i^2 d_i = f[x_i, x_{i+1}], \quad i = 0, \ldots, n-1. \tag{11.3b}$$

Now for the smoothness conditions. From (11.1c) and (11.2b) we get for the first derivative continuity the condition

$$b_i + 2h_i c_i + 3h_i^2 d_i = b_{i+1}, \quad i = 0, \ldots, n-2. \tag{11.3c}$$

Likewise, for the second derivative continuity, we get from (11.1d) and (11.2c) the relation

$$c_i + 3h_i d_i = c_{i+1}, \quad i = 0, \ldots, n-2. \tag{11.3d}$$

Note that we may extend (11.3c) and (11.3d) to apply also for $i = n-1$ by defining

$$b_n = v'(x_n), \ \ c_n = \frac{1}{2} v''(x_n).$$

Now, we can eliminate the d_i from (11.3d), obtaining

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \quad i = 0, \ldots, n-1, \tag{11.4a}$$

and then the b_i from (11.3b), which yields

$$b_i = f[x_i, x_{i+1}] - \frac{h_i}{3}(2c_i + c_{i+1}), \quad i = 0, \ldots, n-1. \tag{11.4b}$$

Thus, if we can determine the c_i coefficients, then all the rest are given by (11.4) (and (11.3a)).

Next, let us shift the counter $i \leftarrow i+1$ in (11.3c), so we have

$$b_{i-1} + 2h_{i-1}c_{i-1} + 3h_{i-1}^2 d_{i-1} = b_i, \quad i = 1, \ldots, n-1.$$

Plugging in (11.4b) yields

$$f[x_{i-1}, x_i] - \frac{h_{i-1}}{3}(2c_{i-1} + c_i) + 2h_{i-1}c_{i-1} + h_{i-1}(c_i - c_{i-1}) = f[x_i, x_{i+1}] - \frac{h_i}{3}(2c_i + c_{i+1}),$$

and this, rearranged, finally reads

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]),$$
$$i = 1, \ldots, n-1. \tag{11.5}$$

(We could further divide these equations by $h_{i-1} + h_i$, obtaining $3f[x_{i-1}, x_i, x_{i+1}]$ on the right-hand side, but let's not.)

In (11.5) we have a set of $n-1$ linear equations for $n+1$ unknowns. To close this system, we need two more conditions: freedom's just another word for something left to do. This is where the boundary conditions discussed earlier (see page 339) come in. An overview in algorithm form of the cubic spline construction is given on the current page. Next, we concentrate on the important choice of these two extra conditions.

Algorithm: Cubic Spline.
Given data pairs (x_i, y_i), $i = 0, 1, \ldots, n$:

1. Identify $f(x_i) \equiv y_i$, $i = 0, 1, \ldots, n$, and set $a_i = y_i$, $i = 0, 1, \ldots, n-1$.
2. Construct a tridiagonal system of equations for the unknowns $c_0, c_1, \ldots, c_n$ using the $n-1$ equations (11.5) and two more boundary conditions.
3. Solve the linear system, obtaining the coefficients c_i.
4. Set the coefficients d_i, $i = 0, 1, \ldots, n-1$, by equations (11.4a); set the coefficients b_i, $i = 0, 1, \ldots, n-1$, by equations (11.4b).
5. The desired spline $v(x)$ is given by

$$v(x) = s_i(x), \quad x_i \le x \le x_{i+1}, \quad i = 0, 1, \ldots, n-1,$$

where s_i are given by equations (11.2a).

Two extra conditions

The free boundary choice, yielding the natural spline, is the easiest; this perhaps accounts for its pedagogical popularity. It translates into simply setting

$$c_0 = c_n = 0.$$

Then we write (11.5) in matrix form, obtaining

$$\begin{pmatrix} 2(h_0+h_1) & h_1 & & & & \\ h_1 & 2(h_1+h_2) & h_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & h_{n-3} & 2(h_{n-3}+h_{n-2}) & h_{n-2} & \\ & & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{n-2} \\ \psi_{n-1} \end{pmatrix},$$

where ψ_i is shorthand for

$$\psi_i = 3(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]).$$

The matrix in the above system is nonsingular, because it is strictly diagonally dominant, i.e., in each row, the magnitude of the diagonal element is larger than the sum of magnitudes of all other elements in that row; see Section 5.3. Hence, there is a unique solution for $c_1, \ldots, c_{n-1}$ and thus a unique interpolating natural spline, whose coefficients are found by solving this linear system and then using $c_0 = 0$ and the formulas (11.3a) and (11.4). For those readers who have diligently read Chapter 5, let us add that the matrix is tridiagonal and symmetric positive definite. Hence, solving for the coefficients c_i by Gaussian elimination without pivoting is stable and simple. It requires about $4n$ flops. The entire construction process of the spline therefore costs $\mathcal{O}(n)$ operations.

These matrix equations represent a *global* coupling of the unknowns. Thus, if one data value $f(x_{i_0})$ is changed, all of the coefficients change: a bummer! However, as it turns out, the magnitude of the elements of the inverse of the matrix decay exponentially as we move away from the main diagonal. (This, incidentally, is a consequence of the strict diagonal dominance.) Consequently, c_i depends most strongly upon elements of f at abscissae closest to x_i. So, if $f(x_{i_0})$ is modified, it will cause the largest changes in the c_i which are near $i = i_0$; although all coefficients change, these changes rapidly decay in magnitude as we move away from the spot of modification of f. The nature of the approximation is therefore "almost local."

For clamped boundary conditions we must specify

$$b_0 = f'(x_0), \;\; b_n = f'(x_n).$$

This is then plugged into (11.4b) and (11.3c). After simplification we get

$$h_0(2c_0 + c_1) = 3(f[x_0, x_1] - f'(x_0)),$$
$$h_{n-1}(2c_n + c_{n-1}) = 3(f'(x_n) - f[x_{n-1}, x_n]).$$

These relations, together with (11.5), can be written as a tridiagonal system of size $n+1$ for the $n+1$ unknowns $c_0, c_1, \ldots, c_n$.

For the not-a-knot condition we set

$$d_0 = d_1, \;\; d_{n-1} = d_{n-2}.$$

This basically says that $s_0(x)$ and $s_1(x)$ together are really one cubic polynomial, and likewise $s_{n-1}(x)$ and $s_{n-2}(x)$ together are just one cubic polynomial. Inserting these conditions into (11.4a) allows elimination of c_0 in terms of c_1 and c_2, and elimination of c_n in terms of c_{n-1} and c_{n-2}. Substituting these expressions in (11.5), the obtained linear system is still tridiagonal and strictly diagonally dominant. The details are left as Exercise 5.

The error in the complete spline approximation (i.e., with clamped ends) satisfies

$$\max_{a \le x \le b} |f(x) - v(x)| \le c \max_{a \le t \le b} |f''''(t)| \max_{0 \le i \le n-1} h_i^4, \tag{11.6}$$

where $c = \frac{5}{384}$. Equivalently, we use the maximum norm notation (see page 366) to write

$$\|f - v\| \le c\|f''''\|h^4.$$

This bound is comparable (up to factor 5) with the error *bound* for the Hermite piecewise cubics (see page 336), which is quite impressive given that we are using much less information on f. The error in the not-a-knot approximation is also of similar quality, obeying a similar expression with just a different constant c.

In contrast, the natural spline is generally only second order accurate near the endpoints! Indeed, it can be shown for a general piecewise polynomial approximation that if $\max_{a\le x\le b} |f(x) - v(x)| = \mathcal{O}(h^q)$, then the error in the jth derivative satisfies

$$\max_{a\le x\le b} |f^{(j)}(x) - v^{(j)}(x)| = \mathcal{O}(h^{q-j}).$$

Thus, if it happens that $f''(x_0) = 1$, say, then the error in the second derivative of the natural spline near x_0 is $\approx 1 = \mathcal{O}(h^0)$, so $q - 2 = 0$ and this translates to an $\mathcal{O}(h^2)$ error in $v(x)$ near x_0. The lesson is that not everything "natural" is good for you.

Having described no less than four favorite piecewise polynomial interpolants, we summarize their respective properties in a table:

Interpolant	Local?	Order	Smooth?	Selling features
Piecewise constant	yes	1	bounded	Accommodates general f
Broken line	yes	2	C^0	Simple, max and min at data values
Piecewise cubic Hermite	yes	4	C^1	Elegant and accurate
Spline (not-a-knot)	not quite	4	C^2	Accurate, smooth, requires only f data

Construction cost

When contemplating taking a relatively large number of subintervals n, it is good to remember that the construction cost of the approximation, i.e., the part of the algorithm that does not depend on a particular evaluation point, is *linear* in n. In fact, the flop count is αn with a small proportionality constant α for all practical piecewise polynomial interpolation methods. This fact is obvious in particular for the methods described in Section 11.2. (Why?)

In contrast, recall that the construction cost for the Chebyshev polynomial interpolation considered in Section 10.6 (the only place in Chapter 10 where we contemplate mildly large polynomial degrees) is proportional to n^2, where now n is the polynomial degree; see Exercise 8.

Specific exercises for this section: Exercises 5–9.

11.4 Hat functions and B-splines

In Section 10.1 we have discussed the representation of an interpolant using basis functions $\phi_j(x)$. The three forms of polynomial interpolation that followed in Sections 10.2–10.4 have all been presented with references to their corresponding basis functions. In the next two chapters basis functions play a central role. But for the piecewise polynomial approximations presented in Sections 11.2

Note: The material in the present section may be studied almost independently from the material in Sections 11.2 and 11.3.

and 11.3 no basis functions have been mentioned thus far. Do such bases exist, and if they do, aren't they relevant?

Let us first hasten to reconfirm that such basis functions do exist. The general linear form for the approximant, written as

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x) = c_0\phi_0(x) + \cdots + c_n\phi_n(x),$$

does apply also to piecewise polynomial interpolation. The reason we have not made a fuss about it thus far is that a simpler and more direct determination of such interpolants is often achieved using local representation, building upon polynomial basis functions.

But in other applications, for instance, in the solution of boundary value problems for differential equations, the approximated function is given only indirectly, through the solution of the differential problem, and the need to express the discretization process in terms of basis functions is more prevalent.

In such cases, it is important to choose basis functions that have as narrow a support as possible. The **support** of a function is the subinterval on which (and only on which) it may have nonzero values. If basis functions with narrow support can be found, then this gives rise to local operations in x or, if not entirely local, it typically translates into having a sparse matrix for the linear system of equations that is assembled in the discretization process. This, in turn, gives rise to savings in the computational work required to solve the linear system.

Hat functions

As usual, everything is simple for the case of continuous, piecewise linear approximation. Recall from Section 11.2 that in this case we have

$$(x_0, x_1, x_2, \ldots, x_{n-1}, x_n) = (t_0, t_1, t_2, \ldots, t_{r-1}, t_r),$$

and $r = n$.

For any linear combination of the basis functions ϕ_j to be a continuous, piecewise linear function, the basis functions themselves must be of this type. Insisting on making their support as narrow as possible leads to the famous *hat functions* given by

$$\phi_j(x) = \begin{cases} \frac{x - x_{j-1}}{x_j - x_{j-1}}, & x_{j-1} \le x < x_j, \\ \frac{x - x_{j+1}}{x_j - x_{j+1}}, & x_j \le x < x_{j+1}, \\ 0 & \text{otherwise.} \end{cases}$$

The end functions ϕ_0 and ϕ_n are not defined outside the interval $[x_0, x_n]$. See Figure 11.6.

Like the Lagrange polynomials, these basis functions satisfy

$$\phi_j(x_i) = \begin{cases} 0, & i \ne j, \\ 1, & i = j. \end{cases}$$

Therefore, like the Lagrange polynomial representation, the coefficients c_j are none other than the data ordinates themselves, so we write

$$c_j = y_j = f(x_j), \quad j = 0, 1, \ldots, n.$$

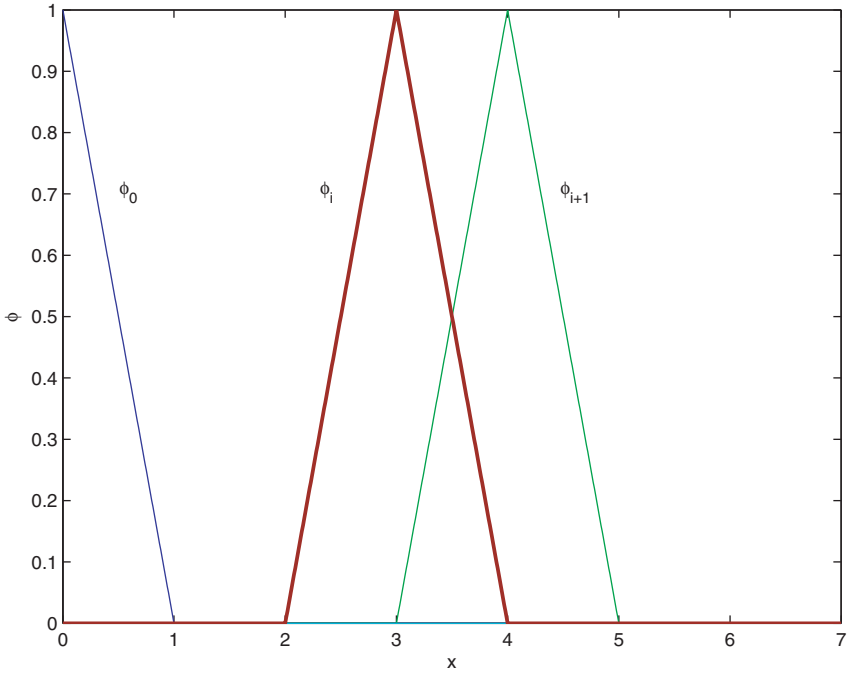


Figure 11.6. *Hat functions: a compact basis for piecewise linear approximation.*

Hat functions appear in almost any text on the **finite element method** (FEM). In that context, representations that satisfy $c_j = f(x_j)$ are called **nodal methods**.

Hat functions form an instance of basis functions with *local support*: the support of each basis function $\phi_j(x)$ spans only a few (two in this instance) mesh subintervals about x_j. Moreover, a change in one data value y_i obviously affects only $c_i\phi_i(x)$ and thus only values of $v(x)$ in the very narrow vicinity of x_i, namely, the subinterval (x_{i-1}, x_{i+1}).

Often in finite element methods and other techniques related to piecewise interpolation, everything is defined and constructed on a particularly simple interval, say, $[-1, 1]$. In the present case the basis functions can in fact be viewed as simple scalings and translations of one *mother hat function*, given by

$$\phi(z) = \begin{cases} 1+z, & -1 \le z < 0, \\ 1-z, & 0 \le z < 1, \\ 0 & \text{otherwise.} \end{cases}$$

The two polynomial segments of this mother function, $\psi_1(z) = 1+z$ and $\psi_2(z) = 1-z$, are the building blocks for constructing all hat functions.

Hermite cubic basis

Moving on to more challenging piecewise polynomial spaces, let us consider those based on cubic polynomials. The next simplest case is that of Hermite piecewise cubics. Like the piecewise linear approximation, the Hermite interpolation is completely local, as we have seen in Section 11.2. Indeed, the observations of the previous paragraph apply also for the Hermite interpolation, except that

the notation and the notion of "mother function" are more involved. In fact, there are two mothers here!

Let us define four interpolating cubic polynomials (as compared to two in the piecewise linear case) on the interval $[0,1]$ by

$$\begin{aligned}
&\psi_1(0)=1,\ \psi_1'(0)=0,\ \psi_1(1)=0,\ \psi_1'(1)=0,\\
&\psi_2(0)=0,\ \psi_2'(0)=1,\ \psi_2(1)=0,\ \psi_2'(1)=0,\\
&\psi_3(0)=0,\ \psi_3'(0)=0,\ \psi_3(1)=1,\ \psi_3'(1)=0,\\
&\psi_4(0)=0,\ \psi_4'(0)=0,\ \psi_4(1)=0,\ \psi_4'(1)=1.
\end{aligned}$$

Figuring out the cubic polynomials defined in this way is straightforward based on the Hermite interpolation procedure described in Section 10.7. We obtain $\psi_1(z)=1-3z^2+2z^3$, $\psi_2(z)=z-2z^2+z^3$, $\psi_3(z)=3z^2-2z^3$, and $\psi_4(z)=-z^2+z^3$. See Exercise 10.

A global basis function may now be constructed from $\psi_l(z), l=1,2,3,4$. Recall that for Hermite interpolation we have

$$(x_0,x_1,\ldots,x_n)=(t_0,t_0,\ldots,t_r,t_r),$$

from which it follows that $n=2r+1$. We thus need $2r+2$ basis functions. Define for $0\le k\le r$ the functions

$$\xi_k(x)=\begin{cases}\psi_1\left(\frac{x-t_k}{t_{k+1}-t_k}\right), & t_k\le x<t_{k+1},\\ \psi_3\left(\frac{x-t_{k-1}}{t_k-t_{k-1}}\right), & t_{k-1}\le x<t_k,\\ 0 & \text{otherwise,}\end{cases}$$

and

$$\eta_k(x)=\begin{cases}\psi_2\left(\frac{x-t_k}{t_{k+1}-t_k}\right)\cdot(t_{k+1}-t_k), & t_k\le x<t_{k+1},\\ \psi_4\left(\frac{x-t_{k-1}}{t_k-t_{k-1}}\right)\cdot(t_k-t_{k-1}), & t_{k-1}\le x<t_k,\\ 0 & \text{otherwise.}\end{cases}$$

The functions $\xi_k(x)$ and $\eta_k(x)$ are depicted in Figure 11.7. Clearly, these are piecewise cubic functions in C^1 which have local support. The ξ_k's are all scalings and translations of one mother Hermite cubic, and the η_k's likewise relate to another mother. See Exercise 11.

Our Hermite piecewise interpolant is now expressed as

$$v(x)=\sum_{k=0}^{r}\Big(f(t_k)\xi_k(x)+f'(t_k)\eta_k(x)\Big).$$

With respect to the standard form $v(x)=\sum_{j=0}^{n}c_j\phi_j(x)$, we have here that $n+1=2(r+1)$, $\phi_{2k}(x)=\xi_k(x)$, and $\phi_{2k+1}(x)=\eta_k(x)$, $k=0,1,\ldots,r$. The interpolation properties of the basis functions allow us to identify $c_{2k}=f(t_k)$ and $c_{2k+1}=f'(t_k)$, so in FEM jargon this is a nodal method.

B-splines

While the case for Hermite cubics is almost as easy (in principle!) as for the broken line interpolation, the case for cubic spline interpolation cannot be that simple because the approximation process is no longer entirely local. Rather than looking for a special remedy for cubic splines, we turn instead to a general method for constructing basis functions with local support for piecewise polynomial spaces. The resulting basis functions, called *B-splines*, do not yield a nodal method, but they possess many beautiful properties and have proved useful in CAGD.

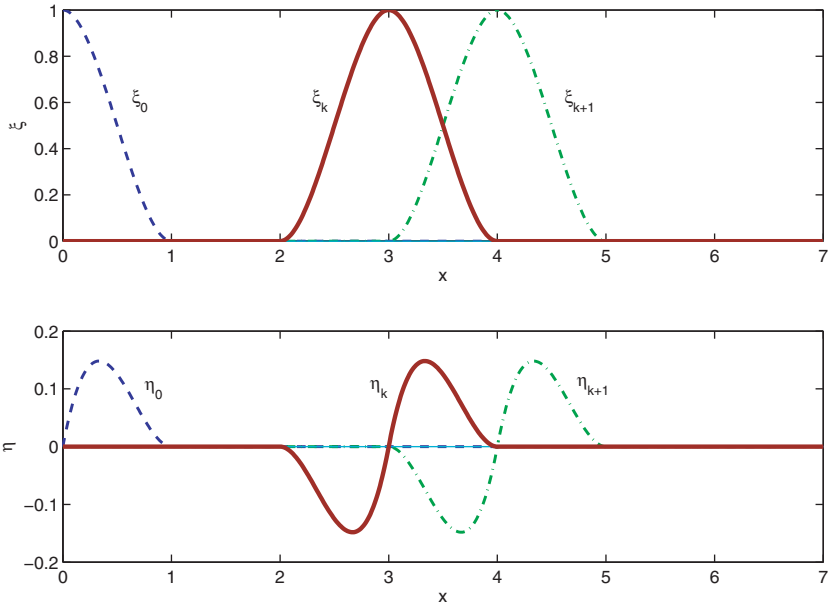


Figure 11.7. *Basis functions for piecewise Hermite polynomials.*

Note: Be warned that the remainder of this section is inherently technical.

Suppose then that we have a sequence of *break points* $t_0, t_1, \ldots, t_r$ and that we wish to form a basis for an approximation space such that in each subinterval $[t_{i-1}, t_i]$ the approximation is a polynomial of degree at most l and such that globally the approximation is in C^m, $m < l$. We form a sequence of *knots* x_i by repeating the values t_j in the sequence, which yields

$$(x_0, x_1, x_2, \ldots, x_n) = \left(\underbrace{t_0, t_0, \ldots, t_0}_{l+1}, \underbrace{t_1, \ldots, t_1}_{l-m}, \ldots, \underbrace{t_{r-1}, \ldots, t_{r-1}}_{l-m}, \underbrace{t_r, \ldots, t_r}_{l+1} \right).$$

The *truncated power function* $w(\tau) = (\tau)_+^k$ is defined as

$$(\tau)_+^k = [\max\{\tau, 0\}]^k.$$

Using this, define

$$M_j(x) = g_x[x_j, \ldots, x_{j+l+1}],$$

where

$$g_x(t) = (t - x)_+^l.$$

(By this notation we mean that each real value x is held fixed while finding the divided difference of g_x in t. See page 308 for the definition of divided differences.) The B-spline of degree l is

$$\phi_j(x) \equiv B_j(x) = (x_{j+l+1} - x_j) M_j(x).$$

These functions form a basis for piecewise polynomials of degree l and global smoothness m. Figure 11.8 depicts the shape of B-spline basis functions for the cubic spline ($m = 2, l = 3$). The basis has a number of attractive features:

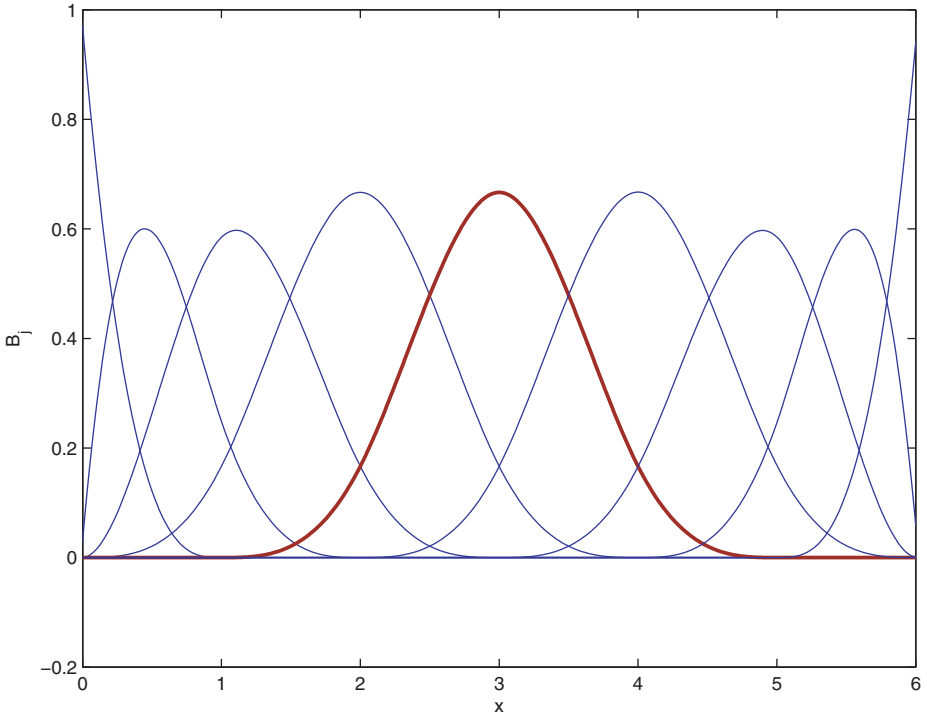


Figure 11.8. *B-spline basis for the C^2 cubic spline.*

- The function B_j is positive on the interval (x_j, x_{j+l+1}) and zero elsewhere, so it has a local support.

- The evaluation of B-splines can be done recursively, a property inherited from the divided difference function used in the definition of M_j above.

- At each point x, $\sum_{j=0}^{n} \phi_j(x) = \sum_j B_j(x) = 1$.

Since the B-spline values are never negative they form a *partition of unity*. Although the value c_j of the obtained interpolant is not equal to the corresponding given data y_j, it is not far from it either. Hence the values of c_j may be used as control points in a CAGD system.

Specific exercises for this section: Exercises 10–12.

11.5 Parametric curves

Suppose we are given a set of data with repeated abscissae and that the data are to be joined in a curve *in that order*—see Figure 11.9. The curve C joining these points cannot be expressed as a function of either coordinate variable. Thus, none of our interpolation techniques thus far can generate C without breaking it up into subproblems, such that on each, y can be expressed as a univalued function of x.

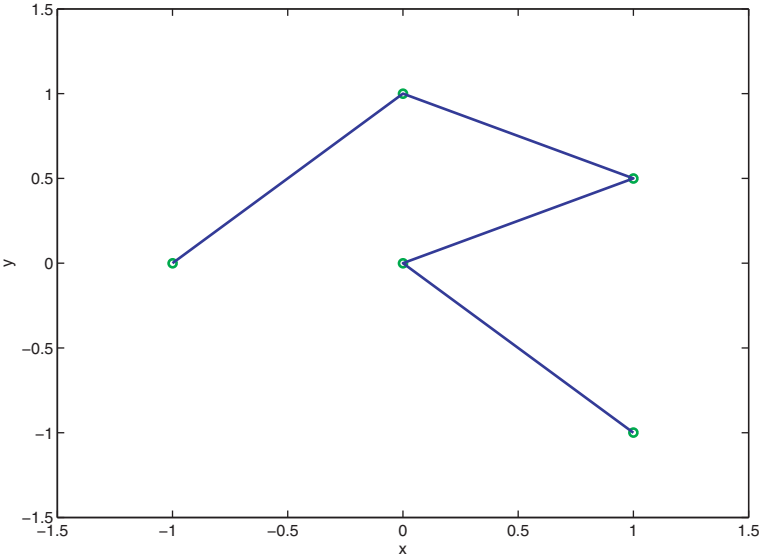


Figure 11.9. *Parametric broken-line interpolation.*

To construct an interpolating approximation in this case, we **parametrize** both abscissa and ordinate. In other words, we find a representation

$$(\tau_0, x_0), (\tau_1, x_1), (\tau_2, x_2), \ldots, (\tau_n, x_n),$$
$$(\tau_0, y_0), (\tau_1, y_1), (\tau_2, y_2), \ldots, (\tau_n, y_n),$$

where $\tau_0 < \tau_1 < \cdots < \tau_n$ is a partition of some interval in the newly introduced common variable τ. For instance, in the absence of a reason not to do so, set

$$\tau_i = i/n, \quad i = 0, 1, \ldots, n.$$

Denote the parametrized curve by

$$C : (X(\tau), Y(\tau)).$$

Then the interpolation conditions are

$$X(\tau_i) = x_i, \qquad Y(\tau_i) = y_i \quad i = 0, 1, \ldots, n,$$

and the curve C is given at any point $\tau \in [\tau_0, \tau_n]$ by $(X(\tau), Y(\tau))$.

For constructing $X(\tau)$ and $Y(\tau)$ we may now use any of the techniques seen so far in this chapter. Different parametric curve fitting techniques differ in the choice of interpolation method used.

Example 11.7. Suppose we are given the data $\{(-1,0), (0,1), (1,0.5), (0,0), (1,-1)\}$. If we issue the MATLAB commands

```
xi = [-1,0,1,0,1];
yi = [0,1,0.5,0,-1];
plot (xi,yi,'go',xi,yi)
axis([-1.5,1.5,-1.5,1.5])
xlabel('x')
ylabel('y')
```

then Figure 11.9 is obtained. The part `plot(xi,yi,'go')` of the plot command merely plots the data points (in green circles), whereas the default `plot(xi,yi)` does a broken line interpolation (in default blue) of $X(\tau)$ vs. $Y(\tau)$.

Suppose instead that we use polynomial interpolation for each of $X(\tau)$ and $Y(\tau)$. Since we have the two MATLAB functions `divdif` and `evalnewt` presented in Section 10.4 already coded, we can obtain Figure 11.10 using the script

```
ti = 0:0.25:1;
coefx = divdif(ti,xi);
coefy = divdif(ti,yi);
tau = 0:.01:1;
xx = evalnewt(tau,ti,coefx);
yy = evalnewt(tau,ti,coefy);
plot (xi,yi,'go',xx,yy)
axis([-1.5,1.5,-1.5,1.5])
xlabel('x')
ylabel('y')
```

Note again that the parametric curves in Figures 11.10 and 11.9 interpolate the same data set! ■

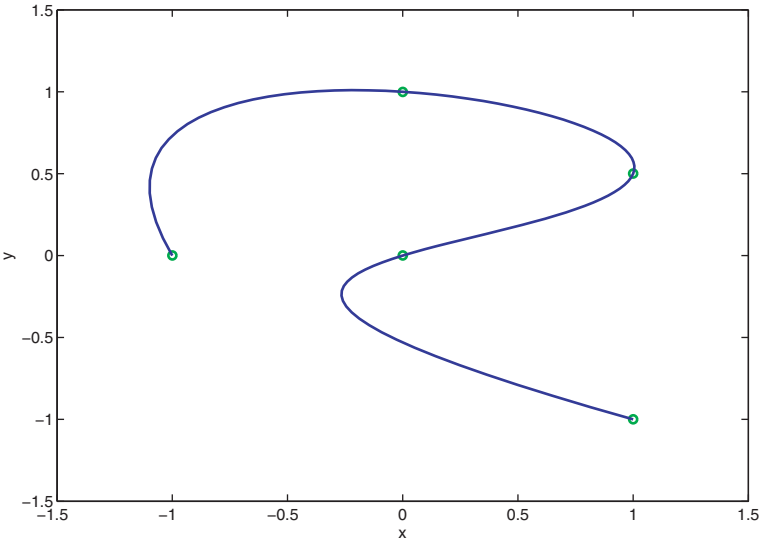


Figure 11.10. *Parametric polynomial interpolation.*

Parametric piecewise cubic hermite polynomials

Computer graphics and CAGD applications make extensive use of parametric curve fitting. These applications require

- rapid generation of curves, and
- easily modified curves. In particular, a modification should be *local*; changing one or two data points should not affect the entire curve.

The second criterion rules out global interpolating polynomials. The natural approach here is to use piecewise cubic Hermite polynomials. In a drawing package, for instance, one may specify, for two consecutive data points (x_0, y_0) and (x_1, y_1), also *guidepoint*s $(x_0+\alpha_0, y_0+\beta_0)$ and $(x_1-\alpha_1, y_1-\beta_1)$. Then the software constructs the two Hermite cubics $X(\tau)$ and $Y(\tau)$ satisfying

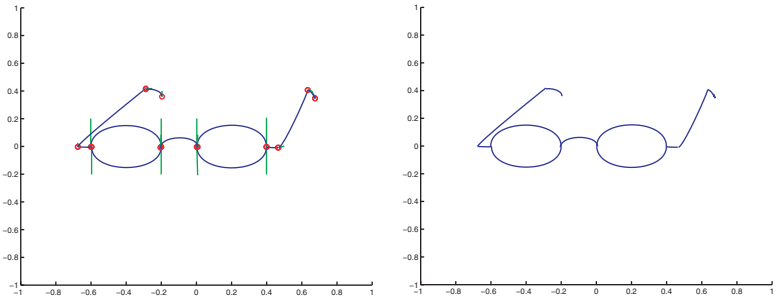
$$X(0) = x_0,\ X(1) = x_1,\ X'(0) = \alpha_0,\ X'(1) = \alpha_1,$$
$$Y(0) = y_0,\ Y(1) = y_1,\ Y'(0) = \beta_0,\ Y'(1) = \beta_1.$$

It is easy to come up with an explicit formula for each of $X(\tau)$ and $Y(\tau)$; see Exercise 4.

Common graphics systems use local **Bézier polynomials**,[46] which are cubic Hermite polynomials with a built-in scaling factor of 3 for the derivatives (i.e., $X'(0) = 3\alpha_0,\ Y'(1) = 3\beta_1$). This scaling is transparent to the user and is done to make curves more sensitive to the guidepoint position; also, the mouse does not have to be moved off screen when specifying the guidepoints.[47] Please verify that for $0 \le \tau \le 1$ we get the formulas

$$X(\tau) = (2(x_0 - x_1) + 3(\alpha_0 + \alpha_1))\,\tau^3 + (3(x_1 - x_0) - 3(\alpha_1 + 2\alpha_0))\,\tau^2 + 3\alpha_0\tau + x_0,$$
$$Y(\tau) = (2(y_0 - y_1) + 3(\beta_0 + \beta_1))\,\tau^3 + (3(y_1 - y_0) - 3(\beta_1 + 2\beta_0))\,\tau^2 + 3\beta_0\tau + y_0.$$

Example 11.8. Figure 11.11 shows a somewhat primitive design of a pair of glasses using 11 Bézier polynomials. MATLAB was employed using the function `ginput` to input point coordinates from the figure's screen.



(a) With guidepoints displayed (green).

(b) Guidepoints hidden.

Figure 11.11. *A simple curve design using* 11 *Bézier polynomials.*

The design was deliberately kept simple so as to make the point that with more dedication you can do much better than this using the Bézier curves. Indeed, the best way to get the hang of this procedure is to try it out interactively, along the lines of Exercise 14. The programming involved is not difficult! ∎

Specific exercises for this section: Exercises 13–14.

[46] Pierre Bézier came up with these polynomials in the 1970s while designing car shapes for the French auto maker Renault.

[47] Remember that τ is an arbitrarily chosen parametrization. For instance, if we define the parameter $\sigma = c\tau$ for some scaling factor $c \neq 0$, then at any point $\frac{dY/d\tau}{dX/d\tau} = \frac{cdY/d\sigma}{cdX/d\sigma} = \frac{dY/d\sigma}{dX/d\sigma}$.

11.6 *Multidimensional interpolation

All approximating functions that we have seen thus far in Chapter 10 and in the present chapter were essentially functions of one variable: typically, we are looking for

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x),$$

using some basis functions $\phi_j(x)$. The function $v(x)$ describes a **curve**. In Section 11.5 we have extended the definition of our interpolant $v(x)$, using parametrization to allow the curve to roam more flexibly in the plane (in two dimensions). It is also possible to parametrize a function in one variable so as to describe a curve in a box (in three dimensions).

But there are many applications where the sought interpolating function is a function of more than one independent variable—a **surface**. An example is a digitized black-and-white photograph, which is really a function of two variables describing gray levels at each pixel, with the pixels arranged in an underlying two-dimensional array. A color photo is the same except that there are three such functions—for red, green, and blue. Another example is a body such as a sofa that is made of inhomogeneous material, or a box containing dollar bills buried in the sand. The conductivity or permeability at each point of such an article is a function in three space dimensions. Between these there is the case of the surface of a body in three dimensions, which is a manifold in two dimensions: analogous to the parametrized curve, each point of such a surface is described as a function of three variables, but at least in principle it could be described as a function of two variables using a clever parametrization.

Make no mistake: the passage from one to more independent variables is often far from simple or straightforward. For one thing, while in one dimension the domain of definition (where x roams) is just an interval or, at worst, a union of intervals, in two dimensions the domain need not be a rectangle or even a disk, as a casual glance at the shape of islands and continents on a flattened world map reveals. Moreover, the task of finding appropriate parametrization for surfaces and bodies in more than one variable often becomes prohibitively complex. For instance, people often view the surface of a body (such as part of the human body) as a collection of points in $\mathcal{R}^3$, each having three coordinates, that may or may not be connected by some local neighboring relations to form a *surface mesh*.

There is no one quick approach for all interpolation problems here. The following central questions arise:

- Is the data scattered or located on a grid?
- What is the purpose of the interpolating function (i.e., where is it expected to be evaluated and how often)?

Below we quickly survey but a sample of related techniques and ideas.

Interpolation on a rectangular grid

In general, the interpolation problem in two dimensions, extending Chapter 10 and Sections 11.2–11.4, can be written as seeking a function $v(x, y)$ that satisfies

$$v(x_i, y_i) = f_i, \quad i = 0, 1, \ldots, n.$$

Note that here, unlike before, y_i is the abscissa coordinate in direction y, while the corresponding data value is denoted by f_i. The points (x_i, y_i) for which function (or data ordinate) values are given are contained in some domain Ω that need not be rectangular.

Before getting to this general case, however, let us consider a more special though important one. We assume that there are abscissae in x and y separately, e.g., $x_0 < x_1 < \cdots < x_{N_x}$ and $y_0 < y_1 < \cdots < y_{N_y}$, such that the $n+1 = (N_x+1)(N_y+1)$ data points are given as $(x_i, y_j, f_{i,j})$. So, the interpolation problem is to find $v(x,y)$ satisfying

$$v(x_i, y_j) = f_{i,j}, \quad i = 0, 1, \ldots, N_x, \;\; j = 0, 1, \ldots, N_y.$$

In this special case we can think of seeking $v(x,y)$ in the *tensor product* form

$$v(x,y) = \sum_{k=0}^{N_x} \sum_{l=0}^{N_y} c_{k,l} \phi_k(x) \psi_l(y),$$

where ϕ_k and ψ_l are *one-dimensional basis functions*. This is why it is simpler, despite the somewhat cumbersome notation. In fact, extensions of this specific case to more dimensions are in principle straightforward, too. The multidimensional interpolation problem is decoupled into a product of one-dimensional ones.

For a polynomial interpolation we can think, for instance, of a *bilinear* or a *bicubic* polynomial. These are defined on a square or a rectangle, with data typically given at the four corners, and are often considered as a surface *patch*, filling in missing data inside the rectangle. Obviously, without data given inside the rectangle we would need some additional data to determine the bicubic.

Example 11.9 (bilinear interpolation). This example is longish but should be straightforward. Let us consider first a bilinear polynomial interpolation over the unit square depicted on the left of Figure 11.12. The data points are $(0,0,f_{0,0})$, $(1,0,f_{1,0})$, $(0,1,f_{0,1})$, and $(1,1,f_{1,1})$.

We write our interpolant as

$$v(x,y) = c_0 + c_1 x + c_2 y + c_3 xy.$$

Then substituting the data directly gives

$$\begin{aligned} c_0 &= f_{0,0}, \\ c_0 + c_1 = f_{1,0} \;&\Rightarrow\; c_1 = f_{1,0} - f_{0,0}, \\ c_0 + c_2 = f_{0,1} \;&\Rightarrow\; c_2 = f_{0,1} - f_{0,0}, \\ c_0 + c_1 + c_2 + c_3 = f_{1,1} \;&\Rightarrow\; c_3 = f_{1,1} + f_{0,0} - f_{0,1} - f_{1,0}. \end{aligned}$$

This completes the construction of the interpolant $v(x,y)$. Next, suppose the task is to interpolate at mid-edges and at the center of the square. We obtain

$$\begin{aligned} v(.5,0) &= c_0 + .5c_1 = \frac{1}{2}(f_{0,0} + f_{1,0}), \\ v(0,.5) &= c_0 + .5c_2 = \frac{1}{2}(f_{0,0} + f_{0,1}), \\ v(.5,1) &= c_0 + c_2 + .5(c_1 + c_3) = \frac{1}{2}(f_{0,1} + f_{1,1}), \\ v(1,.5) &= c_0 + c_1 + .5(c_2 + c_3) = \frac{1}{2}(f_{1,0} + f_{1,1}), \\ v(.5,.5) &= c_0 + .5c_1 + .5c_2 + .25c_3 = \frac{1}{4}(f_{0,0} + f_{1,0} + f_{0,1} + f_{1,1}). \end{aligned}$$

(You could have probably guessed these expressions directly. But it's good to know how they are obtained in an orderly manner.)

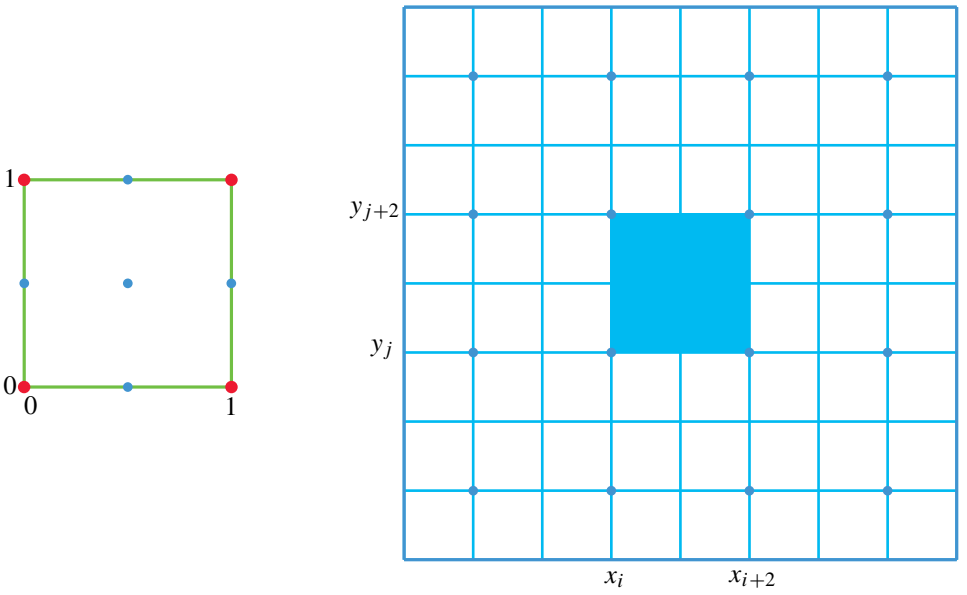


Figure 11.12. *Bilinear interpolation. Left green square: data are given at the unit square's corners (red points), and bilinear polynomial values are desired at mid-edges and at the square's middle (blue points). Right blue grid: data are given at the coarser grid nodes (blue points) and a bilinear interpolation is performed to obtain values at the finer grid nodes, yielding values* $f_{i,j}$ *for all* $i = 0,1,2,\ldots,N_x$, $j = 0,1,2,\ldots,N_y$.

Now, suppose we have a rectangular, uniform grid in the (x,y)-plane and data given at every other node of the grid, as depicted in the right part of Figure 11.12. The data value at point (x_i, y_j) is $f_{i,j}$, and we are required to supply values at all points of the finer grid.

Let us pick the particular grid square $[x_i, x_{i+2}] \times [y_j, y_{j+2}]$, whose corner values are given. To this we fit an interpolating bilinear polynomial, which is a scaled version of the one developed above, given by

$$v(x,y) = c_0 + c_1 \frac{(x-x_i)}{(x_{i+2}-x_i)} + c_2 \frac{(y-y_j)}{(y_{j+2}-y_j)} + c_3 \frac{(x-x_i)}{(x_{i+2}-x_i)} \frac{(y-y_j)}{(y_{j+2}-y_j)}.$$

The same formulas as above follow with the obvious notational change. Thus, we evaluate

$$\begin{aligned}
f_{i+1,j} &= \frac{1}{2}(f_{i,j} + f_{i+2,j}),\\
f_{i,j+1} &= \frac{1}{2}(f_{i,j} + f_{i,j+2}),\\
f_{i+1,j+2} &= \frac{1}{2}(f_{i,j+2} + f_{i+2,j+2}),\\
f_{i+2,j+1} &= \frac{1}{2}(f_{i+2,j} + f_{i+2,j+2}),\\
f_{i+1,j+1} &= \frac{1}{4}(f_{i,j} + f_{i+2,j} + f_{i,j+2} + f_{i+2,j+2}).
\end{aligned}$$

Of course this procedure can (and should) be carried out throughout the grid all at once, using vector operations, rather than one square at a time. The resulting bilinear interpolation procedure from coarse to fine grid has been implemented in our MATLAB multigrid program of Example 7.17 in Section 7.6. ∎

More generally, a *bicubic spline* can be constructed from (many) one-dimensional cubic splines such as described in Section 11.3 for a general-purpose smooth interpolant of data on a grid.

Triangular meshes and scattered data

We now consider piecewise polynomial interpolation in the plane and denote $\mathbf{x} = (x, y)^T$. As is the case for one space variable, there are applications where the abscissae of the data are given and others where these locations are ours to choose.

In the latter case the "data," or function values, are often given implicitly, say, as the solution of some partial differential equation on a domain $\Omega \subset \mathcal{R}^2$. The locations of the interpolation points then form part of a discretization scheme for the differential equation, usually a *finite element* or a **finite volume** method, whose complete description is beyond the scope of this text. Suffice it to say here that the domain Ω is divided into nonoverlapping elements (triangles or rectangles), on each of which the sought solution function is a polynomial in two variables. The solution constructed out of these polynomial pieces should maintain an overall degree of smoothness. The union of these elements should form a good approximation of Ω and they each should be small enough in diameter to allow for a high quality approximation overall. Of course, if the domain Ω is square, then we would happily discretize it with a tensor grid as in Examples 7.1 and 11.9. But if the domain has a more complex boundary, then a tensor grid will not do. In more complex geometries, triangles are often better building blocks than rectangles; see Figure 11.13.

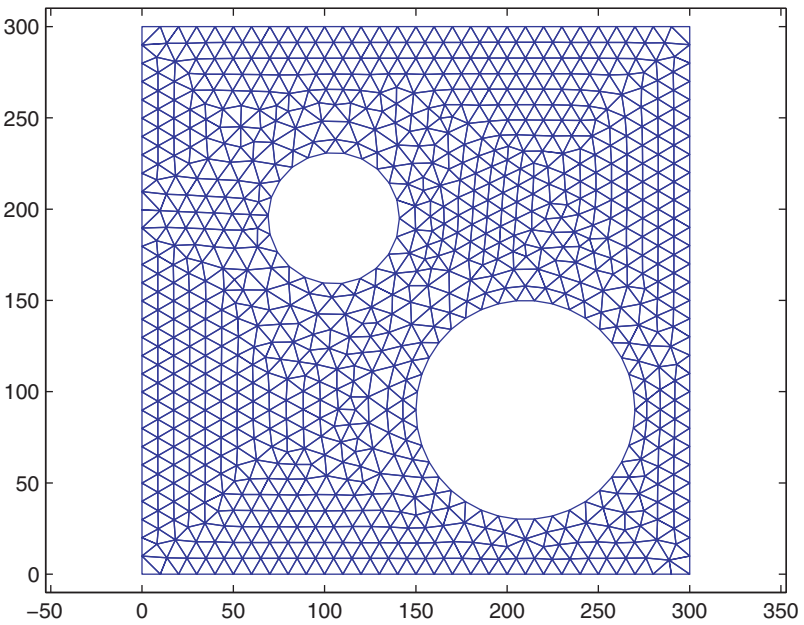


Figure 11.13. *Triangle mesh in the plane. This one is MATLAB's data set* `trimesh2d`.

In the case where the locations of the data are fixed as part of the given interpolation problem, there are also many situations where we would naturally want to form a piecewise polynomial interpolant. Thus we are led, in either case, to the following basic tasks:

1. Construct a triangulation for a given set of points in the plane.
2. Construct a piecewise polynomial interpolant on the given triangle mesh.

Of the constructed triangulation we want if possible not to have small or large (more than 90°) angles: triangles that are not too far from being equilateral allow for good approximation properties. This leads to *Delaunay triangulation*, a subject from *computational geometry* that again we just mention rather than fully explain here. MATLAB has several commands that deal with point set triangulation in both two and three dimensions: check out `delaunaytri`, `triplot`.

Now that we have a triangle mesh, we wish to construct an interpolating piecewise polynomial on it. We must ensure the desired number of continuous derivatives, not only across nodes as in one dimension, but also across triangle edges. This can quickly turn hairy.

By far the most popular option is to use linear polynomials, where everything is still relatively simple. On a triangle T_j having vertices $\mathbf{x}_i^j$, $i = 0,1,2$, we write

$$v(\mathbf{x}) = v(x,y) = c_0^j + c_1^j x + c_2^j y.$$

If the corresponding data ordinates are f_i^j, then the three interpolation conditions

$$v(\mathbf{x}_i^j) = f_i^j, \quad i = 0,1,2,$$

obviously determine the three coefficients c_i^j of the linear polynomial piece. Moreover, continuity across vertices between neighboring triangles is automatically achieved, and so is continuity across edges, because the value of a linear polynomial along an edge is determined by its values at the edge's endpoints (which are the vertices common to the two neighboring triangles). See Figure 11.14. In MATLAB, check out `triscatteredinterp`.

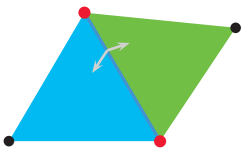


Figure 11.14. *Linear interpolation over a triangle mesh. Satisfying the interpolation conditions at triangle vertices implies continuity across neighboring triangles.*

Smoother, higher order piecewise polynomial interpolants are also possible, and they are discussed in finite element texts. In three dimensions the triangles become tetrahedra while the rectangles become boxes, affording considerably less headache than tetrahedra when tracking them. Thus, engineers often stick with the latter elements after all, at least when the geometry of Ω is not too complicated.

In computer graphics and computer aided geometric design it is popular to construct triangle surface meshes such as in Figure 11.15, both for display and for purposes of further manipulation such as morphing or design.

Radial basis functions for scattered data interpolation

We now leave the land of piecewise polynomials. Given a set of scattered data to interpolate, it may not be worthwhile to triangulate it if the purpose of interpolation is just the display of a pleasingly coherent surface.

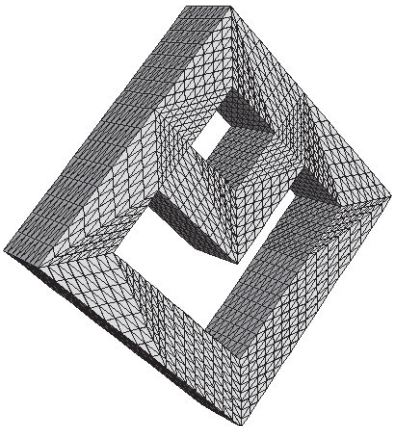


Figure 11.15. *Triangle surface mesh.*

A very popular technique for the latter purpose is radial basis function (RBF) interpolation. Let $|\mathbf{z}|$ denote the ℓ_2-norm of a point $\mathbf{z}$ in two or three dimensions. Given data $(\mathbf{x}_i, f_i)$, $i = 0, 1, \ldots, n$, the values $\phi(|\mathbf{x} - \mathbf{x}_j|)$ of an RBF ϕ measure the distances from a point $\mathbf{x}$ to the data abscissae. Thus, $\phi(\mathbf{z})$ depends only on the radial distance $|\mathbf{z}|$.

In the simplest version of this method we look for an interpolant

$$v(\mathbf{x}) = \sum_{j=0}^{n} w_j \phi(|\mathbf{x} - \mathbf{x}_j|),$$

where the weights w_j are determined by the $n+1$ interpolation conditions

$$f_i = v(\mathbf{x}_i) = \sum_{j=0}^{n} w_j \phi(|\mathbf{x}_i - \mathbf{x}_j|), \quad i = 0, 1, \ldots, n.$$

These are $n+1$ linear equations for $n+1$ unknown weights w_j, so we solve the system once and then construct the interpolant using the formula for $v(\mathbf{x})$.

A good choice for ϕ in two dimensions is the *multiquadric*

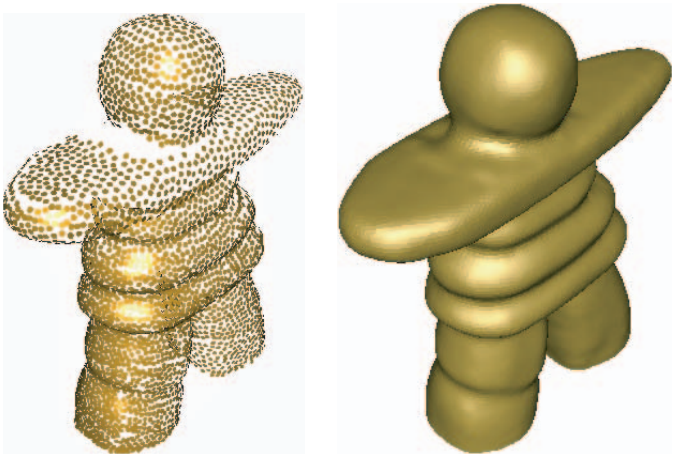
$$\phi(r) = \sqrt{r^2 + c^2},$$

where c is a user-determined parameter. In three dimensions the *biharmonic spline*

$$\phi(r) = r$$

plus an extra linear polynomial is recommended. There are other choices in both two and three dimensions.

The algorithm as presented above works reasonably well if n is not too large and the points are reasonably spread out. But for point clouds that contain many points, or very tightly clustered points, modifications must be made. These are well beyond the scope of our quick presentation, so let us just demonstrate the approach by an example.



(a) Consolidated point cloud.

(b) RBF surface.

Figure 11.16. *RBF interpolation of an upsampling of a consolidated point cloud.*

Example 11.10. Figure 11.16(a) shows a set of points with directions that was obtained from a raw scanned point cloud of an inukshuk-like three-dimensional model[48] following denoising, removal of outlier points, and association of a normal to each point (thus generating a so-called *surfel*; see Example 4.18). The data was further up-sampled, and the resulting scattered surfel data was interpolated by the FastRBF software of Far Field Technology to form the surface in (b). Note that this is just one still shot of a three-dimensional surface.

We don't expect you to be able to reproduce Figure 11.16 based on the above sketchy description. The purpose, rather, is to show that the simple RBF idea can be made to produce impressive practical results. ∎

11.7 Exercises

0. **Review questions**

   (a) What is a piecewise polynomial?

   (b) State three shortcomings of polynomial interpolation that are improved upon by piecewise polynomial interpolation.

   (c) State at least one advantage and one disadvantage of piecewise constant interpolation.

   (d) State at least two advantages and two disadvantages of broken line interpolation.

   (e) What is a piecewise cubic Hermite?

   (f) In what different ways are the cubic spline and piecewise cubic Hermite useful interpolants?

   (g) Define the different end conditions for cubic spline interpolation, giving rise to the natural, complete, and not-a-knot variants. Which of these is most suitable for general-purpose implementation, and why?

[48] The inukshuk is a manmade stone landmark, used by peoples of the Arctic region of North America. It formed the basis of the logo of the 2010 Winter Olympics in Vancouver.

(h) What is a tridiagonal matrix and how does it arise in spline interpolation?

(i) What is the support of a function, and what can be said about the support of basis functions for piecewise interpolation?

(j) How are basis functions for piecewise polynomial interpolation different from basis functions for polynomial interpolation of the sort discussed in Chapter 10?

(k) What is a nodal method?

(l) Define a hat function and explain its use.

(m) In what ways is the B-spline similar to the hat function?

(n) How do parametric curves arise?

(o) Define the Bézier polynomial and explain its use in CAGD.

1. Consider the piecewise constant interpolation of the function

$$f(x) = \sin(x), \quad 0 \le x \le 381,$$

at points $x_i = ih$, where $h = 0.1$. Thus, our interpolant satisfies

$$v(x) = \sin(ih), \quad (i - .5)h \le x < (i + .5)h,$$

for $i = 0, 1, \ldots, 3810$.

(a) Find a bound for the error in this interpolation.

(b) How many leading digits in $v(x)$ are guaranteed to agree with those of $f(x)$, for any $0 \le x \le 381$?

2. Suppose we wish to interpolate $n+1$ data points ($n > 2$) with a piecewise *quadratic* polynomial. How many continuous derivatives at most can this interpolant be guaranteed to have without becoming one global polynomial? Explain.

3. Let $f \in C^3[a,b]$ be given at equidistant points $x_i = a + ih$, $i = 0, 1, \ldots, n$, where $nh = b - a$. Assume further that $f'(a)$ is given as well.

(a) Construct an algorithm for C^1 piecewise quadratic interpolation of the given values. Thus, the interpolating function is written as

$$v(x) = s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2, \quad x_i \le x \le x_{i+1},$$

for $i = 0, \ldots, n-1$, and your job is to specify an algorithm for determining the $3n$ coefficients a_i, b_i and c_i.

(b) How accurate do you expect this approximation to be as a function of h? Justify.

4. Verify that the Hermite cubic interpolating $f(x)$ and its derivative at the points t_i and t_{i+1} can be written explicitly as

$$s_i(x) = f_i + \left(h_i f_i'\right)\tau + \Big(3(f_{i+1} - f_i) - h_i(f_{i+1}' + 2f_i')\Big)\tau^2 + \Big(h_i(f_{i+1}' + f_i') - 2(f_{i+1} - f_i)\Big)\tau^3,$$

where $h_i = t_{i+1} - t_i$, $f_i = f(t_i)$, $f_i' = f'(t_i)$, $f_{i+1} = f(t_{i+1})$, $f_{i+1}' = f'(t_{i+1})$, and $\tau = \frac{x - t_i}{h_i}$.

5. Derive the matrix problem for cubic spline interpolation with the not-a-knot condition. Show that the matrix is tridiagonal and strictly diagonally dominant.

6. The *gamma function* is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt, \quad x > 0.$$

It is known that for integer numbers the function has the value

$$\Gamma(n) = (n-1)! = 1 \cdot 2 \cdot 3 \cdots (n-1).$$

(We define $0! = 1$.) Thus, for example, $(1,1),(2,1),(3,2),(4,6),(5,24)$ can be used as data points for an interpolating polynomial.

(a) Write a MATLAB script that computes the polynomial interpolant of degree four that passes through the above five data points.

(b) Write a program that computes a cubic spline to interpolate the same data. (You may use MATLAB's `spline`.)

(c) Plot the two interpolants you found on the same graph, along with a plot of the gamma function itself, which can be produced using the MATLAB command `gamma`.

(d) Plot the errors in the two interpolants on the same graph. What are your observations?

7. Suppose you are asked to construct a *clamped* cubic spline interpolating the following set of data:

i	0	1	2	3	4	5	6	7
x	1.2	1.4	1.6	1.67	1.8	2.0	2.1	2.2
$f(x)$	4.561	5.217	5.634	5.935	6.562	6.242	5.812	5.367

The function underlying these data points is unknown, but clamped cubic splines require interpolation of the first derivative of the underlying function at the end points $x_0 = 1.2$ and $x_7 = 2.2$. Select the formula in the following list that would best assist you to construct the clamped cubic spline interpolating this set of data:

i. $f'(x_0) = \frac{1}{2h}\Big(-f(x_0-h)+f(x_0+h)\Big) - \frac{h^2}{6} f^{(3)}(\xi)$.

ii. $f'(x_n) = \frac{1}{2h}\Big(f(x_n-2h)-4f(x_n-h)+3f(x_n)\Big) + \frac{h^2}{3} f^{(3)}(\xi)$.

iii. $f'(x_0) = \frac{1}{12h}\Big(f(x_0-2h)-8f(x_0-h)+8f(x_0+h)-f(x_0+2h)\Big) + \frac{h^4}{30} f^{(5)}(\xi)$.

iv. $f'(x_n) = \frac{1}{12h}\Big(3f(x_n-4h)-16f(x_n-3h)+36f(x_n-2h)-48f(x_n-h)$
$+25f(x_n)\Big) + \frac{h^4}{5} f^{(5)}(\xi)$.

v. $f'(x_0) = \frac{1}{180h}\Big(-441f(x_0)+1080f(x_0+h)-1350f(x_0+2h)$
$+1200f(x_0+3h)-675f(x_0+4h)+216f(x_0+5h)$
$-30f(x_0+6h)\Big) + \frac{h^6}{7} f^{(7)}(\xi)$.

Provide an explanation supporting your choice of formula.

8. Consider using cubic splines to interpolate the function

$$f(x) = e^{3x}\sin(200x^2)/(1+20x^2), \quad 0 \le x \le 1,$$

featured in Figure 10.8.

Write a short MATLAB script using `spline`, interpolating this function at equidistant points $x_i = i/n,\ i = 0,1,\ldots,n$. Repeat this for $n = 2^j,\ j = 4,5,\ldots,14$. For each such calculation record the maximum error at the points `x = 0:.001:1`. Plot these errors against n, using `loglog`.

Make observations in comparison to Figure 10.8.

9. Given function values $f(t_0), f(t_1), \ldots, f(t_r)$, as well as those of $f'(t_0)$ and $f'(t_r)$, for some $r \ge 2$, it is possible to construct the complete interpolating cubic spline.

Suppose that we were instead to approximate $f'(t_i)$ by the divided difference $f[t_{i-1}, t_{i+1}]$, for $i = 1,2,\ldots,r-1$, and then use these values to construct a Hermite piecewise cubic interpolant.

State one advantage and one disadvantage of this procedure over a complete cubic spline interpolation.

10. Show that the four sets of interpolation conditions given on page 347 define the four cubic polynomials $\psi_1(z), \psi_2(z), \psi_3(z)$, and $\psi_4(z)$ used to construct the Hermite piecewise cubic basis functions.

11. (a) Verify that the functions $\xi_k(x)$ and $\eta_k(x)$ defined in Section 11.4 are C^1 piecewise cubic with local support interval $[t_{k-1}, t_{k+1}]$.

    (b) What are the mother functions $\xi(z)$ and $\eta(z)$ that give rise to the Hermite basis functions by scalings and translations?

12. Derive a B-spline basis representation for piecewise linear interpolation and for piecewise Hermite cubic interpolation.

13. For the script in Example 11.7 implementing parametric polynomial interpolation, plot $X(\tau)$ vs. τ and $Y(\tau)$ vs. τ.

Is this useful? Discuss.

14. Using parametric cubic Hermite polynomials, draw a crude face in a MATLAB graph. Your picture must include an outline of the head, eyes, ears, a mouth, and a nose. You may use Bézier polynomials to construct the necessary parametric curves. The graph must clearly mark the interpolation and guidepoints used for each curve. Include also a list of the interpolation and guidepoints used for each feature of the face. Finally, include a second plot that shows the curves of your face plot without the guidepoints and interpolation points.

Hints:

- You should not have to use too many curves to draw a crude representation of a face. At the same time, you are hereby given artistic license to do better than the bare minimum.
- Experiment with the parametric curves before you begin to construct your picture.
- You may want to write two generic MATLAB functions that can be used by `plot` to construct all of the parametric curves.
- Use the `help` command in MATLAB to check the features of `plot`, `ginput`, and whatever else you find worthwhile.

15. Consider interpolating the data $(x_0, y_0), \ldots, (x_6, y_6)$ given by

x	0.1	0.15	0.2	0.3	0.35	0.5	0.75
y	3.0	2.0	1.2	2.1	2.0	2.5	2.5

Construct the five interpolants specified below (you may use available software for this), evaluate them at the points $0.05 : 0.01 : 0.8$, plot, and comment on their respective properties:

(a) a polynomial interpolant;

(b) a cubic spline interpolant;

(c) the interpolant

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x) = c_0\phi_0(x) + \cdots + c_n\phi_n(x),$$

where $n = 7$, $\phi_0(x) \equiv 1$, and

$$\phi_j(x) = \sqrt{(x - x_{j-1})^2 + \varepsilon^2} - \varepsilon, \quad j = 1, \ldots, n.$$

In addition to the n interpolation requirements, the condition $c_0 = -\sum_{j=1}^{n} c_j$ is imposed. Construct this interpolant with (i) $\varepsilon = 0.1$, (ii) $\varepsilon = 0.01$, and (iii) $\varepsilon = 0.001$. Make as many observations as you can. What will happen if we let $\varepsilon \to 0$?

11.8 Additional notes

While Chapter 10 lays the foundation for most numerical approximation methods, in the present chapter we reap the rewards: many of the techniques described here see an enormous amount of everyday use by scientists, engineers, and others. Piecewise linear interpolation is the default plotting mode, cubic spline is the default all-purpose data interpolator, and piecewise Hermite cubics are the the default playing units in many CAGD applications.

The complete spline defined in Section 11.3 is a function that approximately minimizes a strain energy functional over all functions (not only piecewise polynomials) which pass through the data. As such, it approximates the trace of a draftsperson's **spline**—a flexible thin beam forced to pass through a set of given pegs. This is the origin of the term "spline" in approximation theory, coined by I. J. Schoenberg in 1946.

B-splines were first shown to be practical by C. de Boor, and we refer to his book [20], where much more theory can be found about polynomial and spline approximation.

Bézier curves and B-splines are used routinely in CAGD and in computer graphics. See, for instance, Chapter 11 in Hill [41]. There is an extension of polynomial bases to rational ones, i.e., bases for *rational interpolation and approximation*. The corresponding extension of B-splines is called NURBs.

In this chapter we are mainly concerned with one-dimensional problems, i.e., curves. But as evident from our brief discussion in Section 11.6 of interpolation in two and three dimensions, the methods and approaches for one dimension may be used as building blocks for the more complex applications. For example, consider methods such as bilinear and bicubic patches, tensor product interpolation, and interpolation on triangular and tetrahedral meshes. This is true for even more dimensions.

The cryptic Example 11.10 is further explained in [45], from which Figure 11.16 has been lifted as well.