CHAPTER

NINE

Curve and Surface Fitting

9.1 Introduction

In Chapters 7 and 8 we showed how to construct NURBS representations of common and relatively simple curves and surfaces such as circles, conics, cylinders, surfaces of revolution, etc. These entities can be specified with only a few data items, e.g., center point, height, radius, axis of revolution, etc. Moreover, the few data items uniquely specify the geometric entity. In this chapter we enter the realm of free-form (or sculptured) curves and surfaces. We study *fitting*, i.e., the construction of NURBS curves and surfaces which fit a rather arbitrary set of geometric data, such as points and derivative vectors. We distinguish two types of fitting, *interpolation* and *approximation*. In interpolation we construct a curve or surface which satisfies the given data precisely, e.g., the curve passes through the given points and assumes the given derivatives at the prescribed points. Figure 9.1 shows a curve interpolating five points and the first derivative vectors at the endpoints. In approximation, we construct curves and surfaces which do not necessarily satisfy the given data precisely, but only approximately. In some applications – such as generation of point data by use of coordinate measuring devices or digitizing tablets, or the computation of surface/surface intersection points by marching methods – a large number of points can be generated, and they can contain measurement or computational noise. In this case it is important for the curve or surface to capture the "shape" of the data, but not to "wiggle" its way through every point. In approximation it is often desirable to specify a maximum bound on the deviation of the curve or surface from the given data, and to specify certain *constraints*, i.e., data which is to be satisfied precisely. Figure 9.2 shows a curve approximating a set of $m + 1$ points. A maximum deviation bound, E, was specified, and the perpendicular distance, e_i, is the approximation error obtained by projecting $\mathbf{Q}_i$ on to the curve. The e_i of each point, $\mathbf{Q}_i$, is less than E. The endpoints $\mathbf{Q}_0$ and $\mathbf{Q}_m$ were specified as constraints, with the result that $e_0 = e_m = 0$.

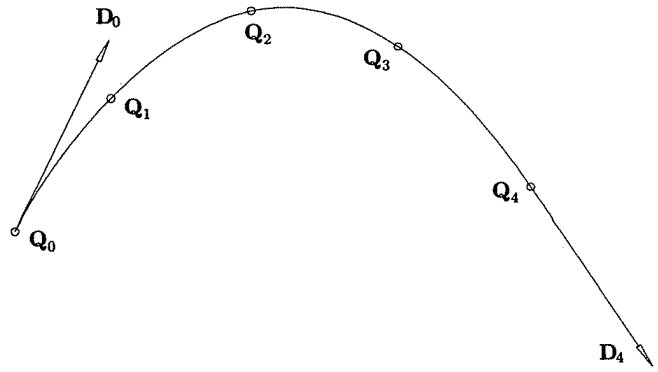


Figure 9.1. A curve interpolating five points and two end derivatives.

There are many subtleties in fitting, and literally hundreds of papers have been written on this topic. Many of the techniques are heuristic, and there are usually no unique or clear-cut "right" answers. A fundamental problem is that the given data never specifies a unique solution. There are infinitely many NURBS curves which can interpolate the seven data items in Figure 9.1, or approximate the $m+1$ items in Figure 9.2. How often has the implementor of a fitting algorithm been told by a designer using the software: "but that's not the curve I wanted!". And the reply is often: "well, it's mathematically correct; it satisfies the data you gave me!".

Input to a fitting problem generally consists of geometric data, such as points and derivatives. Output is a NURBS curve or surface, i.e., control points, knots, and weights. Furthermore, either the degree p (or (p,q) for surfaces) must be input or the algorithm must select an appropriate degree. If C^r continuity is

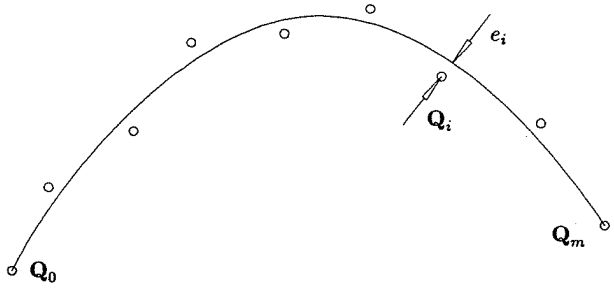


Figure 9.2. A curve approximating $m+1$ points; the curve is constrained to pass through the endpoints, $\mathbf{Q}_0$ and $\mathbf{Q}_m$.

desired for a curve, then the chosen degree p must satisfy

$$p \geq r+1 \tag{9.1}$$

(assuming no interior knots of multiplicity > 1). Assuming no other requirements, choosing $p = r+1$ is generally adequate for interpolation. For approximation, choosing $p > r+1$ may produce better results. For example, suppose we want a C^1 continuous curve approximation to point data representing a twisted curve in three-dimensional space. Using a cubic will likely yield a curve with much fewer control points than a quadratic, although a quadratic gives us the desired continuity. Very little has been published on setting the weights in the fitting process. Most often, all weights are simply set to 1. Indeed, for interpolation there is probably little reason to do otherwise (an exception being when fitting with piecewise circular arcs [Hosc92b]). For approximation, allowing weights to be arbitrary should produce curves and surfaces with fewer control points; we present one such algorithm. Finally, there are many methods for choosing the knots, most of them heuristic.

Most fitting algorithms fall into one of two categories: *global* or *local*. With a global algorithm, a system of equations or an optimization problem is set up and solved. If the given data consists of only points and derivatives, and if the control points are the only unknowns (degree, knots, and weights have been preselected), then the system is linear and hence easy to solve. If more esoteric data, such as curvature, is given, or knots and/or weights are also system unknowns, then the resulting system is nonlinear. Theoretically, a perturbation of any one input data item can change the shape of the entire curve or surface; however, the magnitude of the change decreases with increasing distance from the affected data item. Local algorithms are more geometric in nature, constructing the curve or surface segment-wise, using only local data for each step. A perturbation in a data item only changes the curve or surface locally. These algorithms are usually computationally less expensive than global methods. They can also deal with cusps, straight line segments, and other local data anomalies better. However, achieving desired levels of continuity at segment boundaries is more of a headache, and local methods often result in multiple interior knots.

In subsequent sections we present a number of global and local methods of curve and surface interpolation and approximation. The methods were chosen for their proven utility, and because they represent a broad cross section of techniques. We emphasize that this chapter does not treat interactive curve and surface design, although many of the techniques covered here are applicable to that topic. Interactive design in a NURBS framework generally takes one (or both) of the following approaches:

- given a NURBS curve or surface, interactively change its shape by modifying weights and/or control points; these techniques are treated in Chapter 11;
- interactively construct a NURBS curve or surface. Local fitting techniques are particularly well-suited to this task. Polynomial or rational segments

of any type (e.g., Hermite, Bézier) can be individually constructed and shaped. In addition to local controls, some methods provide global controls, such as a global tension parameter. Whatever the method, if the result is a piecewise polynomial or rational curve or surface, it generally can be converted to NURBS form as the final step.

9.2 Global Interpolation

In this section we present methods to interpolate a curve to point and derivative data, and to interpolate a surface to point data.

9.2.1 GLOBAL CURVE INTERPOLATION TO POINT DATA

Suppose we are given a set of points $\{\mathbf{Q}_k\}$, $k = 0, \ldots, n$, and we want to interpolate these points with a pth-degree nonrational B-spline curve. If we assign a parameter value, $\bar{u}_k$, to each $\mathbf{Q}_k$, and select an appropriate knot vector $U = \{u_0, \ldots, u_m\}$, we can set up the $(n+1) \times (n+1)$ system of linear equations

$$\mathbf{Q}_k = \mathbf{C}(\bar{u}_k) = \sum_{i=0}^{n} N_{i,p}(\bar{u}_k)\mathbf{P}_i \tag{9.2}$$

The control points, $\mathbf{P}_i$, are the $n+1$ unknowns. Let r be the number of coordinates in the $\mathbf{Q}_k$ (typically 2, 3, or 4). Note that this method is independent of r; Eq. (9.2) has one coefficient matrix, with r right hand sides and, correspondingly, r solution sets for the r coordinates of the $\mathbf{P}_i$.

The problem of choosing the $\bar{u}_k$ and U remains, and their choice affects the shape and parameterization of the curve. Throughout this section we assume that the parameter lies in the range $u \in [0, 1]$. Three common methods of choosing the $\bar{u}_k$ are:

- equally spaced:

$$\begin{aligned} \bar{u}_0 &= 0 \qquad \bar{u}_n = 1 \\ \bar{u}_k &= \frac{k}{n} \qquad k = 1, \ldots, n-1 \end{aligned} \tag{9.3}$$

  This method is not recommended, as it can produce erratic shapes (such as loops) when the data is unevenly spaced;

- chord length: Let d be the total chord length

$$d = \sum_{k=1}^{n} |\,\mathbf{Q}_k - \mathbf{Q}_{k-1}\,| \tag{9.4}$$

  Then $\bar{u}_0 = 0 \qquad \bar{u}_n = 1$

$$\bar{u}_k = \bar{u}_{k-1} + \frac{|\,\mathbf{Q}_k - \mathbf{Q}_{k-1}\,|}{d} \qquad k = 1, \ldots, n-1 \tag{9.5}$$

  This is the most widely used method, and it is generally adequate. It also gives a "good" parameterization to the curve, in the sense that it approximates a uniform parameterization.

- centripetal method: Let

$$d = \sum_{k=1}^{n} \sqrt{|\,\mathbf{Q}_k - \mathbf{Q}_{k-1}\,|}$$

  Then $\bar{u}_0 = 0 \qquad \bar{u}_n = 1$

$$\bar{u}_k = \bar{u}_{k-1} + \frac{\sqrt{|\,\mathbf{Q}_k - \mathbf{Q}_{k-1}\,|}}{d} \qquad k = 1, \ldots, n-1 \tag{9.6}$$

  This is a newer method (see [Lee89]) which gives better results than the chord length method when the data takes very sharp turns.

Knots can be equally spaced, that is,

$$\begin{aligned} u_0 = \cdots = u_p &= 0 \qquad u_{m-p} = \cdots = u_m = 1 \\ u_{j+p} &= \frac{j}{n-p+1} \qquad j = 1, \ldots, n-p \end{aligned} \tag{9.7}$$

However, this method is not recommended; if used in conjunction with Eqs. (9.5) or (9.6) it can result in a singular system of equations (Eq. [9.2]). We recommend the following technique of *averaging*

$$\begin{aligned} u_0 = \cdots = u_p &= 0 \qquad u_{m-p} = \cdots = u_m = 1 \\ u_{j+p} &= \frac{1}{p} \sum_{i=j}^{j+p-1} \bar{u}_i \qquad j = 1, \ldots, n-p \end{aligned} \tag{9.8}$$

With this method the knots reflect the distribution of the $\bar{u}_k$. Furthermore, using Eq. (9.8) combined with Eq. (9.5) or (9.6) to compute the $\bar{u}_k$ leads to a system (Eq. [9.2]) which is totally positive and banded with a semibandwidth less than p (see [DeBo78]), that is, $N_{i,p}(\bar{u}_k) = 0$ if $|\, i - k \,| \geq p$. Hence, it can be solved by Gaussian elimination without pivoting.

Figure 9.3 shows control points, parameters, and the knot vector of a cubic curve interpolating seven points. Parameters were chosen by the chord length method, and the knots were obtained by averaging the parameters (Eq. [9.8]). In Figure 9.4 a comparison of different parameterizations is illustrated. Figure 9.5 shows the same comparison using more "wildly" scattered data points. In both cases a cubic curve is passed through seven points, using uniform parameters and uniform knots (solid curve and top knot vector – see Eqs. [9.3] and [9.7]); chord length parameters and knots obtained by averaging (dashed curve and middle

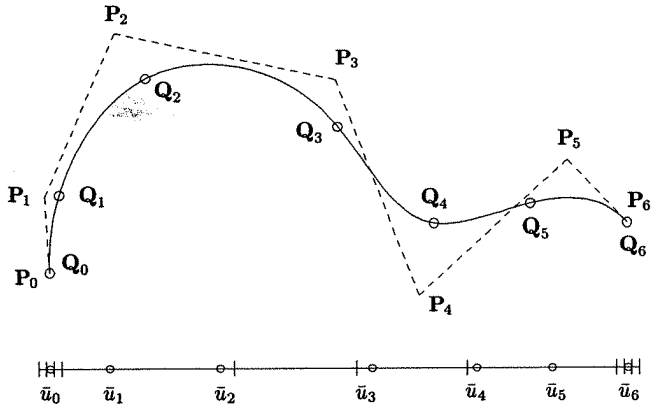


Figure 9.3. A curve interpolation example using chord length parameterization and a knot vector obtained by averaging parameters.

knot vector – see Eqs. [9.5] and [9.8]); and centripetal parameters and knots obtained by averaging (dotted curve and bottom knot vector – see Eqs. [9.6] and [9.8]). In Figure 9.5 notice how the chord length and centripetal parameterized curves adapt to the changes in point spacing.

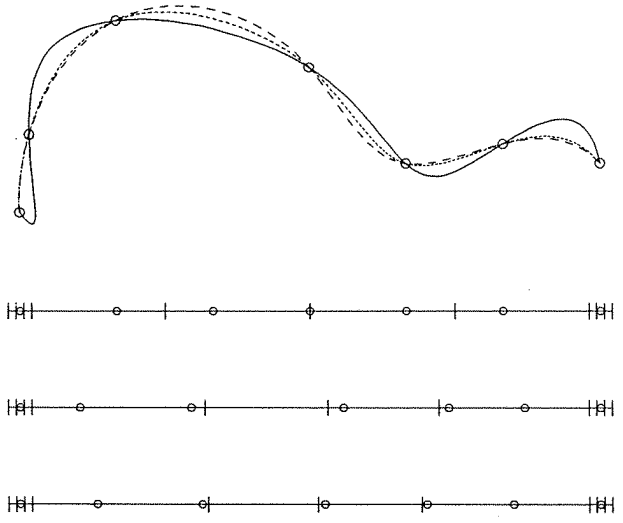


Figure 9.4. A curve interpolation example with different parameterizations and knot vectors.

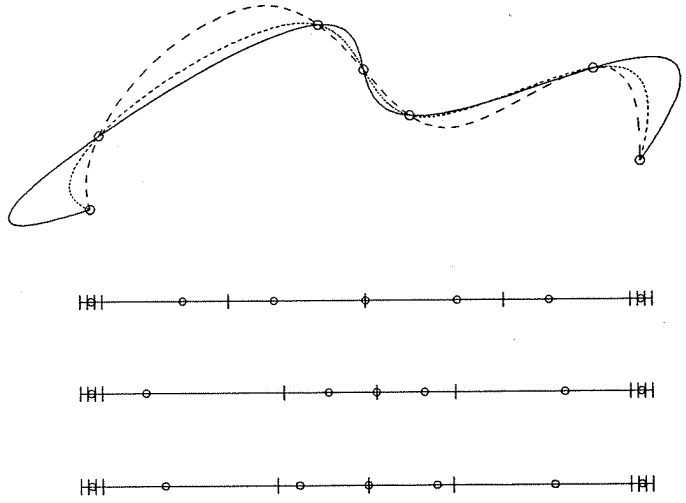


Figure 9.5. A curve interpolation example with different parameterizations and knot vectors.

Figure 9.6 illustrates interpolation with different degrees; solid, dashed, and dotted curves have degrees 2, 3, and 4, respectively. Figure 9.7 shows the inability of the global interpolant to handle collinear point sets.

Once the $\bar{u}_k$ and the knots are computed, the $(n+1)\times(n+1)$ coefficient matrix of the system (Eq. [9.2]) is set up by evaluating the nonzero basis functions at each $\bar{u}_k$, $k = 0, \ldots, n$; use Algorithms `A2.1` and `A2.2`.

Example

`Ex9.1` Let $\{\mathbf{Q}_k\} = \{(0,0),(3,4),(-1,4),(-4,0),(-4,-3)\}$, and assume that we want to interpolate the $\mathbf{Q}_k$ with a cubic curve. We use Eqs. (9.5) and (9.8) to compute the $\bar{u}_k$ and u_j, and then set up the system of linear equations, Eq. (9.2). The separate chord lengths are

$$|\mathbf{Q}_1 - \mathbf{Q}_0| = 5 \quad |\mathbf{Q}_2 - \mathbf{Q}_1| = 4 \quad |\mathbf{Q}_3 - \mathbf{Q}_2| = 5 \quad |\mathbf{Q}_4 - \mathbf{Q}_3| = 3$$

and the total chord length is $d = 17$. Thus

$$\bar{u}_0 = 0 \quad \bar{u}_1 = \frac{5}{17} \quad \bar{u}_2 = \frac{9}{17} \quad \bar{u}_3 = \frac{14}{17} \quad \bar{u}_4 = 1$$

Using Eq. (9.8)

$$u_4 = \frac{1}{3}\left(\frac{5}{17} + \frac{9}{17} + \frac{14}{17}\right) = \frac{28}{51}$$

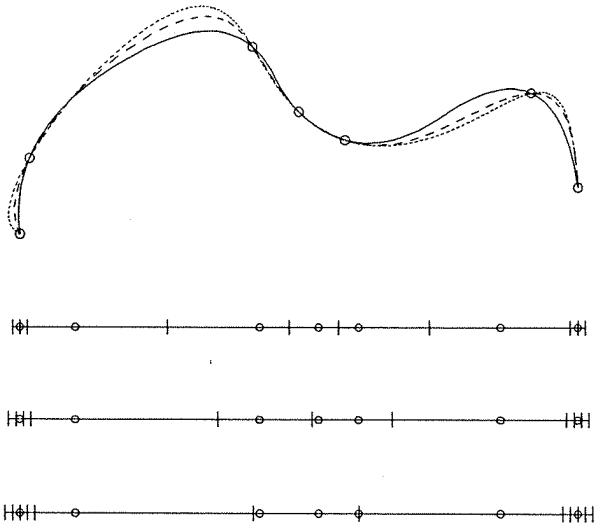


Figure 9.6. A curve interpolation with different degrees using chord length parameterization and knots obtained by averaging.

hence $$U = \left\{0,0,0,0,\frac{28}{51},1,1,1,1\right\}$$

The system of linear equations is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ N_{0,3}\left(\frac{5}{17}\right) & N_{1,3}\left(\frac{5}{17}\right) & N_{2,3}\left(\frac{5}{17}\right) & N_{3,3}\left(\frac{5}{17}\right) & 0 \\ N_{0,3}\left(\frac{9}{17}\right) & N_{1,3}\left(\frac{9}{17}\right) & N_{2,3}\left(\frac{9}{17}\right) & N_{3,3}\left(\frac{9}{17}\right) & 0 \\ 0 & N_{1,3}\left(\frac{14}{17}\right) & N_{2,3}\left(\frac{14}{17}\right) & N_{3,3}\left(\frac{14}{17}\right) & N_{4,3}\left(\frac{14}{17}\right) \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \\ \mathbf{P}_4 \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_0 \\ \mathbf{Q}_1 \\ \mathbf{Q}_2 \\ \mathbf{Q}_3 \\ \mathbf{Q}_4 \end{bmatrix}$$

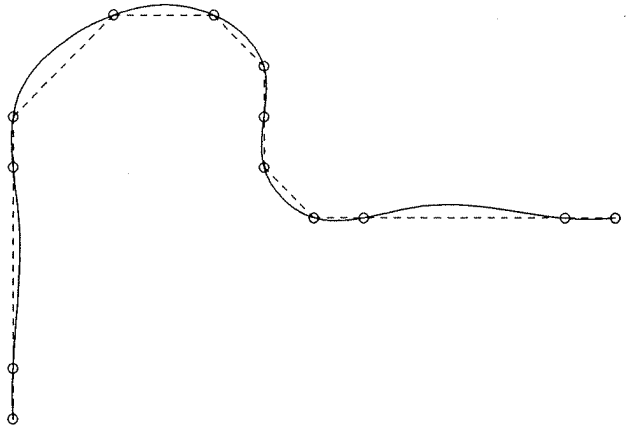


Figure 9.7. A global cubic curve interpolant to data containing collinear points (see the dashed line).

We now summarize the global interpolation algorithm. We assume two utility routines to solve the system of linear equations:

- `LUDecomposition(A,q,sbw)` to decompose the $q \times q$ coefficient matrix with semibandwidth `sbw` into lower and upper triangular components; for simplicity we assume `A` is an $q \times q$ square array, but a utility should be used which only stores the nonzero band;
- `ForwardBackward(A,q,sbw,rhs,sol)` to perform the forward/backward substitution (see [Pres88]); `rhs[]` is the right hand side of the system (the coordinates of the $\mathbf{Q}_k$), and `sol[]` is the solution vector (coordinates of the $\mathbf{P}_i$).

The algorithm also makes use of `FindSpan()` and `BasisFuns()` from Chapter 2.

```
ALGORITHM A9.1
  GlobalCurveInterp(n,Q,r,p,m,U,P)
    { /*  Global interpolation through n+1 points  */
      /*  Input:  n,Q,r,p  */
      /*  Output: m,U,P  */
    m = n+p+1;
    Compute the uk;     /* Eq.(9.5) or (9.6) */
    Compute the knot vector U;   /* Eq.(9.8) */
    Initialize array A to zero;
    for (i=0; i<=n; i++)
      {  /* Set up coefficient matrix */
      span = FindSpan(n,p,uk[i],U);
      BasisFuns(span,uk[i],p,U,A[i][span-p]); /* Get ith row */
      }
```

```
LUDecomposition(A,n+1,p-1);
for (i=0; i<r; i++)   /* r is the number of coordinates */
  {
  for (j=0; j<=n; j++)   rhs[j] = ith coordinate of Q[j];
  ForwardBackward(A,n+1,p-1,rhs,sol);
  for (j=0; j<=n; j++)   ith coordinate of P[j] = sol[j];
  }
}
```

9.2.2 GLOBAL CURVE INTERPOLATION WITH END DERIVATIVES SPECIFIED

It is not uncommon for derivative vectors to be specified as input data. In this case, the solution process is similar to Algorithm A9.1:

1. compute parameters $\bar{u}_k$ corresponding to the points $\mathbf{Q}_k$;
2. compute a knot vector U;
3. evaluate the basis functions to set up a system of linear equations, with the control points as unknowns;
4. solve the system.

Only Steps 2 and 3 are slightly different from the previous case. Each derivative gives rise to one additional knot and control point, and hence to one additional linear equation. The derivative formulas of Chapters 2 and 3 are used to set up the additional equations, e.g., Eqs. (2.7),(2.10), and (3.3)–(3.10). We now illustrate this with examples using first derivatives.

Again let $\{\mathbf{Q}_k\}$, $k = 0, \ldots, n$, be points, and assume that $\mathbf{D}_0$ and $\mathbf{D}_n$ are the first derivative vectors at the start point and end point of the curve, respectively. We want to interpolate this data with a pth-degree curve

$$\mathbf{C}(u) = \sum_{i=0}^{n+2} N_{i,p}(u)\mathbf{P}_i$$

As before, compute the $\bar{u}_k$, $k = 0, \ldots, n$, using Eq. (9.5) or Eq. (9.6). Then set $m = n + p + 3$ and obtain the $m + 1$ knots by

$$u_0 = \cdots = u_p = 0 \qquad u_{m-p} = \cdots = u_m = 1$$
$$u_{j+p+1} = \frac{1}{p}\sum_{i=j}^{j+p-1} \bar{u}_i \qquad j = 0, \ldots, n-p+1 \tag{9.9}$$

Notice that Eq. (9.9) is analogous to Eq. (9.8) except that we pick up two additional knots by running j from 0 to $n - p + 1$. As before, Eq. (9.2) yields $n + 1$ equations

$$\mathbf{Q}_k = \mathbf{C}(\bar{u}_k) = \sum_{i=0}^{n+2} N_{i,p}(\bar{u}_k)\mathbf{P}_i \tag{9.10}$$

Equations (3.7) produce two additional equations, i.e.

$$-\mathbf{P}_0 + \mathbf{P}_1 = \frac{u_{p+1}}{p}\mathbf{D}_0 \tag{9.11}$$

$$-\mathbf{P}_{n+1} + \mathbf{P}_{n+2} = \frac{1 - u_{m-p-1}}{p}\mathbf{D}_n \tag{9.12}$$

Inserting Eqs. (9.11) and (9.12) into Eq. (9.10) as the second and next to last equations, respectively, yields an $(n + 3) \times (n + 3)$ banded linear system. Figure 9.8 shows cubic curve interpolants with end tangents specified. The small, medium, and large magnitudes of the derivatives produce the solid, the dashed, and the dotted curves, respectively. Figure 9.9 illustrates interpolants to the same data points, but with different end tangent directions and magnitudes for the derivatives.

9.2.3 CUBIC SPLINE CURVE INTERPOLATION

The previous method (Eqs. [9.9]–[9.12]) is valid for any degree $p > 1$. If $p = 3$, there is an algorithm which is more efficient and yields the traditional C^2 cubic spline. The $\bar{u}_k$ are computed as previously. The knots are

$$u_0 = \cdots = u_3 = 0 \qquad u_{n+3} = \cdots = u_{n+6} = 1$$
$$u_{j+3} = \bar{u}_j \qquad j = 1, \ldots, n-1 \tag{9.13}$$

In other words, the interpolation of the $\mathbf{Q}_k$ occurs at the knots. The first two and last two equations are, respectively

$$\mathbf{P}_0 = \mathbf{Q}_0$$
$$-\mathbf{P}_0 + \mathbf{P}_1 = \frac{u_4}{3}\mathbf{D}_0 \tag{9.14}$$

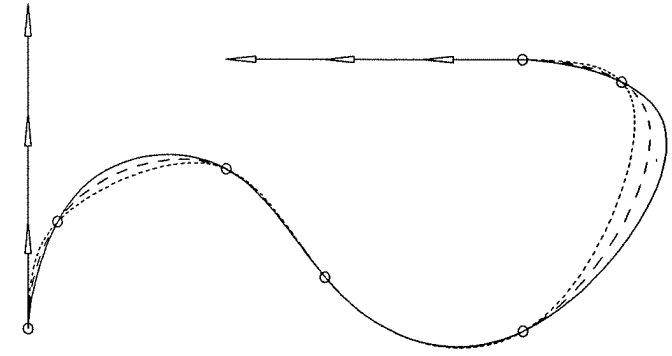


Figure 9.8. Global cubic curve interpolants with varying end derivative magnitudes.

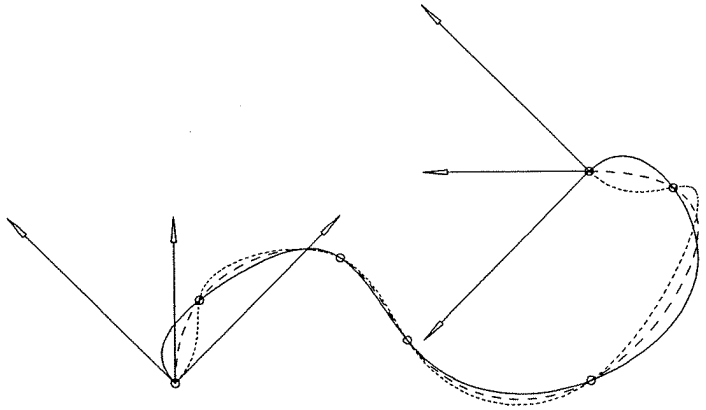


Figure 9.9. A global cubic curve interpolation with different end derivative directions.

and

$$-\mathbf{P}_{n+1} + \mathbf{P}_{n+2} = \frac{1-u_{n+2}}{3}\mathbf{D}_n$$
$$\mathbf{P}_{n+2} = \mathbf{Q}_n \tag{9.15}$$

These can be solved directly. Recalling that there are only three nonzero cubic basis functions at an interior knot, the remaining $n-1$ equations have the form

$$\mathbf{Q}_k = \mathbf{C}(\bar{u}_k) = N_{k,3}(\bar{u}_k)\mathbf{P}_k + N_{k+1,3}(\bar{u}_k)\mathbf{P}_{k+1} + N_{k+2,3}(\bar{u}_k)\mathbf{P}_{k+2} \tag{9.16}$$

for $k = 1, \ldots, n-1$. Setting

$$a_k = N_{k,3}(\bar{u}_k) \quad b_k = N_{k+1,3}(\bar{u}_k) \quad c_k = N_{k+2,3}(\bar{u}_k)$$

yields the tridiagonal system

$$\begin{bmatrix} \mathbf{R}_2 \\ \mathbf{R}_3 \\ \vdots \\ \mathbf{R}_{n-1} \\ \mathbf{R}_n \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_1 - a_1\mathbf{P}_1 \\ \mathbf{Q}_2 \\ \vdots \\ \mathbf{Q}_{n-2} \\ \mathbf{Q}_{n-1} - c_{n-1}\mathbf{P}_{n+1} \end{bmatrix}$$

$$= \begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-2} & b_{n-2} & c_{n-2} \\ 0 & 0 & 0 & \cdots & 0 & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} \mathbf{P}_2 \\ \mathbf{P}_3 \\ \vdots \\ \mathbf{P}_{n-1} \\ \mathbf{P}_n \end{bmatrix} \tag{9.17}$$

This system can be solved by a simple algorithm which computes the rows of the matrix in Eq. (9.17) on-the-fly. It requires two local arrays, `R[]` and `dd[]`, of length $n+1$, and an array `abc[]` of length 4, which stores $a_k, b_k, c_k, 0$. It assumes that $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_{n+1}, \mathbf{P}_{n+2}$ are already computed and loaded into array `P`. The algorithm can be adapted to handle points with any number of coordinates.

```
ALGORITHM A9.2
  SolveTridiagonal(n,Q,U,P)
    { /*  Solve tridiagonal system for C2 cubic spline  */
      /*  Input:  n,Q,U,P[0],P[1],P[n+1],P[n+2]  */
      /*  Output: P  */
    for (i=3; i<n; i++)   R[i] = Q[i-1];
    BasisFuns(4,U[4],3,U,abc);
    den = abc[1];
    P[2] = (Q[1]-abc[0]*P[1])/den;
    for (i=3; i<n; i++)
      {
      dd[i] = abc[2]/den;
      BasisFuns(i+2,U[i+2],3,U,abc);
      den = abc[1]-abc[0]*dd[i];
      P[i] = (R[i]-abc[0]*P[i-1])/den;
      }
    dd[n] = abc[2]/den;
    BasisFuns(n+2,U[n+2],3,U,abc);
    den = abc[1]-abc[0]*dd[n];
    P[n] = (Q[n-1]-abc[2]*P[n+1]-abc[0]*P[n-1])/den;
    for (i=n-1; i>=2; i--)   P[i] = P[i]-dd[i+1]*P[i+1];
    }
```

Figure 9.10 shows data interpolated with this cubic spline algorithm.

Often, only tangent directions but not magnitudes for derivative vectors are specified. We denote unit length tangent vectors by $\mathbf{T}_0$ and $\mathbf{T}_n$. In this case, magnitudes α_0 and α_n must be estimated. Set

$$\mathbf{D}_0 = \alpha_0\mathbf{T}_0 \qquad \mathbf{D}_n = \alpha_n\mathbf{T}_n \tag{9.18}$$

and proceed as before. A reasonable choice is to set α_0 and α_n equal to the total chord length, d, as computed in Eq. (9.4). In an interactive environment, α_0 and α_n can be used as additional shape controls.

9.2.4 GLOBAL CURVE INTERPOLATION WITH FIRST DERIVATIVES SPECIFIED

Suppose now that the first derivative, $\mathbf{D}_k$, is given at every point, $\mathbf{Q}_k$, $k = 0, \ldots, n$. There are $2(n+1)$ data items and that many unknown control points.

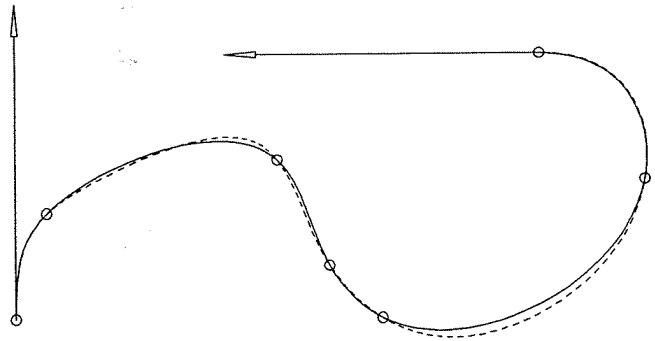


Figure 9.10. A cubic spline interpolant (solid line) compared to a regular cubic interpolant (dashed line) obtained by using chord length parameterization and knots computed by averaging.

The $n+1$ equations expressing the $\mathbf{Q}_k$ are

$$\mathbf{Q}_k = \mathbf{C}(\bar{u}_k) = \sum_{i=0}^{2n+1} N_{i,p}(\bar{u}_k)\mathbf{P}_i \tag{9.19}$$

and from Eqs. (2.10) and (3.3), the $n+1$ equations expressing the $\mathbf{D}_k$ are

$$\mathbf{D}_k = \mathbf{C}'(\bar{u}_k) = \sum_{i=0}^{2n+1} N'_{i,p}(\bar{u}_k)\mathbf{P}_i \tag{9.20}$$

The parameters $\bar{u}_k$ are computed using Eq. (9.5) or Eq. (9.6). The number of knots is

$$2(n+1)+p+1 \tag{9.21}$$

and they should be chosen to reflect the distribution of the $\bar{u}_k$. For $p=2$ and $p=3$, satisfactory choices are

$$U = \left\{0,0,0,\frac{\bar{u}_1}{2},\bar{u}_1,\frac{\bar{u}_1+\bar{u}_2}{2},\bar{u}_2,\ldots,\bar{u}_{n-1},\frac{\bar{u}_{n-1}+1}{2},1,1,1\right\} \tag{9.22}$$

and

$$U = \left\{0,0,0,0,\frac{\bar{u}_1}{2},\frac{2\bar{u}_1+\bar{u}_2}{3},\frac{\bar{u}_1+2\bar{u}_2}{3},\ldots,\frac{\bar{u}_{n-2}+2\bar{u}_{n-1}}{3},\frac{\bar{u}_{n-1}+1}{2},1,1,1,1\right\} \tag{9.23}$$

respectively. Equations (9.19) and (9.20) are merged in an alternating fashion to yield a $2(n+1)\times 2(n+1)$ banded linear system. For example, let $p=3$. Denote the first and last interior knots of Eq. (9.23) by

$$u_4 = \frac{\bar{u}_1}{2} \qquad u_{m-p-1} = \frac{\bar{u}_{n-1}+1}{2}$$

The system of equations is

$$\begin{aligned}
\mathbf{P}_0 &= \mathbf{Q}_0\\
-\mathbf{P}_0+\mathbf{P}_1 &= \frac{u_4}{3}\mathbf{D}_0\\
N_{1,3}(\bar{u}_1)\mathbf{P}_1+\cdots+N_{4,3}(\bar{u}_1)\mathbf{P}_4 &= \mathbf{Q}_1\\
N'_{1,3}(\bar{u}_1)\mathbf{P}_1+\cdots+N'_{4,3}(\bar{u}_1)\mathbf{P}_4 &= \mathbf{D}_1\\
&\vdots\\
N_{2n-3,3}(\bar{u}_{n-1})\mathbf{P}_{2n-3}+\cdots+N_{2n,3}(\bar{u}_{n-1})\mathbf{P}_{2n} &= \mathbf{Q}_{n-1}\\
N'_{2n-3,3}(\bar{u}_{n-1})\mathbf{P}_{2n-3}+\cdots+N'_{2n,3}(\bar{u}_{n-1})\mathbf{P}_{2n} &= \mathbf{D}_{n-1}\\
-\mathbf{P}_{2n}+\mathbf{P}_{2n+1} &= \frac{1-u_{m-p-1}}{3}\mathbf{D}_n\\
\mathbf{P}_{2n+1} &= \mathbf{Q}_n
\end{aligned} \tag{9.24}$$

Use Algorithm `A2.3` (`DersBasisFuns()`) to compute the $N_{i,3}(\bar{u}_k)$ and $N'_{i,3}(\bar{u}_k)$.

If unit length tangent vectors $\mathbf{T}_k$ are given instead of the $\mathbf{D}_k$, then magnitudes α_k must be estimated. Setting all $\alpha_k = d$, where d is the total chord length, is a reasonable choice. Figure 9.11 shows quadratic (solid line) and cubic (dashed line) curve interpolants to points and derivatives. Figure 9.12 illustrates a comparison among different curve interpolants obtained by using various derivative magnitudes at the data points. The solid curve applies $\alpha_k = d$ (middle tangent);

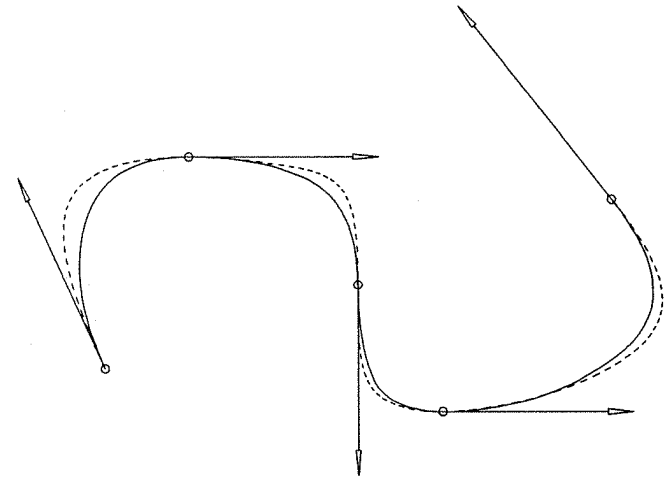


Figure 9.11. Quadratic (solid line) and cubic (dashed line) curve interpolants with derivatives specified at each data point.

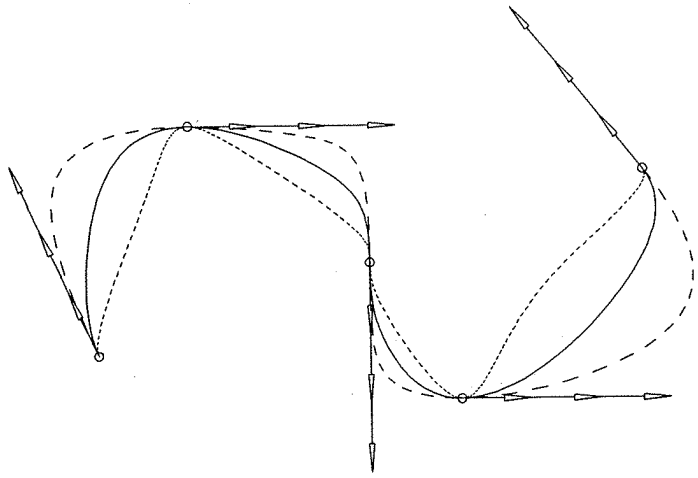


Figure 9.12. Cubic curve interpolants with varying tangent derivatives at each point.

the dashed curve has $\alpha_k = {}^3/_2 d$ (large tangent); and the dotted curve was obtained by $\alpha_k = {}^1/_2 d$ (small tangent). Figure 9.13 shows the control points of a quadratic curve interpolant to specified points and first derivatives.

9.2.5 GLOBAL SURFACE INTERPOLATION

We now turn to global interpolation of surfaces. We are given a set of $(n+1) \times (m+1)$ data points $\{\mathbf{Q}_{k,\ell}\}$, $k = 0, \ldots, n$ and $\ell = 0, \ldots, m$, and we want to construct a nonrational (p, q)th-degree B-spline surface interpolating these points, i.e.

$$\mathbf{Q}_{k,\ell} = \mathbf{S}(\bar{u}_k, \bar{v}_\ell) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(\bar{u}_k) N_{j,q}(\bar{v}_\ell) \mathbf{P}_{i,j} \tag{9.25}$$

Again, the first order of business is to compute reasonable values for the $(\bar{u}_k, \bar{v}_\ell)$ and the knot vectors U and V. We show how to compute the $\bar{u}_k$; the $\bar{v}_\ell$ are analogous. A common method is to use Eq. (9.5) or Eq. (9.6) to compute parameters $\bar{u}_0^\ell, \ldots, \bar{u}_n^\ell$ for each ℓ, and then to obtain each $\bar{u}_k$ by averaging across all $\bar{u}_k^\ell$, $\ell = 0, \ldots, m$, that is

$$\bar{u}_k = \frac{1}{m+1} \sum_{\ell=0}^{m} \bar{u}_k^\ell \qquad k = 0, \ldots, n$$

where for each fixed ℓ, $\bar{u}_k^\ell$, $k = 0, \ldots, n$, was computed by Eq. (9.5) or Eq. (9.6). An efficient algorithm using Eq. (9.5) follows. It computes both the $\bar{u}_k$ and $\bar{v}_\ell$. It requires one local array,`cds[]`, of length $\max(n+1, m+1)$, to store chordal

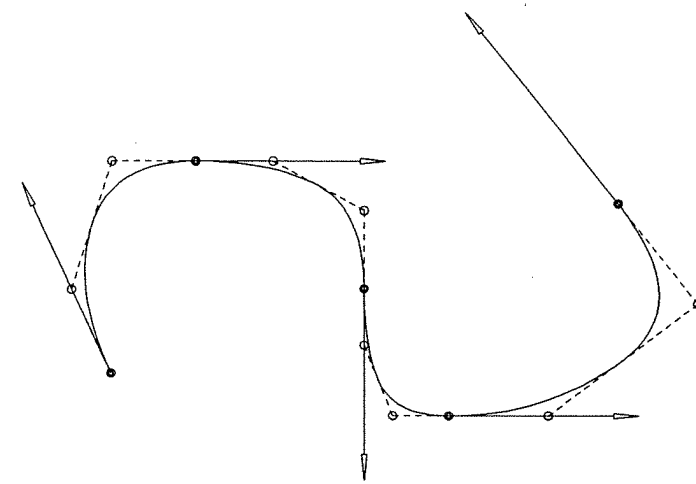


Figure 9.13. Control points of a quadratic B-spline curve interpolant to data with specified derivatives.

distances $|\mathbf{Q}_{k,\ell} - \mathbf{Q}_{k-1,\ell}|$, $k = 1, \ldots, n$ (and similarly for the v direction). The function `Distance3D()` computes the distance between two three-dimensional points. This algorithm handles the case when the total chord length of a row is equal to zero, a necessary condition if a three-sided surface is being constructed.

```
ALGORITHM A9.3
  SurfMeshParams(n,m,Q,uk,vl)
    { /*  Compute parameters for  */
      /*        global surface interpolation  */
      /*  Input:  n,m,Q  */
      /*  Output: uk,vl  */
     /* First get the uk */
    num = m+1;    /* number of nondegenerate rows */
    uk[0] = 0.0;     uk[n] = 1.0;
    for (k=1; k<n; k++)   uk[k] = 0.0;
    for (l=0; l<=m; l++)
      {
      total = 0.0;    /* total chord length of row */
      for (k=1; k<=n; k++)
        {
        cds[k] = Distance3D(Q[k][l],Q[k-1][l]);
        total = total + cds[k];
        }
      if (total == 0.0)    num = num-1;
```

```
        else
          {
          d = 0.0;
          for (k=1; k<n; k++)
            {
            d = d + cds[k];
            uk[k] = uk[k] + d/total;
            }
          }
      }
    if (num == 0)   return(error);
    for (k=1; k<n; k++)   uk[k] = uk[k]/num;
       /* Now do the same for vl */
    }
```

Once the $(\bar{u}_k, \bar{v}_\ell)$ are computed, the knot vectors U and V can be obtained by Eq. (9.8).

Now to the computation of the control points. Clearly, Eq. (9.25) represents $(n+1) \times (m+1)$ linear equations in the unknown $\mathbf{P}_{i,j}$. However, since $\mathbf{S}(u,v)$ is a tensor product surface, the $\mathbf{P}_{i,j}$ can be obtained more simply and efficiently as a sequence of curve interpolations. For fixed ℓ write Eq. (9.25) as

$$\mathbf{Q}_{k,\ell} = \sum_{i=0}^{n} N_{i,p}(\bar{u}_k) \left(\sum_{j=0}^{m} N_{j,q}(\bar{v}_\ell) \mathbf{P}_{i,j} \right) = \sum_{i=0}^{n} N_{i,p}(\bar{u}_k) \mathbf{R}_{i,\ell} \qquad (9.26)$$

where

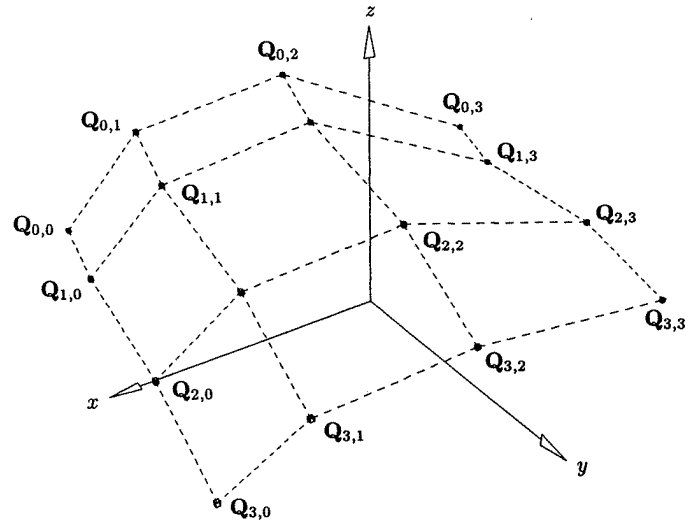
$$\mathbf{R}_{i,\ell} = \sum_{j=0}^{m} N_{j,q}(\bar{v}_\ell) \mathbf{P}_{i,j} \qquad (9.27)$$

Notice that Eq. (9.26) is just curve interpolation through the points $\mathbf{Q}_{k,\ell}$, $k = 0, \ldots, n$. The $\mathbf{R}_{i,\ell}$ are the control points of the isoparametric curve on $\mathbf{S}(u,v)$ at fixed $v = \bar{v}_\ell$. Now fixing i and letting ℓ vary, Eq. (9.27) is curve interpolation through the points $\mathbf{R}_{i,0}, \ldots, \mathbf{R}_{i,m}$, with $\mathbf{P}_{i,0}, \ldots, \mathbf{P}_{i,m}$ as the computed control points. Thus, the algorithm to obtain all the $\mathbf{P}_{i,j}$ is (see Figure 9.14a–9.14d):

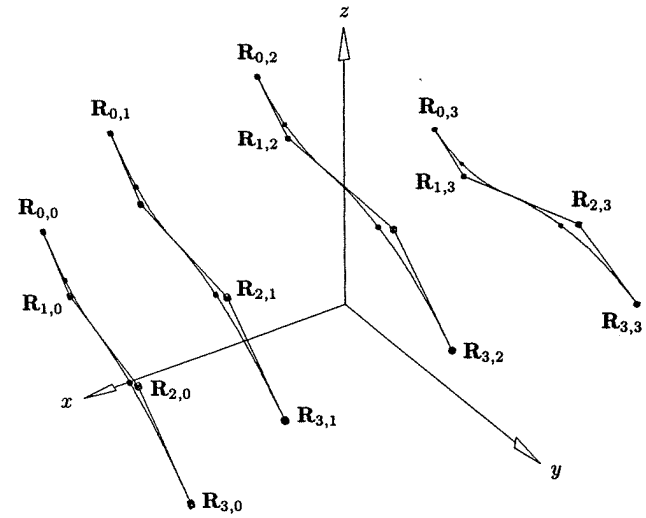
1. using U and the $\bar{u}_k$, do $m+1$ curve interpolations through $\mathbf{Q}_{0,\ell}, \ldots, \mathbf{Q}_{n,\ell}$ (for $\ell = 0, \ldots, m$); this yields the $\mathbf{R}_{i,\ell}$ (Figure 9.14b);
2. using V and the $\bar{v}_\ell$, do $n+1$ curve interpolations through $\mathbf{R}_{i,0}, \ldots, \mathbf{R}_{i,m}$ (for $i = 0, \ldots, n$); this yields the $\mathbf{P}_{i,j}$ (Figure 9.14c).

Figure 9.14d portrays the resulting surface. Clearly, the algorithm is symmetric; the same surface is obtained by:

1. doing $n+1$ curve interpolations through the $\mathbf{Q}_{k,0}, \ldots, \mathbf{Q}_{k,m}$ to obtain the $\mathbf{R}_{k,j}$ (control points of isoparametric curves $\mathbf{S}(\bar{u}_k, v)$);
2. then doing $m+1$ curve interpolations through the $\mathbf{R}_{0,j}, \ldots, \mathbf{R}_{n,j}$ to obtain the $\mathbf{P}_{i,j}$.

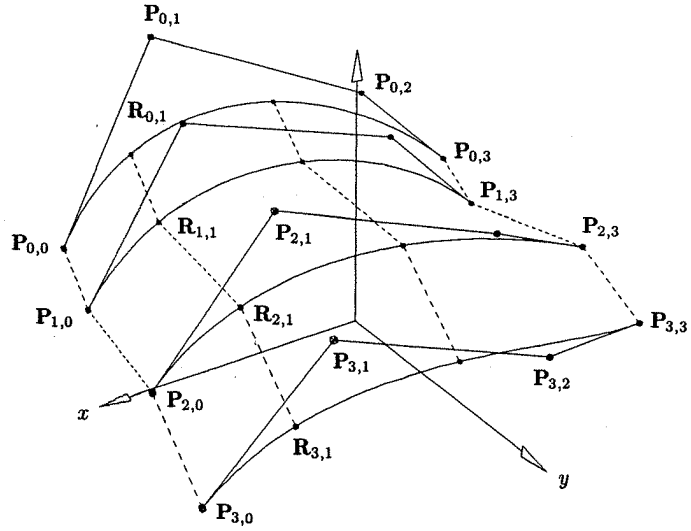


(a)



(b)

Figure 9.14. Global surface interpolation. (a) The data points; (b) interpolating the u-directional data points; (c) interpolating in the v direction through control points of u-directional interpolants; (d) the surface interpolant showing control points.



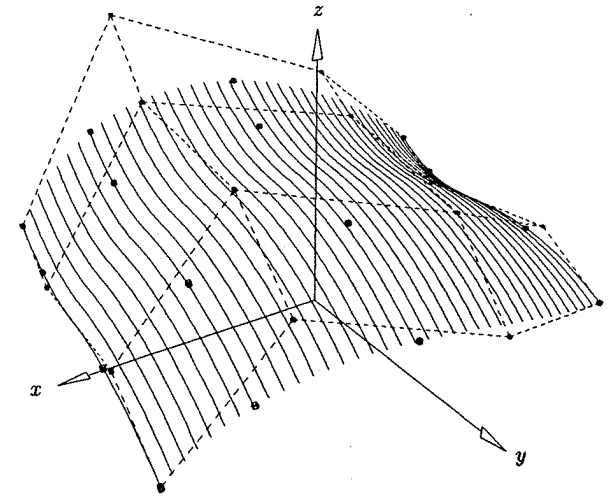
(c)

Figure 9.14. (*Continued.*)

Summarizing, surface interpolation proceeds as follows:

```
ALGORITHM A9.4
  GlobalSurfInterp(n,m,Q,p,q,U,V,P)
    { /*  Global surface interpolation  */
      /*  Input:  n,m,Q,p,q  */
      /*  Output: U,V,P  */
    SurfMeshParams(n,m,Q,uk,vl);  /* get parameters */
    Compute U using Eq.(9.8);
    Compute V using Eq.(9.8);
    for (l=0; l<=m; l++)
      {
      Do curve interpolation through Q[0][l],...,Q[n][l];
      This yields R[0][l],...,R[n][l];
      }
    for (i=0; i<=n; i++)
      {
      Do curve interpolation through R[i][0],...,R[i][m];
      This yields P[i][0],...,P[i][m];
      }
    }
```

Several surface interpolation examples are shown in Figures 9.15–9.18. The data set is shown in Figure 9.15. Surfaces with different parameterizations and



(d)

Figure 9.14. (*Continued.*)

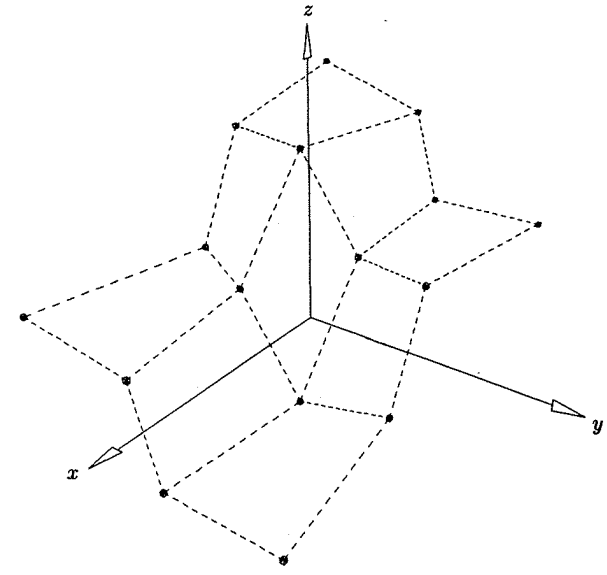


Figure 9.15. Data points to test surface interpolation.

knot vectors are illustrated in Figures 9.16a–9.16c. The degree is $(2,3)$, and the parameterizations and knot vectors are uniform (Figure 9.16a); chord length, knots obtained by averaging (Figure 9.16b); and centripetal, knots obtained by averaging (Figure 9.16c). In Figure 9.17 surface interpolants with different degrees are depicted. The degrees in each are: Figure 9.17a, degree $(2,3)$; Figure 9.17b, degree (3,2); and Figure 9.17c, degree $(3,3)$. In Figure 9.18 the inability to handle coplanar data is demonstrated.

Derivative constraints can also be incorporated into global surface interpolation. Conceptually, the derivative formulas of Chapters 2 and 3 can be used to define one additional linear equation for each derivative constraint specified. Unfortunately, if the number of data constraints is not the same in every row or column it becomes more difficult to solve for the unknown surface control points using curve interpolations in two directions, as described previously. However, with clever use of curve knot insertion and surface knot removal, one can adapt Algorithm **A9.4** to handle partial derivative constraints at individual data points. Finally, local surface interpolation methods (see Section 9.3) are well-suited to handling derivative constraints, as they require derivatives at every point anyway.

9.3 Local Interpolation

In this section we present local interpolation of curves and surfaces to point and tangent data. We start with some preliminary concepts.

9.3.1 LOCAL CURVE INTERPOLATION PRELIMINARIES

Let $\{\mathbf{Q}_k\}$, $k = 0,\ldots,n$, be given. By local curve interpolation we mean a method which constructs n polynomial or rational curve segments, $\mathbf{C}_i(u)$, $i = 0,\ldots,n-1$, such that $\mathbf{Q}_i$ and $\mathbf{Q}_{i+1}$ are the endpoints of $\mathbf{C}_i(u)$. Neighboring segments are joined with some prescribed level of continuity, and the construction proceeds segment-wise, generally from left to right. Any equations which arise are local to only a few neighboring segments. In the framework of NURBS, we construct the segments using polynomial or rational Bézier curves, then obtain a NURBS curve by selecting a suitable knot vector.

Now let $\bar{u}_i$ denote the start parameter of $\mathbf{C}_i(u)$ and the end parameter of $\mathbf{C}_{i-1}(u)$. $\mathbf{C}_i(u)$ and $\mathbf{C}_{i-1}(u)$ meet at $\bar{u}_i$ with G^1 *continuity* (G for *geometric*) if their tangent directions coincide there, that is, if $\mathbf{C}_i'(\bar{u}_i)$ and $\mathbf{C}_{i-1}'(\bar{u}_i)$ are pointing in the same direction. However, their magnitudes may be different. G^1 continuity implies that the curve is visually continuous (smooth) but may have a discontinuity in the parameterization. For a deeper study of G^n continuity, $n \geq 1$, we refer the reader to Barsky and DeRose [Bars89, 90]. Local fitting algorithms are designed to provide some level of either G or C continuity; in this chapter we present only algorithms providing G^1 or C^1 continuity. Local methods yielding higher continuity are not widely used. Although G^2 and C^2

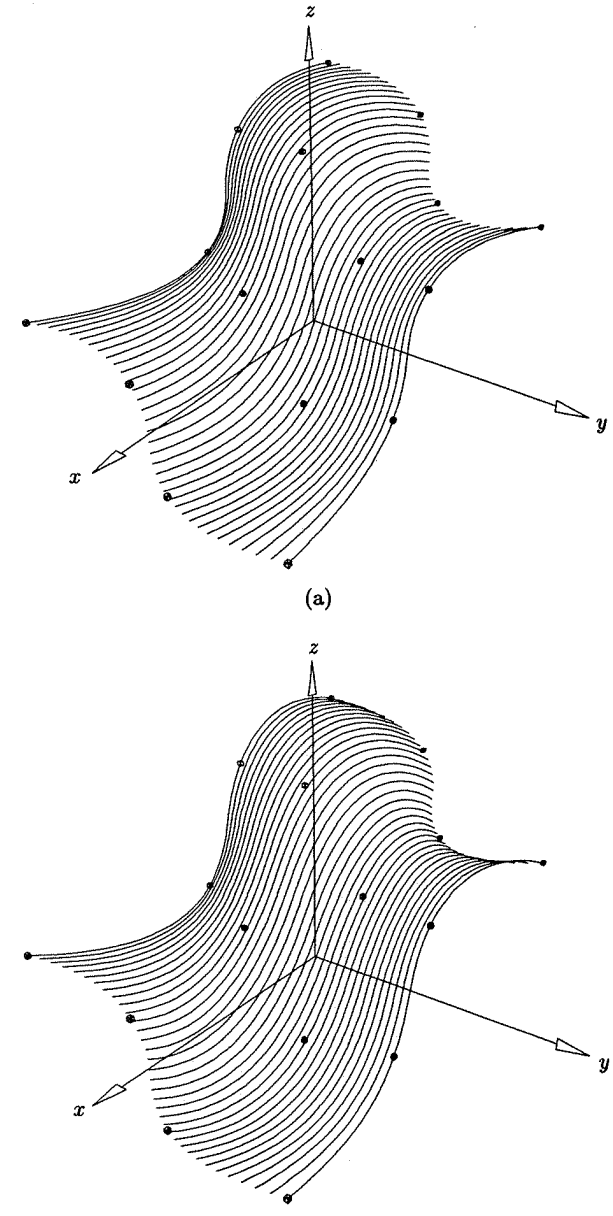
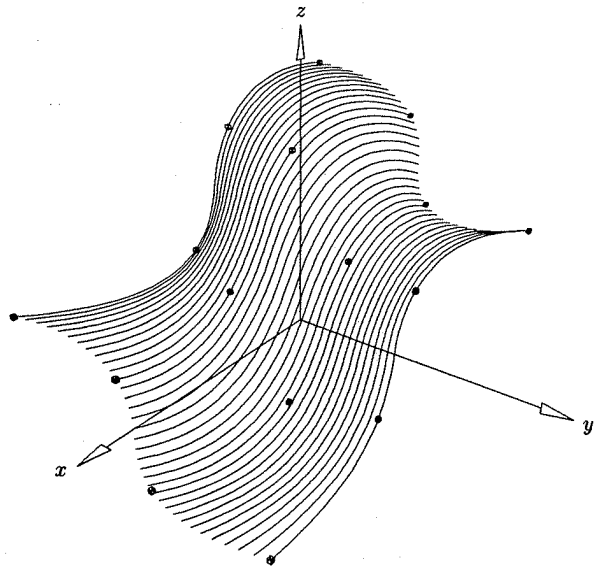


Figure 9.16. A surface interpolation with degree $(2,3)$ surfaces obtained by using different parameterizations and knot vectors.



(c)

Figure 9.16. (*Continued.*)

are possible with cubics, higher degree must be used if a reasonable amount of flexibility is to be maintained.

Obtaining the Bézier segments, $\mathbf{C}_i(u)$, requires computation of the inner Bézier control points, one point for quadratics, two for cubics. These control points lie on the lines which are tangent to the curve at the $\mathbf{Q}_k$; thus, we require tangent vectors $\mathbf{T}_k$ at each $\mathbf{Q}_k$. In some cases they can be input along with the $\mathbf{Q}_k$; e.g., they are easily obtained when computing points of intersection between two surfaces. However, if not input then they must be computed as part of the fitting algorithm. A number of methods exist; Boehm et al [Boeh84] gives a survey of various methods. Let

$$\Delta\bar{u}_k = \bar{u}_k - \bar{u}_{k-1} \quad \mathbf{q}_k = \mathbf{Q}_k - \mathbf{Q}_{k-1} \quad \mathbf{d}_k = \frac{\mathbf{q}_k}{\Delta\bar{u}_k}$$

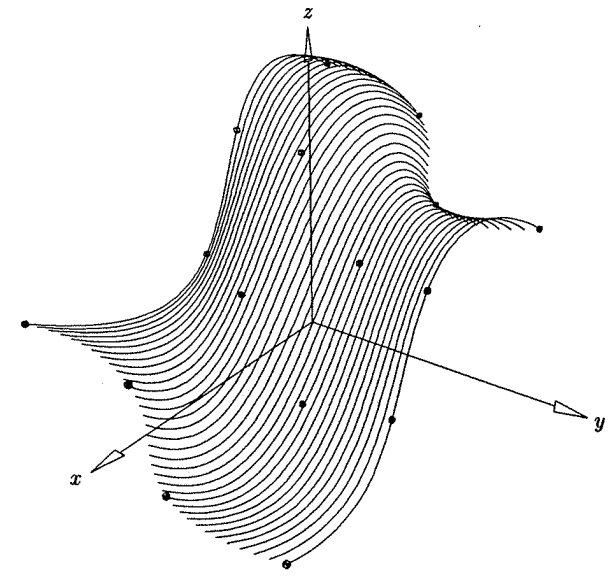
All methods have one of two forms

$$\mathbf{D}_k = (1-\alpha_k)\mathbf{d}_k + \alpha_k\mathbf{d}_{k+1} \tag{9.28}$$

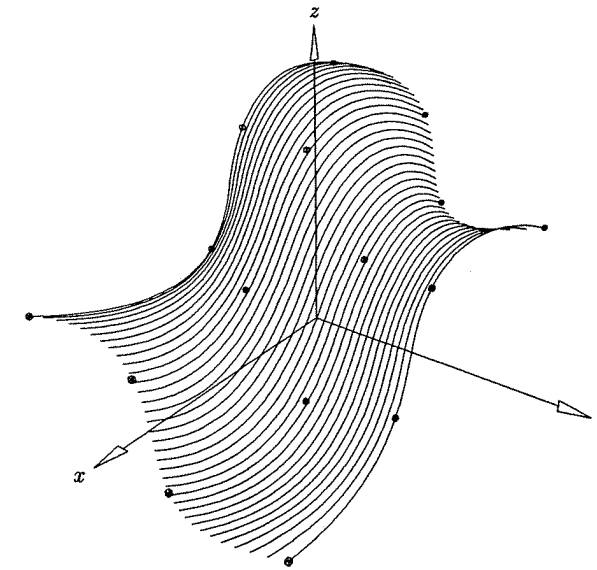
or

$$\mathbf{T}_k = \frac{\mathbf{V}_k}{|\,\mathbf{V}_k\,|} \qquad \mathbf{V}_k = (1-\alpha_k)\mathbf{q}_k + \alpha_k\mathbf{q}_{k+1} \tag{9.29}$$

(see Figure 9.19). Notice that Eq. (9.28) assumes that values for the $\bar{u}_k$ have been assigned. The vectors $\mathbf{D}_k$ can be viewed as estimates for the derivatives.

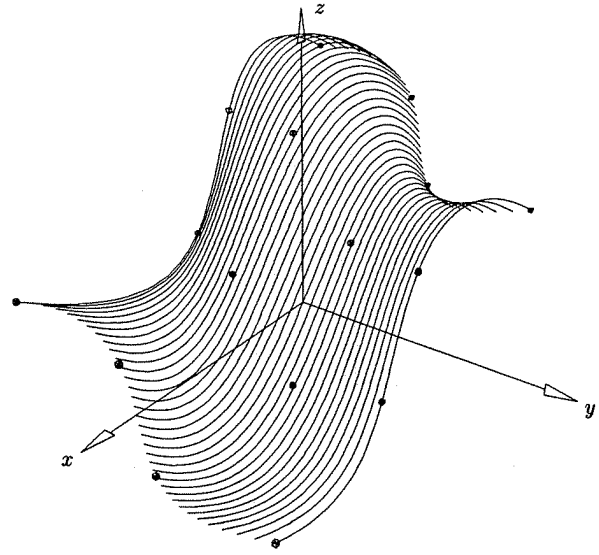


(a)



(b)

Figure 9.17. Surface interpolants using different degrees; the parameterization is chord length and knot vector obtained by averaging.



(c)

Figure 9.17. (*Continued.*)

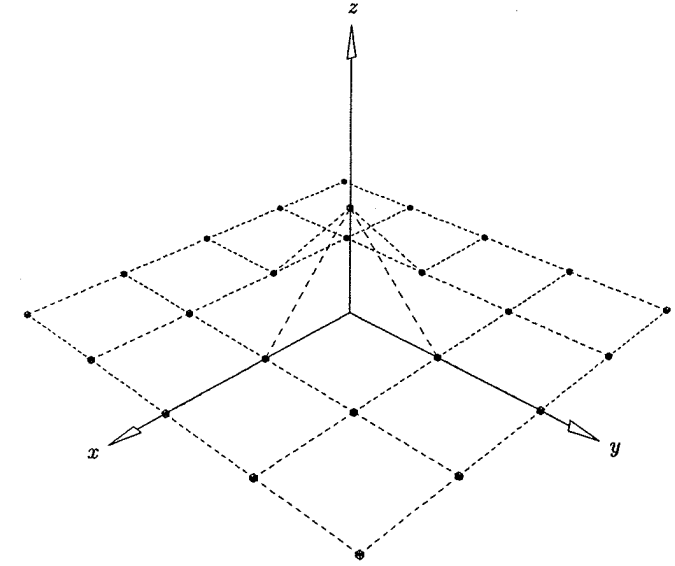
Equation (9.29) does not use parameters $\bar{u}_k$, and the resulting vectors should be viewed as tangent directions only. Assignment of magnitudes and parameters must be done in conjunction with one another; they are not independent. Notice also that we use the notation $\mathbf{T}$ strictly for *unit length* tangent vectors. Equations (9.28) and (9.29) are linear interpolations. The various schemes differ in the way they compute the interpolation parameter α_k; it typically depends on either three or five neighboring points. For example, the Bessel method [DeBo78] is a three-point method using

$$\alpha_k = \frac{\Delta\bar{u}_k}{\Delta\bar{u}_k + \Delta\bar{u}_{k+1}} \qquad k = 1, \ldots, n-1 \tag{9.30}$$

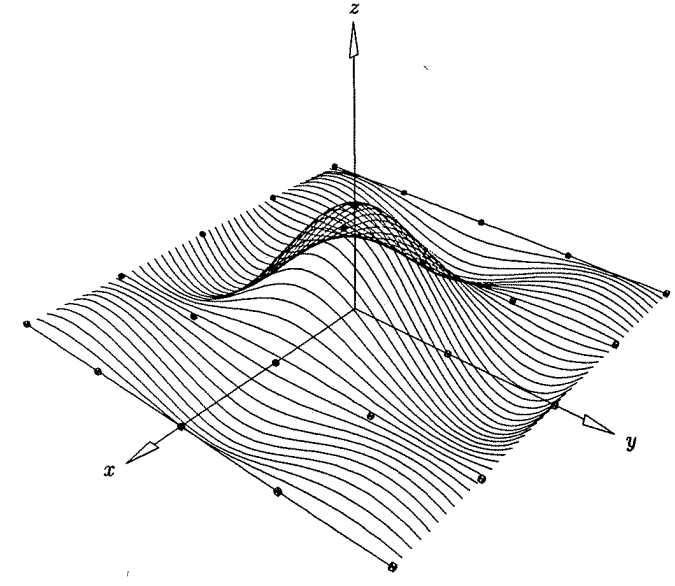
together with Eq. (9.28). Setting

$$\alpha_k = \frac{|\mathbf{q}_{k-1} \times \mathbf{q}_k|}{|\mathbf{q}_{k-1} \times \mathbf{q}_k| + |\mathbf{q}_{k+1} \times \mathbf{q}_{k+2}|} \qquad k = 2, \ldots, n-2 \tag{9.31}$$

together with Eq. (9.29) yields a five-point method to obtain $\mathbf{T}_k$ [Akim70; Renn82; Pieg87d]. It has the advantage that three collinear points, $\mathbf{Q}_{k-1}$, $\mathbf{Q}_k$, $\mathbf{Q}_{k+1}$, yield a $\mathbf{T}_k$ which is parallel to the line segment. The denominator of Eq. (9.31) vanishes if $\mathbf{Q}_{k-2}, \mathbf{Q}_{k-1}, \mathbf{Q}_k$ are collinear and $\mathbf{Q}_k, \mathbf{Q}_{k+1}, \mathbf{Q}_{k+2}$ are collinear. This implies either a corner at $\mathbf{Q}_k$ or a straight line segment from $\mathbf{Q}_{k-2}$ to $\mathbf{Q}_{k+2}$. In these cases α_k can be defined in a number of ways; we choose



(a)



(b)

Figure 9.18. Global surface interpolation example. (a) Data containing coplanar points; (b) interpolating surface.

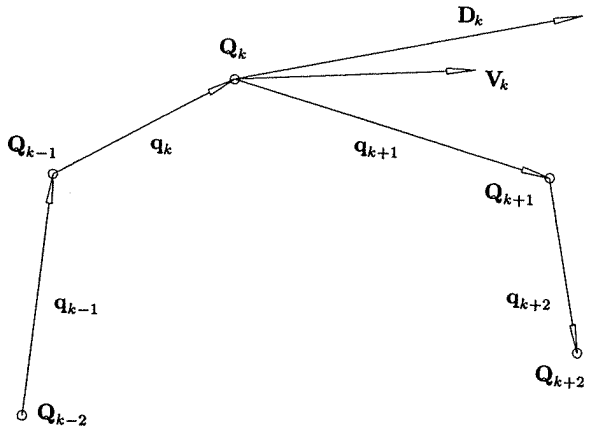


Figure 9.19. Computation of tangent ($\mathbf{V}_k$) and derivative ($\mathbf{D}_k$) vectors for local curve interpolation.

- $\alpha_k = 1$, which implies $\mathbf{V}_k = \mathbf{q}_{k+1}$; this produces a corner at $\mathbf{Q}_k$ if one is implied by the data;
- $\alpha_k = 1/2$, which implies $\mathbf{V}_k = 1/2(\mathbf{q}_k + \mathbf{q}_{k+1})$; this choice smoothes out a corner if one is implied.

Based on these choices, a local curve interpolation routine can accept an input flag, indicating whether or not to preserve corners. All methods require special treatment at the ends. For the three-point scheme we can set

$$\mathbf{D}_0 = 2\mathbf{d}_1 - \mathbf{D}_1 \qquad \mathbf{D}_n = 2\mathbf{d}_n - \mathbf{D}_{n-1} \tag{9.32}$$

and for the five-point scheme we can use

$$\begin{aligned} \mathbf{q}_0 &= 2\mathbf{q}_1 - \mathbf{q}_2 & \mathbf{q}_{-1} &= 2\mathbf{q}_0 - \mathbf{q}_1 \\ \mathbf{q}_{n+1} &= 2\mathbf{q}_n - \mathbf{q}_{n-1} & \mathbf{q}_{n+2} &= 2\mathbf{q}_{n+1} - \mathbf{q}_n \end{aligned} \tag{9.33}$$

to substitute into Eqs. (9.31) and (9.29) in order to obtain $\mathbf{T}_0, \mathbf{T}_1$ and $\mathbf{T}_{n-1}, \mathbf{T}_n$. We are now ready to present several local interpolation methods.

9.3.2 LOCAL PARABOLIC CURVE INTERPOLATION

We start with a simple two-dimensional, nonrational, quadratic scheme. Let $\{\mathbf{Q}_k\}$, $k = 0, \ldots, n$, be points in the xy plane. Use some tangent estimation scheme to compute the corresponding $\{\mathbf{T}_k\}$, e.g., Eqs. (9.29), (9.31), and (9.33). Let L_k denote the directed line defined by $(\mathbf{Q}_k, \mathbf{T}_k)$, and $\mathbf{R}_k$ the intersection

point of L_{k-1} and L_k. Assume for the moment that the intersection exists, and that

$$\gamma_{k-1} > 0 \qquad \gamma_k < 0 \tag{9.34}$$

where $\mathbf{R}_k = \mathbf{Q}_{k-1} + \gamma_{k-1}\mathbf{T}_{k-1} \qquad \mathbf{R}_k = \mathbf{Q}_k + \gamma_k \mathbf{T}_k$

(we drop this restriction later). Then we choose as our control points

$$\mathbf{Q}_0, \mathbf{R}_1, \mathbf{Q}_1, \mathbf{R}_2, \ldots, \mathbf{R}_n, \mathbf{Q}_n \tag{9.35}$$

Let $\bar{u}_0 = 0$ and $\bar{u}_n = 1$. If $\{\bar{u}_i\}$, $i = 1, \ldots, n-1$, is any sequence of numbers satisfying $\bar{u}_{i-1} < \bar{u}_i < \bar{u}_{i+1}$, then the control points of Eq. (9.35), together with the knot vector

$$U = \{0, 0, 0, \bar{u}_1, \bar{u}_1, \bar{u}_2, \bar{u}_2, \ldots, \bar{u}_{n-1}, \bar{u}_{n-1}, 1, 1, 1\} \tag{9.36}$$

define a nonrational, G^1 continuous, quadratic B-spline curve interpolating the $\{\mathbf{Q}_k\}$ (see Figure 9.20). The choice of the internal knots does not affect the shape of the curve, only the parameterization. It is possible to choose the knots so that the resulting curve is C^1 continuous; in doing so, we can remove the control points $\mathbf{Q}_1, \ldots, \mathbf{Q}_{n-1}$ and one occurrence of each of the interior knots. We simply equate start and end derivatives at $\mathbf{Q}_{k-1}$ (using Eq. [3.7] transformed to the intervals $[\bar{u}_{k-2}, \bar{u}_{k-1}]$ and $[\bar{u}_{k-1}, \bar{u}_k]$) and obtain the formula

$$\bar{u}_0 = 0 \qquad \bar{u}_1 = 1$$

$$\bar{u}_k = \bar{u}_{k-1} + (\bar{u}_{k-1} - \bar{u}_{k-2}) \frac{|\mathbf{R}_k - \mathbf{Q}_{k-1}|}{|\mathbf{Q}_{k-1} - \mathbf{R}_{k-1}|} \qquad k = 2, \ldots, n \tag{9.37}$$

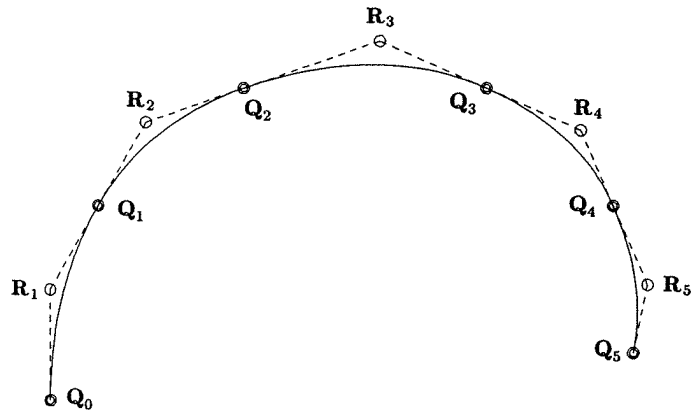


Figure 9.20. G^1 continuous quadratic curve interpolant.

The C^1 continuous curve interpolating the $\{\mathbf{Q}_k\}$ is then defined by the control points

$$\mathbf{Q}_0, \mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_{n-1}, \mathbf{R}_n, \mathbf{Q}_n \tag{9.38}$$

and the knots

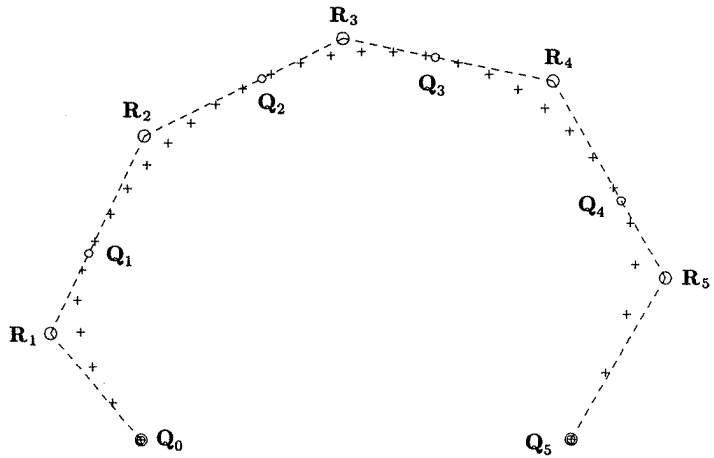
$$U = \left\{0,0,0, \frac{\bar{u}_1}{\bar{u}_n}, \frac{\bar{u}_2}{\bar{u}_n}, \ldots, \frac{\bar{u}_{n-2}}{\bar{u}_n}, \frac{\bar{u}_{n-1}}{\bar{u}_n}, 1,1,1\right\} \tag{9.39}$$

One must pay a price for the C^1 continuity and the reduction in the number of control points, namely in the parameterization; Figures 9.21a and 9.21b show this. The curve in these figures is marked at equally spaced parameter values. In Figure 9.21a the knots were computed using Eq. (9.37) and in Figure 9.21b using the chord length parameterization (Eq. [9.5]). Note the spacing of the $\mathbf{Q}_k$.

We now drop the restriction given in Eq. (9.34) and detail two special cases which arise when computing the $\mathbf{R}_k$:

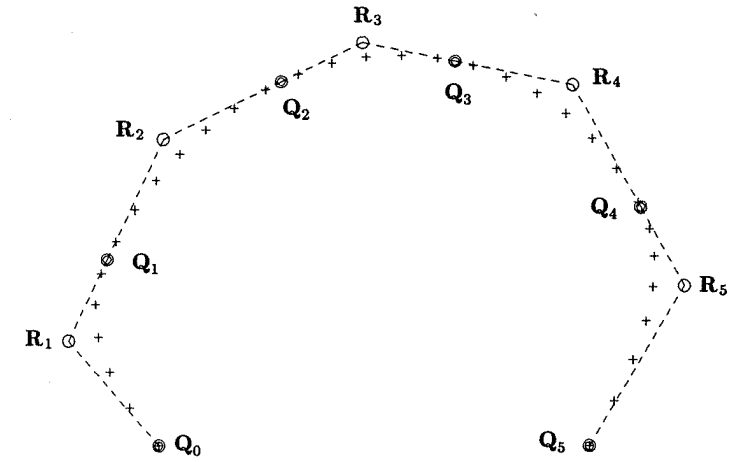
- $\mathbf{T}_{k-1}$ and $\mathbf{T}_k$ are parallel, hence $\mathbf{R}_k$ cannot be computed by intersection; this can indicate collinear segments, an inflection point, or a 180° turn in the curve;
- $\mathbf{R}_k$ can be computed, but γ_{k-1} and γ_k do not satisfy Eq. (9.34); this indicates either an inflection point or a turn of more than 180°.

The collinear segments case applies if $\mathbf{T}_{k-1}$ and $\mathbf{T}_k$ are both parallel to the chord $\mathbf{Q_{k-1}Q_k}$. We handle this case simply by setting



(a)

Figure 9.21. Parameterizations of the curve interpolant in Figure 9.20. (a) C^1 parameterization (Eqs. [9.39]–[9.41]); (b) G^1 parameterization using chord length parameters.



(b)

Figure 9.21. (*Continued.*)

$$\mathbf{R}_k = \frac{1}{2}(\mathbf{Q}_{k-1} + \mathbf{Q}_k) \tag{9.40}$$

All other special cases are handled by creating two parabolic segments between $\mathbf{Q_{k-1}}$ and $\mathbf{Q}_k$, instead of one. Thus, we must compute three additional points, $\mathbf{R}'_k$, $\mathbf{Q}'_k$ and $\mathbf{R}'_{k+1}$ (see Figures 9.22a and 9.22b). Setting

$$\mathbf{R}'_k = \mathbf{Q}_{k-1} + \gamma_k \mathbf{T}_{k-1} \qquad \mathbf{R}'_{k+1} = \mathbf{Q}_k - \gamma_{k+1}\mathbf{T}_k \tag{9.41}$$

and

$$\mathbf{Q}'_k = \frac{\gamma_k \mathbf{R}'_{k+1} + \gamma_{k+1}\mathbf{R}'_k}{\gamma_k + \gamma_{k+1}} \tag{9.42}$$

it remains to determine reasonable choices for γ_k and γ_{k+1}. If $\mathbf{T}_{k-1}$ and $\mathbf{T}_k$ are parallel (but not to $\mathbf{Q}_{k-1}\mathbf{Q}_k$), then set

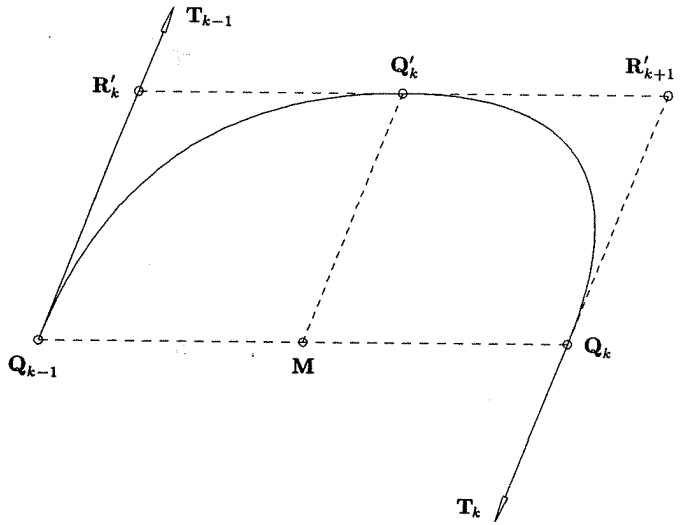
$$\gamma_k = \gamma_{k+1} = \frac{1}{2} \mid \mathbf{Q}_{k-1}\mathbf{Q}_k \mid \tag{9.43}$$

This is illustrated in Figure 9.22a for the case of a 180° turn. Now consider the case when $\mathbf{T}_{k-1}$ and $\mathbf{T}_k$ are not parallel, but Eq. (9.34) does not hold (see Figure 9.22b). Intuitively, γ_k and γ_{k+1} should depend on the angles θ_{k-1} and θ_k subtended by the chord $\mathbf{Q}_{k-1}\mathbf{Q}_k$ and the tangents $\mathbf{T}_{k-1}$ and $\mathbf{T}_k$, respectively ($0 \le \theta_{k-1}, \theta_k \le 90°$). Figure 9.22c shows a parabola defined by an isosceles triangle. For this figure one can easily derive

$$\gamma_k = \gamma_{k+1} = \frac{1}{4} \frac{\mid \mathbf{Q}_{k-1}\mathbf{Q}_k \mid}{\cos\theta}$$

This suggests setting

$$\gamma_k = \frac{1}{4} \frac{\mid \mathbf{Q}_{k-1}\mathbf{Q}_k \mid}{\alpha \cos\theta_k + (1-\alpha)\cos\theta_{k-1}}$$



(a)

Figure 9.22. Computing piecewise parabolic arcs to tangents at neighboring data points. (a) Parallel end tangents; (b) a turning point implied by tangents $\mathbf{T}_{k-1}$ and $\mathbf{T}_k$; (c) parabola defined by an isosceles control triangle.

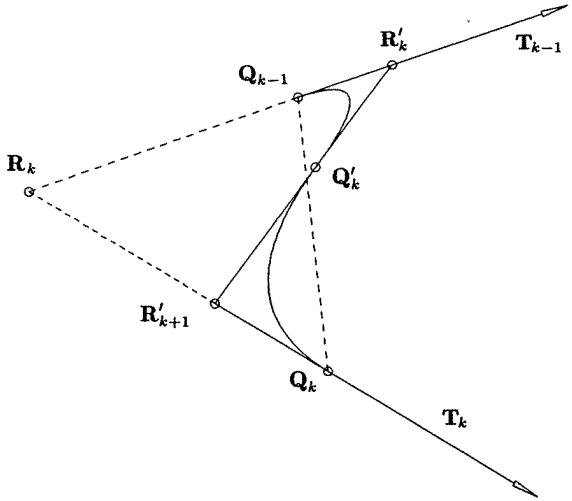
$$\gamma_{k+1} = \frac{1}{4} \frac{| \mathbf{Q}_{k-1}\mathbf{Q}_k |}{\alpha \cos\theta_{k-1} + (1-\alpha)\cos\theta_k} \tag{9.44}$$

where α is some constant between 0 and 1. Based on experimental observation, we choose $\alpha = 2/3$. The two curve segments of Figure 9.22b are obtained using Eqs. (9.41), (9.42), and (9.44).

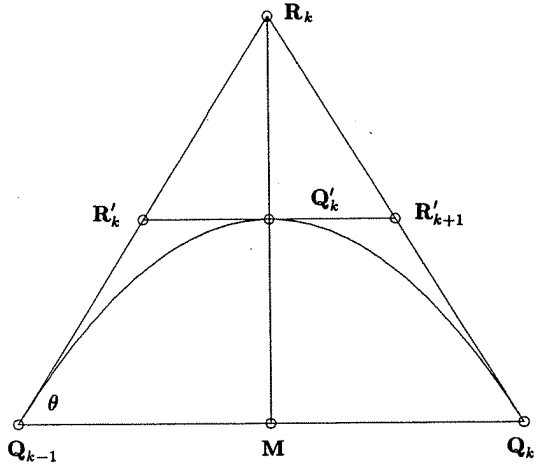
Figures 9.23–9.25 show several examples. In Figure 9.23a the tangents are computed by Bessel's method [DeBo78], while in Figure 9.23b Akima's method [Akim70] is used. Note how collinearity is maintained in Figure 9.23b. In Figure 9.24a the "no corner" option was selected, resulting in a (perhaps unwanted) smooth connection of each segment. Figure 9.24b shows the corner preserving case, using the same algorithm with only a different flag setting. Figure 9.25 shows the profile of a shoe sole, obtained using this interpolation method.

9.3.3 LOCAL RATIONAL QUADRATIC CURVE INTERPOLATION

The preceding method can be modified to produce rational quadratic curves. All weights at the $\mathbf{Q}_k$ are set to 1; weights at the $\mathbf{R}_k$ can be freely chosen. For example, some applications may desire that the segments be more "circular" rather than parabolic. More specifically, for $k = 1, \ldots, n$, a conic arc is defined



(b)



(c)

Figure 9.22. (*Continued.*)

by $\mathbf{Q}_{k-1}$, $\mathbf{R}_k$,$\mathbf{Q}_k$, and the weight w_k at $\mathbf{R}_k$. The w_k can be set as follows:

1. if $\mathbf{Q}_{k-1}$, $\mathbf{R}_k$, and $\mathbf{Q}_k$ are collinear, set $w_k = 1$;
2. if the triangle is isosceles ($| \mathbf{Q}_{k-1}\mathbf{R}_k | = | \mathbf{Q}_k\mathbf{R}_k |$), set w_k according to Eq. (7.33) (a precise circular arc);

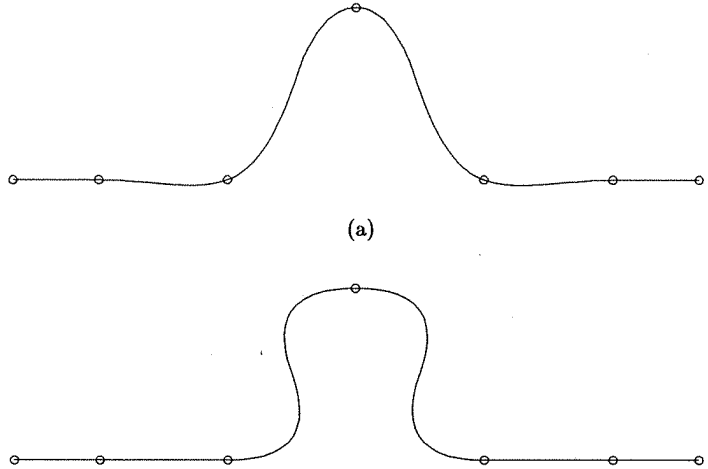


Figure 9.23. Local quadratic curve interpolants. (a) Tangents computed by Bessel's method; (b) tangents computed by Akima's method.

3. if the triangle is not isosceles, set w_k as follows (see Figure 9.26):
   a. let $\mathbf{M} = \frac{1}{2}(\mathbf{Q}_{k-1} + \mathbf{Q}_k)$;
   b. set $\mathbf{S}_1$ to be the intersection of $\mathbf{MR}_k$, with the bisector of the angle $\angle \mathbf{R}_k \mathbf{Q}_{k-1} \mathbf{Q}_k$;
   c. set $\mathbf{S}_2$ to be the intersection of $\mathbf{MR}_k$, with the bisector of the angle $\angle \mathbf{Q}_{k-1} \mathbf{Q}_k \mathbf{R}_k$;
   d. set $\mathbf{S} = \frac{1}{2}(\mathbf{S}_1 + \mathbf{S}_2)$;
   e. with $\mathbf{S}$ as the shoulder point, use Eqs. (7.30) and (7.31) to compute the w_k.

Recognition of the special cases in computing $\mathbf{R}_k$ is exactly as detailed; however, the computation of $\mathbf{R}'_k$ and $\mathbf{R}'_{k+1}$ is slightly different. Referring again to Figure 9.22c but assuming the isosceles triangle defines a circular arc, it follows from Eqs. (7.31)–(7.33) that

$$\gamma_k = \frac{1}{2} \frac{|\mathbf{Q}_{k-1}\mathbf{Q}_k|}{1 + \cos\theta}$$

Thus, we set

$$\gamma_k = \frac{1}{2} \frac{|\mathbf{Q}_{k-1}\mathbf{Q}_k|}{1 + \alpha\cos\theta_k + (1-\alpha)\cos\theta_{k-1}}$$

$$\gamma_{k+1} = \frac{1}{2} \frac{|\mathbf{Q}_{k-1}\mathbf{Q}_k|}{1 + \alpha\cos\theta_{k-1} + (1-\alpha)\cos\theta_k} \tag{9.45}$$

where α is the same constant as in Eq. (9.44).

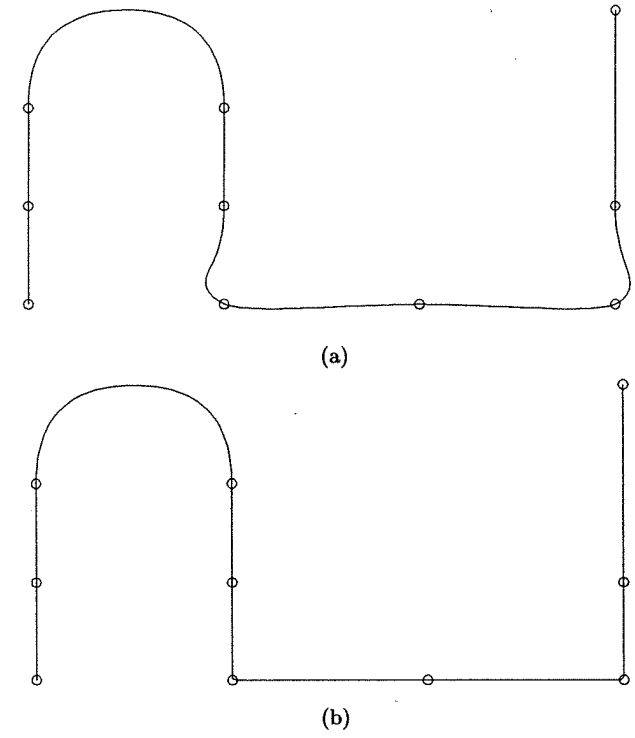


Figure 9.24. Local quadratic curve interpolants. (a) No corner is allowed; (b) the corner is maintained.

Figure 9.27 shows an example of rational quadratic curve interpolation. Notice that circles are precisely reproduced from four points, and the two corners are preserved by setting the corner flag.

9.3.4 LOCAL CUBIC CURVE INTERPOLATION

Cubics easily handle three-dimensional data and inflection points without special treatment. We begin by deriving a general fact about cubic Bézier curves, which is of interest in its own right. Let $\mathbf{P}_0$ and $\mathbf{P}_3$ be two endpoints, and $\mathbf{T}_0$ and $\mathbf{T}_3$ be the corresponding tangent directions with unit length. It is possible to construct a cubic Bézier curve, $\mathbf{C}(u)$, $u \in [0,1]$, with these endpoints and tangent directions, and satisfying

$$\alpha = |\mathbf{C}'(0)| = |\mathbf{C}'\left(\frac{1}{2}\right)| = |\mathbf{C}'(1)| \tag{9.46}$$

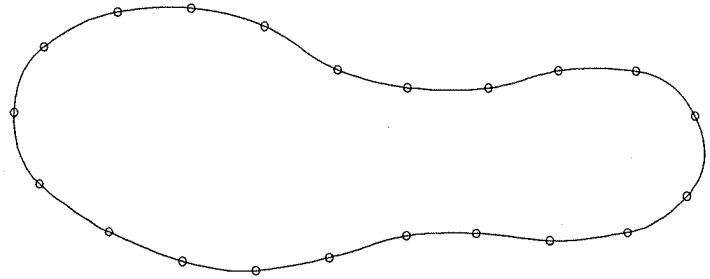


Figure 9.25. A piecewise G^1 parabolic curve interpolant.

(see Figure 9.28), that is, the speed is equal at the start point, midpoint, and endpoint of the curve. Equations (9.46) and (1.10) imply that

$$\mathbf{P}_1 = \mathbf{P}_0 + \frac{1}{3}\alpha\mathbf{T}_0 \qquad \mathbf{P}_2 = \mathbf{P}_3 - \frac{1}{3}\alpha\mathbf{T}_3 \tag{9.47}$$

We apply the deCasteljau Algorithm of Chapter 1 at $u = 1/2$, and, recalling the notation used in that algorithm (see Figure 1.17), we write $\mathbf{P}_0^3 = \mathbf{C}(1/2)$. By backing out of the deCasteljau algorithm one can derive

$$\mathbf{P}_1^2 - \mathbf{P}_0^3 = \frac{1}{8}(\mathbf{P}_3 + \mathbf{P}_2 - \mathbf{P}_1 - \mathbf{P}_0) \tag{9.48}$$

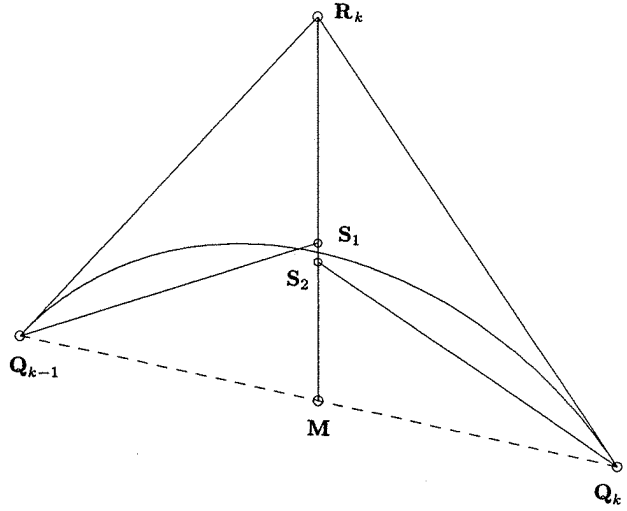


Figure 9.26. Choosing the weight (shoulder point) for rational quadratic interpolation.

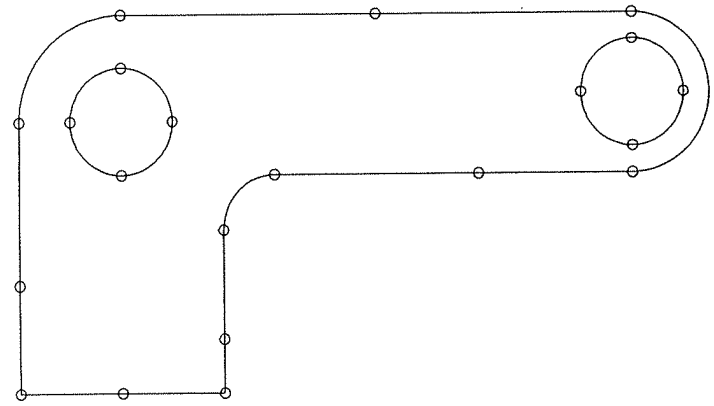


Figure 9.27. Rational quadratic curve interpolants; circles are precisely reproduced, and corners are maintained.

Applying Eq. (1.10) and considering the bisection of the parameter range, it follows that

$$\mathbf{C}'\left(\frac{1}{2}\right) = 6\left(\mathbf{P}_1^2 - \mathbf{P}_0^3\right) \tag{9.49}$$

Setting the speed equal to α, substituting Eq. (9.48) into Eq. (9.49), and using Eq. (9.47), we obtain

$$\begin{aligned}\frac{8}{6}\alpha &= |\,\mathbf{P}_3 + \mathbf{P}_2 - \mathbf{P}_1 - \mathbf{P}_0\,| \\ &= |\,\mathbf{P}_3 + \left(\mathbf{P}_3 - \frac{1}{3}\alpha\mathbf{T}_3\right) - \left(\mathbf{P}_0 + \frac{1}{3}\alpha\mathbf{T}_0\right) - \mathbf{P}_0\,|\end{aligned}$$

which leads to

$$16\alpha^2 = \alpha^2|\,\mathbf{T}_0 + \mathbf{T}_3\,|^2 - 12\alpha(\mathbf{P}_3 - \mathbf{P}_0)\cdot(\mathbf{T}_0 + \mathbf{T}_3) + 36|\,\mathbf{P}_3 - \mathbf{P}_0\,|^2$$

and finally

$$a\alpha^2 + b\alpha + c = 0 \tag{9.50}$$

where

$$a = 16 - |\,\mathbf{T}_0 + \mathbf{T}_3\,|^2 \quad b = 12(\mathbf{P}_3 - \mathbf{P}_0)\cdot(\mathbf{T}_0 + \mathbf{T}_3) \quad c = -36|\,\mathbf{P}_3 - \mathbf{P}_0\,|^2$$

Equation (9.50) has two real solutions for α, one positive and one negative. Substituting the positive solution into Eq. (9.47) yields the desired $\mathbf{P}_1$ and $\mathbf{P}_2$.

We now return to the cubic interpolation problem. Let $\{\mathbf{Q}_k\}$, $k = 0, \ldots, n$, be a set of three-dimensional data points. If tangent vectors are not given, compute them (Eqs. [9.29], [9.31], and [9.33]). We must construct a cubic Bézier

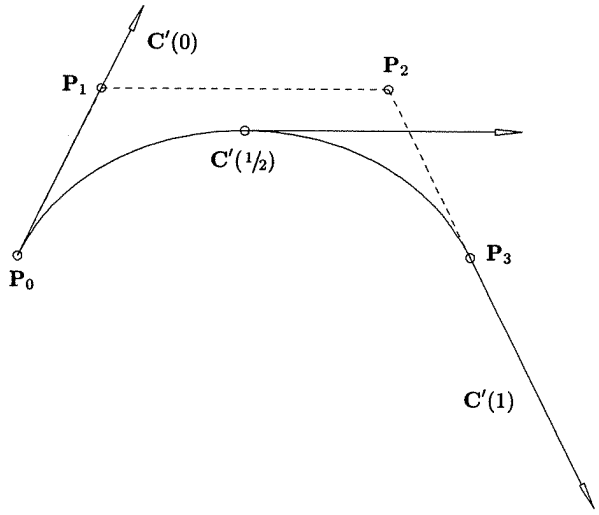


Figure 9.28. A cubic Bézier curve with equal tangent magnitudes at parameters 0, 1/2, and 1.

curve segment, $\mathbf{C}_k(u)$, between each pair, $\mathbf{Q}_k$, $\mathbf{Q}_{k+1}$. Denote the Bézier control points by

$$\mathbf{P}_{k,0} = \mathbf{Q}_k \quad \mathbf{P}_{k,1} \quad \mathbf{P}_{k,2} \quad \mathbf{P}_{k,3} = \mathbf{Q}_{k+1} \tag{9.51}$$

We must determine suitable locations for $\mathbf{P}_{k,1}$ and $\mathbf{P}_{k,2}$ along $\mathbf{T}_k$ and $\mathbf{T}_{k+1}$, respectively. It is possible to obtain a C^1 continuous cubic and to achieve a good approximation to a uniform parameterization. The method is due to Renner [Renn82]. True uniform parameterization means constant speed over the entire parameter range. We construct a curve with equal speed at each $\mathbf{Q}_k$ and at the midpoint of each Bézier segment. Set $\bar{u}_0 = 0$. Now, for $k = 0, \ldots, n-1$, the $\bar{u}_{k+1}$ and the two inner control points of $\mathbf{C}_k(u)$ are computed as follows:

1. use Eq. (9.50) to compute α and Eq. (9.47) to compute $\mathbf{P}_{k,1}$ and $\mathbf{P}_{k,2}$;
2. set

$$\bar{u}_{k+1} = \bar{u}_k + 3 \mid \mathbf{P}_{k,1} - \mathbf{P}_{k,0} \mid \tag{9.52}$$

This algorithm yields n Bézier segments, each having speed equal to 1 at their end- and midpoints with respect to their parameter ranges, $[\bar{u}_k, \bar{u}_{k+1}]$. Thus, a C^1 continuous cubic B-spline curve interpolating the $\mathbf{Q}_k$ is defined by the control points (see Figure 9.29)

$$\mathbf{Q}_0, \mathbf{P}_{0,1}, \mathbf{P}_{0,2}, \mathbf{P}_{1,1}, \mathbf{P}_{1,2}, \ldots, \mathbf{P}_{n-2,2,}, \mathbf{P}_{n-1,1}, \mathbf{P}_{n-1,2,}, \mathbf{Q}_n \tag{9.53}$$

and the knots

$$U = \left\{0,0,0,0, \frac{\bar{u}_1}{\bar{u}_n}, \frac{\bar{u}_1}{\bar{u}_n}, \frac{\bar{u}_2}{\bar{u}_n}, \frac{\bar{u}_2}{\bar{u}_n}, \ldots, \frac{\bar{u}_{n-1}}{\bar{u}_n}, \frac{\bar{u}_{n-1}}{\bar{u}_n}, 1,1,1,1\right\} \tag{9.54}$$

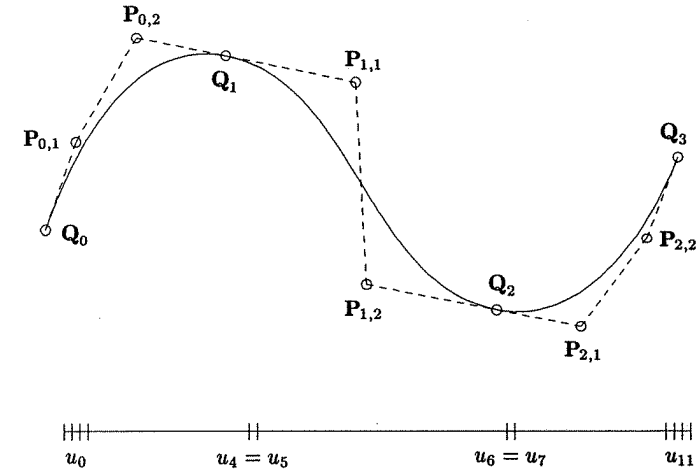


Figure 9.29. C^1 local cubic curve interpolant.

Figure 9.30 shows an example of cubic curve interpolation using this method. The curve is marked at points corresponding to equally spaced parameter values. Figure 9.31 is another example of this interpolation, using the same data set as in Figure 9.7. Notice how collinear points are fitted by straight cubic segments.

9.3.5 LOCAL BICUBIC SURFACE INTERPOLATION

A $C^{(1,1)}$ continuous, bicubic, local surface interpolation scheme is also possible (C^1 continuous in both the u and the v directions). Let $\{\mathbf{Q}_{k,\ell}\}$, $k = 0, \ldots, n$

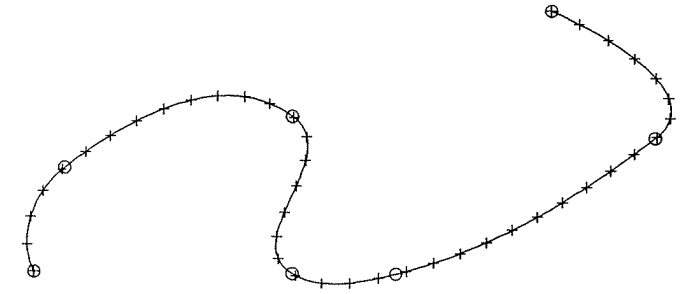


Figure 9.30. Parameterization of a local cubic curve interpolant; the points are computed at equally spaced parameter values.

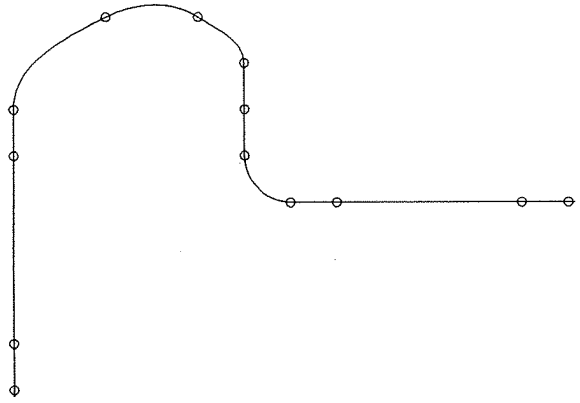


Figure 9.31. The cubic curve interpolant to data shown in Figure 9.7; note how collinearity is preserved.

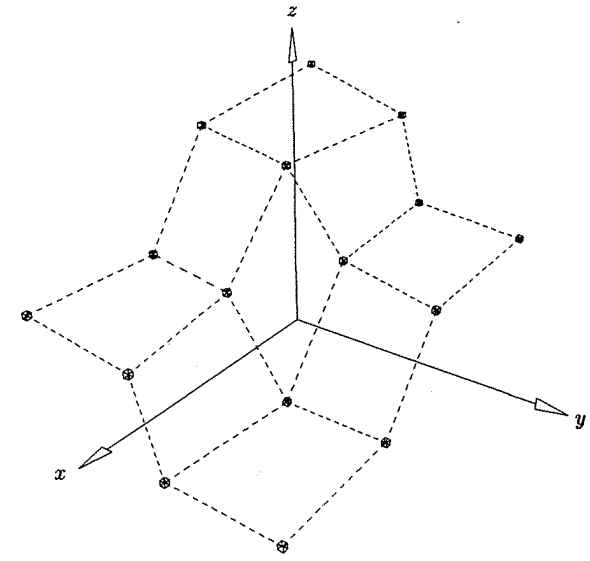
and $\ell = 0, \ldots, m$, be a set of data points, and let $\{(\bar{u}_k, \bar{v}_\ell)\}$ be the corresponding parameter pairs, computed by chord length averaging (as in Algorithm `A9.3`). The following method produces a bicubic surface, $\mathbf{S}(u, v)$, satisfying

$$\mathbf{S}(\bar{u}_k, \bar{v}_\ell) = \sum_{i=0}^{2n+1} \sum_{j=0}^{2m+1} N_{i,3}(\bar{u}_k) N_{j,3}(\bar{v}_\ell) \mathbf{P}_{i,j} \tag{9.55}$$

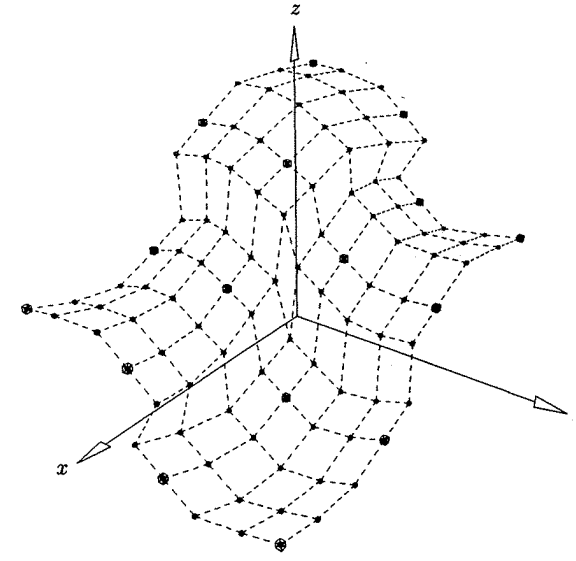
We obtain the surface by constructing nm bicubic Bézier patches, $\{\mathbf{B}_{k,\ell}(u, v)\}$, $k = 0, \ldots, n-1$, $\ell = 0, \ldots, m-1$, where $\mathbf{Q}_{k,\ell}$, $\mathbf{Q}_{k+1,\ell}$, $\mathbf{Q}_{k,\ell+1}$, $\mathbf{Q}_{k+1,\ell+1}$ are the corner points of the patch, and the patches join with $C^{(1,1)}$ continuity across their boundaries. Except for the surface boundaries, all rows and columns of control points containing the original $\{\mathbf{Q}_{k,\ell}\}$ are removed (see Figures 9.32b and 9.32c), leaving $(2n+2)(2m+2)$ control points in the final B-spline surface. The knot vectors are

$$\begin{aligned} U &= \{0, 0, 0, 0, \bar{u}_1, \bar{u}_1, \bar{u}_2, \bar{u}_2, \ldots, \bar{u}_{n-1}, \bar{u}_{n-1}, 1, 1, 1, 1\} \\ V &= \{0, 0, 0, 0, \bar{v}_1, \bar{v}_1, \bar{v}_2, \bar{v}_2, \ldots, \bar{v}_{m-1}, \bar{v}_{m-1}, 1, 1, 1, 1\} \end{aligned} \tag{9.56}$$

A bicubic Bézier patch has 16 control points. The 12 boundary control points are obtained by initially looping through the $m+1$ rows and $n+1$ columns of data and using a cubic curve interpolation scheme. The scheme is slightly different from the one detailed previously, because we already have the parameters $\{(\bar{u}_k, \bar{v}_\ell)\}$. Notice that these had to be computed up front, since all rows (columns) must have the same parameterization in a tensor product surface. Thus, we can still force C^1 continuity at the segment endpoints, but we cannot force equal speed at the midpoint of the Bézier curve segments. More specifically, let $\ell = \ell_0$ be fixed, and consider the cubic curve interpolating the points

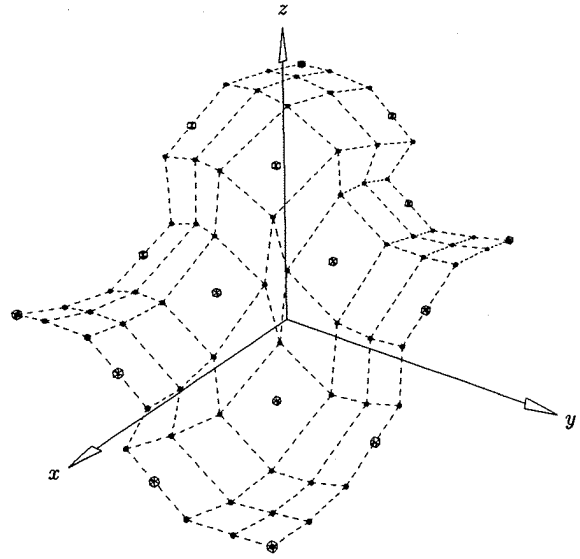


(a)

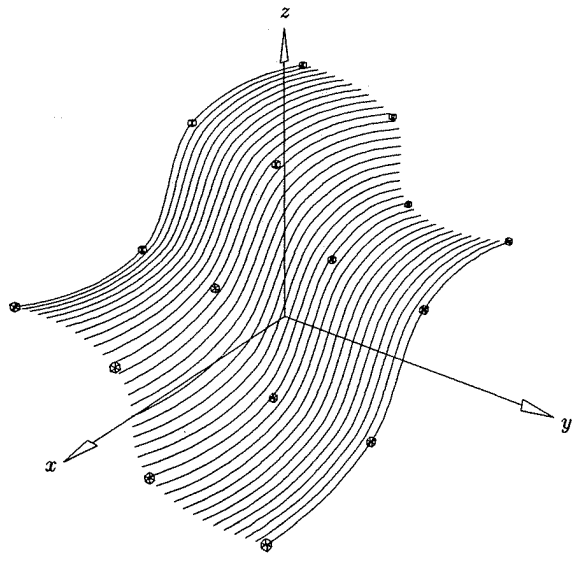


(b)

Figure 9.32. $C^{(1,1)}$ local bicubic surface interpolation. (a) The data set; (b) the Bézier net of the interpolant.



(c)



(d)

Figure 9.32. (*Continued.*) (c) a B-spline net (large cubes mark data points, and the small ones are the B-spline control points); (d) the surface interpolant.

$\mathbf{Q}_{0,\ell_0}, \ldots, \mathbf{Q}_{n,\ell_0}$. Let r_{ℓ_0} denote the total chord length of the ℓ_0th row. At each point, $\mathbf{Q}_{k,\ell_0}$, compute the $\mathbf{T}^u_{k,\ell_0}$ (the unit tangent in the u direction) as previously (Eqs. [9.29],[9.31], and [9.33]). Then the interior Bézier points on this row are computed by

$$\mathbf{P}^{k,\ell_0}_{1,0} = \mathbf{Q}_{k,\ell_0} + a\mathbf{T}^u_{k,\ell_0} \qquad \mathbf{P}^{k,\ell_0}_{2,0} = \mathbf{Q}_{k+1,\ell_0} - a\mathbf{T}^u_{k+1,\ell_0} \tag{9.57}$$

where
$$a = \frac{r_{\ell_0}(\bar{u}_{k+1} - \bar{u}_k)}{3} = \frac{r_{\ell_0}\Delta\bar{u}_{k+1}}{3}$$

The resulting curve is C^1 continuous, with derivative magnitude equal to r_{ℓ_0} at all $\mathbf{Q}_{k,\ell_0}$. The same technique is applied to the $n+1$ columns of data.

It remains to compute the four interior control points of each Bézier patch. This requires estimates for the mixed partial derivative, $\mathbf{D}^{uv}_{k,\ell}$, at each $\mathbf{Q}_{k,\ell}$, as well as Eq. (3.24) (and analogous formulas for the other three corners). We derive a formula for $\mathbf{D}^{uv}_{k,\ell}$ based on the three-point Bessel method (Eqs. [9.28], [9.30], and [9.32]). Let r_ℓ and s_k denote the total chord length of the ℓth row (kth column). Then

$$\mathbf{D}^u_{k,\ell} = r_\ell \mathbf{T}^u_{k,\ell} \qquad \mathbf{D}^v_{k,\ell} = s_k \mathbf{T}^v_{k,\ell} \tag{9.58}$$

Then set
$$\mathbf{d}^{vu}_{k,\ell} = (1-\alpha_k)\frac{\mathbf{D}^v_{k,\ell} - \mathbf{D}^v_{k-1,\ell}}{\Delta\bar{u}_k} + \alpha_k \frac{\mathbf{D}^v_{k+1,\ell} - \mathbf{D}^v_{k,\ell}}{\Delta\bar{u}_{k+1}}$$

and
$$\mathbf{d}^{uv}_{k,\ell} = (1-\beta_\ell)\frac{\mathbf{D}^u_{k,\ell} - \mathbf{D}^u_{k,\ell-1}}{\Delta\bar{v}_\ell} + \beta_\ell \frac{\mathbf{D}^u_{k,\ell+1} - \mathbf{D}^u_{k,\ell}}{\Delta\bar{v}_{\ell+1}}$$

with
$$\alpha_k = \frac{\Delta\bar{u}_k}{\Delta\bar{u}_k + \Delta\bar{u}_{k+1}} \qquad \beta_\ell = \frac{\Delta\bar{v}_\ell}{\Delta\bar{v}_\ell + \Delta\bar{v}_{\ell+1}}$$

Finally
$$\mathbf{D}^{uv}_{k,\ell} = \frac{\alpha_k \mathbf{d}^{uv}_{k,\ell} + \beta_\ell \mathbf{d}^{vu}_{k,\ell}}{\alpha_k + \beta_\ell} \tag{9.59}$$

The appropriate end formulas (Eq. [9.32]) must be used on the boundaries. The four interior control points of the (k,ℓ)th patch are now computed using Eqs. (9.59) and (3.24)

$$\mathbf{P}^{k,\ell}_{1,1} = \gamma\mathbf{D}^{uv}_{k,\ell} + \mathbf{P}^{k,\ell}_{0,1} + \mathbf{P}^{k,\ell}_{1,0} - \mathbf{P}^{k,\ell}_{0,0}$$

$$\mathbf{P}^{k,\ell}_{2,1} = -\gamma\mathbf{D}^{uv}_{k+1,\ell} + \mathbf{P}^{k,\ell}_{3,1} - \mathbf{P}^{k,\ell}_{3,0} + \mathbf{P}^{k,\ell}_{2,0}$$

$$\mathbf{P}^{k,\ell}_{1,2} = -\gamma\mathbf{D}^{uv}_{k,\ell+1} + \mathbf{P}^{k,\ell}_{1,3} - \mathbf{P}^{k,\ell}_{0,3} + \mathbf{P}^{k,\ell}_{0,2}$$

$$\mathbf{P}^{k,\ell}_{2,2} = \gamma\mathbf{D}^{uv}_{k+1,\ell+1} + \mathbf{P}^{k,\ell}_{2,3} + \mathbf{P}^{k,\ell}_{3,2} - \mathbf{P}^{k,\ell}_{3,3}$$

where
$$\gamma = \frac{\Delta\bar{u}_{k+1}\Delta\bar{v}_{\ell+1}}{9}$$

We now summarize with an algorithm. For the sake of efficiency, the sequence of events is slightly different than detailed here. The algorithm requires local arrays: td[n+1][m+1][3], which contains the $\mathbf{T}^u_{k,\ell}$, $\mathbf{T}^v_{k,\ell}$, and $\mathbf{D}^{uv}_{k,\ell}$; ub[n+1] and vb[m+1], which contain the parameter values; and r[m+1] and s[n+1], which store the total chord lengths of the rows and columns, respectively.

```
ALGORITHM A9.5
  LocalSurfInterp(n,m,Q,U,V,P)
    { /*  Local surface interpolation */
    { /*  through (n+1)(m+1) points  */
      /*  Input:  n,m,Q  */
      /*  Output: U,V,P  */
    total = 0.0;   /* get ub[], r[] and u direction tangents */
    for (k=0; k<=n; k++)   ub[k] = 0.0;
    for (l=0; l<=m; l++)
      {
      Compute and load T^u_{0,l} into td[0][l][0];
      r[l] = 0.0;
      for (k=1; k<=n; k++)
        {
        Compute and load T^u_{k,l} into td[k][l][0]
            /* Eqs.(9.31) and (9.33) */
        d = | Q_{k,l} - Q_{k-1,l} |;
        ub[k] = ub[k]+d;
        r[l] = r[l]+d;
        }
      total = total + r[l];
      }
    for (k=1; k<n; k++)   ub[k] = ub[k-1]+ub[k]/total;
    ub[n] = 1.0;
    total = 0.0;   /* get vb[], s[] and v direction tangents */
    for (l=0; l<=m; l++)   vb[l] = 0.0;
    for (k=0; k<=n; k++)
      {
      Compute and load T^v_{k,0} into td[k][0][1];
      s[k] = 0.0;
      for (l=1; l<=m; l++)
        {
        Compute and load T^v_{k,l} into td[k][l][1]
            /* Eqs.(9.31) and (9.33) */
        d = | Q_{k,l} - Q_{k,l-1} |;
        vb[l] = vb[l]+d;
        s[k] = s[k]+d;
        }
      total = total + s[k];
      }
```



```
    for (l=1; l<m; l++)   vb[l] = vb[l-1]+vb[l]/total;
    vb[m] = 1.0;
    Load the U knot vector;
    Load the V knot vector;
    Compute all Bézier control points along each row and
        column of data points.
    for (k=0; k<=n; k++)
      for (l=0; l<=m; l++)
        {
        Compute the D^{uv}_{k,l} by Eq.(9.59) and load into td[k][l][2].
        }
    for (k=0; k<n; k++)
      for (l=0; l<m; l++)
        {
        Compute the four inner control points of the (k,l)th
            Bezier patch and load them into P.
        }
    Load the NURBS control points by discarding Bézier points
        along inner rows and columns.   /* Figure 9.32c */
    }
```

Figure 9.32 shows the surface construction process. In Figure 9.32a data points are shown marked by small cubes. Figure 9.32b illustrates the control points of the Bézier patches. Eliminating Bézier points along inner rows and columns yields the NURBS surface control net depicted in Figure 9.32c. The surface is shown in Figure 9.32d.

This interpolation scheme easily generates closed $C^{(1,1)}$ continuous surfaces. Figures 9.33a and 9.33b show an example of a surface interpolation where the data indicates a closed surface in one direction. In Figures 9.34a and 9.34b the surface is closed in both directions (a toroidal-like interpolant). A nice property of this method is that flat (planar) patches are generated where the data points indicate planarity. An example of this is shown in Figures 9.35a and 9.35b.

Obtaining pleasing surfaces when the data is unevenly spaced is more difficult for surfaces than for curves. Figures 9.36a and 9.36b show an interpolation to very unevenly spaced data. Although the surface is not particularly pleasing, the results using the global methods of the previous section are not much different.

9.4 Global Approximation

Approximation is more difficult than interpolation. In interpolation, the number of control points is automatically determined by the chosen degree and the number of data items, knot placement is straightforward, and there is no curve or surface error to be checked. In approximation, a curve/surface error bound, E, is input along with the data to be fit. It is usually not known in advance how many control points are required to obtain the desired accuracy, E, and

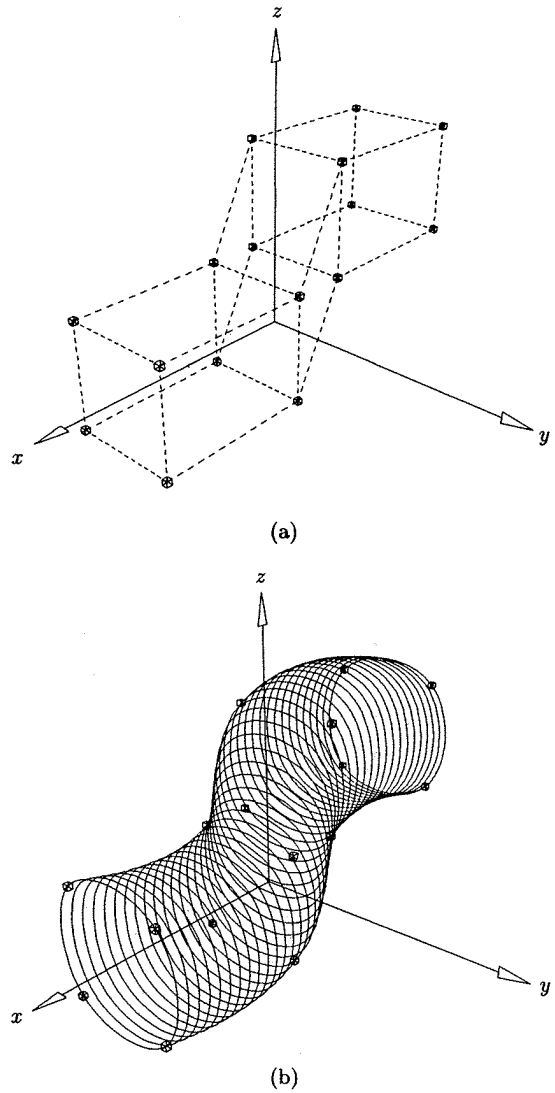


Figure 9.33. A surface interpolation example using a local bicubic surface interpolant. (a) Data points; (b) an interpolating surface.

hence approximation methods are generally iterative. Roughly speaking, global methods proceed in one of two ways:

1a. start with the minimum or a small number of control points;

b. fit an approximating curve or surface to the data, using a global fit method;

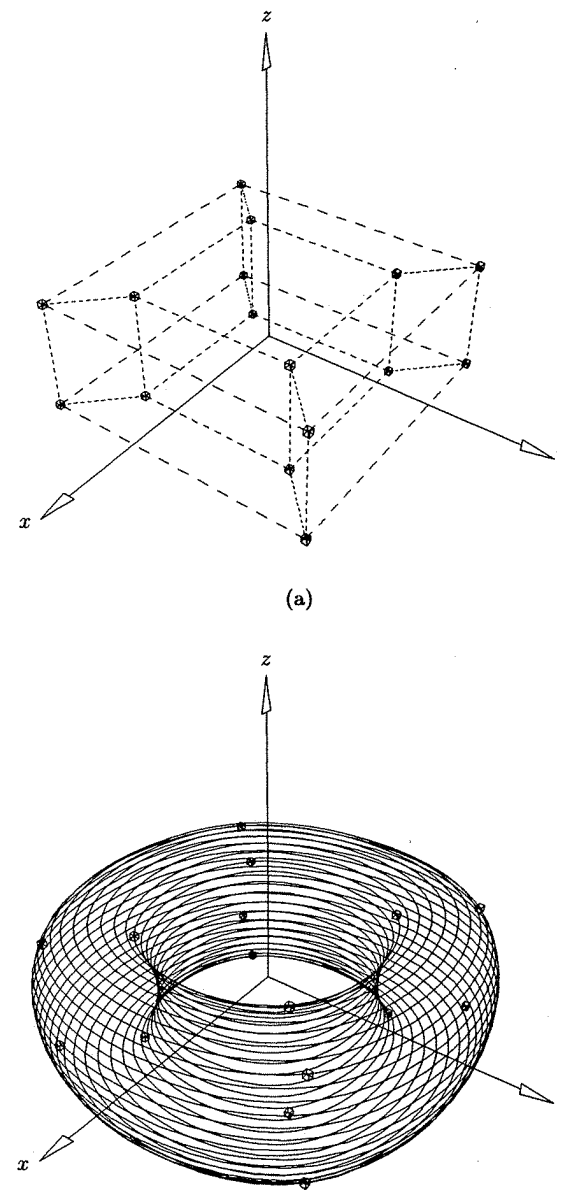
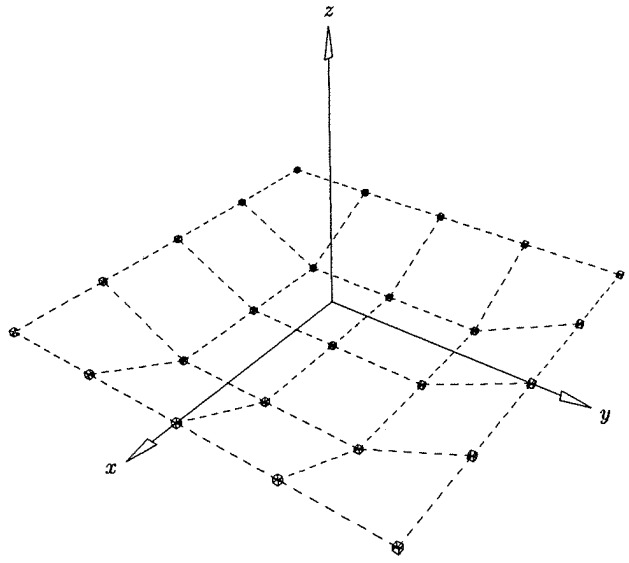
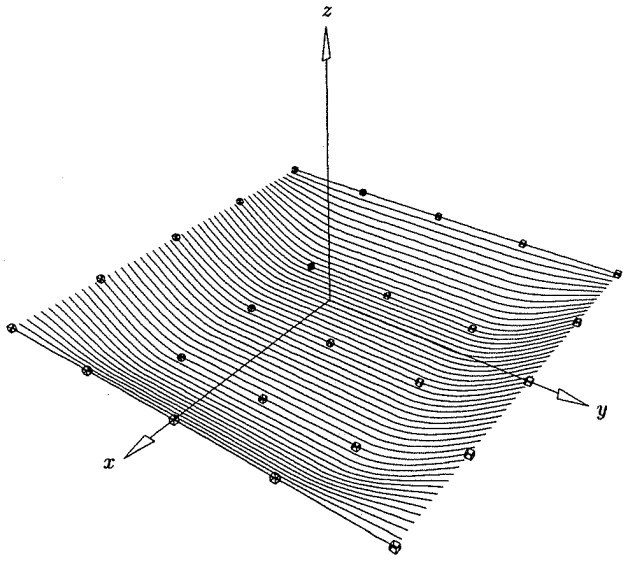


Figure 9.34. A surface interpolation example using a local bicubic surface interpolant. (a) Data points; (b) interpolating surface.

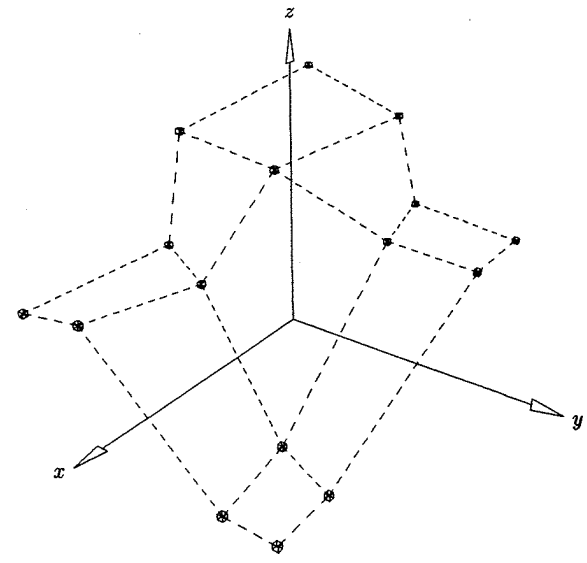


(a)

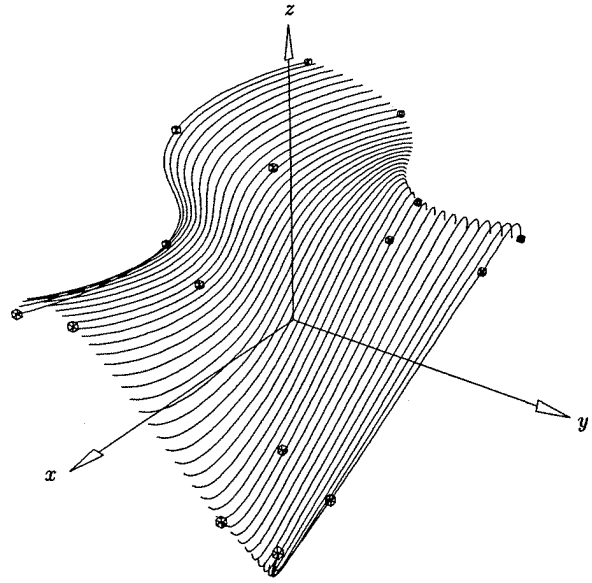


(b)

Figure 9.35. A surface interpolation example using a local bicubic surface interpolant. (a) Data points; (b) an interpolating surface.



(a)



(b)

Figure 9.36. A surface interpolation example using a local bicubic surface interpolant. (a) Data points; (b) an interpolating surface.

c. check the deviation of the curve/surface from the data;

d. if the deviation everywhere is less than E, return; else increase the number of control points and go back to Step 1(b).

2a. start with the maximum or "many" control points, enough that the first approximation satisfies E;

b. fit a curve/surface to the data using a global method;

c. check if the deviation E is satisfied everywhere;

d. if E is no longer satisfied return the previous curve or surface, else reduce the number of control points and go back to Step 2(b).

We return to these two methods in Section 9.4.4, and in particular we give an example of the second type. Clearly, approximation is also more expensive than interpolation. Multiple fits are required, and deviation checking can be expensive.

A central ingredient in both of these methods is that given a *fixed number* of control points, say n, we fit an approximating curve (surface) to the data. There are many ways to do this. For example, a nonlinear optimization problem can be set up, with the control points, parameters $(\bar{u}_k)$, knots, and even the weights as unknowns. The objective function to be minimized must measure the error in some way, e.g., least squares or maximum deviation (e.g., see [Laur93]). In Sections 9.4.1–9.4.3, we present curve and surface fitting methods in which a fixed number of control points are the only unknowns, and they are solved using linear least squares techniques.

9.4.1 LEAST SQUARES CURVE APPROXIMATION

To avoid the nonlinear problem, we set the weights to 1, and precompute values for the parameters and knots. We then set up and solve the (unique) linear least squares problem for the unknown control points. We start with a simple curve fit to point data. Assume that $p \geq 1$, $n \geq p$, and $\mathbf{Q}_0, \ldots, \mathbf{Q}_m$ $(m > n)$ are given. We seek a pth degree nonrational curve

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i \qquad u \in [0,1] \tag{9.61}$$

satisfying that:

- $\mathbf{Q}_0 = \mathbf{C}(0)$ and $\mathbf{Q}_m = \mathbf{C}(1)$;
- the remaining $\mathbf{Q}_k$ are approximated in the least squares sense, i.e.

$$\sum_{k=1}^{m-1} |\,\mathbf{Q}_k - \mathbf{C}(\bar{u}_k)\,|^2 \tag{9.62}$$

is a minimum with respect to the $n+1$ variables, $\mathbf{P}_i$; the $\{\bar{u}_k\}$ are the precomputed parameter values.

We emphasize that the resulting curve generally does not pass precisely through $\mathbf{Q}_k$, and $\mathbf{C}(\bar{u}_k)$ is not the closest point on $\mathbf{C}(u)$ to $\mathbf{Q}_k$. Let

$$\mathbf{R}_k = \mathbf{Q}_k - N_{0,p}(\bar{u}_k)\mathbf{Q}_0 - N_{n,p}(\bar{u}_k)\mathbf{Q}_m \qquad k = 1, \ldots, m-1 \tag{9.63}$$

Then set

$$\begin{aligned} f &= \sum_{k=1}^{m-1} |\,\mathbf{Q}_k - \mathbf{C}(\bar{u}_k)\,|^2 = \sum_{k=1}^{m-1} |\mathbf{R}_k - \sum_{i=1}^{n-1} N_{i,p}(\bar{u}_k)\mathbf{P}_i|^2 \\ &= \sum_{k=1}^{m-1} \left(\mathbf{R}_k - \sum_{i=1}^{n-1} N_{i,p}(\bar{u}_k)\mathbf{P}_i\right) \cdot \left(\mathbf{R}_k - \sum_{i=1}^{n-1} N_{i,p}(\bar{u}_k)\mathbf{P}_i\right) \\ &= \sum_{k=1}^{m-1} \Bigg[\mathbf{R}_k \cdot \mathbf{R}_k - 2\sum_{i=1}^{n-1} N_{i,p}(\bar{u}_k)(\mathbf{R}_k \cdot \mathbf{P}_i) \\ &\qquad + \left(\sum_{i=1}^{n-1} N_{i,p}(\bar{u}_k)\mathbf{P}_i\right) \cdot \left(\sum_{i=1}^{n-1} N_{i,p}(\bar{u}_k)\mathbf{P}_i\right)\Bigg] \end{aligned}$$

f is a scalar-valued function of the $n-1$ variables, $\mathbf{P}_1, \ldots, \mathbf{P}_{n-1}$. Now we apply the standard technique of linear least squares fitting [DeBo78; Vand83; Lanc86]; to minimize f we set the derivatives of f with respect to the $n-1$ points, $\mathbf{P}_\ell$, equal to zero. The ℓth derivative is

$$\frac{\partial f}{\partial \mathbf{P}_\ell} = \sum_{k=1}^{m-1} \left(-2N_{\ell,p}(\bar{u}_k)\mathbf{R}_k + 2N_{\ell,p}(\bar{u}_k)\sum_{i=1}^{n-1} N_{i,p}(\bar{u}_k)\mathbf{P}_i\right)$$

which implies that

$$-\sum_{k=1}^{m-1} N_{\ell,p}(\bar{u}_k)\mathbf{R}_k + \sum_{k=1}^{m-1}\sum_{i=1}^{n-1} N_{\ell,p}(\bar{u}_k)N_{i,p}(\bar{u}_k)\mathbf{P}_i = 0$$

It follows that

$$\sum_{i=1}^{n-1} \left(\sum_{k=1}^{m-1} N_{\ell,p}(\bar{u}_k)N_{i,p}(\bar{u}_k)\right)\mathbf{P}_i = \sum_{k=1}^{m-1} N_{\ell,p}(\bar{u}_k)\mathbf{R}_k \tag{9.64}$$

Equation (9.64) is one linear equation in the unknowns $\mathbf{P}_1, \ldots, \mathbf{P}_{n-1}$. Letting $\ell = 1, \ldots, n-1$ yields the system of $n-1$ equations in $n-1$ unknowns

$$(N^T N)\mathbf{P} = \mathbf{R} \tag{9.65}$$

where N is the $(m-1) \times (n-1)$ matrix of scalars

$$N = \begin{bmatrix} N_{1,p}(\bar{u}_1) & \cdots & N_{n-1,p}(\bar{u}_1) \\ \vdots & \ddots & \vdots \\ N_{1,p}(\bar{u}_{m-1}) & \cdots & N_{n-1,p}(\bar{u}_{m-1}) \end{bmatrix} \tag{9.66}$$

$\mathbf{R}$ is the vector of $n-1$ points

$$\mathbf{R} = \begin{bmatrix} N_{1,p}(\bar{u}_1)\mathbf{R}_1 + \cdots + N_{1,p}(\bar{u}_{m-1})\mathbf{R}_{m-1} \\ \vdots \\ N_{n-1,p}(\bar{u}_1)\mathbf{R}_1 + \cdots + N_{n-1,p}(\bar{u}_{m-1})\mathbf{R}_{m-1} \end{bmatrix} \tag{9.67}$$

and

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_1 \\ \vdots \\ \mathbf{P}_{n-1} \end{bmatrix}$$

Note that Eq. (9.65) is one coefficient matrix, with three right hand sides and three sets of unknowns (x, y, z coordinates). In order to set up Eqs. (9.66) and (9.67), a knot vector $U = \{u_0, \ldots, u_r\}$ and parameters $\{\bar{u}_k\}$ are required. The $\{\bar{u}_k\}$ can be computed using Eq. (9.5). The placement of the knots should reflect the distribution of the $\{\bar{u}_k\}$. If d is a positive real number, denote by $i = \text{int}(d)$ the largest integer such that $i \leq d$. We need a total of $n+p+2$ knots; there are $n-p$ internal knots, and $n-p+1$ internal knot spans. Let

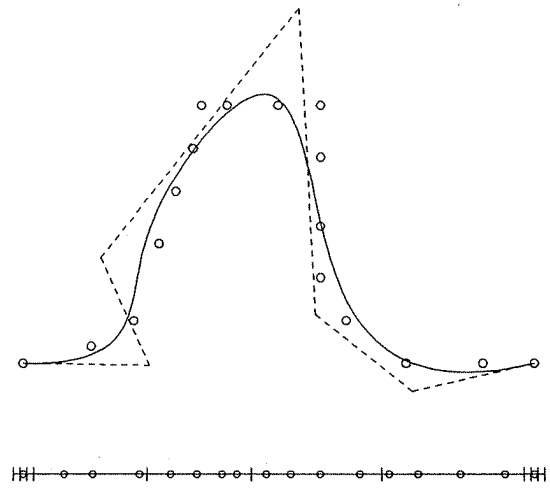
$$d = \frac{m+1}{n-p+1} \tag{9.68}$$

Then define the internal knots by

$$\begin{aligned} i &= \text{int}(jd) \qquad \alpha = jd - i \\ u_{p+j} &= (1-\alpha)\bar{u}_{i-1} + \alpha\bar{u}_i \qquad j = 1, \ldots, n-p \end{aligned} \tag{9.69}$$

Equation (9.69) guarantees that every knot span contains at least one $\bar{u}_k$, and under this condition deBoor [DeBo78] shows that the matrix $(N^T N)$ in Eq. (9.65) is positive definite and well-conditioned. It can be solved by Gaussian elimination without pivoting. Furthermore, $(N^T N)$ has a semibandwidth less than $p+1$ (if $N_{i,j}$ is the element in the ith row, jth column, then $N_{i,j} = 0$ if $\mid i - j \mid > p$). Finally, concerning implementation, note that the matrix N (Eq. [9.66]) can be very large, and most of its elements are zero; in some applications it is not uncommon for $m = 1000$ and $n = 100$. Hence, the implementor may want to develop a storage scheme that stores only the nonzero elements of N. Once $(N^T N)$ and $\mathbf{R}$ (Eq. [9.67]) are computed, N can be deallocated. Similarly, only the nonzero band of $(N^T N)$ should be stored.

Figures 9.37a–9.37c show examples of simple least squares curve fits. In Figure 9.37a seven control points were specified. Figure 9.37b shows a nine control point fit, whereas Figure 9.37c depicts an eleven control point fit. Notice that, in a distance sense, the approximation improves as the number of control points increases. However, as the number of control points approaches the number of data points, undesirable shapes can occur if the data exhibits noise or unwanted wiggles.



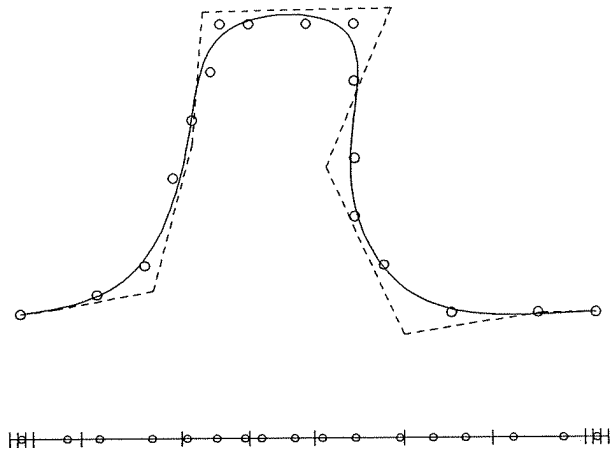
(a)

Figure 9.37. Global curve fitting. (a) Seven control points; (b) nine control points; (c) eleven control points.

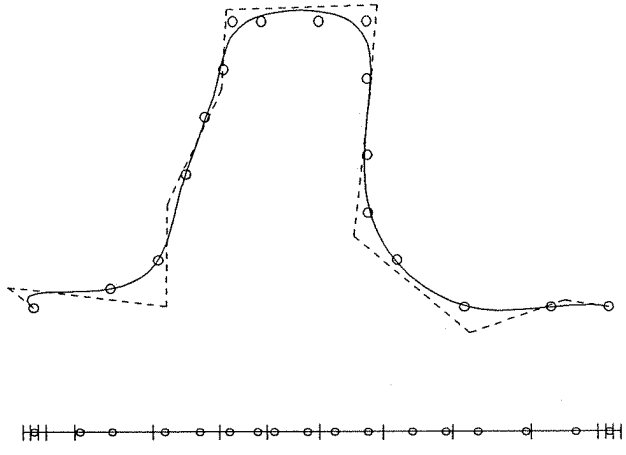
9.4.2 WEIGHTED AND CONSTRAINED LEAST SQUARES CURVE FITTING

We now present a very general, weighted and constrained least squares curve fitting method. It is a NURBS adaptation of an algorithm by Smith et al [Smit74]. Let $\{\mathbf{Q}_i\}$, $i = 0, \ldots, r$, be the points to be approximated. Optionally, the first derivative $\mathbf{D}_i$ at any $\mathbf{Q}_i$ can be specified. Let $\{\mathbf{D}_{i(j)}\}$, $j = 0, \ldots, s$, be the set of derivatives; $-1 \leq s \leq r$, where $s = -1$, means no derivatives are specified. Furthermore, any $\mathbf{Q}_i$ or $\mathbf{D}_{i(j)}$ can be constrained (precisely interpolated). We divide the $\{\mathbf{Q}_i\}$ and $\{\mathbf{D}_{i(j)}\}$ into unconstrained and constrained sets. Denote by $\mathbf{Q}^u_{i(0)}, \ldots, \mathbf{Q}^u_{i(r_u)}$ and $\mathbf{D}^u_{i(0)}, \ldots, \mathbf{D}^u_{i(s_u)}$ the unconstrained items, and by $\mathbf{Q}^c_{i(0)}, \ldots, \mathbf{Q}^c_{i(r_c)}$ and $\mathbf{D}^c_{i(0)}, \ldots, \mathbf{D}^c_{i(s_c)}$ the constrained items (s_u, s_c, or r_c equal to -1 means no data corresponding to this index). Note that $r = r_u + r_c + 1$ and $s = s_u + s_c + 1$. We also allow a positive weight to be assigned to each unconstrained item, hence we have $w^q_{i(0)}, \ldots, w^q_{i(r_u)}$ and $w^d_{i(0)}, \ldots, w^d_{i(s_u)}$ ($w^q_i, w^d_i > 0$). These weights allow additional influence over the "tightness" of the approximation to each data item relative to its neighbors. $w^q_i, w^d_i = 1$ is the default. Increasing a weight increases the tightness of the approximation to that item, whereas decreasing the weight loosens the approximation to that item. Notice that these weights have nothing to do with weights in the NURBS sense.

Now let $m_u = r_u + s_u + 1$ and $m_c = r_c + s_c + 1$. We want to approximate the unconstrained data in the least squares sense and interpolate the constrained



(b)



(c)

Figure 9.37. (*Continued.*)

data with a pth degree nonrational curve, $\mathbf{C}(u)$, with $n+1$ control points. The next algorithm requires that

$$m_c < n \qquad m_c + n < m_u + 1 \tag{9.70}$$

Now let:

$\mathbf{S}_k, k = 0, \ldots, m_u$, be the kth unconstrained data item (a $\mathbf{Q}^u$ or a $\mathbf{D}^u$);

$\mathbf{T}_k, k = 0, \ldots, m_c$, be the kth constrained data item (a $\mathbf{Q}^c$ or a $\mathbf{D}^c$);

$w_k, k = 0, \ldots, m_u$, be the kth weight (a w^q or a w^d).

Define the vectors/matrices:

$\mathbf{S}$ = $[\mathbf{S}_k]$ (a vector with $m_u + 1$ elements);

$\mathbf{T}$ = $[\mathbf{T}_k]$ (a vector with $m_c + 1$ elements);

W = $[w_k]$ (an $(m_u + 1) \times (m_u + 1)$ diagonal matrix, with the w_k on the diagonal);

$\mathbf{P}$ = $[\mathbf{P}_k]$ (the vector of $n + 1$ unknown control points);

N = $[ND_{i,p}(\bar{u}_k)]$, where $ND_{i,p}(\bar{u}_k)$ is the ith basis function or its first derivative, evaluated at $\bar{u}_k$, which is a parameter value corresponding to an unconstrained data item; this is an $(m_u + 1) \times (n + 1)$ matrix of scalars;

M = $[MD_{i,p}(\bar{u}_k)]$, where $MD_{i,p}(\bar{u}_k)$ is the ith basis function or its first derivative, evaluated at $\bar{u}_k$, which is a parameter value corresponding to a constrained data item; this is an $(m_c+1)\times(n+1)$ matrix of scalars.

The $\bar{u}_k$ can be computed using Eq. (9.5), and the knots u_j according to Eq. (9.68) and Eq. (9.69) (replace m by r in Eq. [9.68]).

Determining the curve is basically a constrained minimization problem, involving $n+1$ unknowns (the $\mathbf{P}_i$) and m_c+1 constraints. The standard method of solution is to use Lagrange multipliers (see [Kapl52]). The idea is to introduce m_c+1 additional unknowns, λ_k (the Lagrange multipliers), thereby obtaining a partitioned $(n+m_c+2) \times (n+m_c+2)$ system of equations. From this system a solution is first obtained for the m_c+1 λ_k and subsequently for the $n+1$ $\mathbf{P}_i$. Let

$$\mathbf{A} = [\lambda_k]$$

be the vector of Lagrange multipliers; note that each λ_k is a vector with the same dimensionality as the $\{\mathbf{Q}_i\}$ and $\{\mathbf{D}_{i(j)}\}$.

The unconstrained equations are

$$N\mathbf{P} = \mathbf{S}$$

and the constrained equations are

$$M\mathbf{P} = \mathbf{T}$$

The error in the unconstrained equations is $\mathbf{S} - N\mathbf{P}$, and we want to minimize the sum of the weighted squares of this error, subject to $M\mathbf{P} = \mathbf{T}$. Hence, using Lagrange multipliers we want to minimize the expression

$$(\mathbf{S}^T - \mathbf{P}^T N^T) W (\mathbf{S} - N\mathbf{P}) + \mathbf{A}^T (M\mathbf{P} - \mathbf{T})$$

with respect to the unknowns, $\mathbf{A}$ and $\mathbf{P}$. Differentiating and setting equal to zero yields

$$-2(\mathbf{S}^T W N - \mathbf{P}^T N^T W N) + \mathbf{A}^T M = 0$$

$$M\mathbf{P} - \mathbf{T} = 0$$

Transposing and factoring 2 out of $\mathbf{A}$ yields

$$N^T W N \mathbf{P} + M^T \mathbf{A} = N^T W \mathbf{S} \tag{9.71}$$

$$M\mathbf{P} = \mathbf{T} \tag{9.72}$$

or in partitioned matrix form

$$\begin{bmatrix} N^T W N & M^T \\ M & 0 \end{bmatrix} \begin{bmatrix} \mathbf{P} \\ \mathbf{A} \end{bmatrix} = \begin{bmatrix} N^T W \mathbf{S} \\ \mathbf{T} \end{bmatrix} \tag{9.73}$$

We must solve Eq. (9.73) for $\mathbf{A}$ and $\mathbf{P}$. There is a unique solution if $N^T W N$ and $M(N^T W N)^{-1} M^T$ are both invertible. This is seen by solving Eq. (9.71) for $\mathbf{P}$, that is

$$\mathbf{P} = (N^T W N)^{-1} N^T W \mathbf{S} - (N^T W N)^{-1} M^T \mathbf{A} \tag{9.74}$$

Multiplying by M yields

$$M\mathbf{P} = M(N^T W N)^{-1} N^T W \mathbf{S} - M(N^T W N)^{-1} M^T \mathbf{A}$$

Finally, from Eq. (9.72) we find that

$$M(N^T W N)^{-1} M^T \mathbf{A} = M(N^T W N)^{-1} N^T W \mathbf{S} - \mathbf{T} \tag{9.75}$$

or

$$\mathbf{A} = \left(M(N^T W N)^{-1} M^T\right)^{-1} \left(M(N^T W N)^{-1} N^T W \mathbf{S} - \mathbf{T}\right) \tag{9.76}$$

Hence, we solve first for $\mathbf{A}$ and then for $\mathbf{P}$ by substituting $\mathbf{A}$ into Eq. (9.74).

Algorithm A9.6 implements weighted and constrained least squares curve fitting. The input is:

`Q[r+1]`: an array containing the points to be fit (constrained and unconstrained);

`r`: the upper index of `Q[]`;

`Wq[r+1]`: `Wq[i] > 0` means $\mathbf{Q}_i$ is unconstrained, and `Wq[i]` is the weight w_i^q; `Wq[i] < 0` means $\mathbf{Q}_i$ is to be constrained;

`D[s+1]`: an array containing the derivatives; `s = -1` means no derivatives are specified;

`s`: there are $s+1$ derivatives in `D[]`;

`I[s+1]`: an integer array; `I[j]` gives the index into `Q[]` of the point corresponding to the derivative in `D[j]` (i.e., `D[j]`$= \mathbf{D}_{i(j)}$);

`Wd[s+1]`: `Wd[j] > 0` means `D[j]` is unconstrained, and `Wd[j]` is the weight $w_{i(j)}^d$; `Wd[j] < 0` means `D[j]` is to be constrained;

`n`: a fit with $n+1$ control points;

`p`: a fit with degree p curve.

The knots `U` and control points `P` are output. The algorithm uses `LUDecomposition()` and `ForwardBackward()` to invert the matrix $(N^T W N)$ and to solve for the Lagrange multipliers, $\mathbf{A}$. $(N^T W N)$ has a semibandwidth less than $p+1$, due to the way in which the algorithm constructs N. In general $\mathbf{A}$ can have full bandwidth, but it is usually a small matrix. The appropriate algorithms from Chapter 2 are used to determine knot spans and to compute basis functions and their derivatives. The following local arrays are required:

`ub[r+1]`: the parameters;

`N[mu+1][n+1],M[mc+1][n+1],S[mu+1],T[mc+1],A[mc+1]`: the arrays corresponding to the matrices defined previously (except `S[]`, which is $W\mathbf{S}$);

`W[mu+1]`: the weights; since the matrix W is diagonal, only a vector of length $m_u + 1$ is required;

`funs[2][p+1]`: contains the basis functions, and their derivatives where specified (see Algorithms A2.2 and A2.3).

Furthermore, several local arrays are required for temporary storage of matrices computed in Eqs. (9.71)–(9.76).

```
ALGORITHM A9.6
  WCLeastSquaresCurve(Q,r,Wq,D,s,I,Wd,n,p,U,P)
    { /*  Weighted & constrained least squares curve fit  */
      /*  Input:  Q,r,Wq,D,s,I,Wd,n,p  */
      /*  Output: U,P  */
    ru = -1;     rc = -1;
    for (i=0; i<=r; i++)
      if (Wq[i] > 0.0)   ru = ru+1;
        else             rc = rc+1;
    su = -1;     sc = -1;
    for (j=0; j<=s; j++)
      if (Wd[j] > 0.0)   su = su+1;
        else             sc = sc+1;
    mu = ru+su+1;     mc = rc+sc+1;
    if (mc >= n || mc+n >= mu+1)   return(error);
    Compute and load parameters ū_k into ub[] (Eq.[9.5]);
    Compute and load the knots into U[] (Eqs.[9.68],[9.69]);
          /* Now set up arrays N,W,S,T,M */
    j = 0;   /* current index into I[] */
    mu2 = 0;   mc2 = 0;   /* counters up to mu and mc */
    for (i=0; i<=r; i++)
      {
      span = FindSpan(n,p,ub[i],U);
      dflag = 0;
      if (j <= s)
        if (i == I[j])   dflag = 1;
```

```
  if (dflag == 0)   BasisFuns(span,ub[i],p,U,funs);
    else            DersBasisFuns(span,ub[i],p,1,U,funs);
  if (Wq[i] > 0.0)
    {  /* Unconstrained point */
    W[mu2] = Wq[i];
    Load the mu2th row of N[][] from funs[0][];
    S[mu2] = W[mu2]*Q[i];
    mu2 = mu2+1;
    }
    else
    {  /* Constrained point */
    Load the mc2th row of M[][] from funs[0][];
    T[mc2] = Q[i];
    mc2 = mc2+1;
    }
  if (dflag == 1)
    {  /* Derivative at this point */
    if (Wd[j] > 0.0)
      {  /* Unconstrained derivative */
      W[mu2] = Wd[j];
      Load the mu2th row of N[][] from funs[1][];
      S[mu2] = W[mu2]*D[j];
      mu2 = mu2+1;
      }
      else
      {  /* Constrained derivative */
      Load the mc2th row of M[][] from funs[1][];
      T[mc2] = D[j];
      mc2 = mc2+1;
      }
    j = j+1;
    }
  }  /* End of for-loop i=0,...,r */
Compute the matrices N^TWN and N^TWS;
LUDecomposition(N^TWN,n+1,p);
if (mc < 0)
  {  /* No constraints */
  Use ForwardBackward() to solve for the control points P[].
      Eq.(9.71) reduces to (N^TWN)P = N^TWS.
  return;
  }
Compute the inverse (N^TWN)^-1, using ForwardBackward().
Do matrix operations to get: M(N^TWN)^-1 M^T and
    M(N^TWN)^-1 (N^TWS) - T;
Solve Eq.(9.75) for the Lagrange multipliers, load into A[];
```



```
Then P = (N^TWN)^-1 ((N^TWS) - M^TA), Eq.(9.74);
}
```

Figures 9.38–9.43 show several examples of weighted and constrained curve fitting. Figure 9.38 shows an unconstrained fit with all weights set to 1. In Figure 9.39 the weights w_4^q, w_5^q, w_6^q, and w_7^q were increased to 100 each, resulting in a curve pull towards the data points $\mathbf{Q}_4, \ldots, \mathbf{Q}_7$. Note that, despite the misleading graphics, the curve does not pass through the endpoints. Figure 9.40 shows a constrained fit with point (marked with squares) as well as derivative (arrowed vectors) constraints. The curve passes through the square-marked points and is tangential to the specified vectors at these points. In Figure 9.41 two additional weights were introduced, $w_5^q = 100$ and $w_{15}^q = 100$. Figure 9.42 shows the same curve as in Figure 9.40, except that two new inner derivatives were used. Similarly, Figure 9.43 is the same as Figure 9.42 except that the weights $w_4^q = 50$ and $w_{16}^q = 20$ were added.

9.4.3 LEAST SQUARES SURFACE APPROXIMATION

We turn now to surface approximation with a fixed number of control points. Let $\{\mathbf{Q}_{k,\ell}\}$, $k = 0, \ldots, r$ and $\ell = 0, \ldots, s$, be the $(r+1) \times (s+1)$ set of points to be approximated by a (p, q)th degree nonrational surface, with $(n+1) \times (m+1)$ control points. Although it is possible to set up and solve a general least squares surface fitting problem, linear or nonlinear, and with or without weights and constraints, the task is much more complex than for curves. We present a surface approximation scheme here which builds upon our least squares curve scheme, is

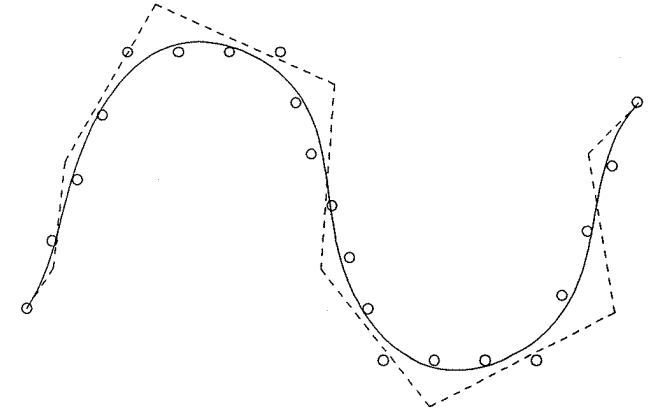


Figure 9.38. Unconstrained fit with all weights equal to one; note that the curve does not pass through the endpoints.

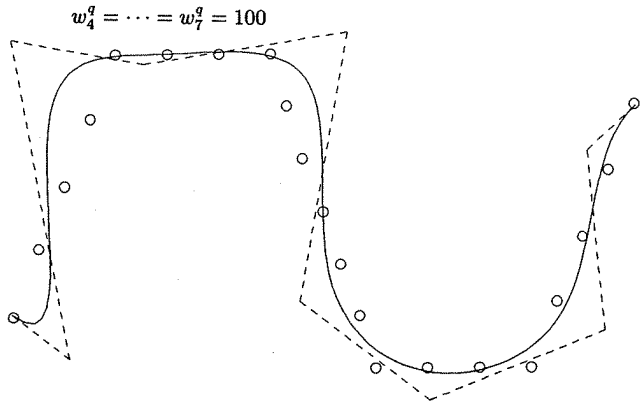


Figure 9.39. Unconstrained fit with weights $w_4^q = w_5^q = w_6^q = w_7^q = 100$; the curve does not pass through the endpoints.

very simple, and is quite adequate for most applications. As was done for global surface interpolation, we simply fit curves across the data in one direction, then fit curves through the resulting control points across the other direction.

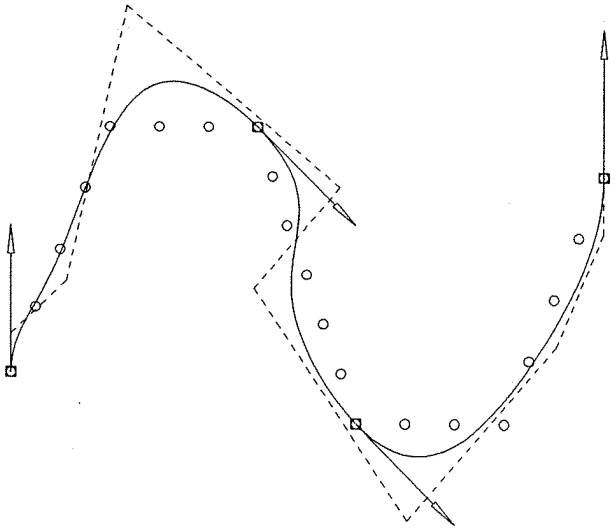


Figure 9.40. A constrained curve fit; point constraints are marked by squares, tangent constraints are shown by arrows, and all weights are one.

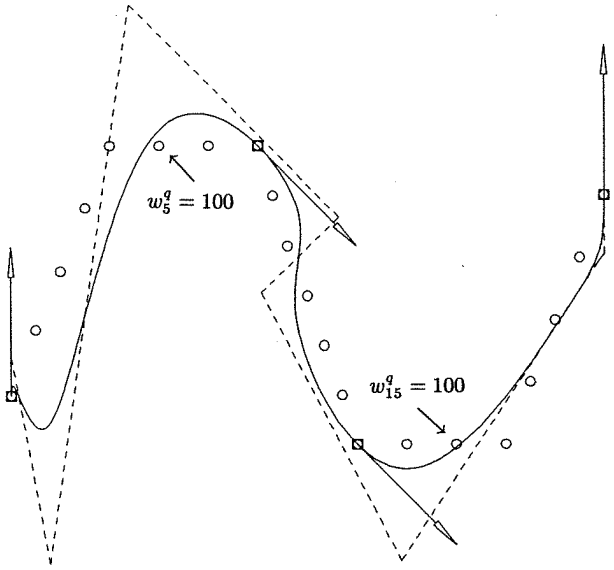


Figure 9.41. The curve from Figure 9.40, with weights $w_5^q = w_{15}^q = 100$.

Algorithm `A9.7` interpolates the corner points $\mathbf{Q}_{0,0}$, $\mathbf{Q}_{r,0}$, $\mathbf{Q}_{0,s}$, and $\mathbf{Q}_{r,s}$ precisely, and approximates the remaining $\{\mathbf{Q}_{k,l}\}$. It uses repeated least squares curve fits given by Eqs. (9.63) and (9.65)–(9.67). It fits the $s+1$ rows of data

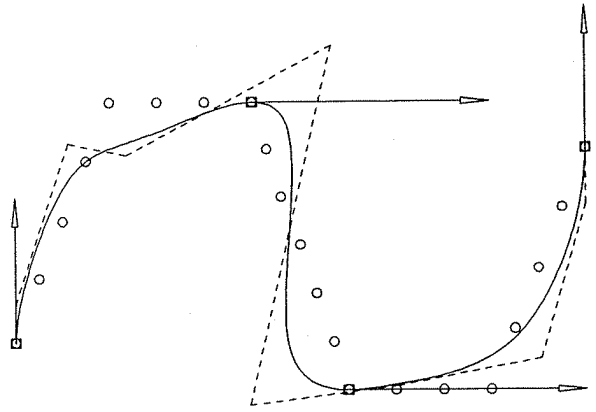


Figure 9.42. The curve from Figure 9.40, but using two new tangent constraints.

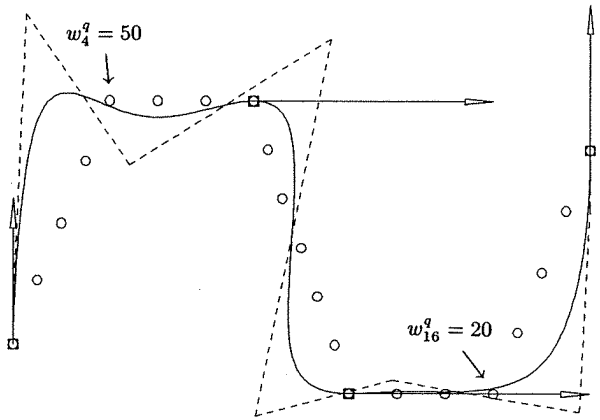


Figure 9.43. The curve from Figure 9.42, but with weights $w_4^q = 50$ and $w_{16}^q = 20$.

first, storing the resulting control points in the local array, `Temp[n+1][s+1]`. Then it fits across these control points to produce the $(n+1) \times (m+1)$ surface control points. Notice that the matrices N and $N^T N$ only have to be computed once for each direction, and the LU decomposition of $N^T N$ is only done once for each direction. We require local arrays `Nu[r-1][n-1]` and `NTNu[n-1][n-1]` for the u direction fits, and `Nv[s-1][m-1]` and `NTNv[m-1][m-1]` for the v direction fits. The vector $\mathbf{R}$ of Eq. (9.68) is stored in `Ru[n-1]` and `Rv[m-1]`.

```
ALGORITHM A9.7
  GlobalSurfApproxFixednm(r,s,Q,p,q,n,m,U,V,P)
    { /*  Global surface approx with fixed num of ctrl pts  */
      /*  Input:  r,s,Q,p,q,n,m  */
      /*  Output: U,V,P  */
    SurfMeshParams(r,s,Q,ub,vb);
    Compute knots U by Eqs.(9.68),(9.69);
    Compute knots V by Eqs.(9.68),(9.69);
    Compute Nu[][] and NTNu[][] using Eq.(9.66);
    LUDecomposition(NTNu,n-1,p);
    for (j=0; j<=s; j++)
      { /* u direction fits */
      Temp[0][j] = Q0,j;     Temp[n][j] = Qr,j;
      Compute and load Ru[] (Eqs.[9.63],[9.67]);
      Call ForwardBackward() to get the control points
          Temp[1][j],...,Temp[n-1][j];
      }
    Compute Nv[][] and NTNv[][] using Eq.(9.66);
    LUDecomposition(NTNv,m-1,q);
```



```
    for (i=0; i<=n; i++)
      { /* v direction fits */
      P[i][0] = Temp[i][0];     P[i][m] = Temp[i][s];
      Compute and load Rv[] (Eqs.[9.63],[9.67]);
      Call ForwardBackward() to get the control points
          P[i][1],...,P[i][m-1];
      }
    }
```

Figures 9.44–9.47 illustrate surface fitting. Figure 9.44 shows the original data joined by straight lines. Figure 9.45a depicts a degree $(2,3)$ least squares surface fit using a (5×6) control net. The error of approximation is shown in Figure 9.45b. Figure 9.46a shows an approximation with an (8×8) net. Its quality of approximation is much better, as seen in Figure 9.46b. For comparison, Figure 9.47 shows a degree $(2,3)$ global interpolation (using chord length parameters) to the data in Figure 9.44.

Of course, one can fit the $r+1$ columns first, and then fit across the resulting $m+1$ rows of control points. In general the results are not the same. We know of no criteria to decide in advance which approach will yield the "best" fit for a particular set of data. Clearly, this technique can easily be extended to accomodate point and derivative constraints and weighting on the boundaries. Interior constraints are more difficult. Suppose $\mathbf{Q}_{k_0,\ell_0}$, $0 < k_0 < r$ and $0 < \ell_0 < s$, is to be constrained. The point is constrained during the u direction

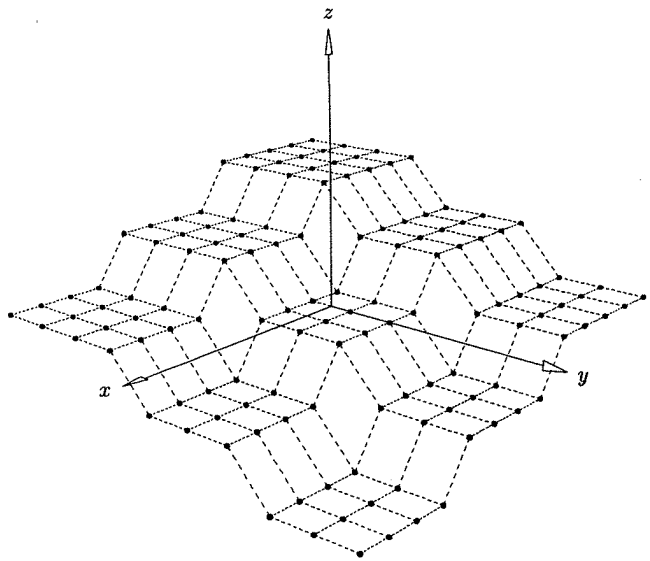
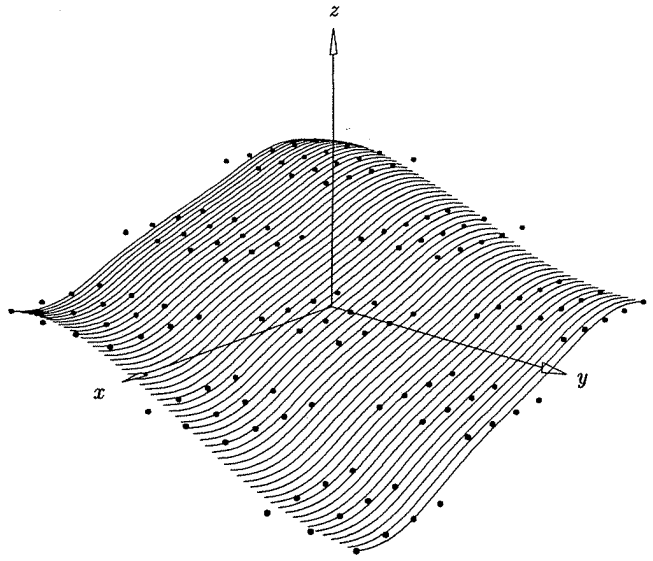


Figure 9.44. Data set for surface approximation.



(a)

Figure 9.45. A least squares surface fit with a degree $(2,3)$ surface using a (5×6) control net. (a) Surface approximant; (b) error surface (error measured at data points).

curve fit, as in Algorithm `A9.6`. Then, if $\bar{u}_{k_0} \in [u_i, u_{i+1})$, the points $\texttt{Temp}[i-p][\ell_0], \ldots, \texttt{Temp}[i][\ell_0]$ must be constrained in the v direction fitting.

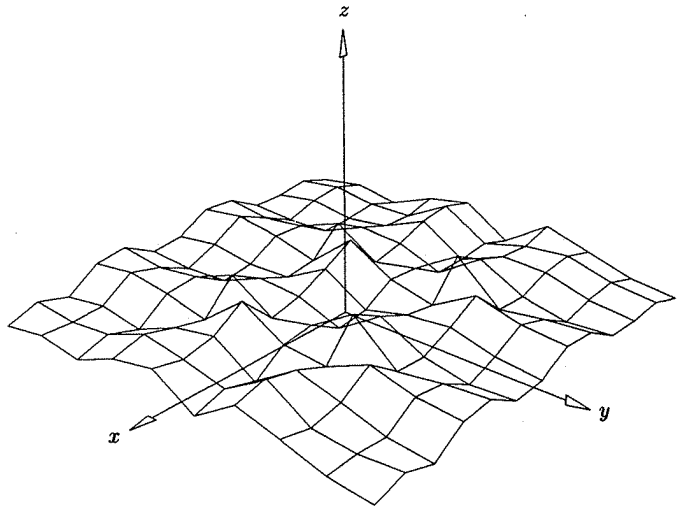
9.4.4 APPROXIMATION TO WITHIN A SPECIFIED ACCURACY

We now return to the problem of approximating data to within some user-specified error bound, E. As indicated at the beginning of this section, iterative methods either start with few control points, fit, check deviation, and add control points if necessary (Type 1); or they start with many control points, fit, check deviation, and discard control points if possible (Type 2). The least squares based methods given earlier are appropriate for the fitting step. The deviation checking step usually measures maximum distance, either as

$$\max_{0\le k\le m} |\, \mathbf{Q}_k - \mathbf{C}(\bar{u}_k)\,| \quad \text{or} \quad \max_{\substack{0\le k\le r\\ 0\le \ell\le s}} |\, \mathbf{Q}_{k,\ell} - \mathbf{S}(\bar{u}_k, \bar{v}_\ell)\,| \tag{9.77}$$

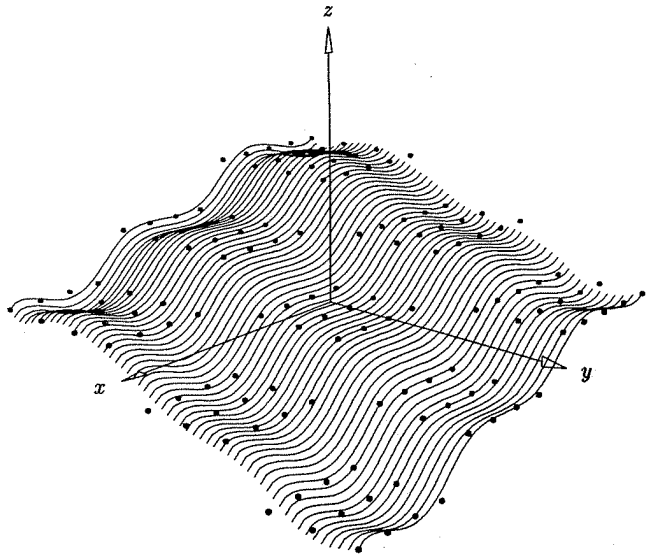
or as

$$\max_{0\le k\le m} \left(\min_{0\le u\le 1} |\, \mathbf{Q}_k - \mathbf{C}(u)\,| \right) \quad \text{or} \quad \max_{\substack{0\le k\le r\\ 0\le \ell\le s}} \left(\min_{\substack{0\le u\le 1\\ 0\le v\le 1}} |\, \mathbf{Q}_{k,\ell} - \mathbf{S}(u,v)\,| \right) \tag{9.78}$$



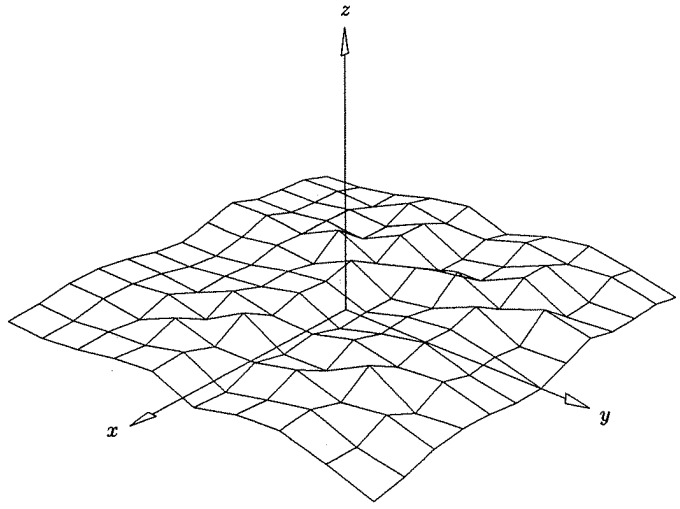
(b)

Figure 9.45. (*Continued.*)



(a)

Figure 9.46. A least squares surface fit with a degree $(2,3)$ surface using an (8×8) control net. (a) Surface approximant; (b) error surface (error measured at data points).



(b)

Figure 9.46. (*Continued.*)

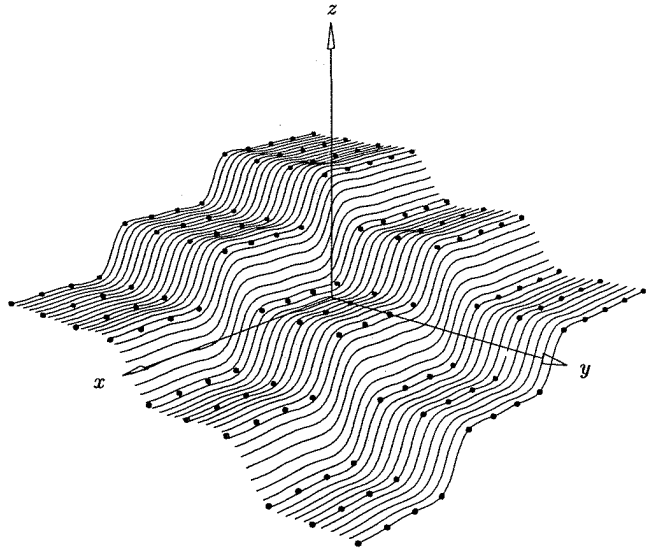


Figure 9.47. The surface interpolating points shown in Figure 9.44.

We call Eq. (9.78) the *max norm deviation.* It uses Eqs. (6.4)–(6.9) to compute the minimum distance from each $\mathbf{Q}_k$ ($\mathbf{Q}_{k,\ell}$) to $\mathbf{C}(u)$ ($\mathbf{S}(u,v)$). The $\bar{u}_k$ ($\bar{u}_k, \bar{v}_\ell$) serve as good start points for the Newton iterations. Obviously, Eq. (9.78) is more expensive to apply than Eq. (9.77). However, it is usually what a user wants measured, and it generally leads to curves and surfaces with fewer control points, since

$$\min_{0 \le u \le 1} |\, \mathbf{Q}_k - \mathbf{C}(u) \,| \;\le\; |\, \mathbf{Q}_k - \mathbf{C}(\bar{u}_k) \,| \tag{9.79}$$

We emphasize that both Type 1 and Type 2 methods can fail to converge, and this eventuality must be dealt with in a software implementation.

A Type 1 curve approximation method can proceed as follows. Start with $p+1$ control points (the minimum). Fit a curve using Eqs. (9.63) and (9.65)–(9.67), or Algorithm `A9.6` if interior constraints are desired. After each fit, use Eq. (9.78) to check if the curve deviation is less than E. A record is maintained for each knot span, indicating whether or not it has converged; a span $[u_j, u_{j+1})$ has converged if Eq. (9.78) is satisfied for all k such that $\bar{u}_k \in [u_j, u_{j+1})$. After each fit and subsequent deviation check, a knot is added at the midpoint of each nonconvergent span, thus adding control points. This algorithm must deal with the case in which a span can become empty as knots are added (i.e., the span contains no $\bar{u}_k$), thus producing a singular system of equations in the fitting step.

A Type 1 surface approximation method can be implemented in a similar fashion, with an efficiency improvement. During the deviation check, determine the row ℓ_0 and column k_0 which have the greatest numbers of points with deviations greater than E. Then extract isoparametric curves $\mathbf{C}_{\bar{u}_{k_0}}(v)$ and $\mathbf{C}_{\bar{v}_{\ell_0}}(u)$, and independently approximate them to within E. This yields new (increased) knot vectors, U and V, to use for the next surface fit. Using this method, usually only a few surface fits and subsequent deviation checks are required.

We now develop a Type 2 curve approximation method, which combines least squares fitting with the knot removal techniques of Chapter 5. First we need a few more knot removal tools. Let u_r be an interior knot of a pth degree nonrational curve, $\mathbf{C}(u)$, $u_r \neq u_{r+1}$, and denote the multiplicity of u_r by s, i.e., $u_{r-s+j} = u_r$ for $j = 1, \ldots, s$. Let $\hat{\mathbf{C}}(u)$ denote the curve obtained by removing one occurrence of u_r. Recall that the new control points, $\mathbf{P}_i^1$ and $\mathbf{P}_j^1$, are computed from the left and the right, respectively, by Eq. (5.28). After removal, $\hat{\mathbf{C}}(u)$ differs from $\mathbf{C}(u)$; Tiller [Till92] derives error bounds for two cases:

- assume $(p+s) \mod 2 = 0$; set $k = (p+s)/2$ and

$$B_r = |\, \mathbf{P}_{r-k} - \alpha_{r-k}\, \mathbf{P}_{r-k+1}^1 - (1-\alpha_{r-k})\, \mathbf{P}_{r-k-1}^1 \,| \tag{9.80}$$

where $$\alpha_{r-k} = \frac{u_r - u_{r-k}}{u_{r-k+p+1} - u_{r-k}}$$

Then $$|\mathbf{C}(u) - \hat{\mathbf{C}}(u)| \;\le\; N_{r-k,p}(u) B_r \qquad \text{for } u \in [0,1] \tag{9.81}$$

- assume $(p+s) \mod 2 = 1$; set $k = (p+s+1)/2$ and

$$B_r = |\mathbf{P}^1_{r-k} - \mathbf{P}^1_{r-k+1}| \tag{9.82}$$

Then $$|\mathbf{C}(u) - \hat{\mathbf{C}}(u)| \leq (1-\alpha_{r-k+1}) N_{r-k+1,p}(u) B_r \quad \text{for } u \in [0,1] \tag{9.83}$$

with $$\alpha_{r-k+1} = \frac{u_r - u_{r-k+1}}{u_{r-k+p+2} - u_{r-k+1}}$$

The B_r are just the distances computed in knot removal Algorithm A5.8. Algorithm A9.8 is a modification of Algorithm A5.8 to compute just the B_r.

```
ALGORITHM A9.8
GetRemovalBndCurve(n,p,U,P,u,r,s,Br)
  {  /*  Get knot removal error bound (nonrational)  */
     /*  Input:  n,p,U,P,u,r,s  */
     /*  Output: Br  */
  ord = p+1;
  last = r-s;
  first = r-p;
  off = first-1; /* difference in index between temp and P */
  temp[0] = P[off];   temp[last+1-off] = P[last+1];
  i = first;      j=last;
  ii = 1;      jj = last-off;
  while (j-i > 0)
    { /* Compute new control points for one removal step */
    alfi = (u-U[i])/(U[i+ord]-U[i]);
    alfj = (u-U[j])/(U[j+ord]-U[j]);
    temp[ii] = (P[i]-(1.0-alfi)*temp[ii-1])/alfi;
    temp[jj] = (P[j]-alfj*temp[jj+1])/(1.0-alfj);
    i = i+1;     ii = ii+1;
    j = j-1;     jj = jj-1;
    } /* End of while-loop */
  if (j-i < 0)  /* now get bound */
    {   /* Eq.(9.82) */
    Br = Distance3D(temp[ii-1],temp[jj+1]);
    }
    else
    {   /* Eq.(9.80) */
    alfi = (u-U[i])/(U[i+ord]-U[i]);
    Br = Distance3D(P[i],alfi*temp[ii+1]+(1.0-alfi)
                                          *temp[ii-1]);
    }
  }
```


Now let $\{\mathbf{Q}_k\}$ be a set of points and $\{\bar{u}_k\}$ the associated set of parameters. Denote by $\{e_k\}$ a set of errors associated with the points, and let E be a maximum error bound. Given a curve $\mathbf{C}(u)$ with the property that the max norm deviation of each $\mathbf{Q}_k$ from $\mathbf{C}(u)$ is less than or equal to e_k, Algorithm A9.9 removes (roughly) as many knots as possible from $\mathbf{C}(u)$ while maintaining $e_k \leq E$ for all k. It also updates the $\{e_k\}$, i.e., it accumulates the error. Output is the new curve, represented by $\hat{n}$, $\hat{U}$, and $\hat{\mathbf{P}}_i$. The algorithm maintains two lists, the knot removal error, B_r, for each *distinct* interior knot; and for each basis function, $N_{i,p}(u)$, the range of indices representing the parameters $\bar{u}_k$ which fall within the domain of that function. Clearly, when a knot is removed these lists must be updated; however, the effect is local. The bound, B_r, of only a few neighboring knots changes, and only a few neighboring basis functions (and their domains) change. Suppose we remove u_r, which has multiplicity s before removal. After removal there is one less basis function, and in the *new* numbering scheme the range of $\bar{u}_k$ indices for only the basis functions

$$N_{i,p}(u) \qquad i = r-p-1, \ldots, r-s \tag{9.84}$$

must be recomputed. B_r values change for all knots whose old span index lies in the range

$$\max\{r-p, p+1\}, \ldots, \min\{r+p-s+1, n\} \tag{9.85}$$

Example

Ex9.3 See Figure 9.48: Let the cubic curve be defined by $\mathbf{P}_0, \ldots, \mathbf{P}_{11}$ and $U = \{0,0,0,0,1,2,3,4,4,5,6,7,8,8,8,8\}$. Suppose we remove one occurrence of the knot $u_8 = 4$ ($r = 8$, $s = 2$). By Eq. (9.85) new B_r values must be computed for knots 2, 3, 4, 5, 6. By Eq. (9.84) new $\bar{u}_k$ ranges of indices must be computed for $N_{4,3}$, $N_{5,3}$, and $N_{6,3}$.

Ex9.4 See Figure 9.49: Let the cubic curve be defined by $\mathbf{P}_0, \ldots, \mathbf{P}_{10}$ and $U = \{0,0,0,0,1,2,3,4,5,6,7,8,8,8,8\}$. Suppose we remove the knot $u_7 = 4$ ($r = 7$, $s = 1$). By Eq. (9.85) new B_r values must be computed for knots 1,2,3,5,6,7 (all remaining interior knots). By Eq. (9.84) new $\bar{u}_k$ ranges of indices must be computed for $N_{3,3}$, $N_{4,3}$, $N_{5,3}$, and $N_{6,3}$.

```
ALGORITHM A9.9
RemoveKnotsBoundCurve(n,p,U,P,ub,ek,E,nh,Uh,Ph)
  {  /*  Remove knots from curve, bounded  */
     /*  Input:  n,p,U,P,ub,ek,E  */
     /*  Output: ek,nh,Uh,Ph  */
  Inf = ∞;   /* Big number */
  Get the values Br for all distinct interior knots (A9.8).
  For each basis function, get range of parameter indices.
  while (1)
    {
    Find knot with the smallest Br bound.  Set r and s.
```

```
    if (Br == Inf)  break;   /* Finished */
    Using Eqs.(9.81),(9.83), and Algorithm A2.4 to compute
        NewError[k], form temp[k] = ek[k]+NewError[k] at
        all ub[k] values falling within the relevant domain.
    If knot is removable   /* All temp[k] <= E */
      {
      Update ek[] :  ek[k] = temp[k] for relevant range.
      Call routine similar to A5.8 to remove knot
          (remove without tolerance check).
      If no more internal knots, break.   /* Finished */
      Using Eq.(9.84), compute new index ranges for
          affected basis functions.
      Using Eq.(9.85), compute new error bounds for the
          relevant knots.
      }
      else
      {
      Set this Br to Inf.
      }
    }
}
```

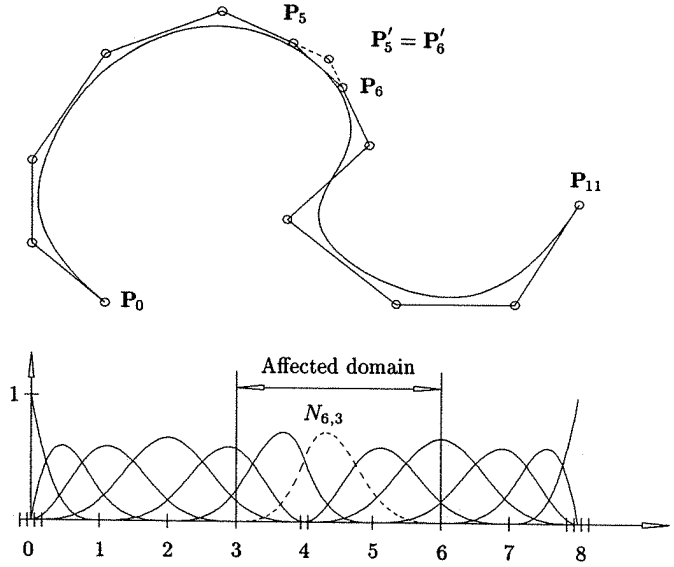


Figure 9.48. Removal of the knot $u_8 = 4$ one time.

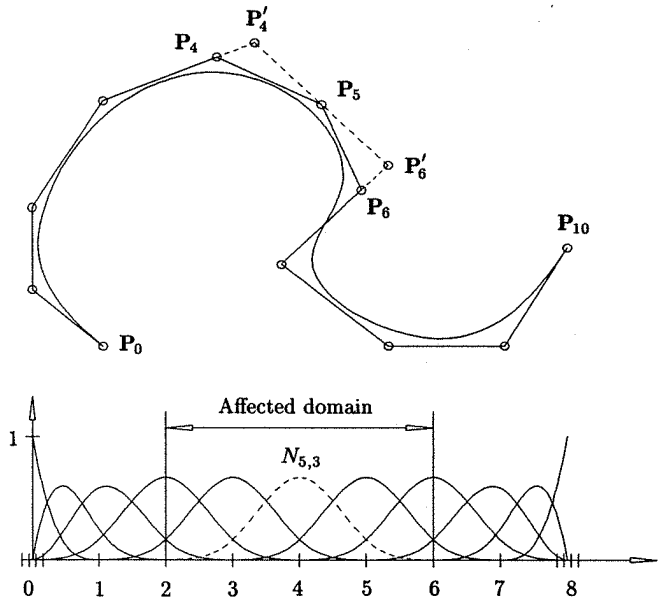


Figure 9.49. Removal of the knot $u_7 = 4$ one time.

We can now put together a Type 2 curve approximation method.

```
ALGORITHM A9.10
GlobalCurveApproxErrBnd(m,Q,p,E,n,U,P)
  { /*  Global curve approximation to within bound E  */
    /*  Input:  m,Q,p,E  */
    /*  Output: n,U,P  */
  Compute ū_k and load into ub[] (Eq.[9.5]).
  Set U and P to be the degree 1 curve interpolating the Q_k:
      U = {0,0,ū_1,...,ū_{m-1},1,1} and P_k = Q_k.
  Initialize the e_k:  ek[0],...,ek[m] = 0.
  n = m;
  for (deg=1; deg<=p; deg++)
    {
    RemoveKnotsBoundCurve(n,p,U,P,ub,ek,E,nh,Uh,Ph);
    if (deg == p)   break;
    Let U be the knot vector obtained by degree elevating Uh
        from deg to deg+1 (increase all multiplicities by 1).
        Reset n (Eq.[5.32]).
    Fit a least squares curve to the Q_k, using n, ub, degree
        deg+1, and knot vector U; this yields new ctrl pts P.
```

```
    Using Eqs.(6.4),(6.5), project all Q_k to current curve
        to get R_k = C(u_k).  Then update ek[] and ub[]:
        e_k =| Q_k, R_k | and ū_k = u_k.
    } /* End of for-loop */
  if (n == nh)   return(n,U,P);
  Set U = Uh and n = nh, and fit one final time with degree p
      and knots U.
  Project the Q_k and update ek[] and ub[].
  RemoveKnotsBoundCurve(n,p,U,P,ub,ek,E,nh,Uh,Ph);
  return(nh,Uh,Ph);
  }
```

The advantages of starting with a linear curve and working up to degree p are three-fold:

- geometric characteristics (e.g., cusps and discontinuities in curvature) inherent in the data tend to be captured at the appropriate stage; if the data is known to be smooth, then Algorithm A9.10 can be started at degree 2 (quadratic interpolation before entering the for-loop);
- the evolving curve tends to "settle" into a natural parameterization;
- a general rule when globally fitting large numbers of points is that the higher the degree, and the more knots (interpolation is the limit), the worse the wiggle; Algorithm 9.10 reduces the number of knot spans as it increases the degree, thereby tending to decrease wiggle in the final curve.

Figure 9.50a shows a cubic curve fitting example using Algorithm A9.10. The tolerance is $E = {}^{7}/_{100}$. Figure 9.50b shows the parameterization of this curve, along with the control polygon. Note the mix of double and single knots, as apparent by the alternating touching and nontouching polygon legs. Figure 9.51 shows another example of cubic curve approximation, with $E = {}^{8}/_{100}$.

The max norm error bound E is guaranteed in the knot removal steps of Algorithm A9.10. The error resulting from the least squares fit is guaranteed to be less than or equal to the error in the previous (knot removed curve) *in the least squares sense.* In practice it also seems to always be better in the max norm sense. In any case, the max norm error can be easily monitored during the point projection step when the e_k and $\bar{u}_k$ are updated.

Constraints can be easily incorporated into the algorithm by using Algorithm A9.6 for the fitting, and by initializing the relevant e_k to ∞ and not updating it (or its $\bar{u}_k$). Lyche [Lych87, 88] used similar knot removal techniques for curve and surface approximation and data reduction.

An analogous Type 2 surface approximation method can be developed. Let

$$\mathbf{S}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)\mathbf{P}_{i,j} \qquad (u,v) \in [0,1]\times[0,1]$$

be a (p,q)th degree B-spline surface. Assume u_r is a u knot of multiplicity s,

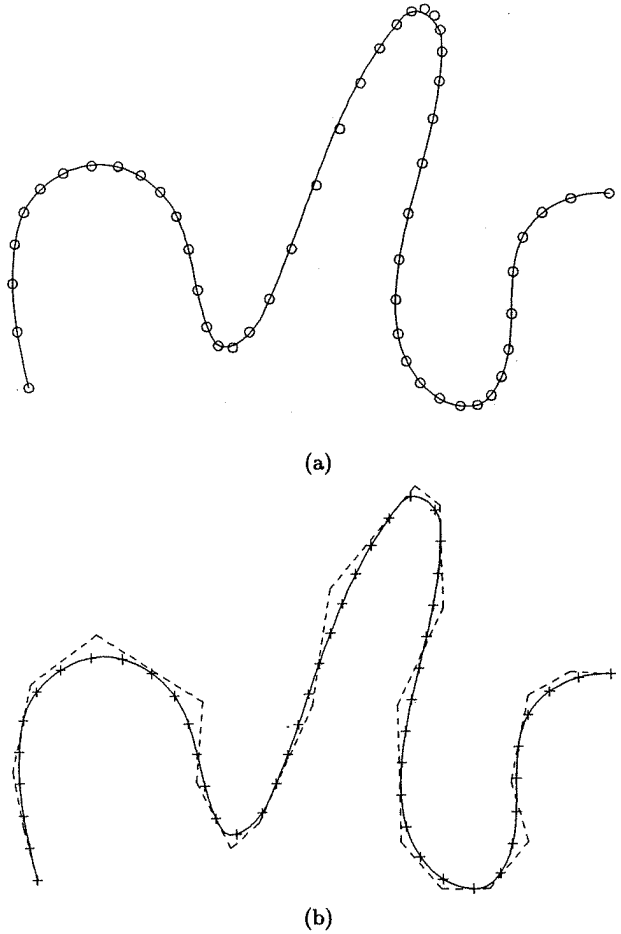


Figure 9.50. A cubic curve fit using algorithm A9.10.

and let $\hat{\mathbf{S}}(u,v)$ denote the surface obtained by removing one occurrence of u_r. Considering the tensor product structure of $\mathbf{S}(u,v)$, and using the techniques of Tiller [Till92], it can be shown that:

- if $(p+s) \mod 2 = 0$, set $k = (p+s)/2$ and

$$B_r^j = |\,\mathbf{P}_{r-k,j} - \alpha_{r-k}\mathbf{P}^1_{r-k+1,j} - (1-\alpha_{r-k})\mathbf{P}^1_{r-k-1,j}\,| \qquad j = 0,\ldots,m \qquad (9.86)$$

where $$\alpha_{r-k} = \frac{u_r - u_{r-k}}{u_{r-k+p+1} - u_{r-k}}$$

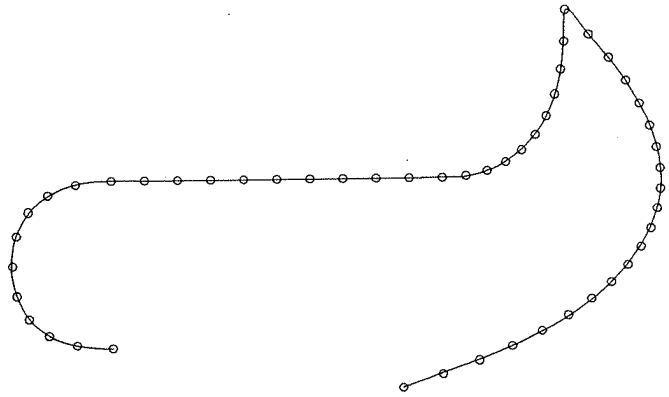


Figure 9.51. A cubic curve fit using algorithm A9.10 and $E = {}^{8}/_{100}$.

Then $$|\mathbf{S}(u,v) - \hat{\mathbf{S}}(u,v)| \leq N_{r-k,p}(u)\sum_{j=0}^{m} N_{j,q}(v)B_r^j$$
$$\text{for } (u,v) \in [0,1] \times [0,1] \qquad (9.87)$$

- if $(p+s) \mod 2 = 1$, set $k = {}^{(p+s+1)}/_{2}$ and

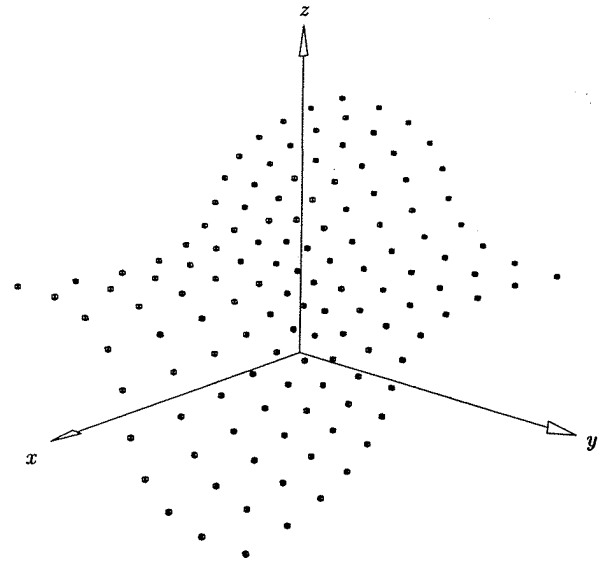
$$B_r^j = |\mathbf{P}_{r-k,j}^1 - \mathbf{P}_{r-k+1,j}^1| \qquad j = 0, \ldots, m \qquad (9.88)$$

Then $$|\mathbf{S}(u,v) - \hat{\mathbf{S}}(u,v)| \leq (1-\alpha_{r-k+1})N_{r-k+1,p}(u)\sum_{j=0}^{m} N_{j,q}(v)B_r^j$$
$$\text{for } (u,v) \in [0,1] \times [0,1] \qquad (9.89)$$

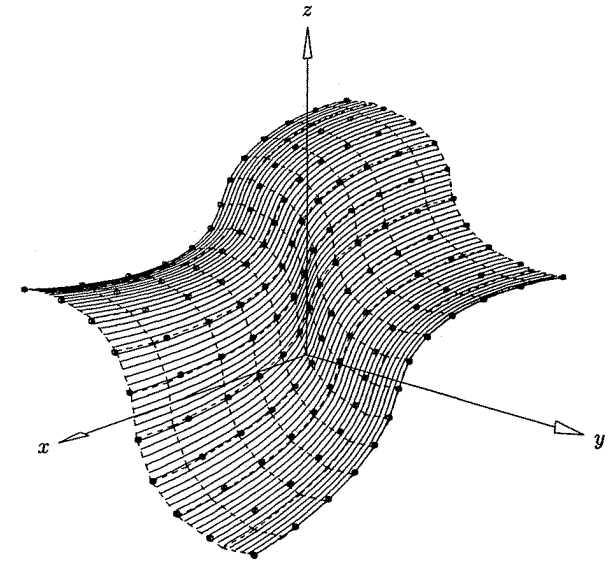
with $$\alpha_{r-k+1} = \frac{u_r - u_{r-k+1}}{u_{r-k+p+2} - u_{r-k+1}}$$

Analogous equations hold for v direction knot removal. Using these equations, one can develop a knot removal stategy for surfaces similar to Algorithms A9.8 and A9.9. This, combined with the least squares based surface fitting algorithm A9.7, provides a method for approximating a set of points $\{\mathbf{Q}_{k,\ell}\}$ to within an E tolerance.

Figures 9.52a–9.52e show several examples of surface approximation. The data set to be approximated is shown in Figure 9.52a. Figure 9.52b shows a degree $(2,2)$ surface that is an interpolant obtained by using $E = 0$. Increasing E to ${}^1/_2$ yields the surface in Figure 9.52c. Further increasing E to 1 produces the surface of Figure 9.52d. Finally, a degree $(3,3)$ approximation using $E = 1$ is shown in Figure 9.52e.

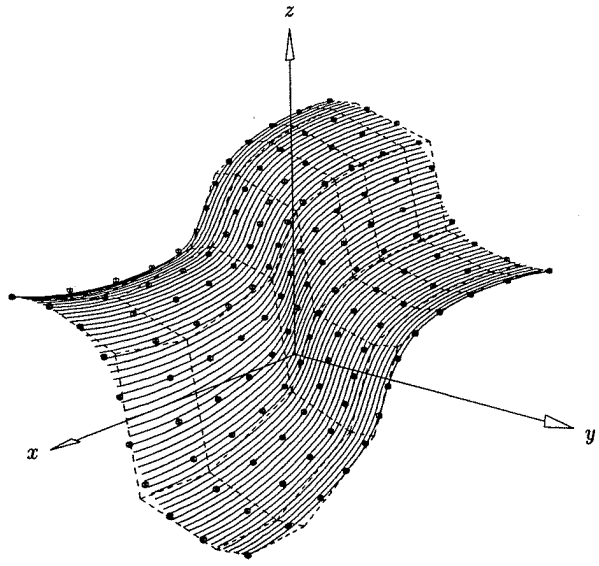


(a)

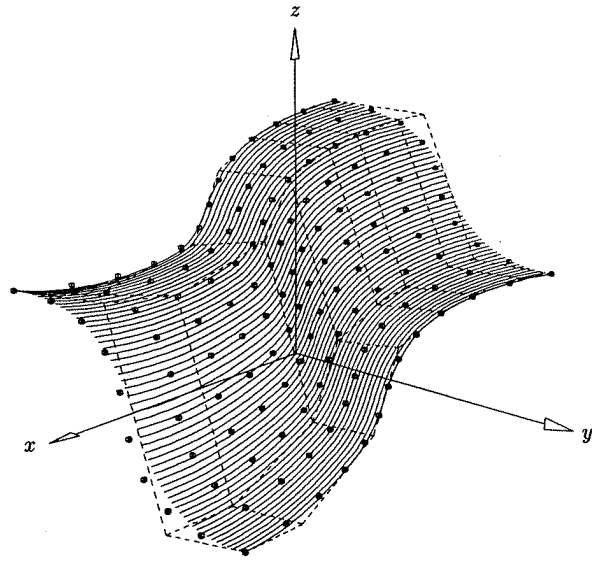


(b)

Figure 9.52. Various surface fitting examples using the extension of algorithm A9.10 to surfaces.

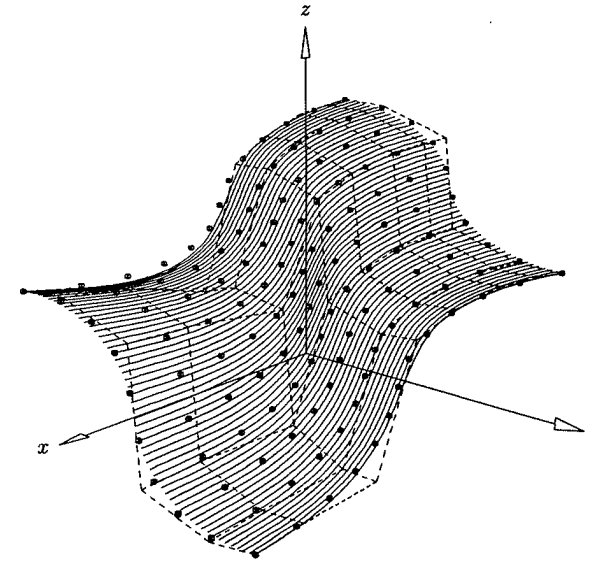


(c)



(d)

Figure 9.52. (*Continued.*)



(e)

Figure 9.52. (*Continued.*)

Algorithm `A9.10` and the corresponding surface algorithm are, in fact, data reduction algorithms, i.e., they work best on large data sets. They assume that during the knot removal phase (Algorithm `A9.9`) a significant number of knots can be removed. If this is not the case the least squares approximation step can fail, either because degree elevation requires more control points than there are data points, or because the many multiple knots make the system of equations singular. A simple fix is to restart the process with a higher degree curve.

9.5 Local Approximation

Let $\mathbf{Q}_0, \ldots, \mathbf{Q}_m$ and E be given. Here we give two examples of local curve approximation; the first produces a rational quadratic B-spline curve, and the second produces a nonrational cubic B-spline curve. In both cases the curves are G^1 continuous by construction; they can be made C^1 continuous, but this is not recommended for the quadratic case because it can produce very poor parameterizations. The quadratic curve fits planar data, while the cubic curve fits arbitrary three-dimensional data. The algorithms are variations on the methods presented in [Pieg87b; Chou92].

A good approximation algorithm attempts to generate as few segments as possible. One way to do this is to let $k_s = 0$, then find the largest $k_e \leq m$

such that $\mathbf{Q}_{k_s}, \ldots, \mathbf{Q}_{k_e}$ can be fit to within the E tolerance by one pth degree segment. Once found, set $k_s = k_e$ and repeat the process to find the largest $k_e \leq m$ such that $\mathbf{Q}_{k_s}, \ldots, \mathbf{Q}_{k_e}$ can be fit with one pth degree segment. This continues until $k_s = m$. Since m is usually rather large in approximation, a binary search is much faster than a linear search. As a practical matter, it may be desirable to bound the number of data points which can be approximated by one segment. We denote this bound by $K_{\max}$, that is, the binary search algorithm is modified to guarantee that

$$k_e - k_s \leq K_{\max}$$

always holds. This is accomplished by initializing k_e to $k_s + K_{\max}$ at the start of each search for the next segment. There are two advantages to setting $K_{\max} < m$:

- for a fixed tolerance E, a smaller $K_{\max}$ produces a more accurate fit;
- if m is large and E is quite small (which is often the case), then the average number of points approximated with each segment is small relative to m; in this case the algorithm is much more efficient if $K_{\max}$ is set substantially smaller than m.

We develop routines `FitWithConic()` and `FitWithCubic()` in Sections 9.5.1 and 9.5.2, respectively. These routines attempt to fit the points $\mathbf{Q}_{k_s}, \ldots, \mathbf{Q}_{k_e}$ to within the tolerance E with a conic or cubic segment, respectively. If the fit is successful, the calling (search) routine either accepts the segment or it increases k_e and calls the fit routine again to try to find an even larger segment (depending on the search strategy). If the fit is unsuccessful, the calling routine decreases k_e and calls the fit routine again.

For both the quadratic and cubic cases, the knot vector is constructed as the segments are accepted. Double internal knots are used in both cases, producing G^1 quadratic and C^1 cubic curves. Accumulated chord lengths corresponding to segment boundaries are used for the knot values of quadratic curves; Eq. (9.37) is used to compute knots for cubic curves.

9.5.1 LOCAL RATIONAL QUADRATIC CURVE APPROXIMATION

We now develop the routine `FitWithConic()`. We assume points $\mathbf{Q}_{k_s}, \ldots, \mathbf{Q}_{k_e}$ lie in a unique plane, and that the end tangents $\mathbf{T}_s$ and $\mathbf{T}_e$ are given.

Consider Figure 9.53. Assume for the moment that

$$\mathbf{R} = \mathbf{Q}_{k_s} + \gamma_s \mathbf{T}_s \qquad \mathbf{R} = \mathbf{Q}_{k_e} + \gamma_e \mathbf{T}_e \tag{9.90}$$

with

$$\gamma_s > 0 \qquad \gamma_e < 0 \tag{9.91}$$

Then to determine an approximating conic, proceed in three steps:

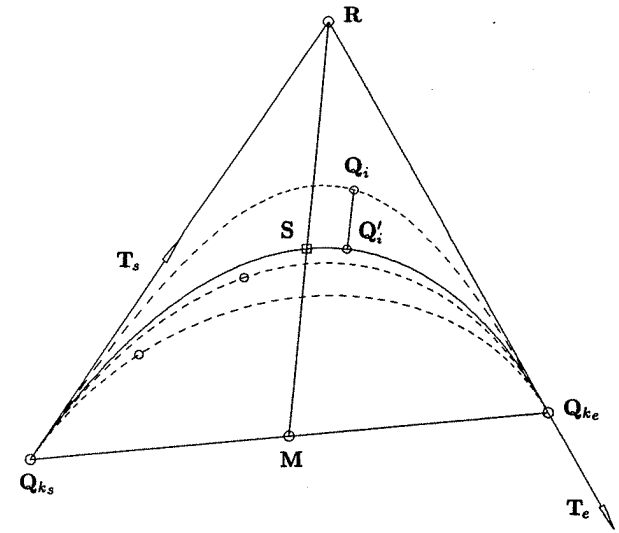


Figure 9.53. A curve fit and scatter check for conic curve fitting.

1. interpolate each point $\mathbf{Q}_i$, $i = k_s + 1, \ldots, k_e - 1$, with a conic, using $\mathbf{Q}_{k_s}$, $\mathbf{R}$, and $\mathbf{Q}_{k_e}$ as the control polygon, as in Algorithm `A7.2`. This yields the middle weight, w_i, and the corresponding shoulder point coordinate, $s_i = w_i/(1 + w_i)$;
2. compute an approximating conic by averaging the s_i, i.e.

$$s = \frac{1}{k_e - k_s - 1} \sum_{i=k_s+1}^{k_e-1} s_i \tag{9.92}$$

   By Eq. (7.30) the shoulder point of the approximating conic is $\mathbf{S} = (1 - s)\mathbf{M} + s\mathbf{R}$, and by Eq. (7.31) the middle weight is $w = s/(1 - s)$;
3. project each $\mathbf{Q}_i$ onto the approximating conic to check if all of them are within tolerance.

There are three special cases which are handled differently:

- $k_e - k_s = 1$, i.e., there are no interior points; the algorithm computes an interpolating segment between $\mathbf{Q}_{k_s}$ and $\mathbf{Q}_{k_e}$ exactly as described in Section 9.3.3;
- $\mathbf{T}_s$ and $\mathbf{T}_e$ are parallel; if they are parallel to the chord $\mathbf{Q}_{k_s}\mathbf{Q}_{k_e}$ and the points $\mathbf{Q}_{k_s}, \ldots, \mathbf{Q}_{k_e}$ are collinear, then a conic segment is returned which is actually a line segment; otherwise no fit is attempted;
- $\mathbf{R}$ is computed but Eq. (9.91) does not hold; no fit is attempted.

FitWithConic() returns the value 1 for a successful fit, or the value 0 for no fit. In the case of a successful fit, the weighted middle control point, $\mathbf{R}^w$, is also returned.

```
ALGORITHM A9.11
  FitWithConic(ks,ke,Q,Ts,Te,E,Rw)
    { /*  Fit to tolerance E with conic segment  */
      /*  Input:  ks,ke,Q,Ts,Te,E  */
      /*  Output: Rw  */
    if (ke-ks == 1)
      { /* No interior points to interpolate */
      Fit an interpolating segment as in Section 9.3.3.
      return(1);
      }
    i = Intersect3DLines(Q[ks],Ts,Q[ke],Te,alf1,alf2,R);
    if (i != 0)
      { /* No intersection */
      if (Q_ks,...,Q_ke not collinear)  return(0);
        else
          {
          Rw = (Q[ks]+Q[ke])/2.0;
          return(1);
          }
      }
    if (alf1 <= 0.0 || alf2 >= 0.0)   return(0);
    s = 0.0;    V = Q[ke]-Q[ks];
    for (i=ks+1; i<=ke-1; i++)
      { /* Get conic interpolating each interior point */
      V1 = Q[i] - R;
      j = Intersect3DLines(Q[ks],V,R,V1,alf1,alf2,dummy);
      if (j != 0 || alf1 <= 0.0 || alf1 >= 1.0 || alf2 <= 0.0)
        return(0);
      Compute the weight wi.   /* Algorithm A7.2 */
      s = s + wi/(1.0+wi);
      }
    s = s/(ke-ks-1);    w = s/(1.0-s);
    if (w < WMIN || w > WMAX)   /* system bounds on weights */
      return(0);
    Create Bezier segment with Q[ks],R,Q[ke] and w.
    for (i=ks+1; i<=ke-1; i++)
      {
      Project Q[i] onto the Bezier segment.
      if (distance > E)   return(0);
      }
    Rw = w*R;
    return(1);
    }
```


Figures 9.54a–9.54e show local quadratic curve fitting examples. The logarithmic spiral

$$r = \frac{1}{2} e^{\theta/5}$$

was sampled and approximated. Input parameters for Figures 9.54a–9.54d were:

Figure 9.54a	$E = 1/100$,	$K_{\max} = 50$
Figure 9.54b	$E = 1/1000$,	$K_{\max} = 50$
Figure 9.54c	$E = 1/10$,	$K_{\max} = 6$
Figure 9.54d	$E = 1/10$,	$K_{\max} = 10$

Figure 9.54e shows the parameterization of the curve in Figure 9.54b.

9.5.2 LOCAL NONRATIONAL CUBIC CURVE APPROXIMATION

We now develop FitWithCubic(). Let $\mathbf{P}_0 = \mathbf{Q}_{k_s}$ and $\mathbf{P}_3 = \mathbf{Q}_{k_e}$ denote the start and endpoints of the cubic Bézier segment, and $\mathbf{T}_s$ and $\mathbf{T}_e$ the start and end unit tangents (see Figure 9.55). We must determine the two inner control points, $\mathbf{P}_1$ and $\mathbf{P}_2$. Setting

$$\mathbf{P}_1 = \mathbf{P}_0 + \alpha \mathbf{T}_s \qquad \mathbf{P}_2 = \mathbf{P}_3 + \beta \mathbf{T}_e$$

we must determine values for α and β such that the cubic Bézier curve defined by $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ passes within the E tolerance of all $\mathbf{Q}_k$, $k_s < k < k_e$. If $k_e - k_s = 1$, then theoretically, any α and β will do; however, they should be chosen so that the derivative magnitudes blend naturally with neighboring magnitudes. Using Eq. (9.29) and Exercise 9.2, reasonable estimates for the derivatives $\mathbf{D}_{k_s}$ and $\mathbf{D}_{k_e}$ at $\mathbf{Q}_{k_s}$ and $\mathbf{Q}_{k_e}$ are

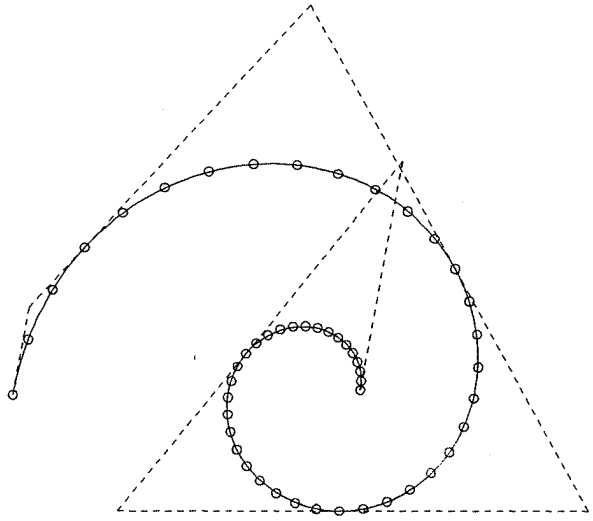
$$\begin{aligned}
\mathbf{D}_{k_s} &= \mathbf{V}_{k_s} \qquad \text{if } k_s = 0 \\
\mathbf{D}_j &= \frac{c_{j+1} - c_j}{c_j - c_{j-1}} \mathbf{V}_j \qquad j \neq \{0, m\} \quad j = k_s \text{ or } j = k_e \\
\mathbf{D}_{k_e} &= \mathbf{V}_{k_e} \qquad \text{if } k_e = m
\end{aligned} \tag{9.93}$$

where $\{c_j\}$ are the accumulated chord lengths. Then set

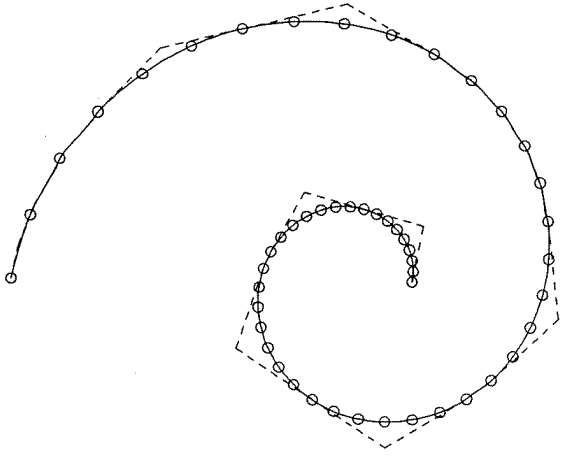
$$\alpha = \frac{|\,\mathbf{D}_{k_s}\,|}{3} \qquad \beta = -\frac{|\,\mathbf{D}_{k_e}\,|}{3} \tag{9.94}$$

Now assume $k_e - k_s > 1$. Our algorithm consists of three steps:

1. for each k, $k_s < k < k_e$, interpolate a cubic Bézier segment to the data $\mathbf{P}_0, \mathbf{T}_s, \mathbf{Q}_k, \mathbf{P}_3, \mathbf{T}_e$, thus obtaining an α_k and a β_k; the Bézier segments, $\mathbf{C}_k(u)$, are defined on the normalized interval $u \in [0, 1]$;



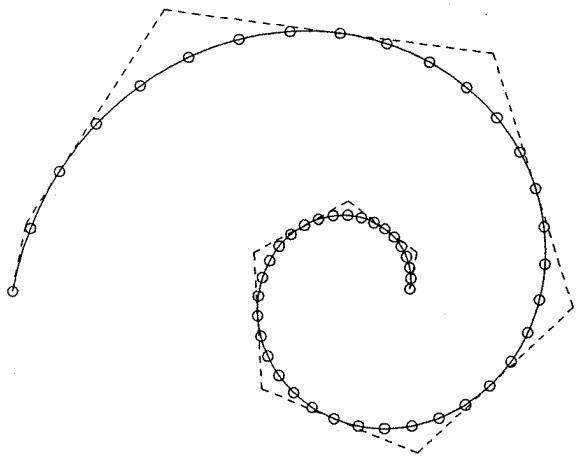
(a)



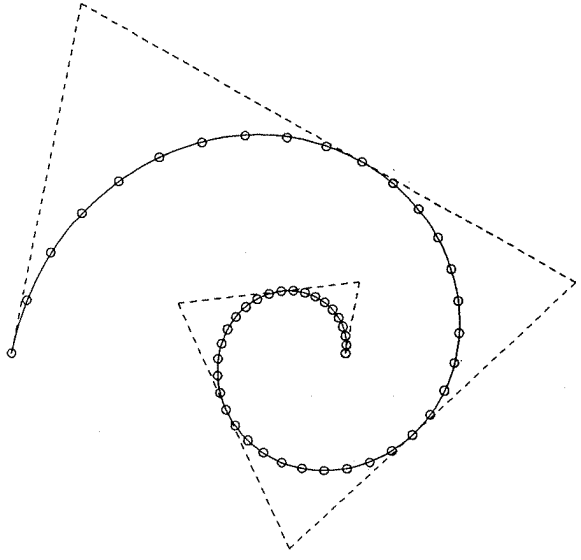
(b)

Figure 9.54. Local quadratic curve fitting examples.

2. average the α_ks and β_ks to obtain α and β; these define the candidate Bézier segment, $\mathbf{C}(u)$;
3. check if all $\mathbf{Q}_k$, $k_s < k < k_e$, are within E distance of $\mathbf{C}(u)$.



(c)

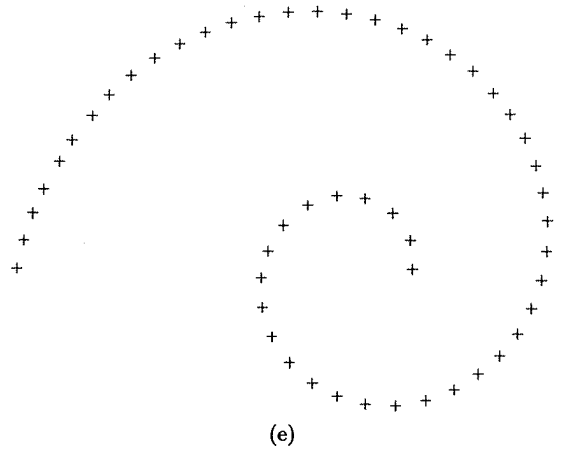


(d)

Figure 9.54. (*Continued.*)

Let $\mathbf{P} = \mathbf{Q}_k$ be the point to interpolate, and denote by $B_{i,3}$ the cubic Bernstein polynomials on $u \in [0, 1]$. We write $\mathbf{P}$ as

$$\mathbf{P} = B_{0,3}\mathbf{P}_0 + B_{1,3}\mathbf{P}_1 + B_{2,3}\mathbf{P}_2 + B_{3,3}\mathbf{P}_3$$



(e)

Figure 9.54. (*Continued.*)

$$= \mathbf{P}_0(B_{0,3} + B_{1,3}) + \alpha \mathbf{T}_s B_{1,3} + \beta \mathbf{T}_e B_{2,3} + \mathbf{P}_3(B_{2,3} + B_{3,3}) \quad (9.95)$$

and finally

$$\mathbf{P} = \mathbf{P}_c + a\mathbf{T}_s + b\mathbf{T}_e \quad (9.96)$$

where the point

$$\mathbf{P}_c = \mathbf{P}_0(B_{0,3} + B_{1,3}) + \mathbf{P}_3(B_{2,3} + B_{3,3}) \quad (9.97)$$

lies on the line $\mathbf{P}_0\mathbf{P}_3$, and

$$a = \alpha B_{1,3} \qquad b = \beta B_{2,3} \quad (9.98)$$

Thus

$$\alpha = \frac{a}{B_{1,3}} \qquad \beta = \frac{b}{B_{2,3}} \quad (9.99)$$

Consider Figure 9.55. Equation (9.96) says that we obtain $\mathbf{P}$ by starting at $\mathbf{P}_c$, going along a direction parallel to $\mathbf{T}_s$ for some distance (to $\mathbf{P}_d$), then continuing from there in the opposite direction of $\mathbf{T}_e$ for another distance. Since $\mathbf{T}_s$ and $\mathbf{T}_e$ have unit length, it follows that

$$a = |\ \mathbf{P}_c\mathbf{P}_d\ | \qquad b = -|\ \mathbf{P}_d\mathbf{P}\ | \quad (9.100)$$

Hence, we must compute $\mathbf{P}_c$ and $\mathbf{P}_d$, then a and b from Eq. (9.100), and finally α and β from Eq. (9.99).

There are two cases to consider, coplanar and not coplanar. First assume the vectors $\mathbf{T}_s$, $\mathbf{T}_e$, and $\mathbf{P}_0\mathbf{P}_3$ are not coplanar. It turns out that there is a unique cubic Bézier curve passing through $\mathbf{P}$, and $\mathbf{P}_c$ and $\mathbf{P}_d$ can be constructed as follows (see Figure 9.55):

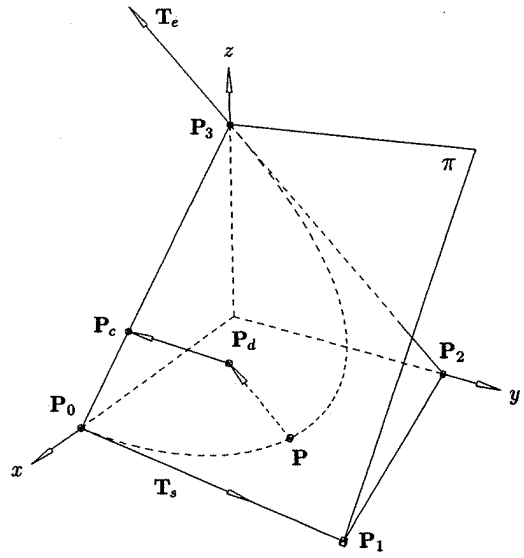


Figure 9.55. A unique cubic three-dimensional (twisted) Bézier curve specified by endpoints, end tangents, and one point on the curve.

1. let π be the plane defined by $\mathbf{P}_0$, $\mathbf{P}_3$, and $\mathbf{T}_s$;
2. intersect π with the line passing through $\mathbf{P}$ and parallel to $\mathbf{T}_e$ – this yields $\mathbf{P}_d$;
3. obtain $\mathbf{P}_c$ by intersecting the line $\mathbf{P}_0\mathbf{P}_3$ with the line passing through $\mathbf{P}_d$ and parallel to $\mathbf{T}_s$.

Equation (9.100) now yields a and b. In order to compute α and β by Eq. (9.99), we require the (unique) parameter value, $\hat{u}$, at which the curve passes through $\mathbf{P}$. From Eq. (9.97) and the fact that the $B_{i,3}$ sum to 1, we have

$$\mathbf{P}_c = \gamma\mathbf{P}_0 + (1-\gamma)\mathbf{P}_3 \quad (9.101)$$

where

$$\gamma = \frac{|\ \mathbf{P}_c\mathbf{P}_3\ |}{|\ \mathbf{P}_0\mathbf{P}_3\ |} = B_{0,3} + B_{1,3} \quad (9.102)$$

γ is computed as a quotient of directed distances, i.e., $\gamma \in [0,1]$ if and only if $\mathbf{P}_c$ lies between $\mathbf{P}_0$ and $\mathbf{P}_3$. The polynomial $B_{0,3} + B_{1,3}$ is cubic in u (the reader should sketch this function), and for any given $\gamma \in [0,1]$ there is a unique solution to Eq. (9.102) that lies within $[0,1]$. If $\gamma \notin [0,1]$, the fit is aborted (return "no fit" to the calling routine). The cubic equation, Eq. (9.102), can be solved by Newton iteration. One way to obtain good start values is to tabulate some number of $\hat{u}$ values corresponding to γ values between 0 and 1. Then, given

any γ, a quick search in the table yields a start $\hat{u}$. Finally, substituting $\hat{u}$ into Eq. (9.99) yields α and β.

If $\mathbf{T}_s$, $\mathbf{T}_e$, and $\mathbf{P}_0\mathbf{P}_3$ are coplanar, then more than one cubic curve passing through $\mathbf{P}$ can exist, and the previous geometric construction of $\mathbf{P}_c$ and $\mathbf{P}_d$ fails. There are several special cases which complicate matters, hence we derive a set of equations which always yields α and β. We first assign a parameter, $\hat{u}$, to $\mathbf{P}$. A good estimate is obtained using chord length parameterization, i.e.

$$\begin{aligned} \hat{u}_{k_s} &= 0 \\ \hat{u}_k &= \hat{u}_{k-1} + \frac{c_k - c_{k-1}}{c_{k_e} - c_{k_s}} \end{aligned} \tag{9.103}$$

The $\hat{u}_k$ are, in fact, the normalized chord length parameters for the $\mathbf{Q}_k$ on $\mathbf{Q}_{k_s}, \ldots, \mathbf{Q}_{k_e}$. Now if $\mathbf{P} = \mathbf{Q}_k$, $k_s < k < k_e$, then $\hat{u} = \hat{u}_k$.

Next we compute the tangent $\mathbf{T}_p$ at $\mathbf{P}$ (Eqs. [9.29] and [9.31]). Let

$$s = 1 - \hat{u} \qquad t = \hat{u}$$

By definition

$$\mathbf{P} = s^3\mathbf{P}_0 + 3s^2t\mathbf{P}_1 + 3st^2\mathbf{P}_2 + t^3\mathbf{P}_3 \tag{9.104}$$

and recalling the deCasteljau algorithm (Eq. [1.12], Algorithm `A1.5`, and Figure 1.17) one can derive

$$\begin{aligned} \mathbf{P}_0^2 &= s^2\mathbf{P}_0 + 2st\mathbf{P}_1 + t^2\mathbf{P}_2 \\ \mathbf{P}_1^2 &= s^2\mathbf{P}_1 + 2st\mathbf{P}_2 + t^2\mathbf{P}_3 \end{aligned} \tag{9.105}$$

where $\mathbf{P}_0^2$ and $\mathbf{P}_1^2$ lie on the line defined by $\mathbf{P}$ and $\mathbf{T}_p$ (Figure 9.56). Thus

$$\begin{aligned} 0 &= \mathbf{T}_p \times \left(\mathbf{P}_1^2 - \mathbf{P}_0^2\right) \\ &= \mathbf{T}_p \times \left(t^2\mathbf{P}_3 + t(2s-t)\mathbf{P}_2 + s(s-2t)\mathbf{P}_1 - s^2\mathbf{P}_0\right) \end{aligned} \tag{9.106}$$

Substituting $\mathbf{P}_1 = \mathbf{P}_0 + \alpha\mathbf{T}_s$ and $\mathbf{P}_2 = \mathbf{P}_3 + \beta\mathbf{T}_e$ into Eqs. (9.104) and (9.106) yields

$$(3s^2t\mathbf{T}_s)\alpha + (3st^2\mathbf{T}_e)\beta = \mathbf{P} - (s^3 + 3s^2t)\mathbf{P}_0 - (t^3 + 3st^2)\mathbf{P}_3 \tag{9.107}$$

$$s(s-2t)(\mathbf{T}_p \times \mathbf{T}_s)\alpha + t(2s-t)(\mathbf{T}_p \times \mathbf{T}_e)\beta = 2st(\mathbf{T}_p \times (\mathbf{P}_0 - \mathbf{P}_3)) \tag{9.108}$$

Equation (9.107) involves the point constraint, $\mathbf{P}$, and Eq. (9.108) the tangent constraint, $\mathbf{T}_p$. They each contribute three equations in two unknowns, for a total of six equations (or four, when the data is xy planar). Some of the special cases are:

- if $\mathbf{T}_s \parallel \mathbf{T}_e$, then Eq. (9.107) yields only one equation; Eq. (9.108) supplies the necessary additional equation;

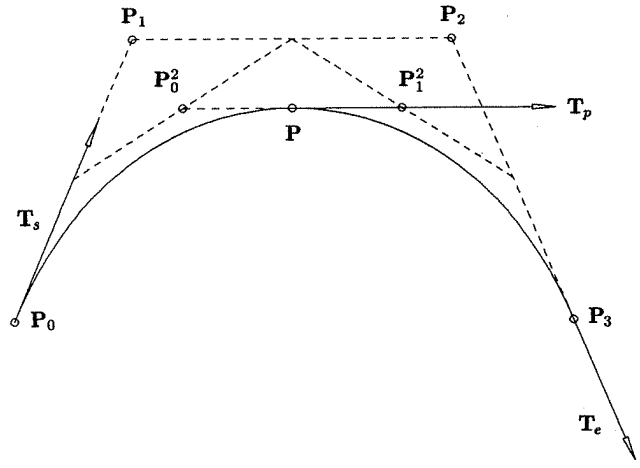


Figure 9.56. A planar cubic Bézier curve as specified by endpoints, end tangents, a point on the curve, and a tangent direction at this point.

- if $\mathbf{P} = (s^3 + 3s^2t)\mathbf{P}_0 + (t^3 + 3st^2)\mathbf{P}_3$, then Eq. (9.107) is a homogeneous system; in this case, $\alpha = \beta = 0$ may be the only solution, or there may exist infinitely many nontrivial solutions. Assuming we have ruled out the straight line case, we have $\mathbf{T}_p \times (\mathbf{P}_0 - \mathbf{P}_3) \neq 0$, hence Eq. (9.108) supplies the necessary nonhomogeneous equations.

A simple way to solve Eqs. (9.107) and (9.108), which avoids worrying about the special cases, is to put the six equations together and solve them by least squares. After solving, it must be checked that $\alpha > 0$ and $\beta < 0$; if this does not hold, the fit is aborted and control is returned to the calling program.

Once the equations are solved the α_ks and β_ks are simply averaged, that is

$$\alpha = \frac{1}{k_e - k_s - 1}\sum_{i=k_s+1}^{k_e-1}\alpha_i \qquad \beta = \frac{1}{k_e - k_s - 1}\sum_{i=k_s+1}^{k_e-1}\beta_i \tag{9.109}$$

The candidate cubic Bézier curve, $\mathbf{C}(u)$, is then defined by $\mathbf{P}_0$, $\mathbf{P}_1 = \mathbf{P}_0 + \alpha\mathbf{T}_s$, $\mathbf{P}_2 = \mathbf{P}_3 + \beta\mathbf{T}_e$, and $\mathbf{P}_3$.

We now must check if each $\mathbf{Q}_k$ is within E distance from the curve $\mathbf{C}(u)$ $(k_s < k < k_e)$. This requires Newton iterations, and the $\hat{u}$s computed previously are good start values. Recall that $\mathbf{Q}_k = \mathbf{C}_k(\hat{u}_k)$. Let $\Delta\alpha_k = \alpha_k - \alpha$ and $\Delta\beta_k = \beta_k - \beta$. Then from Eq. (9.95) we have

$$\begin{aligned} |\mathbf{Q}_k - \mathbf{C}(\hat{u}_k)| &= |\mathbf{C}_k(\hat{u}_k) - \mathbf{C}(\hat{u}_k)| \\ &= |\Delta\alpha_k B_{1,3}(\hat{u}_k)\mathbf{T}_s - \Delta\beta_k B_{2,3}(\hat{u}_k)\mathbf{T}_e| \end{aligned} \tag{9.110}$$

Equation (9.110) can be used as a fast acceptance test: if it is less than or equal to E, then Newton iterations are not required. If iterations are required, the technique given in Section 6.1 is appropriate.

The routine `FitWithCubic()` follows. It returns the value 1 for a successful fit, or the value 0 for no fit. We remark that the construction of $\mathbf{P}_c$ and $\mathbf{P}_d$ in the noncoplanar case is unstable when $\mathbf{T}_s$, $\mathbf{T}_e$, and $\mathbf{P}_0\mathbf{P}_3$ are close to being coplanar. Hence, this test should be made with a rather loose tolerance. The algorithm uses local arrays `uh[]`, `alfak[]`, `betak[]` to store the $\hat{u}_k$, α_k, and β_k.

```
ALGORITHM A9.12
  FitWithCubic(ks,ke,Q,Ts,Te,E,P1,P2)
    { /*  Fit to tolerance E with cubic segment  */
      /*  Input:  ks,ke,Q,Ts,Te,E  */
      /*  Output: P1,P2  */
    if (ke-ks == 1)
      { /* No interior points to interpolate */
      Compute α and β by Eqs.(9.93),(9.94).
      Set P1 = Q_ks + αT_s
      and P2 = Q_ke + βT_e
      return(1);
      }
    dk = ke-ks;
    line = Collinear(dk+1,Q[ks]);
    if (line == 1)
      { /* Collinear points case; create straight line */
      P1 = (2.0*Q[ks]+Q[ke])/3.0;
      P2 = (Q[ks]+2.0*Q[ke])/3.0;
      return(1);
      }
    for (k=1; k<dk; k++)
      {
      Get plane π defined by Q[ks],Q[ke],Ts.
      if (Line(Q[k+ks],Te) lies in π)
        { /* Coplanar case */
        Compute û_k by Eq.(9.103) and load into uh[k].
        Set up Eqs.(9.107) and (9.108) and solve for α_k and β_k
            (by least squares).
        if (α_k > 0 && β_k < 0)
          {
          alfak[k] = α_k;     betak[k] = β_k;
          }
          else
            return(0);
        }
        else
```



```
        { /* Noncoplanar case */
        Pd = Intersection of π with Line(Q[k+ks],Te);
        Pc = Intersection of Line(Q[ks],Q[ke]-Q[ks]) with
            Line(Pd,Ts);
        gamma = Distance3D(Pc,Q[ke])/Distance3D(Q[ks],Q[ke]);
        if (gamma < 0.0 || gamma > 1.0)   return(0);
        Use Newton iteration to solve Eq.(9.102) for uh[k].
        if (uh[k] < 0.0 || uh[k] > 1.0)   return(0);
          else
          {
          a = Distance3D(Pc,Pd);   b = -Distance3D(Pd,Q[k+ks]);
          Evaluate Bernstein polynomials and use Eq.(9.99)
            to get alfak[k] and betak[k].
          }
        }
      } /* End of for-loop:  k=1,...,dk-1 */
      /* Step 2: average the α_k s and β_k s */
  alpha = beta = 0.0;
  for (k=1; k<dk; k++)
    { alpha = alpha+alfak[k];   beta = beta+betak[k]; }
  alpha = alpha/(dk-1);     beta = beta/(dk-1);
  P1 = Q[ks]+alpha*Ts;     P2 = Q[ke]+beta*Te;
      /* Step 3: check deviations */
  for (k=1; k<dk; k++)
    {
    u = uh[k];
    if (Eq.[9.110] less than E)   continue;
      else
      { /* Must do Newton iterations. u is start value */
      Project Q[k+ks] to curve to get error ek.
      if (ek > E)   break;
      }
    }
  if (k == dk)   return(1);   /* segment within tolerance */
     else        return(0);   /* not within tolerance */
  }
```

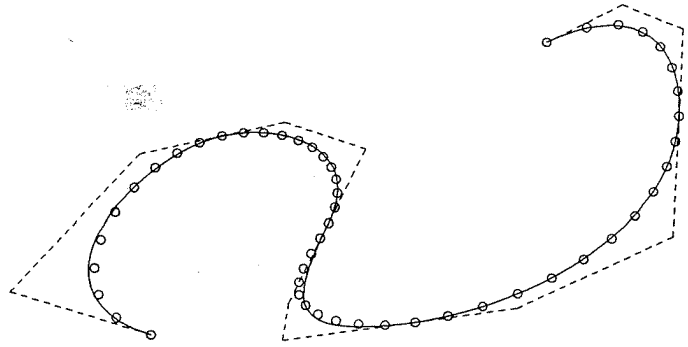
Figures 9.57 and 9.58 show examples of planar and nonplanar fits, respectively. The input data for Figure 9.57 was

Figure 9.57a $E = 1/10$, $K_{\max} = 11$

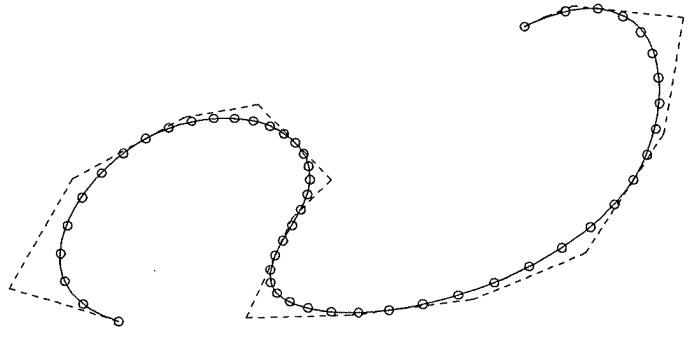
Figure 9.57b $E = 1/10$, $K_{\max} = 8$

Figure 9.57c $E = 1/100$, $K_{\max} = 11$

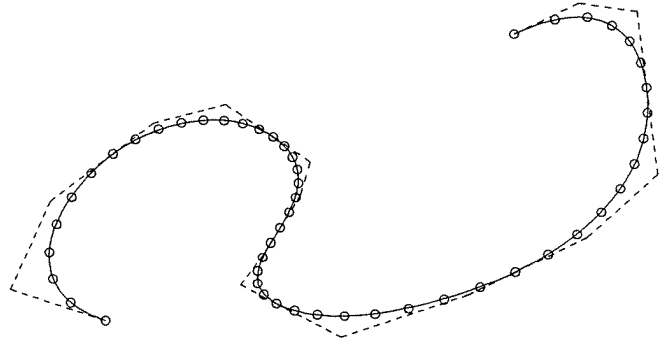
Notice the change in the quality of the approximation from Figure 9.57a to Figure 9.57b. For the same tolerance, allowing a maximum of eight points per



(a)

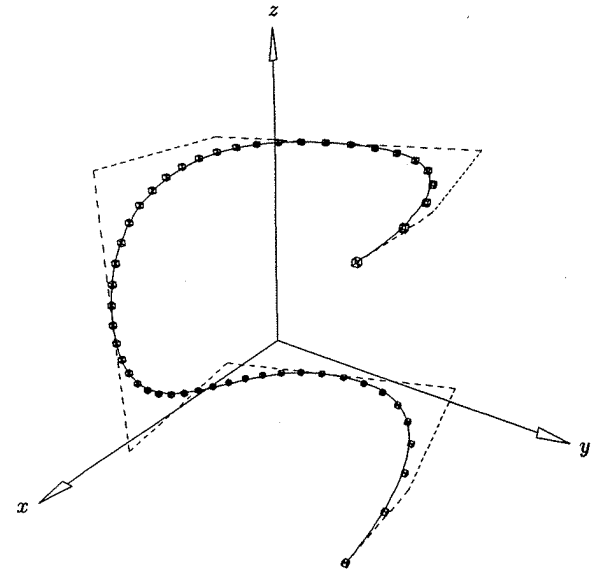


(b)

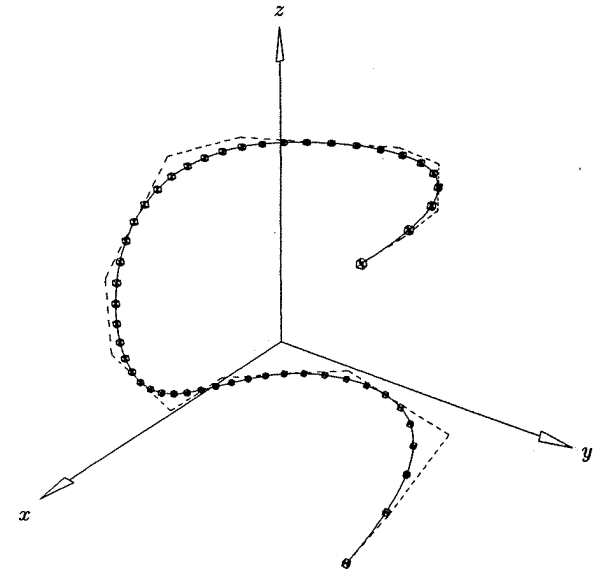


(c)

Figure 9.57. Local cubic approximation to planar data.

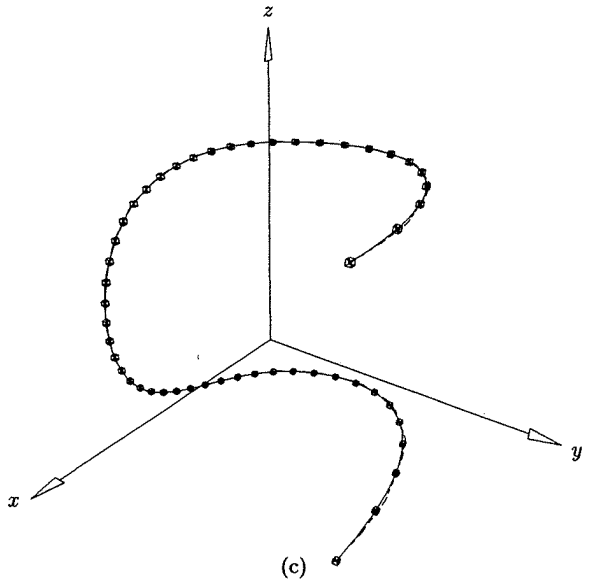


(a)



(b)

Figure 9.58. Local cubic approximation to three-dimensional data containing planar segments.



(c)

Figure 9.58. (*Continued.*)

segment instead of eleven improved the approximation noticeably. The input data for Figures 9.58a–9.58c was

Figure 9.58a	$E = 1/10$,	$K_{\max} = 20$
Figure 9.58b	$E = 1/100$,	$K_{\max} = 20$
Figure 9.58c	$E = 0$,	$K_{\max} = 20$

The zero tolerance resulted in an interpolatory curve. Given 51 points, one might expect 50 segments. However, only 25 segments were required, due to the fact that a cubic can always be constructed to pass through one interior point.

Exercises

9.1. You want to interpolate the points $\mathbf{Q}_{k,\ell}$, $k, \ell = 0, 1, 2, 3$ with a biquadratic surface ($p = q = 2$). Instead of using repeated curve interpolation, use Eq. (9.25) directly to set up a system of 16 linear equations in the 16 unknown $\mathbf{P}_{i,j}$s.

9.2. Assuming that the α_k were computed using Eq. (9.31), consider the unnormalized $\mathbf{V}_k$ of Eq. (9.29). Convince yourself that the quantities $\mathbf{V}_k/\Delta\bar{u}_k$ are good estimates for the derivatives, if the $\Delta\bar{u}_k$ are computed using Eq. (9.5) (the normalized relative chord length). Compute all the $\{\mathbf{V}_k\}$, $k = 0, \ldots, 4$, for $\{\mathbf{Q}_k\} = \{(0,0), (0,20), (20,40), (30,30), (30,20)\}$. What are their magnitudes? Compute the total chord length and the $\Delta\bar{u}_k$ according to Eq. (9.5). Do the quantities $\mathbf{V}_k/\Delta\bar{u}_k$ seem like reasonable estimates for derivatives?

9.3. Draw figures as in Figures 9.48 and 9.49 and determine which B_r values and parameter index ranges must be recomputed for two quadratic cases: $U = \{0,0,0,1,2,3,4,5,6,6,6\}$, with $r = 5$ and $s = 1$; and $U = \{0,0,0,1,2,3,3,4,5,6,6,6\}$, with $r = 6$ and $s = 2$.