Check for updates

Curve-Fitting with Piecewise Parametric Cubics

Michael Plass and Maureen Stone
IMAGING SCIENCES LABORATORY
Xerox Palo Alto Research Center

Abstract: Parametric piecewise-cubic functions are used throughout the computer graphics industry to represent curved shapes. For many applications, it would be useful to be able to reliably derive this representation from a closely spaced set of points that approximate the desired curve, such as the input from a digitizing tablet or a scanner. This paper presents a solution to the problem of automatically generating efficient piecewise parametric cubic polynomial approximations to shapes from sampled data. We have developed an algorithm that takes a set of sample points, plus optional endpoint and tangent vector specifications, and iteratively derives a single parametric cubic polynomial that lies close to the data points as defined by an error metric based on least-squares. Combining this algorithm with dynamic programming techniques to determine the knot placement gives good results over a range of shapes and applications.

CR Categories and Subject Descriptors: G.1.2 [**Numerical Analysis**]: Approximation–Least squares approximation; Spline and piecewise polynomial approximation; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations–Iterative methods; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search–Dynamic programming; I.3.3 [**Computer Graphics**]: Picture/Image Generation–Digitizing and scanning; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling–Curve, surface, solid, and object representations

General Terms: Algorithms

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0-89791-109-1/83/007/0229 $00.75

1. Introduction.

We are interested in the problems that arise when trying to use two-dimensional curved shapes in an interactive design environment. In particular, we are interested in finding the best ways for a designer to define such shapes. Our immediate focus is the production of publication-quality documents. The documents are designed using an interactive system with a raster display, then camera-ready copy is digitally produced at very high resolutions. Our design methodology is to represent all shapes analytically using piecewise parametric cubic polynomials [27]. There are many published methods for specifying such curves [2,12,15,23]. These methods all have in common the approach that the shape designer provides a small set of controlling points and parameters which act as handles to shape the curve. At Xerox PARC such algorithms, and similar ones for conics, have been used in design systems for making illustrations [4,3,16,26] and digital typefaces [5]. We conclude that it would be a powerful addition to the set of tools for specifying shapes to be able to reliably derive an efficient piecewise parametric cubic polynomial representation from a sequence of closely-spaced points that approximate the desired curve. The points could be produced in a variety of ways such as sketching with a digitizing pen, scanning and finding edges in an existing drawing, or writing a program to compute a set of data points. In this paper we will present a method deriving such an analytical representation from digitized data.

For the data in our curve-fitting work we have experimented with hand-tuned digital curves, optically-scanned images and simple hand-drawn sketches. We have found that the source of the digitized curve is relatively unimportant. The differences principally affect how tightly you want to follow the data; is it noisy, does it have pronounced rastering effects, are corners well defined? A scanned image will not have as sharp corners as a hand-tuned bitmap, for example. Also, the tolerance is a function of the image resolution. We feel our results are generally applicable to the problem of generating piecewise approximations to shapes from an ordered set of discrete data points.

The heart of our method is an algorithm that takes a set of sample points and derives a parametric cubic curve that lies close to the data points; this algorithm is presented in section 4. A more elaborate version of the algorithm,

presented in section 5, allows the endpoints and/or the tangent directions at the endpoints of the piece to be specified.

A piecewise approximation to a shape consists of a number of single cubic pieces connected end-to-end. The location and smoothness of the joints or *knots* between the pieces is critical to getting a good representation of the shape. Sometimes we want a smooth connection between pieces, and sometimes we want to produce a sharp "corner." If we somehow knew in advance the position of the knots, and the tangent direction at each non-corner knot, we could simply apply the single-piece fitting algorithm to the points in each span, yielding a piecewise approximation. Finding the best knot positions and tangents, however, is no easy task. Our most promising results were obtained by setting tangents using fairly local information, and then using dynamic programming to search the space of potential knot positions.

The single-piece fitting routine is used in several places in the overall process. In the dynamic programming phase it is used as a way of measuring how well a span of sample points may be fit with a single cubic piece, and, after the knots have been found, it is used to find the piecewise approximation. The algorithm may also be used in the initial phase to compute tangent directions, by fitting a range of samples around the point of interest and finding the tangent of the resulting curve. This method works well for estimating tangent directions in noisy data.

In this paper we want to emphasize our parametric cubic curve-fitting algorithm and the use of dynamic programming to find the knots. While it is important to accurately find corners and tangents, we feel the choice of algorithms to do so is highly application dependent and we are not in a position to give a complete analysis here.

The next section of this paper is a general discussion of the problem of fitting shapes with piecewise functions. In section 3 we describe how we use dynamic programming to find the knot positions. Sections 4 and 5 will present in detail our algorithm for computing a single parametric cubic given a set of data points. We conclude with a summary and examples of our results.

2. Piecewise function approximation to shapes.

What do we mean by the best piecewise approximation to a shape such as Figure 1(a)? The curve must lie near the data points, of course, and the representation will be most efficient if we use as few pieces as possible. This presents an obvious trade-off between a tight fit and an efficient representation. Also, the desired tightness of the fit will vary with the amount of noise in the data. Usually the curve should be smooth, and the pieces of the curve should fit smoothly together. In some cases, however, we find that the shape will look better if we introduce sharp bends at locations that are perceived as "corners". There are other issues of perception as well. For example, if the samples lie on a straight line, most users will object if the approximating curve is slightly wavy even if it lies quite close to the sample points.

In this paper we will parameterize the problem as follows: The closeness of the fit is specified by a user-defined tolerance; it is the user's job to consider the resolution and accuracy of the input when specifying the tolerance. The fit is smooth if the direction of the tangent vector is continuous. Restricting only the direction of this vector to maintain smoothness gives us the maximum flexibility for two-dimensional curves. We use one of a number of algorithms to find the corners before finding any of the other knot locations. These methods all have the property that the user sets some parameters to tune the algorithm for a particular set of data. If a point is identified as a corner, we put a knot at that location and simply do not force tangent continuity there. While a parametric cubic is flexible enough to have a sharp cusp in it, the behavior of the curve near the cusp is quite constrained so we generally do not try to model corners with cusps.

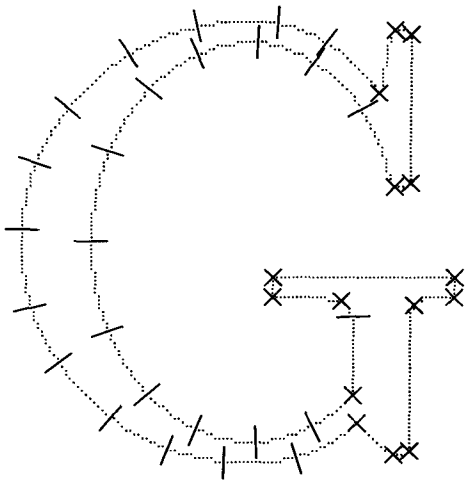
Figure 1 is an example of the process we use to fit a shape. The original data, which in this case is derived from a high resolution bitmap, is shown in figure 1(a). We need to limit the choice of knot locations to a finite set so we can use a finite search technique like dynamic programming. This set is defined to be the set of sample points. For efficiency, we want to reduce the number of possible knot locations even further. In this example we restrict the potential knot locations to the vertices of a polygon fit to the sample points. Any vertex with an angle sharper than 135° is taken to be a corner. For this kind of data we usually filter the samples between corners to remove the more pronounced rastering effects. After filtering, the tangent vectors are set on each potential knot that is not a corner. Figure 1(b) shows the unfiltered samples, the potential knots, and the normal vectors to the tangents at each potential knot. If the potential knot is a corner, it has no tangent set and is marked with a cross. To calculate the tangents in this example we used the curve-fitting algorithm to fit a small number of points around each potential knot and determined the tangent to the resulting curve at the point nearest the potential knot. The computed curved shape is shown in figure 1(c). Figure 1(d) shows the result in more detail. The small dots are the samples, and the large dots are the knots that were actually chosen.

Further examples of shapes fit with our techniques are included in the conclusions.

3. Choosing the knots.

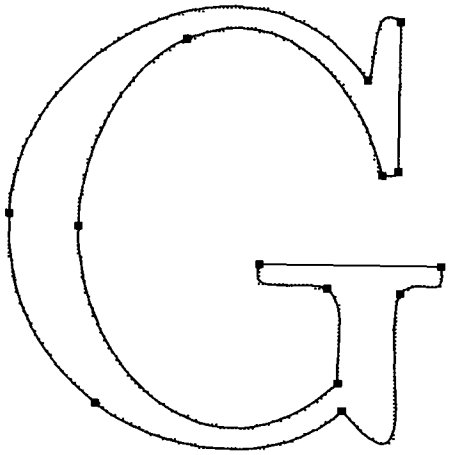
Before discussing methods for choosing knots, let us look first at the problem of predicting whether a set of samples can be closely fit by a single piece of the curve. A parametric cubic polynomial is quite flexible, but has some obvious limits. For example, it can contain at most one loop or, if it has no loop, at most two inflection points. It is tempting, therefore, to try to derive a set of algorithms that measure space-curve characteristics such as slope and curvature and use this information to set the knot positions. However, as you add constraints to guarantee continuity, the degrees of freedom available to fit the cubic to the samples are reduced. Furthermore, the fit is very vulnerable

(a): Original data, approximately 150 by 160 bits.



(b): Samples, potential knots and normal vectors.

(c): Result of curve fitting algorithm, 14 cubic pieces.



(d): Samples, curve and knots.

Figure 1. An example showing the steps involved in fitting a parametric cubic to a sampled shape. Figure (a) is the original data taken from a high resolution bitmap. Figure (b) shows the sample points, the potential knots and the corresponding normal vectors. Potential knots marked with a cross are corners. Figure (c) is the final shape. Figure (d) shows the curve, knots and samples in more detail.

to a poorly chosen continuity constraint such as a bad tangent vector. The only reliable way we have found to tell if a set of samples can be fit by a single cubic with constraints is to try it.

Given that we are going to pay the cost of trying out different cubic pieces, one simple method for defining the knots is to "grow" the pieces out until the fit for that piece exceeds some threshold. In other words, starting with the previous knot, keep adding data points until the piece is as long as possible. Each new piece must be constrained to maintain continuity. This works, in the sense that the resulting curve will fit everywhere within the specified tolerance, but is quite vulnerable to local phenomena such as noisy data or a badly defined tangent. We found this method works only moderately well in terms of the total number of knots in the final solution especially when trying to maintain tangent continuity. Subdivision is another obvious choice which has the same problem. These two techniques are also described by Reeves [20].

If we want to maintain continuity, the globally-best set of knots cannot be found by a method that only optimizes locally. For example, if the first and last sample points are constrained to lie on the curve, the error in fitting a parametric cubic piece is not a monotone function of the number of samples; it may be possible to get a better fit by using a longer run of samples with better endpoint locations. This is especially true if the data is noisy. In theory, to find the best solution we could try all possible arrangements of knots and then take the best fit. This solution is exponential in the number of data points, and is clearly out of the question for any useful cases. It is possible, however, to reduce the complexity to $O(n^3)$ by using dynamic programming.

Dynamic programming [1,6] is a method for efficiently searching a solution space where the problem can be recursively broken up into a set of subproblems. Since the same subproblems contribute to the solution of several larger subproblems, the total number of computations can be reduced by storing the results of each subproblem. We can apply this method to finding the knots as follows: Let e_{ij} be the error obtained when a single piece is fit to samples i through j. The exact definition of e_{ij} does not matter insofar as the dynamic programming algorithm is concerned, and we have obtained good results with several different ones. One reasonable choice for e_{ij} is simply the sum of the squares of the distances from the sample points to the curve, plus a constant $\tau > 0$. Increasing the value of τ will encourage fewer knots, at the expense of not fitting the samples as closely. The total error for a given arrangement of knots is just the sum of the errors for the individual pieces. Let E_{ij} be the least total error over all possible arrangements of knots for samples i thru j. This can be computed as

$$E_{ij} = \min(E_{ik} + e_{kj}), \quad i < k < j,$$

where we have already computed and tabulated the values for E_{ik} for $i < k < j - 1$. We can compute all the values for e_{kj} by working backwards and fitting single pieces from samples j to k for all values of k in the range.

Even using dynamic programming it is very expensive to test all possible subranges for the best fit. We can do several things to prune the search space and improve the performance at the cost of no longer guaranteeing the optimum solution. For example, it is usually true that for each piece the shape of the curve distant from the area of interest will contribute little to the local best solution. Therefore, we limit how far back along the curve we search when computing E_{kj} above. As in the example of Figure 1, we generally use a subset of the sample points as potential knot positions to further reduce the cost.

The main contribution of dynamic programming to this algorithm is a way of searching in a more global manner for the best solution set of knots. However, one of the principal advantages of piecewise functions is that they are controlled mostly by local information. Said another way, for a particular subsection of the curve, the best locations for the knots is not going to be affected much by the shape of the curve far away. So it may be possible to construct a more efficient algorithm than ours that looks only in a small region of the curve for each knot location. What our experience indicates is that it is important to do some searching to give immunity from local aberrations.

The next two sections will describe our algorithm for fitting a parametric cubic polynomial to a set of data points.

4. Least-squares curve fitting with a parametric cubic polynomial.

For computer graphics applications, the common specification of a parametric cubic polynomial is $F(X(t), Y(t))$, where X and Y are cubic polynomials and t lies in the range 0 to 1. Such a curve can be thought of as a three-dimensional curve in the space X, Y, t. The curve is constrained such that the projections on the X-t and Y-t planes are cubic polynomials. What is usually displayed is the projection of this curve on the X-Y plane. Our problem is: given a set of points on the X-Y plane, find the 3-space curve whose X-Y projection is such that the sum of the squares of the distances between the points and the projected curve is minimized. Note that there is not a unique solution to this problem. A simple example of this is that if X is a scalar multiple of Y, the resulting projected curve is a straight line, and many different (X, Y) pairs will yield the same line.

Let us look at this problem a bit differently. We are given a set of data points. If we knew the parametric curve that gave the best fit, we could find the points on the projected curve that were nearest each of the data points. If we assigned the corresponding value of t to the data point, we are in effect positioning the data points near the curve in X, Y, t space. The interesting thing about this representation is that if we consider the projection of the points and curve on, for example, the X-t plane, we have the cubic polynomial that minimizes the sum of the squares of the distances to the points in X. This means if we only knew the correct t values for our data points, we could solve for our curve by performing simple least-squares fitting in X and Y independently. Our algorithm is an iterative technique that converges to this set of t values.

Our algorithm is simple to state: Assign an initial approximation to t for each data point. Solve $X(t)$ and $Y(t)$ independently using conventional least-squares techniques. Measure the result, and if necessary, adjust the t values for the data points and repeat. This is a fixed-point iteration that, if it converges, will converge to a local minimum for our non-linear system. For the class of curves we are using it usually does converge and gives a good solution.

We use an initial approximation based on the Euclidean distance between the data points in the $X-Y$ plane. That is, let

$$s_k = \sum_{i=1}^{k-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}, \qquad (1)$$

be the total length of the polygonal segment connecting points 1 through k. We set the initial value of t_k be s_k/s_n.

To solve for $X(t)$ and $Y(t)$, simple least-squares curve-fitting [8] will be fine for this problem as stated. We will defer any further discussion of this part of the algorithm until section 5, where we will extend this technique so that it is possible to specify endpoint and tangent conditions for the final curve.

Given X and Y, we adjust the t_i values by finding for each data point the point on the curve closest to it. We then assign this new t value to the data point. In X, Y, t space, we are moving the data point closer to the current curve, changing only the value for t. We stop iterating when the values for t no longer change significantly, or when some other condition is satisfied; for example, if the maximum distance from the points to the curve falls below some threshold. We also set an upper bound on the number of iterations.

The square of the distance between a given point (x, y) and any point $(X(t), Y(t))$ that lies on the curve is

$$(X(t) - x)^2 + (Y(t) - y)^2 \qquad (2)$$

To find where this is minimum we differentiate and equate to zero, yielding

$$2(X(t) - x)X'(t) + 2(Y(t) - y)Y'(t) = 0 \qquad (3)$$

The left side of this equation is a fifth-degree polynomial in t. Since we have an approximation to the desired root, the previous value, we can use Newton-Raphson iteration to find the root. The formula for Newton-Raphson iteration for solving $f(t) = 0$ is

$$t \leftarrow t - \frac{f(t)}{f'(t)} \qquad (4)$$

so each iteration should decrease t by the amount

$$\frac{(X(t) - x)X'(t) + (Y(t) - y)Y'(t)}{X'(t)^2 + Y'(t)^2 + (X(t) - x)X''(t) + (Y(t) - y)Y''(t)} \qquad (5)$$

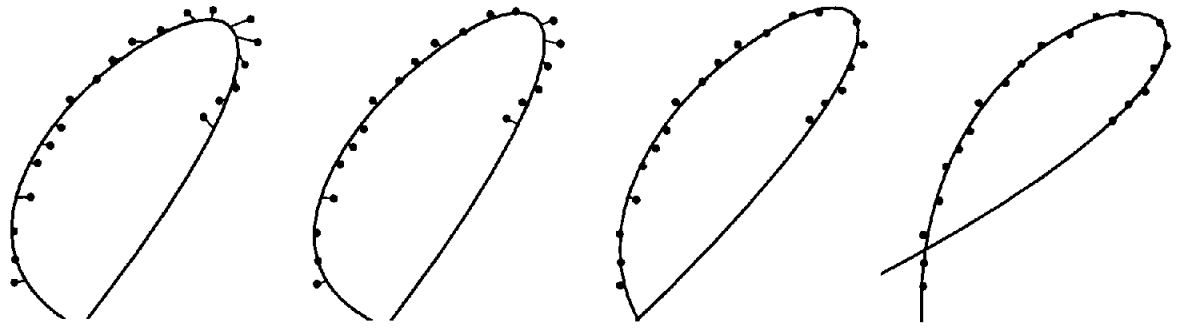
Because Newton-Raphson iteration converges quickly, only a few steps are needed to find a close approximation to the root. In fact, the algorithm performs well when only one iteration step is used to make the adjustment.

After all the t_i have been adjusted, it is important that they still lie in the range 0 to 1. Therefore, we linearly scale the values before each re-calculation of X and Y.

Figure 2 shows an example of the algorithm in operation. The small dots mark the sample points, and lines have been drawn from each data point (x_i, y_i) to the corresponding point $(X(t_i), Y(t_i))$ on the curve. When the algorithm has converged, it can be seen that the point $(X(t_i), Y(t_i))$ is the closest point on the curve to (x_i, y_i).

5. Handling continuity constraints.

The algorithm as stated in the previous section is not suitable for fitting smooth, continuous, piecewise functions because we cannot easily constrain the endpoints and tangents for each cubic piece. To understand how to extend this method, we must look at least-squares curve fitting in some more detail.



Initial fit. 1 iteration. 10 iterations. 100 iterations.

Figure 2. This figure shows several iterations of the curve-fitting algorithm. The sample points are shown as small dots, which are connected by straight lines to the points on the curve with the corresponding value of t. A portion of the curve outside the region $0 \leq t \leq 1$ is included give a better feel for the overall shape of the cubic.

Fitting a polynomial of one variable using least squares curve fitting means the following: given a set of data points $\{(x_i, y_i), 1 \leq i \leq n\}$, find the polynomial $p(x)$ that minimizes the sum

$$\sum_{i=1}^{n} (p(x_i) - y_i)^2 \tag{6}$$

This problem has a unique solution that can be derived as follows. Let the polynomial $p(x)$ be expressed as a linear combination of polynomial basis functions

$$\sum_{j=0}^{d} a_j \phi_j(x) \tag{7}$$

The problem at hand is to determine $A = \langle a_0, a_1, \ldots a_d \rangle$ such that the function

$$S = \sum_{i=1}^{n} \left(\sum_{j=0}^{d} a_j \phi_j(x_i) - y_i \right)^2 \tag{8}$$

is minimized. To do this we take the partial derivatives of S with respect to each value of A and set them to zero to obtain a set of $d+1$ linear equations.

The partial derivatives of S are given by

$$\frac{\partial S}{\partial a_k} = \sum_{i=1}^{n} 2 \left(\sum_{j=0}^{d} a_j \phi_j(x_i) - y_i \right) \phi_k(x_i), \tag{9}$$

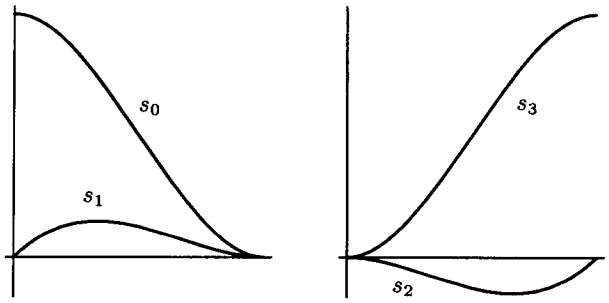
and by setting these to zero and rearranging, we obtain the linear system

$$\sum_{j=0}^{d} a_j \sum_{i=1}^{n} \phi_j(x_i)\phi_k(x_i) = \sum_{i=1}^{n} y_i \phi_k(x_i), \quad 0 \leq k \leq d. \tag{10}$$

This linear system may be easily solved, for example by using Gaussian elimination. Furthermore, for a small linear system like this, it is seldom necessary to worry about numerical instability problems.

The basis functions $\phi_j = x^j$, where $j = \langle 0, 1, 2, 3 \rangle$ are typically used for cubic polynomials. However, the values of A multiplying these basis functions have no intuitive meaning with respect to the shape of the curve. Since any set of linearly independent polynomials would work as basis functions, consider what happens if we use instead the the Hermite polynomials $\langle s_0, s_1, s_2, s_3 \rangle$ shown in Figure 3. These polynomials have the property that if you combine scalar multiples of them to produce a curve, the values of the position and tangent at each endpoint of the curve are each controlled by exactly one of the basis functions. The values of A are just scalar multiples for the basis functions, so this means that by setting an element of A to some fixed value, we can predefine the endpoint or tangent of the resulting curve.

Stated more formally: We may treat just a subset of $\{a_0, a_1, \ldots, a_d\}$ as variables, and the rest as constants. To simplify the notation, rename the basis functions as needed so that $\{a_0, a_1, \ldots, a_{m-1}\}$ are the free variables,



$s_0(t) = 2t^3 - 3t^2 + 1$ $\quad$ $s_2(t) = t^3 - t^2$

$s_1(t) = t^3 - 2t^2 + t$ $\quad$ $s_3(t) = -2t^3 + 3t^2$

Figure 3. The Hermite basis functions.

and $\{a_m, a_{m+1}, \ldots, a_d\}$ are the constants. Then equation 10 becomes, for $0 \leq k < m$,

$$\sum_{j=0}^{m-1} a_j \sum_{i=1}^{n} \phi_j(x_i)\phi_k(x_i) = \sum_{i=1}^{n} \left(y_i - \sum_{j=m}^{d} a_j \phi_j(x_i) \right) \phi_k(x_i) \tag{11}$$

which specifies an $m \times m$ linear system.

This means that by using the Hermite basis in our least-squares fitting, we can preset the endpoints of the resulting cubic. The tangents in the above discussion, however, are the slopes of X or Y with respect to t. If we try to fix the slope of our curve by fixing those two values, the resulting curve is overconstrained for our purposes. The only value we need to fix is the slope of Y with respect to X; that is, the slope of the curve as projected on the X-Y plane. The length of this vector can be left free and used to improve the fit. We would like a representation where it is possible to have the length of this tangent be a variable in A.

As long as we are fitting the x and y components of the parametric curve independently, we cannot design our basis functions such that a single value of A controls the length of the tangent vector in the X-Y plane. The solution is to fit both x and y simultaneously by specifying the curve as a linear combination of basis functions that are vector-valued functions of the variable t. In other words, $\mathbf{F}(t_i) = (x, y)$. Formally, given a sequence of data points $\{\mathbf{z}_i \in \mathbb{R}^2 \mid 1 \leq i \leq n\}$, a corresponding sequence of parameters $\{t_i \in \mathbb{R} \mid 1 \leq i \leq n\}$, a set of linearly independent basis functions $\{\Phi_j : \mathbb{R} \to \mathbb{R}^2 \mid 0 \leq j \leq d\}$, and real numbers $\{a_m, a_{m+1}, \ldots, a_d\}$, we will find real numbers $\{a_0, a_1, \ldots, a_{m-1}\}$ that minimize the function

$$S = \sum_{i=1}^{n} \left(\sum_{j=0}^{d} a_j \Phi_j(t_i) - \mathbf{z}_i \right)^2 \tag{12}$$

The square denotes the dot product of the argument with itself.

For each point $\mathbf{z}_i$ we are computing the square of the magnitude of the vector between the point and the value of the curve at t_i. In other words, once $\mathbf{F}(t_i)$ is the closest point on the curve to $\mathbf{z}_i$, we are minimizing the projected

distance between the data point and the curve, just as we wanted. As before, however, we don't know which values of t are the best to use in our minimization. If we think of each data point as a point in space at $(\mathbf{z}_x, \mathbf{z}_y, t)$, the inner term in equation 12 is just the square of the distance between the curve and the sample point computed in the plane $t = t_i$. Using this distance in our algorithm is equivalent to solving for X and Y independently. At each t_i, we are now computing $(\mathbf{F}_x(t_i) - \mathbf{z}_x)^2 + (\mathbf{F}_y(t_i) - \mathbf{z}_y)^2$. Before, we were computing for X the value $(X(t_i) - \mathbf{z}_x)^2$ and for Y similarly. Since

$$\sum_{i=1}^{n}(\mathbf{F}(t_i) - \mathbf{z}_i)^2 = \sum_{i=1}^{n}(X(t_i) - x_i)^2 + \sum_{i=1}^{n}(Y(t_i) - y_i)^2$$

if $X(t_i) = \mathbf{F}_x(t_i)$ and $Y(t_i) = \mathbf{F}_y(t_i)$ the results will be identical.

The derivation goes through as before, and the resulting linear system is

$$\sum_{j=0}^{m-1} a_j \sum_{i=1}^{n} \Phi_j(t_i)\cdot\Phi_k(t_i) = \sum_{i=1}^{n}\left(\mathbf{z}_i - \sum_{j=m}^{d} a_j\Phi_j(t_i)\right)\cdot\Phi_k(t_i) \tag{13}$$

for $0 \le k < m$.

Once again we will use basis functions that are combinations of the Hermite polynomials. Each basis function will be of the form

$$\Phi(t) = (q(t), r(t))$$

where q and r are scalar multiples of Hermite polynomials. There are up to 8 basis functions, one for each degree of freedom in the curve. For example, if we choose the functions

$$\Phi_j(t) = (s_j(t), 0); \Phi_{j+3}(t) = (0, s_j(t)), \quad j = \langle 0, 1, 2, 3\rangle$$

we get the same results as solving X and Y independently using the Hermite basis functions.

To pick a basis function such that direction of the tangent in the X-Y plane is constrained to a value, for example, (ξ, υ) at $t = 0$, we use a function of the form

$$\Phi(t) = (\xi s_1(t), \upsilon s_1(t))$$

where $s_1(t)$ is the Hermite polynomial that has a slope of 1 at $t = 0$. If we pick the rest of our basis functions such there is no other contribution to the slope at that point, the slope of the final curve is guaranteed to lie along the specified vector.

Let us give a complete example. Suppose we have the data points as shown in Figure 4(a), along with the desired position of the endpoints (x_0, y_0), (x_1, y_1) and the desired tangent direction (ξ, υ) at $t = 1$. The basis function that will control the tangent is $\Phi_0(t) = (\xi s_2(t), \upsilon s_2(t))$ and its associated free variable is a_0. To leave the other tangent free we need two basis functions: $\Phi_1(t) = (s_1(t), 0)$, $\Phi_2(t) = (0, s_1(t))$ and two free variables: a_1, a_2.

To fix the endpoint positions we use

$$\Phi_3(t) = (x_0 s_0(t), y_0 s_0(t)), \quad \Phi_4(t) = (x_1 s_3(t), y_1 s_3(t))$$

and preset the values of a_3 and a_4 to be 1. We now have 5 basis functions, 3 values of A that will vary and 2 fixed values of A that specify the solution for this example. There are 8 degrees of freedom in a parametric cubic piece, but since our goal is to restrict the shape of the curve that can result, we should not be surprised that we end up with fewer than 8 basis functions.

In this example, we have a 3×3 linear system to solve at each iteration of the general algorithm. The values of the t_i are still adjusted at each step, in accordance with equation 5. However, it is no longer necessary to rescale the t_i to lie between 0 and 1, because those extra degrees of freedom have already been specified by fixing the positions of the endpoints. When the algorithm is run, the result is the curve shown in Figure 4(b). To show that the tangent vector at (x_1, y_1) is really parallel to (ξ, υ), we can calculate this vector at $t = 1$ as

$$\sum_{j=0}^{d} a_j\Phi_j'(1) = a_0(\xi, \upsilon) \tag{14}$$

The value of $a_j\Phi_j'(1)$ is $(0, 0)$ for $j \neq 0$ because either the slope of the Hermite polynomial is 0 at $t = 1$, or the corresponding $a_j = 0$.

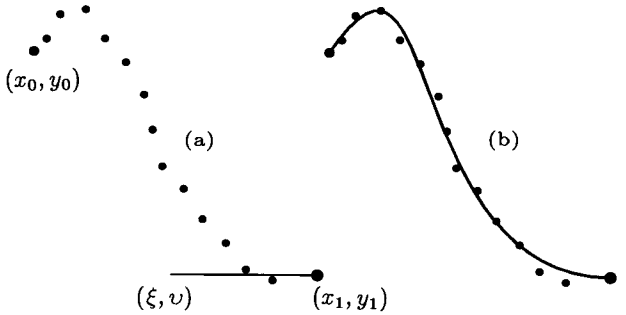


Figure 4. (a) Data points with both end positions and the tangent vector at $t = 1$ specified. The tangent direction is (ξ, υ). (b) The result of applying the fitting algorithm to (a).

Clearly this method of specifying parametric curves is not restricted to basis functions that are combinations of Hermite cubic polynomials—any set of linearly independent twice-differentiable functions will serve as well. If quadratic basis functions are substituted for the cubic ones, least-squares fitting of parabolas may be obtained. Circles or ellipses may be fit by using an appropriate set of trigonometric basis functions.

It is interesting to note how this representation for piecewise cubic polynomials relates to more common specifications, specifically Bézier curves and B-splines. In these representations, the variables are vector values (often called *control points*) which are multiplied by real-valued functions of t to fix the shape of the curve. Such a specification can be converted to a scheme of vector-valued basis function and real-valued coefficients as follows: let the control points be specified as $\mathbf{a}_j = a_{1j}\mathbf{b}_{1j} + a_{2j}\mathbf{b}_{2j}$, each control point having its own coordinate system with basis $\{\mathbf{b}_{1j}, \mathbf{b}_{2j}\}$. Furthermore, let the original basis functions be $\{\phi_j(t)\}$. Then the parametric curve is

$$\sum_j \mathbf{a}_j\phi_j(t) = \sum_j (a_{1j}\mathbf{b}_{1j} + a_{2j}\mathbf{b}_{2j})\phi_j(t) = \sum_j (a_{1j}(\mathbf{b}_{1j}\phi_j(t)) + a_{2j}(\mathbf{b}_{2j}\phi_j(t))) \quad (15)$$

so the appropriate set of vector-valued basis functions is $\{\mathbf{b}_{ij}\phi_j(t) \mid i = 1,2\}$. This means we can use this algorithm to solve for cubic polynomials with B-spline or Bernstein basis functions, and we can preset linear constraints on the control points for the resulting curve. Since each control point has been given its own coordinate system, by appropriate choice of the coordinate systems and the free coefficients, each control point can be constrained to be on a particular point or line, or to be free to move anywhere in the plane.

6. Relation to previous work.

There is a wealth of literature on piecewise polynomial functions or *splines*, and much of it is available in textbook form [10,21,24,25]. Spline functions are often used to approximate more complicated functions because they are computationally simpler. Much of the spline literature involves methods for smoothly interpolating a small set of data points. This is similar to the problem of designing with a small number of interpolating points. The applications that are relevant to our problem are those which try to fit a large set of data points with a small number of pieces. The first reference to this problem we can find is by Rice [21]. The paper by Cox [9] gives a nice description of the problem of curve fitting with splines using the least-squares error metric. He also mentions using dynamic programming as the method for finding knots, but does not extend it to spline functions with constraints. Some more recent publications that discuss curve fitting with splines are listed in the references [7,11,14,19,22].

There are two significant issues that separate the problem of fitting shapes from the problem of function approximation. First, shapes are usually represented as *parametric* functions to make the representation independent of the choice of axis. The usual technique in computer graphics applications is to use two functions, X and Y, each of which is a piecewise polynomial function of the parameter t. By assigning some value of t to each of the data points, it is possible to extend methods developed for spline functions to parametric spline functions by treating $X(t)$ and $Y(t)$ independently [11,20]. However, the shape of the curve can be dramatically influenced by the choice of parameterization (figure 5). The second issue is that the continuity constraints at the knots for shapes are quite relaxed by function fitting standards. A polynomial spline function of order k generally has $k-1$ continuous derivatives at the knots. This is not necessary, or in most cases, even desirable for two-dimensional shapes. If we want the most efficient representation, clearly we must not constrain the curve unnecessarily. A more complete discussion of continuity conditions is available in the literature [17,18].

Flegal developed a method for fitting curves to hand-drawn sketches at Xerox PARC in 1974 [13]. In his thesis, Reeves [20] presents and compares several techniques, including Flegal's algorithm, for fitting piecewise parametric cubic polynomial curves to hand-drawn sketches. The focus is on algorithms that are computationally efficient enough to give real-time feedback in a sketching system, so in general the techniques tend to sacrifice output quality for speed. For many of our purposes, especially for deriving analytical representations for digital typefaces, we are willing to trade computational speed for better results.

There are two published algorithms which take a set of samples and produce the B-spline control points for the approximating curve for use in an interactive environment [28,29]. These algorithms create a representation that has the knots evenly spaced with respect to the parameter, which is not the most efficient representation for our purposes. Also, these techniques include a significant amount of user intervention to achieve the final shape. We want to minimize the amount of user interaction necessary.

7. Conclusions.

We have presented a method for fitting shapes defined by a discrete set of data points with a parametric piecewise cubic polynomial curve. Our goal is to give an efficient representation that "looks good" for graphics arts applications. The two new techniques we have introduced are dynamic programming for determining the knot positions, and an iterative method for fitting a parametric cubic with optional endpoint and tangent vector constraints to a set of data points.

Most of our work to date has been with letter-form shapes. One of the goals of our laboratory is to have a uniform, resolution-independent font representation for our digital printers. Many of the fonts we would like to use are currently defined only as a bitmaps. Therefore, we have been motivated to apply our system to convert fonts

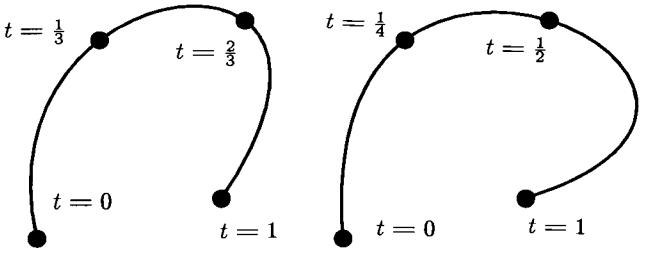


Figure 5. The curves are each defined by the same four data points. The only difference is the values of the parameter associated with the inner points. By comparing the two curves it is easy to see that the shape of the curve can be dramatically influenced by the choice of parameterization.

from bitmaps to curve outlines. Figure 1 and figure 6 are examples of data derived from such bitmaps.

Figure 7 shows an example of a more general graphics arts application. We have also used our methods on very simple hand-drawn sketches. The data points in figures 1 and 2 are hand-drawn, for example.

8. Future work.

There are several ways in which we would like to improve our method, a few of which are worth mentioning here.

We do not completely understand the convergence properties of our algorithm, especially why it sometimes

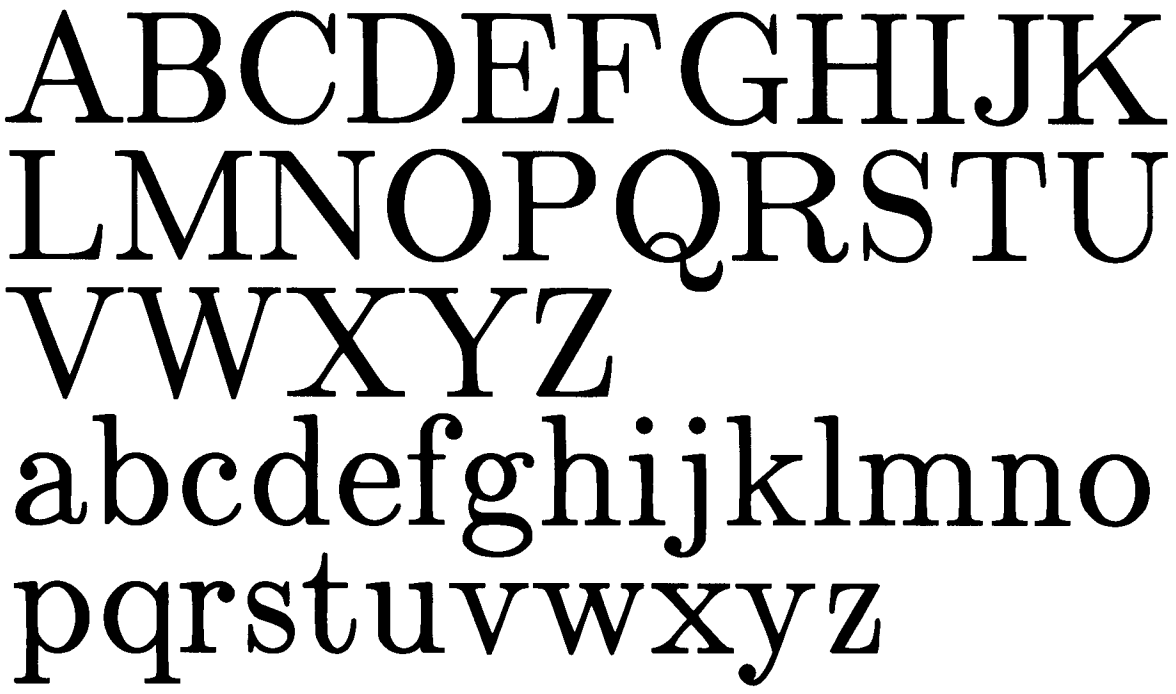


Figure 6. This font was converted from a bitmap (160 pixels per em) which was originally produced using Metafont[15]. Once the tolerance parameters were adjusted, no further user intervention was required to produce the cubic representation.

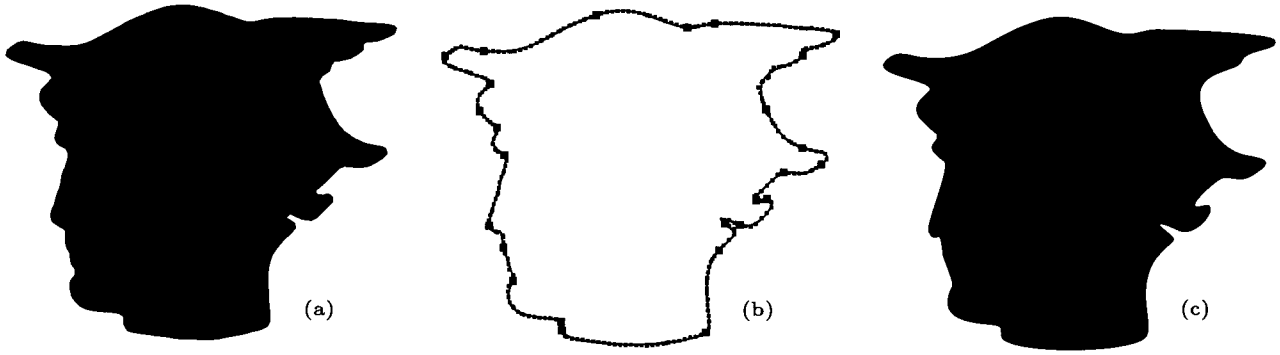


Figure 7. The data for this sample comes scanning a 3/4 inch high original with a 95 line/inch scanner. Some detail was lost in this process, especially with respect to corners. The image was fit using a loose tolerance to achieve further smoothing. (a) The original image, approximately 72 by 72. (b) The sample points and the result of the curve-fitting process. (c) Final curve in more detail, showing the curve, the knots and the samples.

does not converge at all. While our system can easily reject these cases as part of the fitting process (because they are simply instances of a very bad fit), we would like to avoid them. The type of iteration method we use is suitable for convergence acceleration techniques. However, we observe that these methods decrease the stability of the algorithm even further. Further understanding of these properties, especially with respect to the initial approximation for the parameter values, should lead to a more efficient and stable algorithm.

We have chosen a method for fitting piecewise functions that separates somewhat the problems of finding the knots and fitting the curve for each piece. This is not the only way to solve this problem. It is possible to set up a global system that does a least-squares fit for an entire spline [9,11,22]. Then, on each iteration we can adjust the values for t or adjust the knot locations or both. It is easiest to think of solving X and Y separately in this case, although it is also possible to extend the idea of vector basis functions to spline curves. The problem with this approach is that it requires the solution of a large linear system, proportional to the number of pieces, during each iteration. Our experience has been that this quickly becomes too expensive to be practical. Also, it is not clear what is the best way to adjust the number and location of the knots each time. However, a good algorithm of this sort might converge to a globally better solution.

Both the iteration technique and the dynamic programming application presented in this paper can be computationally expensive, and some applications do not need the full power of these algorithms to get acceptable results. We have developed a system for experimenting with methods for fitting shapes. In the future, we would like to get a better understanding of how to parameterize the application-dependent aspects of this problem.

9. Acknowledgements.

We would like to thank John Warnock, Robin Forrest and Lyle Ramshaw for their contributions to this project and for their many helpful comments on this paper.

10. References.

1. Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

2. Barnhill, R.E. and Riesenfeld, R.F., eds., *Computer Aided Geometric Design*, Academic Press, New York, 1974.

3. Baudelaire, P. and Stone, M., "Techniques for Interactive Raster Graphics." *Computer Graphics 14,* 3 (1980), 302-307.

4. Baudelaire, P., *The Draw User's Manual*, internal report. Xerox Palo Alto Research Center, Palo Alto, CA, 1978.

5. Baudelaire, P., *The Fred User's Manual*, internal report. Xerox Palo Alto Research Center, Palo Alto, CA, 1976.

6. Bellman, R., *Dynamic Programming*, University Press, Princeton, N.J., 1957.

7. Chung, W.L., "Automatic Curve Fitting Using and Adaptive Local Algorithm." *ACM Transactions on Mathematical Software 6,* 1 (1980), 45-57.

8. Conte, S.D., and de Boor, C., *Elementary Numerical Analysis: An Algorithmic Approach*, second edition, McGraw-Hill, New York, 1972.

9. Cox, M.G., "Curve Fitting with Piecewise Polynomials." *J. Inst. Maths Applications 8,* (1971), 36-52.

10. de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

11. Dierckx, P., "Algorithms for Smoothing Data with Periodic and Parametric Splines." *Computer Graphics and Image Processing 20,* (1982), 171-184.

12. Faux, I.D., and Pratt, M.J., *Computational Geometry for Design and Manufacture*, Ellis Horwood, 1979.

13. Flegal, R., private communication. Xerox Palo Alto Research Center, Palo Alto, CA, 1978.

14. Ichida, K and Yoshimoto, F., "Curve Fitting by a One-Pass Method with a Piecewise Cubic Polynomial." *ACM Transactions on Mathematical Software 3,* 2 (1977), 164-174.

15. Knuth, D.E., *TEX and Metafont: New Directions in Typesetting*, Digital Press and American Mathematical Society, 1979.

16. Lipkie, D.E., Evans, S.R., Newlin, J.K., and Weissman, R.L., "Star Graphics: An Object-Oriented Implementation." *Computer Graphics 16,* 3 (1982), 115-124.

17. Manning, J.R., "Continuity Conditions for Spline Curves." *Computer Journal 17,* (1974), 181-186.

18. Nielson, G.M. "Some Piecewise Polynomial Alternatives to Splines in Tension" in *Computer Aided Geometric Design*, Barnhill, R.E. and Riesenfeld, R.F., eds., Academic Press, New York, 1974.

19. Pavlidis, T., *Algorithms for Graphics and Image Processing*, Computer Science Press, 1982.

20. Reeves, W.T., "Quantitative Representation of Complex Dynamic Shapes for Motion Analysis." PhD Thesis, Department of Computer Science, University of Toronto, 1980.

21. Rice, J.R., *The Approximation Of Functions*, vols. 1 and 2, Addison-Wesley, Reading, Mass., 1964.

22. Rice, J.R., Algorithm 525. ADAPT, adaptive smooth curve fitting, *ACM Transactions on Mathematical Software 4,* 1 (1978) 82-94.

23. Rogers, D.F., and Adams, J.A., *Mathematical Elements for Computer Graphics*, McGraw-Hill, New York, 1976.

24. Schultz, M., *Spline Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

25. Schumaker, Larry L., *Spline Functions: Basic Theory*, John Wiley & Sons, New York, 1981.

26. Stone, M., *The Griffin User's Manual*, internal report. Xerox Palo Alto Research Center, Palo Alto, CA, 1979.

27. Warnock, J. and Wyatt, D., "A Device Independent Graphics Imaging Model for use with Raster Devices." *Computer Graphics 16,* 3 (1982), 313-320.

28. Wu, S., Abel, J.F., and Greenburg, D.P., "An Interactive Computer Graphics Approach to Surface Representation." *CACM 20,* 10 (1977), 703-712.

29. Yamaguchi, F., "A New Curve Fitting Method Using a CRT Computer Display." *Computer Graphics and Image Processing,* 7 (1978), 425-437.