AN ALGORITHM FOR AUTOMATICALLY FITTING DIGITIZED CURVES

Philip J. Schneider
University of Geneva
Geneva, Switzerland

Introduction

A new curve-fitting method is introduced. This adaptive algorithm automatically (that is, with no user intervention) fits a piecewise cubic Bézier curve to a digitized curve; this has a variety of uses, from drawing programs to creating outline fonts from bitmap fonts. Of particular interest is the fact that it fits geometrically continuous (G^1) curves, rather than parametrically continuous (C^1) curves, as do most previous approaches.

Curve fitting has been the subject of a fair amount of attention, even before computer graphics came along. The term *spline* comes from drafting jargon: to draw a smooth curve passing through a set of points, the draftsman would place a weight (also a term that has survived into CAGD methods) on each point, and then place a flexible wooden strip (the spline) onto the weights. The spline had slots running lengthwise, which fitted onto the top of the weights, allowing the spline to assume a "natural" and smooth shape. Pavlidis (1983) notes that theories of mechanical elasticity can be used to show that such spline curves exhibit C^2 continuity, and are equivalent to piecewise cubic polynomial curves. Because of this, piecewise polynomial curves are referred to as *splines*, and such curves have been used to mathematically interpolate discrete data sets. Readers interested in interpolation should consult any numerical analysis text, such as Conte and deBoor (1972), or Bartels *et al.* (1987).

This article discusses a method for approximation of digitized curves with piecewise cubic Bézier segments. Such an algorithm is useful in interactive drawing systems, converting bitmapped or otherwise digitized figures (such as fonts) to a parametric curve representation, and the like.

Many techniques have been brought to bear on the problem of this type of curve-fitting: splines (Reinsch, 1967; Grossman, 1970); purely geometric methods (Flegal cited in Reeves, 1981); B-splines (Yamaguchi, 1978; Wu *et al.*, 1977; Giloi, 1978; Lozover and Preiss, 1981; Yang *et al.*, 1986; Dierckx, 1982; Vercken *et al.*, 1987; Ichida *et al.*, 1977; Chong, 1980); hermite polynomials (Plass and Stone, 1983); hyperbolic splines in tension (Kozak, 1986; Schweikert, 1966; Cline, 1974); cubic splines in tension (Cline, 1974; Dube, 1987; Schweikert, 1966); conic sections (Bookstein, 1979); conic splines (Pavlidis, 1983); conic arcs and straight-line segments (Albano, 1974); and circular arcs and straight-line segments (Piegl, 1986). A more detailed description of these solutions may be found in Schneider (1988) and Reeves (1981).

However, each of these approaches has some shortcoming—some of the earlier methods apply only to scalar functions, many require a great deal of intervention by the user, some produce representations that are inappropriate for free-form curves (for example, circular arcs and straight-line segments), and all of the parametric polynomial methods but Plass and Stone's (1983) produce curves that are parametrically continuous, but they note of their method that "... it sometimes does not converge at all."

Bézier Curves

The curve representation that is used in this algorithm in approximating the digitized curve is the Bézier curve. Accordingly, we briefly review the basics: the curves known as Bézier curves were developed (independently) in 1959 by P. de Casteljau and in 1962 by P. Bézier. Numerous references exist; see Boehm *et al.*, (1984) and Davis (1975). A Bézier curve of degree n is defined in terms of Bernstein polynomials:

$$Q(t) = \sum_{i=0}^{n} V_i B_i^n(t), \qquad t \in [0,1],$$

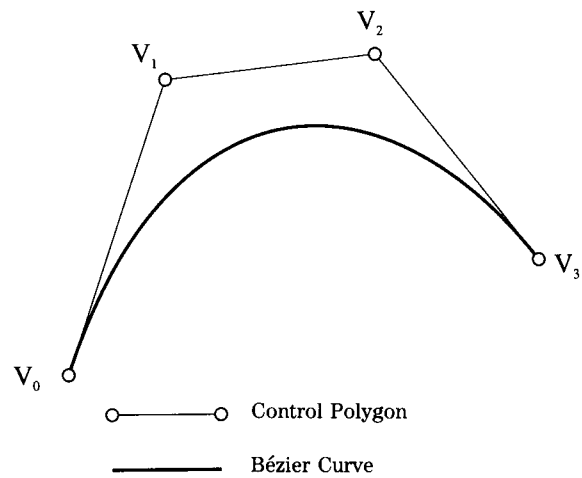

Figure 1. A single cubic Bézier segment.

where the V_i are the control points, and the $B_i^n(t)$ are the Bernstein polynomials of degree n.

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \qquad i = 0, \ldots, n,$$

where $\binom{n}{i}$ is the binomial coefficient $n!/(n-i)!i!$. See Fig. 1 for an example of a cubic Bézier curve. Bézier curves generally are evaluated using a recursive algorithm due to de Casteljau. The algorithm is based on the *recursion* property of Bernstein polynomials:

$$B_i^n(t) = (1-t)B_{i-1}^{n-1} + tB_{i-1}^{n-1}(t).$$

The kth derivative of a Bézier curve is given by

$$\frac{d^k}{dt^k} Q(t) = \frac{n!}{(n-k)!} \sum_{i=0}^{n-k} \Delta^k V_i B_i^{n-k}(t),$$

where $\Delta^1 V_i = \Delta V_i = V_{i+1} - V_i$, and where $\Delta^k V_i = \Delta^k V_{i+1} - \Delta^{k-1} V_i$,

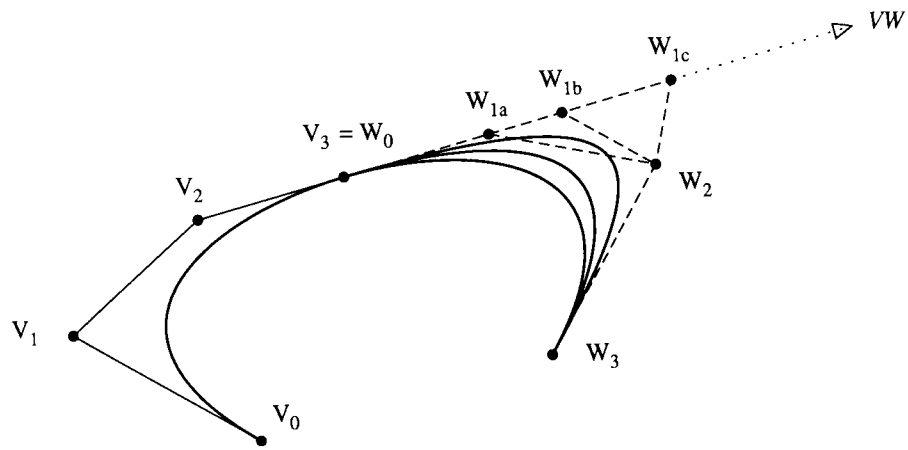

Figure 2. G^1 continuity condition for cubic Bézier curves.

(Watkins, 1987; Boehm *et al.*, 1984). Thus, for $t = 0$ (the left endpoint),

$$\frac{d^k}{dt^k}Q(0) = \frac{n!}{(n-k)!}\Delta^k V_0.$$

For $k = 1$,

$$\frac{dQ}{dt} = Q'(t) = n\sum_{i=0}^{n-1}(V_{i+1} - V_i)B_i^{n-1}(t),$$

which makes more obvious the fact that the tangent vector direction (at the left of the segment) is determined by the line segment from V_0 to V_1. A similar condition holds for the right end.

The implication of this fact is that to enforce continuity at the joints, the second-to-last control point of one segment must be collinear with the last control point (which is also the first control point of the adjoining segment) and the second control point of the adjoining segment. See Fig. 2 for an example; the second control point of the right-hand segment (the one with control points W_i) must be located along the half-line labeled *VW*.

Any sort of complete treatment of continuity conditions is well beyond the scope of this article—interested readers may wish to consult DeRose (1985). Briefly, we note that if the distance between the control point on the left side of the joint and the shared control point is equal to the distance between the shared control point and its neighbor on the right, the tangent vectors will be equal in magnitude and direction. This condition is known as *parametric continuity*, denoted C^1. However, for many applications, this is too restrictive—notice that for the joint to appear smooth, all that is required is that only the tangent *directions* be equivalent, a condition known as *geometric continuity*, denoted G^1. Getting back to the figure, this implies that the shared control point and its two neighbors need only be colinear—the respective distances do not affect the appearance of smoothness at the joint. The curve-fitting algorithm exploits this extra degree of freedom—we employ a least-squares fitting method, which sets these distances so that the error (that is, distance) between the digitized points of the fitted curve is minimized. This has several advantages: first, we can fit the curve with fewer segments; second, parametrically continuous curves correspond to the family of curves drawn by the motion of a particle that moves at a continuous velocity. This is a too much of a restriction for bitmapped fonts, for example; for hand-drawn curves, the restriction is patently wrong.

Fitting the Curve

Following Plass and Stone (1983), a parametric curve $Q(t)$ can be thought of as the projection of a curve in X, Y, t space onto the X–Y plane. Then, we can think of the problem as finding the curve in X, Y, t space whose projection best approximates the digitized curve in the X–Y lane. "Best" is defined in terms of minimization of the sum of squared distances from the digitized curve to the parametric curve.

We state without proof that the curves defined by the projections of the 3-space curve on the X–t and Y–t planes are single-valued (scalar) curves. Thus, if one could devise some scheme relating the X and Y coordinates of each point, one could apply a conventional least-squares function-fitting technique, with the addition constraints of tangent vector direction considered, in X and Y simultaneously. As we are working with

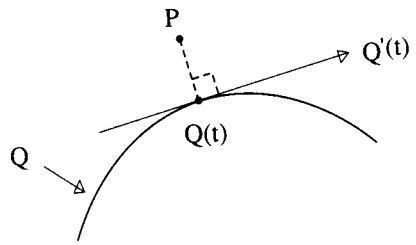

Figure 3. The distance from P to $Q(t)$.

parametric representations, the "scheme" relating the X and Y coordinates is the parametric value t associated with each point. As none are provided, we must look for a way to estimate accurately a value of t for each digitized point. A traditional approach to this problem is to use *chord-length parameterization*.

Once each point has an associated t-value, we can fit a cubic Bézier segment to the set of points (a process described later), and compute the error by comparing the distances between each digitized point p_k and the point with parametric value t_k on the generated curve.

The square distance between a given point P and a point $Q(t)$ on a parametric curve Q is

$$dist = \| P - Q(t) \|. \tag{1}$$

Refer to Fig. 3 for a diagram of this. The general problem can be stated in this manner: given a set of points in the plane, find a curve that fits those points to within some given tolerance. In our case, the curve with which we wish to approximate the points is a cubic Bernstein–Bézier curve, and our fitting criterion is to minimize the sum of the squared distances from the points to their corresponding points on the curve. Formally, we wish to minimize a function S, where S is defined by

$$S = \sum_{i=1}^{n} [d_i - Q(u_i)]^2 \tag{2}$$

$$= \sum_{i=1}^{n} [d_i - Q(u_i)] \cdot [d_i - Q(u_i)], \tag{3}$$

where d_i are the (x, y) coordinates of the given points, and u_i is the parameter value associated with d_i.

In the next set of equations, the following definitions hold:

- V_0 and V_3, the first and last control points, are given—they are set to be equal to the first and last digitized points in the region we are trying to fit.
- $\hat{t}_1$ and $\hat{t}_2$ are the unit tangent vectors at V_0 and V_3, respectively.
- $V_1 = \alpha_1 \hat{t}_1 + V_0$, and $V_2 = \alpha_2 \hat{t}_2 + V_3$; that is, the two inner control points are each some distance α from their the nearest end control point, in the tangent vector direction.

Recall that as we are enforcing G^1 continuity, V_1 and V_2 can be placed anywhere along the half-lines defined by $\hat{t}_1$ and $\hat{t}_2$, respectively. Our problem can be defined as finding α_1 and α_2 to minimize S. That is, we wish to solve these two equations for α_1 and α_2, thereby determining the remaining two control points (that is, V_1 and V_2) of the cubic Bézier segment:

$$\frac{\partial S}{\partial \alpha_1} = 0 \tag{4}$$

$$\frac{\partial S}{\partial \alpha_2} = 0. \tag{5}$$

Expanding Eq. (4),

$$\frac{\partial S}{\partial \alpha_1} = \sum_{i=1}^{n} 2[d_i - Q(u_i)] \cdot \frac{\partial Q(u_i)}{\partial \alpha_1}.$$

Expanding the second term, we get

$$\begin{aligned}\frac{\partial Q(u_i)}{\partial \alpha_1} &= \frac{\partial}{\partial \alpha_1}\left(V_0 B_0^3(u_i) + (\alpha_1 \hat{t}_1 + V_0) B_1^3(u_i)\right. \\ &\qquad \left. + (\alpha_2 \hat{t}_2 + V_3) B_2^3(u_i) + V_3 B_3^3(u_i)\right) \\ &= \hat{t}_1 B_1^3(u_i).\end{aligned}$$

Thus,

$$\frac{\partial S}{\partial \alpha_1} = \sum_{i=1}^{n} 2[d_i - Q(u_i)] \cdot \hat{t}_1 B_1^3(u_i) = 0.$$

Rearranging, we get

$$\sum_{i=1}^{n} B_1^3(u_i) Q(u_i) \cdot \hat{t}_1 = \sum_{i=1}^{n} \hat{t}_1 B_1^3(u_i) \cdot d_i.$$

For convenience, define

$$A_{i,1} = \hat{t}_1 B_1^3(u_i).$$

Then,

$$\sum_{i=1}^{n} Q(u_i) \cdot A_{i,1} = \sum_{i=1}^{n} d_i \cdot A_{i,1}. \tag{6}$$

Expanding $Q(u_i)$,

$$\begin{aligned}
&\sum_{i=1}^{n} Q(u_i) \cdot A_{i,1} \\
&\quad = \sum_{i=1}^{n} A_{i,1} \cdot \left(V_0 B_0^3(u_i) + \alpha_1 A_{i,1} + V_0 B_1^3(u_i)\right. \\
&\qquad\qquad \left. + \alpha_2 A_{i,2} + V_3 B_2^3(u_i) + V_3 B_3^3(u_i)\right) \\
&\quad = \sum_{i=1}^{n} A_{i,1} \cdot V_0 B_0^3(u_i) + \alpha_1 \sum_{i=1}^{n} A_{i,1}^2 + \sum_{i=1}^{n} A_{i,1} \cdot V_0 B_1^3(u_i) \\
&\qquad + \alpha_2 \sum_{i=1}^{n} A_{i,1} \cdot A_{i,2} + \sum_{i=1}^{n} A_{i,1} \cdot V_3 B_2^3(u_i) + \sum_{i=1}^{n} A_{i,1} \cdot V_3 B_3^3(u_i)
\end{aligned}$$

Equation (6) becomes

$$\begin{aligned}
&\left(\sum_{i=1}^{n} A_{i,1}^2\right)\alpha_1 + \left(\sum_{i=1}^{n} A_{i,1} \cdot A_{i,2}\right)\alpha_2 \\
&\quad = \sum_{i=1}^{n} \left(d_i - \left(V_0 B_0^3(u_i) + V_0 B_1^3(u_i) + V_3 B_2^3(u_i) + V_3 B_3^3(u_i)\right)\right) \cdot A_{i,1}
\end{aligned}$$

Similarly, for $\partial S/\partial \alpha_2$,

$$\left(\sum_{i=1}^{n} A_{i,1} \cdot A_{i,2}\right)\alpha_1 + \left(\sum_{i=1}^{n} A_{i,2}^2\right)\alpha_2$$

$$= \sum_{i=1}^{n} \left(d_i - \left(V_0 B_0^3(u_i) + V_0 B_1^3(u_i) + V_3 B_2^3(u_i) + V_3 B_3^3(u_i)\right)\right) \cdot A_{i,2}$$

If we represent the previous two equations by

$$c_{1,1}\alpha_1 + c_{1,2}\alpha_2 = X_1$$

$$c_{2,1}\alpha_1 + c_{2,2}\alpha_2 = X_2,$$

we need only solve

$$\begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix}\begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}$$

for α_1 and α_2. If we let

$$\mathscr{C} = \begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \end{pmatrix}$$

$$\mathscr{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix},$$

Then, using Cramer's Rule, the solution to

$$\begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \mathscr{C}^{-1}\mathscr{X}$$

is

$$\alpha_1 = \frac{\det(\mathscr{X}\ C_2)}{\det(C_1\ C_2)}$$

$$\alpha_1 = \frac{\det(C_1\ \mathscr{X})}{\det(C_1\ C_2)}$$

Our algorithm that attempts to fit a single cubic Bézier segment to a set of points appears in Fig. 4. We begin by computing approximate tangents at the endpoints of the digitized curve. This can be accomplished by fitting a least-squares line to the points in the neighborhood of the endpoints, or by averaging vectors from the endpoints to the next n points, and so on. Next, we assign an initial parameter value u_i to each point d_i, using chord-length parameterization. At this point, we use the technique described to fit a cubic Bézier segment to the points—the first and last control points are positioned at the first and last digitized points in the region we are working on, and the inner two control points are placed a distance α_1 and α_2 away from the first and last control points, in the direction of the unit tangent vectors previously computed. We then compute the error between the Bézier curve and the digitized points, noting the point that is the farthest distance from the curve. If the fit is acceptable, we draw or otherwise output the curve. If the fit is not acceptable, we could break the digitized points into two subsets at the point of greatest error, compute the unit tangent vector at the point of splitting, and recursively try to fit Bézier curves to these two new subcurves. Consider, though, that our initial chord-length parameterization is only a very rough approximation; if we had a better parameterization of the points, we might have been able to fit the curve without further recursive processing. Fortunately, there is a technique available to us. Referring back to Eq. 1, our t is that chord-length-generated approximate parameter. We can use Newton–Raphson iteration to get a better t—in general, the formula is

$$t \leftarrow t - \frac{f(t)}{f'(t)}. \tag{7}$$

Referring back to Fig. 3, we wish to solve

$$[Q(t) - P] \cdot Q'(t) = 0 \tag{8}$$

for t. In our case, then, we reduce the t for each point by

$$\frac{Q_1 t \cdot Q_2 t}{[Q_1(t) \cdot Q_2(t)]'}. \tag{9}$$

```
FitCurve(d, ε)
        d     : array[ ] of point; Array of digitized points
        ε     : double;            User-specified error
begin
        Compute t̂1 and t̂2, the unit tangent
                vectors at the ends of the digitized points;
        FitCubic(d, t̂1, t̂2, ε);
end

FitCubic(d, t̂1, t̂2, ε)
        d     : array[ ] of point; Array of digitized points
        t̂1, t̂2 : vector;           Endpoint tangents
        ε     : double;            User-specified error
begin
    Compute chord-length parameterization of digitized points;
    Fit a single cubic Bezier curve to digitized points;
    Compute the maximum distance from points
    to curve;
    if error < ε
      then begin
            DrawBezierCurve;
            return;
            end;
    if error < ψ
      then begin
            for i: integer ← 0, i ← 1, while i < maxIterations do
               begin
                    Reparameterize the points;
                    Fit a single cubic Bezier curve to digitized points;
                    Compute the maximum distance from points to
                    curve;
                    if error < ε
                      then begin
                                DrawBezierCurve;
                                return;
                                end
                    end
            endloop
      else begin
                Compute unit tangent at point of maximum error;
                Call FitCubic on the "left" side;
                Call FitCubic on the "right" side;
           end
      end
    end
```

Figure 4.

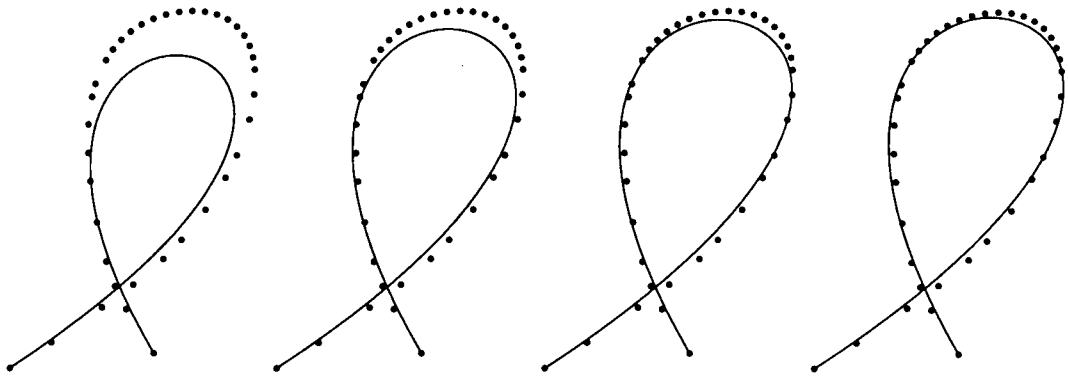

Figure 5. Use of the iterative process to improve the fitting of the cubic.

This technique was first used by Grossman (1970) and later by Plass and Stone (1983) in their algorithms. This iteration can greatly improve the fit of the curve to the points: see Fig. 5, for an example of the process.

Newton–Raphson iteration is fairly inexpensive and tends to converge rather quickly. Thus, one might want to attempt this improvement every time. However, if the initial fit is very poor, it may be best not even to attempt the improvement. So, we compare the error to some value ψ, which we set to some small multiple or power of the user-specified error ϵ. This value ψ is implementation-dependent, but is easy to determine empirically. Additionally, since the incremental improvement decreases quickly with each successive iteration, we set a limit on the number of attempts we make (the author found that a value of four or five is appropriate). Finally, we note that while this Newton–Raphson iteration is cheap, the associated fitting attempts are not. The astute reader may notice, then, that we have placed more emphasis on minimizing the number of segments generated than on execution speed. Even so, the algorithm is generally more than fast enough for interactive use, even with a very large number of points to fit. In addition, it is very stable—the author has not seen a case when the algorithm failed to converge quickly on a satisfactory and correct fit.

One final note: the least-squares mathematics fails when there are only two points in the digitized subcurve to be fitted. In this case, we adopt a

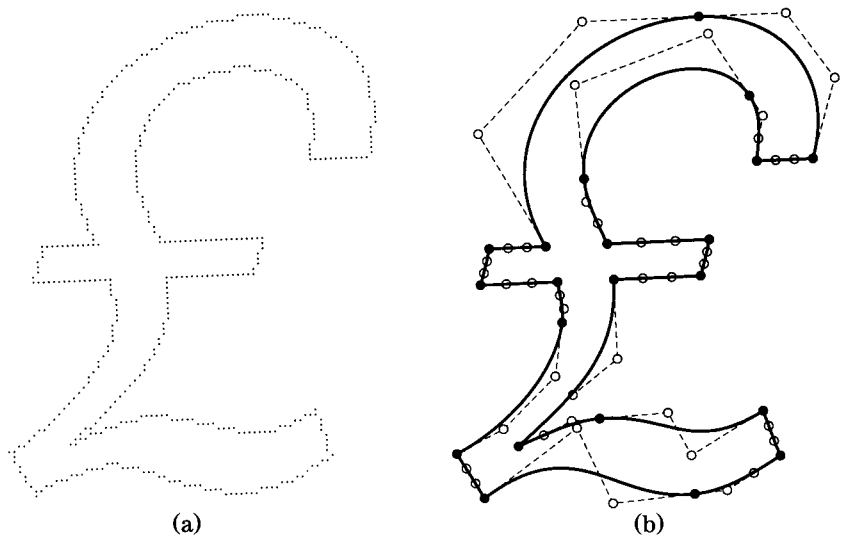

Figure 6. A digitized glyph, showing the original samples and the fitted curves.

method from Schmitt *et al.*, (1986), and plane the inner two control points at a distance from the outer two control points equal to one-third of the distance between the two points, along the unit tangents at each endpoint.

Examples of the fitting algorithm being applied to a digitized font and to a hand-sketched curve appear in Figs. 6 and 7. The large dots indicate the cubic Bézier control points—the filled dots are the first and last control points in each curve (which are shared with the curve's neighbors), and the hollow dots are the "inner" control points.

Implementation Notes

Several points should be made with respect to implementation. First, the implementor may want to preprocess the digitized data prior to calling the fitting routine. Such preprocessing might include: removing coinci-

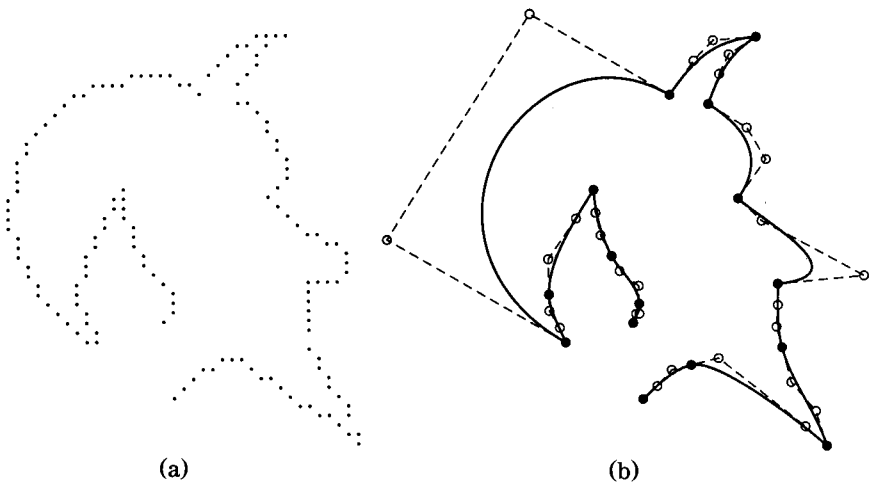

Figure 7. A hand-sketched curve, showing the original samples and the fitted curves.

dent and/or nearly coincident data points, filtering the points with a little convolution (makes tangent computation more reliable), and locating corners. By "corners" we mean regions in the digitized curve where there should be a discontinuity; such points can be located by looking at the angle created by a point and its neighbors. These corners divide the original curve into a number of distinct subcurves, each of which can be fitted independently. Second, negative α values occasionally are generated when the points are very irregularly spaced. In this case, one can either split the points and try to fit each subcurve, or employ the heuristic mentioned earlier.

A sample C implementation of the algorithm is included in the appendix, in the file *fit_cubic.c*. Inputs to the routine *FitCurve* are the array of digitized points, their number, and the desired (squared) error value. When Bézier curves are generated, the routine *DrawBezierCurve* is called to output the curve; this routine must be supplied by the implementor, and simply consists of drawing or otherwise outputting the curve defined by the control points with which it is called.

Conclusion

We have presented an adaptive, automatic algorithm for fitting piecewise cubic Bézier curves to digitized curves. This algorithm elegantly parallels adaptive subdivision for curve evaluation, and shares with that technique that characteristic that regions of lower curvature are more coarsely refined than those of higher curvature. Advantages over previous approaches are complete automaticity, geometric continuity, stability, and extreme ease of implementation.

See Appendix 2 for C Implementation (797)