Phoenix:
An Interactive Curve Design System
Based on the
Automatic Fitting of Hand-Sketched Curves

by

Philip J. Schneider

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

University of Washington

1988

Approved by____

Anthony D. DeRose (Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree____

Date____

Master's Thesis

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature______________________________

Date__________________________________

University of Washington

Abstract

Phoenix: An Interactive Curve Design System Based on the Automatic Fitting of Hand-Sketched Curves

by Philip J. Schneider

Chairperson of the Supervisory Committee: Professor Anthony D. DeRose
Department of Computer Science

Previous methods for the approximation of digitized curves with analytic curves are discussed. A new algorithm , which fits a piecewise cubic Bernstein-Bezier polynomial curve to digitized curves, is presented. This adaptive algorithm generates a curve with first-order geometric continuity (G^1), within an error bound specified by the user, and generates an interpolating curve in the case of exact fit. The use of this algorithm in an interactive curve design system is discussed. Quantitative and qualitative evaluations of the method are presented, using examples generated by the current implementation of the design system.

Contents

List of Figures

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the many people who have supported me in my research efforts:

Professor Tony DeRose, my thesis advisor and mentor, whose contribution to the research efforts resulting in this thesis could not possibly be overstated. Tony is also greatly responsible for the interesting, challenging, and dynamic research environment in which I had the great privelege to work.

Professor Kenneth Sloan, the second member of my thesis committee, for his "no-punches-pulled" approach to offering opinions and answering questions.

My colleagues in the Graphics/Artificial Intelligence Laboratory – Jamie Painter, Charles Loop, Ron Blanford, Daniel O'Donnell, for their friendship, comeraderie, support, advice, and criticism, and for helping me when the mathematics got tough.

My other friends and colleagues in the Department of Computer Science, for their friendship and making me feel like I belonged. In particular, I would like to thank Peter and Anne Damron, Marianne Mueller, Erik Brisson, Simon Kahan, and all the other folks who have put up with my trials and tribulations over the past year.

Allen Akin and the WSE Group at Digital Equipment Corporation, for providing me with the hardware and time I needed to finish this thesis.

My mother, brother, and sister, for providing much of the support and encouragement that I needed in doing this research.

And finally, and most importantly, Katherine Snyder, for being the reason I did all this.

For Kathy, with all my love

Chapter 1

Introduction

1.1 The Problem

Computer-aided geometric design (CAGD) is an application of interactive computer graphics techniques to the design of curves and surfaces. Computer-generated images are used to help design automobiles, ships, airplanes, mechanical parts, etc. CAGD techniques can in many respects be viewed as a modern replacement for the draftsman's paper, pencils, french curves, and the like.

When a designer begins to use an interactive shape-modelling system, he uses an iterative process: he carefully observes the shape he has specified, and decides on changes he must make in order to make the generated shape conform to his mental image of that shape. These changes must be communicated to the computer, within the constraints imposed by the representation of the shape and the parameters available via the user interface. Many of the representation schemes used today are well-understood (e.g., the use of straight lines and circular arcs in designing mechanical parts), but the process of *communicating* shapes and changes to those shapes is perhaps less well-understood.

As this thesis is concerned with two-dimensional curve design, we will address ourselves to this issue only from here on. In conventional CAGD systems, the designer communicates the initial shape of a curve by specifying *control points* interactively (generally with a mouse or digitizing tablet). The computer then generates a curve whose shape depends

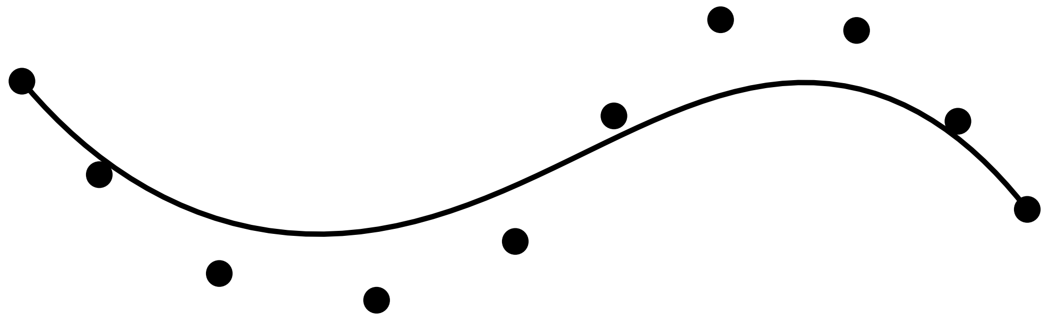


Figure 1.1: Example of a control-point based curve

on the control points. Such a curve may pass through (*interpolate*) each point, or may not, depending on the mathematical formulation defining the curve's shape. The shape of the curve depends, then, on the placement of the control points and the mathematical equations that relate the curve to the points. See Figure 1.1 for an example of such a curve.

Changes to a curve specified in this manner are generally communicated by the designer by his moving the control points interactively, thereby changing the shape of the displayed curve. In some systems, the designer may be able to add new control points, again changing the shape of the curve. These techniques have been the subject of development sufficient to render them generally quite useful for the design of free-form curves.

These techniques are not *completely* non-intuitive, as the generated shapes and changes to them do behave (for the most part) in a consistent and predictable manner. However, there are several drawbacks: first, the characteristics of the curves and the changes are dependent on the underlying representation. This fact has several effects – the user may have to be familiar with the mathematics of the system in order to use it efficiently, the user's intuition may not match that of the designer of the system, some changes may be difficult to make, and so forth. In particular, consider a system that allows the user to interactively move points that lie on the line – as the user tugs on a particular control point, the shape of the curve in the area of the control point changes to follow the moving point. However, the apparent "tension" on the curve as the point is pulled, and the extent of the curve so changed is dependent on the underlying representation, and thus the curve may not move exactly as the designer desires. Consider an artist or draftsman drawing a curve: he or she picks up his pencil, and sketches a first attempt; to change the shape of the curve, he or she inverts the pencil and uses the eraser to obliterate a section of the curve, and draws in that region again. This process is repeated until the curve he has drawn matches his mind's-eye view of the curve. This thesis describes the development of techniques that are intended to simulate this more natural and intuitive process.

We do not wish to abandon the idea of an underlying mathematical representation, for there are numerous advantages of having such a representation. Some of these will be discussed in a later section. Our problem is two-fold:

1. How can we find a mathematical representation of an arbitrary curve that faithfully fits the curve sketched in by the designer?

2. How can we allow the designer to modify such a curve in a natural way?

1.2 Applications

Solutions to the problem of fitting an analytic curve to a digitized (sketched or otherwise sampled) curve in general and two-dimensional curve design systems have a wide variety of applications:

- In free-form drawing systems, such as MacDraw, or Adobe Illustrator.

- In the design of cross-sections or silhouettes of three-dimensional objects, such as an airplane wing.

- In the design of fonts for typographic systems, and for converting bit-mapped font representations to analytic representations (e.g. $\TeX$).
- In the generation of curves which may be swept through space or rotated around an axis, to generate surfaces.

1.3 Goals

The goals of the system described in this thesis are:

1. Intuitive interface: a major goal is simplicity – we wish to develop a methodology that allows the designer to create curves in as natural a fashion as possible. We wish to insulate the user from the control points, and other manifestations of the mathematical representation.

2. Error control: the designer should be able to control the closeness of fit between his sketch and the generated analytic curve, converging to interpolation in the limit of zero error.

3. Mathematical (analytic) representation: as mentioned earlier, there are many motivations for retaining an underlying mathematical representation for the curves, including:

   (a) Data reduction – A digitized curve drawn by a designer may consist of a sequence of hundreds, or even or thousands, of data values, each being the (x, y) coordinates of a point in the Cartesian plane. A mathematical representation may allow the same shape information to be specified by a few tens of values. This can be important for rapid display.

   (b) Shape control – We may wish to control the mathematical characteristics of the curves, such as its smoothness. For example, it may be very important to a designer that the cross-sectional curve of an automobile fender have no discontinuities, but discontinuities (such as corners) may be essential in a mechanical part.

(c) Ease of manipulation – Some mathematical representations, such as the parametric polynomials used in many existing systems, including the author's, allow for easy scaling, translations, rotations, etc., of the curve, without affecting the shape of the curve.

(d) Ease of display – very fast display algorithms exist for displaying mathematical curves.

(e) Useful output – CAGD systems are often used to generate data for controlling manufacturing, such as for numerically-controlled lathes. A curve consisting of a sequence of (x, y) coordinates does not adequately specify a continuous, smooth path for a machine tool to follow; many mathematical representations are ideal for this situation.

(f) Resolution independence – the designer should be able to zoom in and out arbitrarily.

4. Robust and general with respect to shape – we would like to be able to create and manipulate a wide variety of shapes within a common framework.

1.4 Proposed Methodology

The system we set out to create was an attempt to simulate the use of the pencil-and-paper approach to designing curves, using a pointing device (a mouse in our case) as a pencil, and a CRT screen as the paper. The designer should be able to sketch out a line for his "first cut", then have an analytic curve *automatically* fitted to it within a user-specified tolerance. The designer should be able to the modify the curve by erasing a portion and redrawing that portion repeatedly, until he is satisfied with the shape of the curve. As we shall see, this approach was quite successful.

1.5 A Detailed Look at One Session

Figures 1.2 through 1.9 were generated by **Phoenix**, the author's prototype curve design system, and show one "iteration" of the design process. The curve is sketched out, fitted, and a small section is redrawn. One iteration is shown for the sake of simplicity and brevity, though any part of the curve may be redrawn, irrespective of whether or not it has already been so changed. The figures basically correspond to what the user sees in each step.

1.6 How to read this thesis

Chapter 2 begins by introducing the reader to the basic mathematical principles necessary to appreciate both the relevant previous approaches and the system developed by the author. We shall attempt to not overwhelm the reader with gratuitous mathematical formulae; however, the reader must be familiar with the terminology and mathematical issues in order to understand the design decisions and implementation. In particular, we discuss the relation of our work to conventional interpolation and spline approaches, and the techniques involved in describing and specifying smoothness characteristics.

Chapter 3 gives an historical overview of the previous approaches to the problems of approximating digitized curves with analytic curves and of modifying such curves, briefly describing the characteristics of the various mathematics underlying each representation technique. A comprehensive review and evaluation of the entire history of curve approximation is well beyond the scope of this thesis; our intention is to note the relevant "highlights" so that the reasons for our approach can be understood.

Chapter 4, which describes the Phoenix curve design system, begins with an overview of the solution to the problem of fitting a digitized curve with piecewise parametric curves. Next, each of the algorithms is described in detail, and the design decisions related to each.

Figure 1.2: The original sketched curve

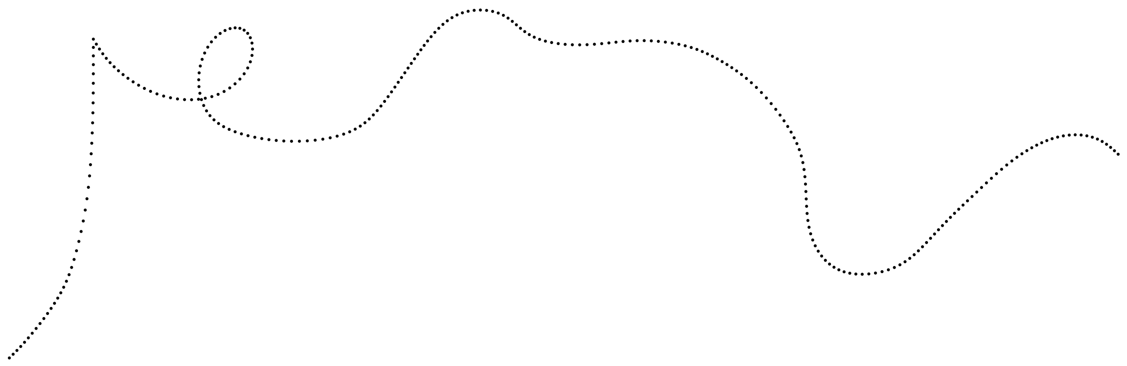


Figure 1.3: The sketched curve after preprocessing

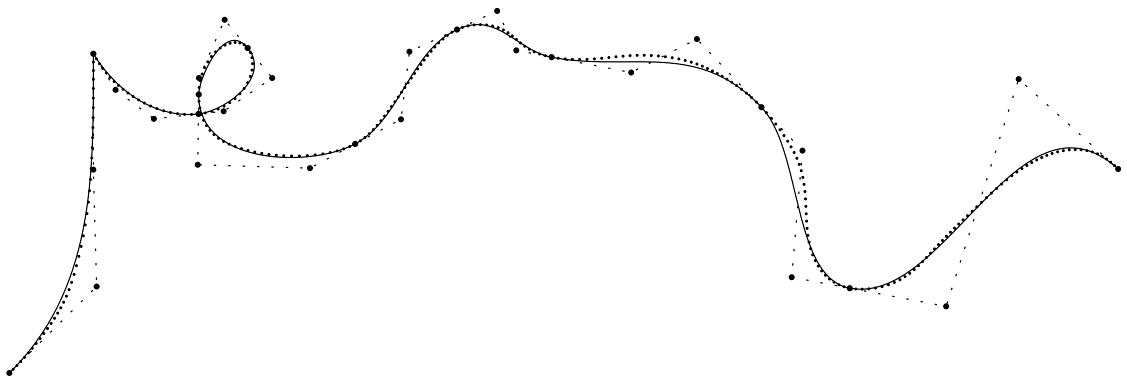


Figure 1.4: Fitted curve, showing the control polygons

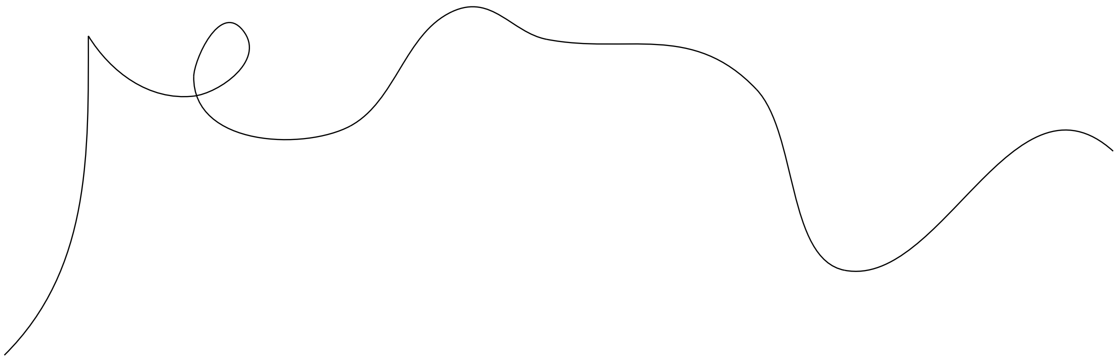


Figure 1.5: The fitted curve

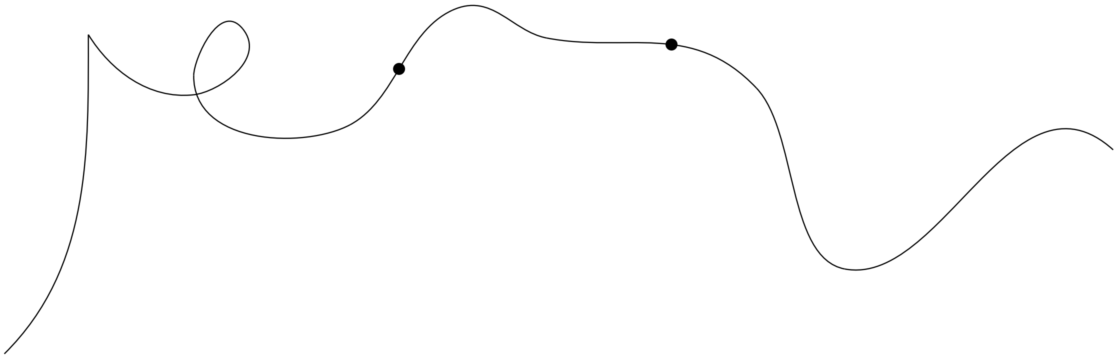


Figure 1.6: The user specifies the area he wishes to redraw

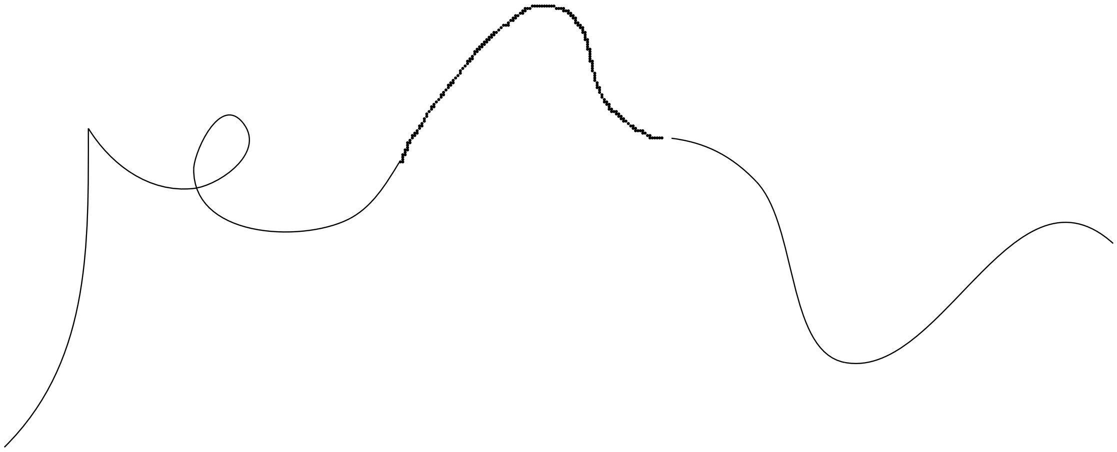


Figure 1.7: The user redraws the erased section

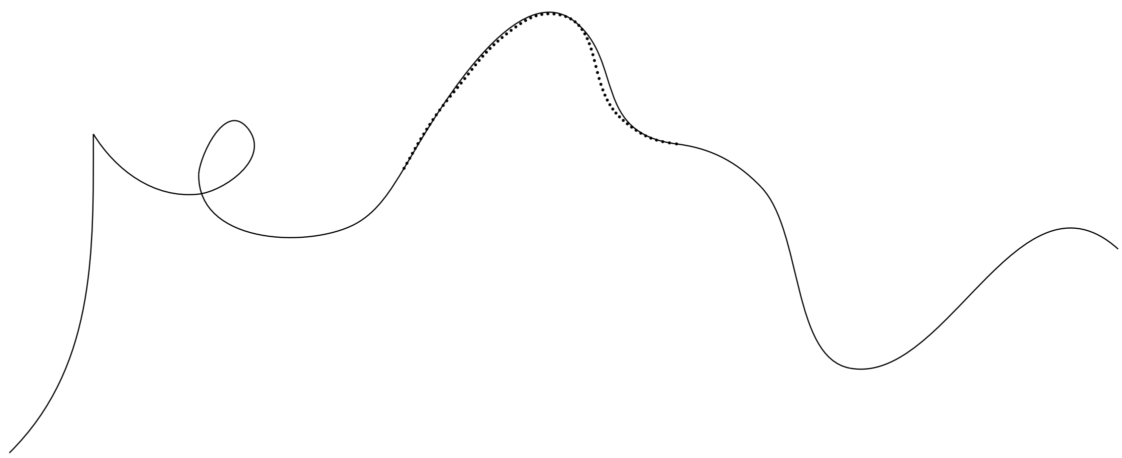


Figure 1.8: The new section is fitted

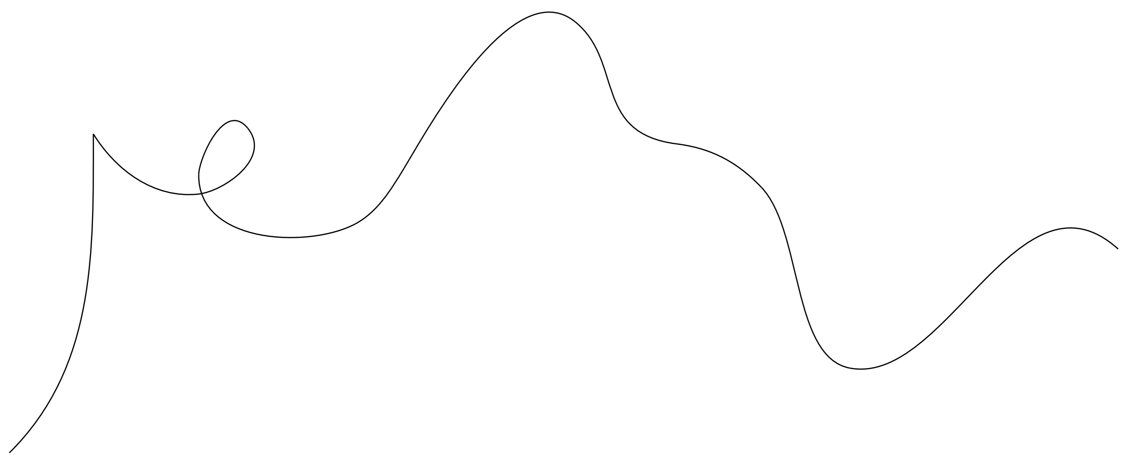


Figure 1.9: The new section is integrated with the entire curve

Chapter 4 concludes with numerous examples taken directly from the system implemented by the author, with quantitative and qualitative evaluations of the system.

Chapter 5 summarizes the work, considers several alternative approaches, and gives some directions for future research in this area.

Chapter 2

Mathematical Background

2.1 Parametric Functions

Much of mathematics is concerned with *functions*, mathematical entities that when given an input, compute some output (e.g. $f(x) = x^2$). As this thesis is concerned with the visual or geometric aspects of functions, we will concentrate on this area. Thus, we consider the *graphs* of functions in the Cartesian plane (the standard (x, y) coordinate system). For example, Figure 2.1 is the graph of the function $f(x) = \sin x$.

Functions such as this are called *scalar* functions; that is, we evaluate *one* input and return a real number. The graphs of such functions cannot loop back on themselves, because the functions are single-valued. As such, scalar functions cannot represent arbitrary plane curves, such as that seen in Figure 2.2. Because of these limitations, the dominant form of functions used in computer graphics is the *parametric* or *vector-valued* function. A point on a curve is represented as a vector (an ordered set of values):

$$P_i = [x_i, y_i]$$

There are corresponding *parametric functions* that may be used to represent arbitrary curves; these are of the form:

$$Q(t) = [x(t), y(t)]$$

The parameter t takes on values from a specified range, conventionally from 0 to 1. The

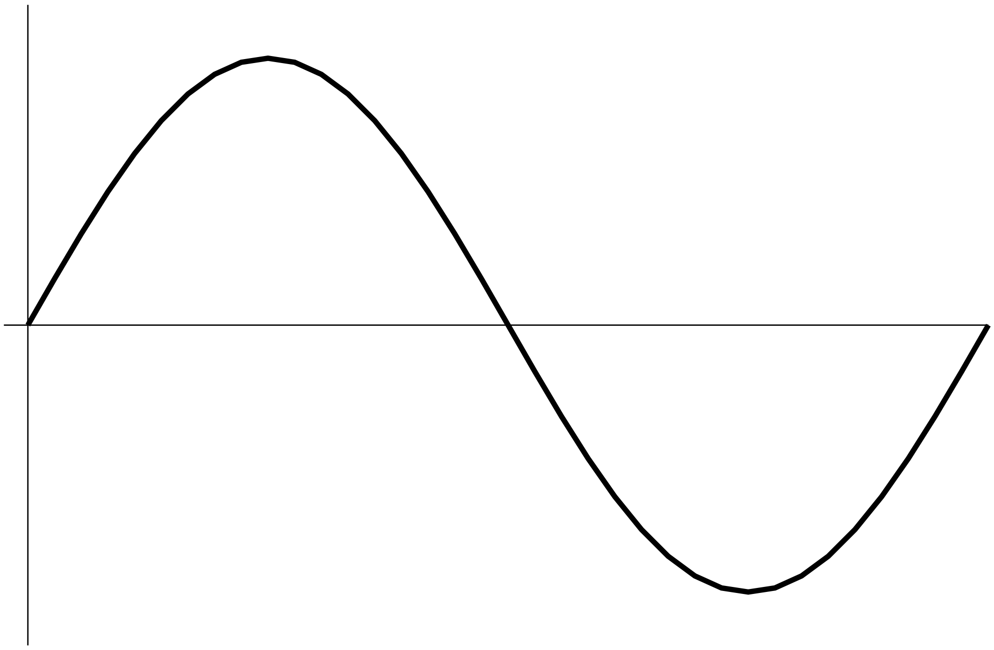


Figure 2.1: Graph of the sine function.

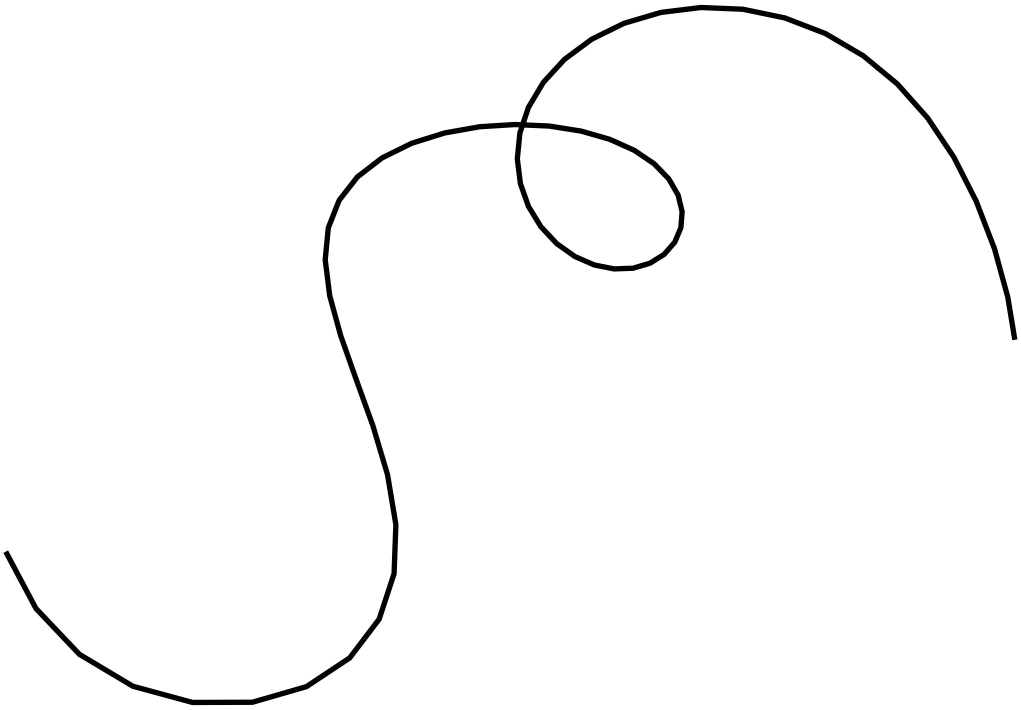


Figure 2.2: An arbitrary planar curve, not representable by a scalar function

more geometrically-minded reader will recognize that a curve represented in this manner may be thought of as the projection of a curve in three-space (the third dimension being t, and the t-axis is perpendicular to the x, y plane) onto the x, y plane. Thus, $x(t)$ and $y(t)$ are scalar functions. This representation allows for multiple-valued curves – there may be many values of y for some or all values of x.

An example of a parametric curve is the circle, which may be represented by the parametric function

$$Q(t) = [\cos(t), \sin(t)].$$

There are other ways of describing curves mathematically. One common representation is by *implicit* equations. For example, the implicit equation for a circle of radius r centered at (x_j, y_j) is

$$(x - x_j)^2 + (y - y_j)^2 - r^2 = 0$$

The mathematical representations of curves we shall be dealing with in the rest of this thesis are parametric functions.

2.2 Piecewise Representations and Continuity Conditions

2.2.1 Piecewise representations

One of the goals of CAGD research in general, and of this thesis in particular, is the development of methodologies for designing "smooth" curves and surfaces. In this section we address the issue of formally defining smoothness. Much of this section is derived from [DeR85].

In general, it is difficult or impossible to devise a single parametric polynomial function to represent an entire arbitrary plane curve. (Polynomials are typically used for efficiency. High-degree polynomials, which would be needed to represent complex curves, are unwieldy.) However, it is not so difficult if the function applies only to a small "well-behaved" portion of a curve. Thus, a complex curve can be represented as a series of functions appropriately pieced together. Such *piecewise* representations generally require

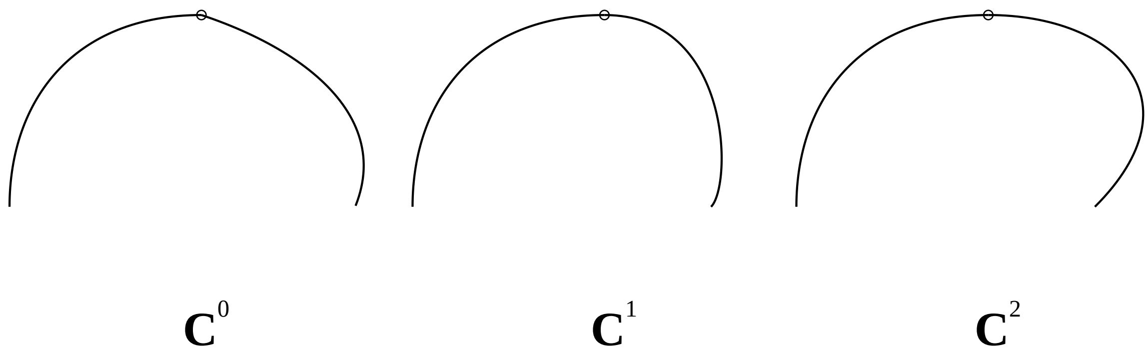


Figure 2.3: Continuity conditions at the joint between two segments

methods to control the behavior of the curve at the *joints* between two pieces or *segments*. Characterizations of the behavior of the curve at the joints are typically given in terms of *continuity* conditions:

- Zero-order continuity (C^0) means that the ends of the two pieces are connected.
- First-order continuity (C^1) means that the tangents at the adjoining ends of the segments are equal, and implies zero-order continuity.
- Second-order continuity (C^2) implies zero-order and first-order continuity, and requires that the curvatures of each segment at the joint are equivalent. Curvature is defined as the inverse of the radius of a circular arc that fits the curves at the joint.

Figure 2.3 shows three similar curves, which are made up of two segments, joined at the small circle.

2.2.2 Parametric Versus Geometric Continuity

Let us "step back" a little and consider parametric curves in general. A curve generated by a parametric function $Q(u)$ may be thought of as the path traced out by a particle travelling in a plane; the parameter u may be thought of as *time* in this analogy [DeR85]. Recall (from calculus) that the first derivative of a function may be thought of as representing the *velocity* (a vector describing both the direction and speed) of such a particle. For parametric curve Q, $Q'(u)$ is a vector whose directional component (dy/dx) indicates the instantaneous direction of the particle at time (i.e. parameter value) u, and whose length (magnitude) is directly proportional to the speed of the particle at time u. A joint will look smooth if the particle's direction does not make a sudden jump, as it does in the C^0 curve in Figure 2.3.

Recall also that the second derivative of a function is the rate of change in velocity, and is commonly called *acceleration*. Geometrically, we can think of this as related to the deviation of the curve from the tangent at a point. If the particle deviates from the tangent line at an increasing rate (i.e., we have an increasing second derivative), then the curvature increases.

Back to parametric curves: Consider two parametric curves $Q_1(u)$ and $Q_2(v)$ that are stitched together as described above. These curves meet with n^{th}-order parametric continuity, denoted C^n, if the first n parametric derivatives match at the joint; that is, if

$$Q_1^{(i)}(1) = Q_2^{(i)}(0), i = 1, \ldots, n.$$

Consider such a situation where $n = 1$ (i.e., C^1 continuity): we know that the particle tracing out these two segments will exhibit smoothness in both direction and speed. In this case, we say that there are no first-order *discontinuities*. If we look at such a curve, we see only the shape so generated, which may have been just as well traced out by a particle moving at constant speed, or widely varying speed. Thus, if we are only interested in the shape of the curve, this C^1 continuity places an unnecessary restriction on what we intuitively think of as "smoothness" in a visual sense. (Note that in some applications, smooth movement of the particle may be indeed quite important – consider a curve describing a camera path in an animation system.) Because of this, a less restrictive definition of this visually-oriented continuity has been developed, known as *geometric continuity*. N^{th}-order geometric continuity is denoted G^n. This term was coined by Barsky and Beatty , though the same concept has been independently developed by a number of authors (.e.g., [FP79], [FW66], etc.).

First-order geometric continuity, G^1, can be formally defined as continuity of *unit tangent vectors*. For a curve generated by Q, the unit tangent vector at the point $Q(u)$, denoted $\hat{Q}'(u)$, has the same directional component as $Q'(u)$, but is of unit length. Thus, $\hat{Q}'(u)$ is defined by

$$\hat{Q}'(u) = \frac{Q'(u)}{\mid Q'(u) \mid}$$

Second-order geometric continuity, G^2 is provided by the *osculating circle*, or equivalently, the *curvature vector*. Intuitively, the osculating circle and the curvature vector measure the rate at which the curve deviates from its tangent direction. The curvature vector, denoted $k(u)$, can be expressed in terms of derivatives of $Q(u)$:

$$k(u) = \frac{Q^1(u) \times Q^2(u) \times Q^1(u)}{\mid Q^1(u) \mid^4}$$

The osculating circle is tangent to the curve at $Q(u)$, having a radius that is the reciprocal of the magnitude of the curvature vector, as shown in Figure 2.4 Less formally, the curvature of a curve at a point is related to the size of the circle that best fits the curve at that point.

2.3 Approximation Versus Interpolation

Depending on the mathematical characteristics of the parametric functions used to represent a curve, the curve associated with a set of points may interpolate every point, interpolate just the endpoints, or just approximate the shape defined by the points. Of these two approaches, interpolation has been the subject of much more research than approximation, and has also been much more widely used in scientific applications.

Consider a situation where we have some function $y(x)$ that we wish to evaluate for some values of x, and that we have a table giving $y(x)$ for various values of x. For values

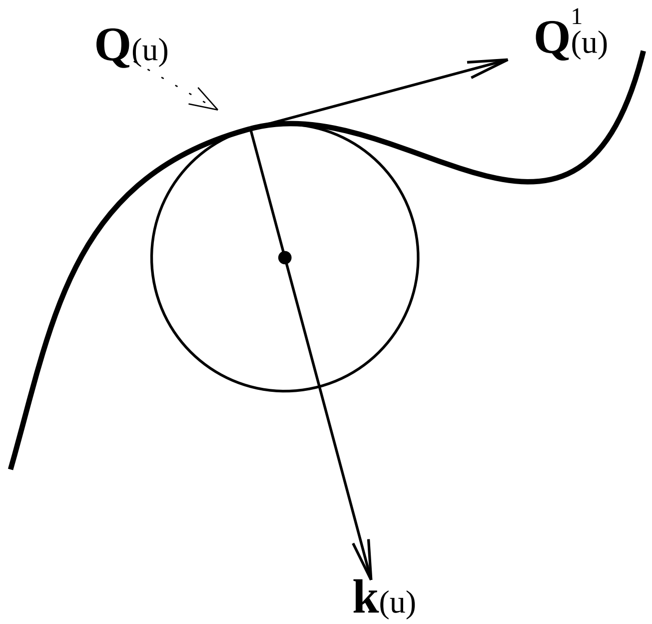


Figure 2.4: The unit tangent vector, the curvature vector, and the osculating circle at a point $Q(u)$.

of x that appear in the table, we simply may look up $y(x)$; for values of x between those that appear as entries in the table, we must devise some scheme for computing $y(x)$ using the information in the table. Then known values may be:

1. Reliable and exact, as in the case of a table of log functions.

2. Of suspect accuracy – they may contain noise, or suffer from statistical or sampling error. This is typically the case for values derived from experimental or other sampling situations.

For situations in which (1) holds, we can construct an *interpolating function* to compute values of $y(x)$. In terms of graphs of functions, an interpolating function corresponds to a curve that passes through the known points $(x, y(x))$ in some smooth fashion. We can then compute $y(x)$ for values of x that are not in the table by evaluating the interpolating function at that x. This technique is particularly useful when the true function is unknown, or difficult or expensive to compute. See any numerical analysis text for a more comprehensive treatment of the classical interpolation techniques, or [FP79] or [BBB87] for discussions of more modern piecewise parametric approaches.

For situations in which (2) holds, consider what would happen if we used an interpolation technique. As the "known" (tabulated) $y(x)$ are of suspect accuracy to begin with, we see that from the start that there may be problems with this approach. Interpolation based on erroneous known values will have the possible effect of amplifying the errors at worst, and of propogating the errors at best. What is really needed is a function that "averages out" or smooths the errors. This idea is based on the common statistical assumption that errors in data are randomly distributed; then, a function/curve that approximates the known values will, on the average, minimize the total errors.

The preceeding logic has lead to what is known as *least-squares* curve-fitting, and a sophisticated variant of this technique is the basis for the author's curve-fitting algorithm. See [FP79] for a detailed explanation of the classical approach. The basic idea for least-squares curve fitting is to derive a function (curve) that minimizes the sum of squared distances from the known values (points) to the fitted curve.

The author has chosen to use an approximating curve to fit to the digitized curve for several reasons. One relates to sampling – not *error*, per se, as we assume that the coordinates of the points are reliable measures of the actual position of the mouse or stylus. In a curve drawing system, a sequence of points may represent any one of an infinite family of actual paths the user's hand may have taken. See Figure 2.5 for an example of several possible actual curves which are represented by the same sequence of digitized points. Note that the severity of the this phenomenon is dependent on the resolution of the devices used. As one of our goals is resolution independence, it is clear we must deal with this problem.

A second justification of the use of approximation is that the user's hand will deviate from the path the user intended, causing the appearance of features in the digitized curve

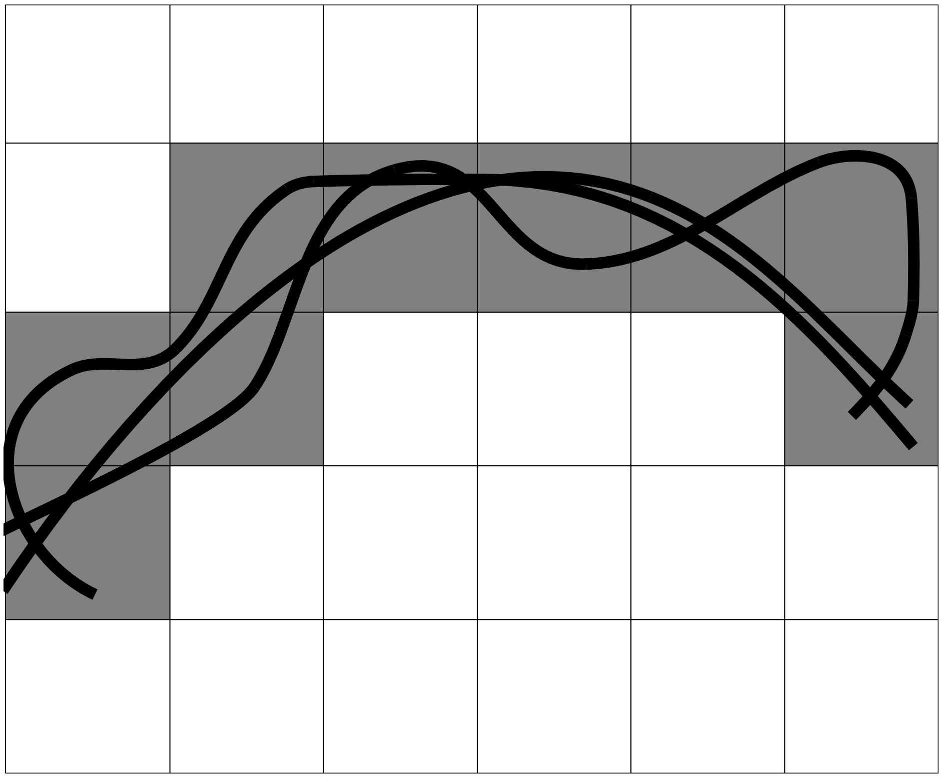


Figure 2.5: Three curves that have the same digitized representation

that the designer did not intend to draw.

A third justification, which will be discussed in more detail in later section , is due to perceptual characteristics – it is simply not necessary for the analytic (fitted) curve to exactly coincide with the drawn curve (or even its digitized representation) for the user to perceive the fit as "close enough" (or even for him to notice, in many cases).

The closeness of fit of the analytic curve to the digitized curve (in the author's system) can range from very far to exact – that is, the designer can tell the system to only approximate his drawn curve, to varying degrees, or to fit it exactly. This second case is interpolation. Thus, we have the best of both worlds. One of the goals of the system is to reduce the size of the representation of the curve, and for digitized curves of typical densities fitted with typical closeness, this goal is met. The author's system was not intended to be an interpolatory system, but an interpolating curve may be generated

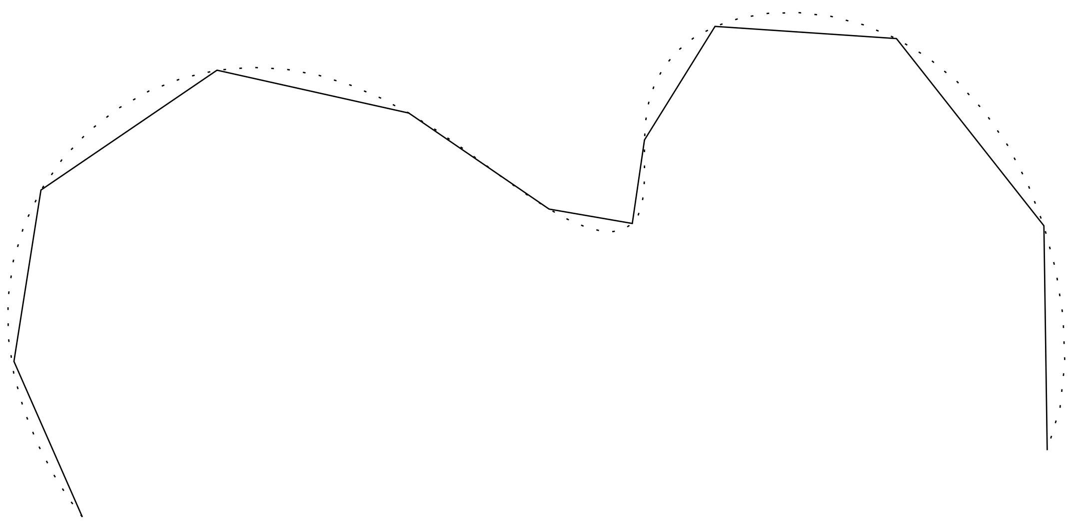


Figure 2.6: A curve approximated by a linear spline

merely by specifying a fitting error of 0.

For those interested in curve design systems based on interpolation, see [Aki70], [EM77], [Dub87], [HN82], [FJ84], [McC70], [McL83], [FB85], [Pie87], and [Rou80].

2.4 Previous approaches

This section presents a number of methods for approximating curves, in a somewhat chronological order.

2.4.1 Polygonal Approximation

Perhaps the earliest approximation method for curves in CAGD is based on polygons, also known as *linear splines*. The curve is represented in a piecewise fashion, where each segment is a straight line segment. See Figure 2.6 for an example of this technique.

Many approaches to solving this general problem have been developed. These include least-squares methods ([Sto61], [Glu62], [Can71], [Cd72]), Chebycheff ([Ric64]), non-linear minimization procedures ([FG69], [Mon70]), and on simpler geometric principles ([Wil78], [SG81], [Ram72]). Some of these methods were developed for single-valued functions, but may be extended to arbitrary planar curves. These approaches pick vertices in different ways :

1. The vertices lie on the curve proper (i.e., for a digitized curve, this means that the vertices are chosen from the points defining the curve)[Ram72].

2. The vertices lie on points of subdivision of the curve.

3. The vertices lie anywhere [SCH72].

As Phoenix (as well as Plass and Stone's system [PS83]), optionally utilizes a linear-splines curve-fitting algorithm as a pre-processing step, we describe one approach based on #1 above.

Formally, let P be an ordered set of points representing a curve:

$$P = \{p_i = (x_i, y_i) \mid i = 1, \ldots, n\} \tag{2.1}$$

We wish to find a subset P' of P :

$$P' = \{p'_i \mid i = 1, \ldots, m; p'_1 = p_1; p'_m = p_n\} \subseteq P \tag{2.2}$$

such that the "curve" formed by the sequence of line segments joining the p'_i's in P' deviate from the points in P by no more than some (perpendicular) distance ε.

For such a situation, there are a number of methods for choosing the p'_i's. Perhaps the most obvious method, which Reeves [Ree81] calls the "brute force" method, is to simply begin at one end, and step along the digitized curve, comparing the distances between the intervening points and a line connecting the last vertex and the current point. If the maximum error exceeds ε, then the point before the current point is included in P', and the process begins again at this new vertex. Sklansky and Gonzalez [SG81] call this the "optimal segment finder". Reeves notes that this method can be somewhat slow (due to the large number of distance calculations that must be made), and suggests that only some percentage of the intervening points be checked, or even just the midpoint.

A second method (the one used by the author), is recursive: a line segment is created, joining the first and last points (which are always vertices), and the distances from this line to the rest of the points are checked. If the maximum error is less than ε, then the approximation is accepted; otherwise, the curve is subdivided at the point of maximum error, and the fitting process is applied recursively to the left and right sides. Ramer [Ram72] used an iterative version of this algorithm. We should note that this recursive technique is somewhat less than ideal in that as the subdivision is done at the point of greatest error (thereby reducing the error at that point to 0), there is a tendancy to over-fit the curve (i.e., you tend to get less error than you asked for).

2.4.2 Splines

Origins

The term *spline* originated in the days before CAGD systems, and 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 [Pav83] 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. We will not, in general, discuss this aspect. Interested readers should consult any numerical analysis text, such as [Cd72], or [BBB87].

Uses

One early reference to the use of splines for approximation of functions appears in [Rei67]. Like many early works in this area, this method applied to single-valued functions. However, the approach may be extended to multiple-valued (parametric) functions, and thus to arbitrary planar curves.

Given a set of points (x_i, y_i), $i = 0, \ldots, n$, construct a function $f(x)$ to minimize

$$\int_{x_0}^{x_n} g''(x^2)dx \tag{2.3}$$

among functions $g(x)$ such that

$$\sum_{i=0}^{n}(\frac{g(x_i) - y_i}{\delta y_i})^2 \leq S \quad g \in C^2[x_0, x_n] \tag{2.4}$$

(Note that $S = 0$ yields interpolation).

Because of the use of the second derivative in Equation 2.4, Reinsch used cubic splines – the Euler-Lagrange equation for this variational problem is y'''', which is the defining equation of a cubic polynomial. Earnshaw [Ear85] notes that "the technique gives good results. However, the majority of scientists with data points containing errors showed no inclination to use the implementation; use of a simple interpolation assuming the errors do not exist, is the preferred approach!"

Grossman [Gro70] describes a parametric curve-fitting technique. He noted that ordinary least-squares fitting takes no account of the ordering of points, and cannot handle multiple-valued functions. His approach was to treat the data points parametrically:

$$\begin{aligned} x' &= f_x(u) \\ y' &= f_y(u), \end{aligned} \tag{2.5}$$

assigning parametric values u by relative chord length: let

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

be the chord length of the digitized curve we are attempting to fit. Then, each point p_k is assigned a u-value (its *parametric* value) of L_k/L_n. Grossman's method may be stated as follows:

1. Parameterize the digitized curve using the chord-length method.

2. Use a least-squares technique independently in x and y, generating a polynomial curve.

3. Improve the parameterization of each (x_i, y_i).

4. Repeat 2 and 3 until a satisfactory fit is obtained.

Grossman's method for reparameterizing the points is as follows: the contribution of the i^{th} point to the total squared error is

$$s(u_i) = w_{x_i}[x_i - f_x(u_i)]^2 + w_{y_i}[y_i - f_y(u_i)]^2,$$

where w_{x_i} and w_{y_i} are the weights assigned the errors associated with point (x_i, y_i). Thus, we wish to find for each point (x_i, y_i) a new parameter value u_i', such that $s(u_i') \leq s(u_i)$. The approach Grossman used was to use a constant step-size search. The position is refined by parabolic interpolation, and the process is then repeated with a smaller step-size.

This basic technique is the basis for both Plass and Stone's and the author's curve-fitting systems. Note that Grossman's method is *not* a piecewise representation. Thus, to fit complex curves, high-order polynomials are required. This can radically increase the computational cost, and potentially induce unwanted oscillations. Grossman did report fairly good results for quadratic and cubic functions used as the bases.

Flegal (cited in [Ree81]) presents an algorithm for piecewise curve-fitting, in the context of fitting hand-drawn curves. His representation scheme used cubic splines, similar to Grossman's use of cubic polynomials. Flegal's knot selection algorithm can be described as follows (this description is derived from [Ree81]): First, scan the curve for points of discontinuity. Then, place a knot at each endpoint, and each point of discontinuity.

Between each pair of knots so chosen, apply the following technique: Label the points, starting at a knot chosen as described above, p_0, p_1, ... Calculate the area of the triangle formed by p_0, p_1, and p_2. If the area is less than some threshold τ, then calculate the area formed by p_0, p_2, and p_3 and add it to the area calculated in the last step. So long as the accumulated area is less than τ, continue to step along the curve. When the area *does* exceed τ, we make the second of the set of three points we are considering a knot, and begin the entire process as described above from that point. Flegal identifies cusps during the above process by looking at the ratio of the area of the current triangle with the sum of the last two areas – if the ratio exceeds some threshold α, then a knot is added if the total area is greater than a noise parameter. Reeves notes that Flegal's knot selection method is intended to select more knots in areas of higher curvature, but notes that the approach can fail for some very simple examples.

2.4.3 B-splines

Background

A B-spline is a parametric curve that approximates a set of control points. Formally,

$$Q(u) = \sum_{i=0}^{n} V_i N_{i,k}(u), \tag{2.6}$$

where

$$N_{i,1}(u) = \begin{cases} 1 & \text{if } t_i \le u < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

$$N_{i,k}(u) = \frac{(u - t_i)N_{i,k-1}(u)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - u)N_{i+1,k-1}(u)}{t_{i+k} - t_{i+1}} \tag{2.8}$$

In the above equations, the *t's* are *knot values*, which relate the parameter u to the control vertices V. A common choice for knot values for open curves are associated with uniform non-periodic B-splines, defined as

$$t_i = \begin{cases} 0 & \text{if } i < k \\ \text{i-k+1} & \text{if } k \le i \le n \\ \text{n-k+2} & \text{if } i > n \end{cases} \tag{2.9}$$

For our purposes, the important properties of B-splines are:

- In general, the curve approximates the control points.
- Local support – the location of the curve depends only on a few nearby control points. This is because the blending functions are non-zero (read: have an effect on the curve's shape) in only a portion of the range of the parameter.
- The curve can be made to interpolate the endpoints by locating more than one knot at the same place. For cubics, placing three knots will do this. This also will cause corners (discontinuities in the tangent) if this is done at points on the interior of the curve.
- The parameter k above, defining the *order* of the B-spline, controls the continuity of the curve – a cubic B-spline, where $k = 3$, is C^2 (assuming no multiple knots).
- The number of knots is unrelated to the order of the B-spline, so single knots may be added or deleted without affecting the continuity characteristics (in general).

See Figure 2.7 for an example of a cubic B-spline curve.

As mentioned earlier, the conventional approach to curve design has been to specify control points and knots, and have the computer calculate the curve based on the equations defined above. The process can be inverted – that is, if we have some set of knots, we can compute the control points, and thereby define a cubic B-spline curve passing through the points. This approach is based on the following relationship between the knot points $t_0, \ldots, t_n$ and control points $V_0, \ldots, V_m$:

$$\frac{1}{6}\begin{pmatrix} 6 & 0 & 0 & . & . & . & 0 \\ 1 & 4 & 1 & . & . & . & 0 \\ \vdots & & & & & & \\ 0 & 0 & 0 & . & 1 & 4 & 1 \\ 0 & 0 & 0 & . & . & . & 6 \end{pmatrix}\begin{pmatrix} V_0 \\ V_1 \\ \vdots \\ V_{n-1} \\ V_n \end{pmatrix} = \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_{n-1} \\ t_n \end{pmatrix} \tag{2.10}$$

We can solve this system for the control points if we have the knots, and thereby define a curve.

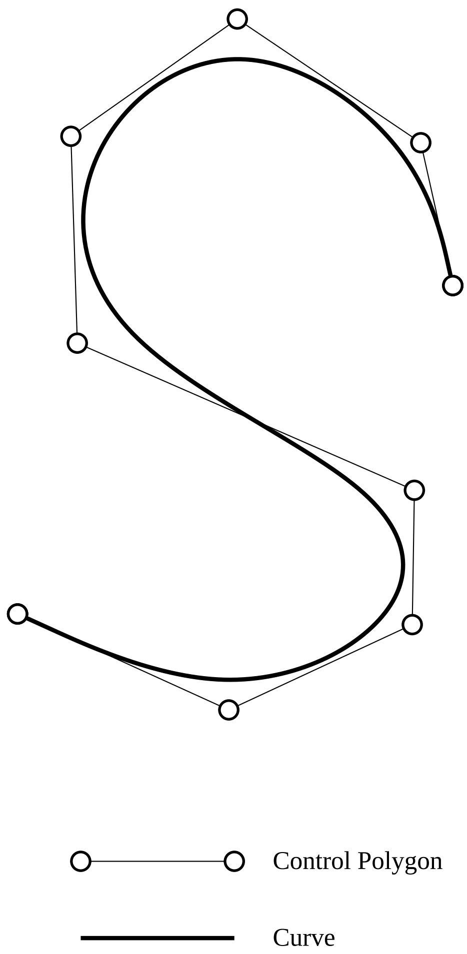


Figure 2.7: A cubic B-spline

Applications

Yamaguchi ([Yam78])developed an interactive curve-fitting and editing system based on this inversion idea. His method proceeds as follows:

1. The designer sketches in a curve.
2. He then picks points on the sketched curve to serve as candidate knots.
3. The computer selects a subset of approximately equidistant candidates as knots.
4. The inversion process is performed, yielding a curve close to the sketched curve.

5. The designer may move knots interactively to change or improve the fit.

6. If the fit is still unsatisfactory, the number of curve segments is automatically increased (i.e., more knots are added), and the above step is repeated.

This method is relatively fast, but requires a good deal of intervention by the user.

Wu, Abel, and Greenburg [WAG77] also used this inversion method for designing sectional curves for surface representation. The designer chooses points as in Yamaguchi's method, but they allow alteration of the curve by moving control points, as in the conventional method. Again, this requires involving the user in the fitting process. Also, the area of the curve affected by moving a control point is based on the fitting, which may not correspond to the designer's idea of what should be affected.

Giloi [Gil78] also describes a method based on the user interactively changing, adding, or deleting knots. This again involves the user in the mathematical representation of the curve.

Clearly, automatic knot-selection would be somewhat of an improvement on the above methods. One approach might be to choose knots on the basis of chord length, and dependent on the curvature of the digitized curve. Lozover and Preiss [LP81] describe an automatic B-spline curve-fitting technique, also based on the inversion idea. Their method selects knots from the set of points comprising the digitized curve, using a method exactly like the "optimal segment finder" described in the section on piecewise linear fitting. Their claim is that a minimal number of knots are generated, and that this results in a 90 to 95 percent reduction in the number of values required to represent a curve.

Yang, et. al. ([YKC*86]) describe an automatic curve-fitting technique based on quadratic B-splines. They choose knots by stepping along the digitized curve, fitting a parabola to longer and longer sequences of points. When the error exceeds some threshold ε, the previous point is deemed a knot, and the procedure starts again from that knot. Once the knots have been chosen, they fit a quadratic B-spline S to the piecewise parabola P such that the error between S and P is below some threshold. Their general algorithm is as follows:

1. Find intersections of the tangents at the knots, and make these the control points.

2. Compute the error between P and S.

3. If the error is less than the threshold, accept the fit, otherwise add additional control points and go back to step 2.

Dierckx [Die82] describes a least-squares cubic B-spline curve approximation technique, using an interesting knot selection method:

1. Fit a least-squares cubic polynomial to the curve, with no interior knots.

2. If the error is within the specified tolerance, then accept the fit. Otherwise, increase the number of knots, and repeat this step until a good fit is obtained.

The number of knots added in step 2 is dependent on the number of knots added in the previous iteration, and the effectiveness of this addition on the closeness of fit. The new knots are placed in regions of greatest error.

Vercken, et. al [VPV87] describe a system similar in intent to Phoenix, which is based on B-splines. In their system, the user draws in a curve using a mouse. The points are then paramaterized using chord-length parameterization. Using a method derived from deBoor [DeB78], they choose a set of knots from the digitized curve in the following fashion: they initially consider the entire curve, and fit two least-squares polynomials in x and y. If the distance from the points to the polynomial curves is greater than some tolerance, then the curve is subdivided into k pieces, where k is the order of the spline. The points so chosen become the knot values in the next step.

Finally, a B-spline is fitted to the curve, using the chosen knots. The technique they use is similar to that used by some of the previous examples – they solve for a set of control points (and thus the fitted curve) by minimizing the following function:

$$I_m(M) = \sum_{j=1}^{n}[M(u_j) - P_j]^2 + \mu \int_0^t (X^m)^2 + (Y^m)^2 dt$$

where:

- M_i are the control points of curve M.

- P_i are the data points.

- μ is a positive parameter controlling the trade-off between the sum of the distances of the points P_i from the curve, and the smoothness of the curve.

- u_j is the parameter value assigned to point P_j.

- X and Y are the least-squares polynomials (B-splines) for which they are solving independently.

They note that this can be done in $O(n)$ time, as the two linear systems resulting from the above formulation can be represented as symmetric band matrices. The resulting curve exhibits parametric continuity of order $m \leq k - 2$.

Following the fitting, the user can manipulate the shape of the curve by moving, supressing, or adding control points, as is done in earlier systems.

2.4.4 Hermite Polynomials

Hermite interpolation is a classical interpolation technique, which may be found in any standard numerical analysis text. Like all the other methods presented, it may be used for approximation by selecting a subset of the data points as knots. These knots/points are interpolated, while the non-knot points are merely approximated.

The important properties of the Hermite representation are:

- Piecewise representation.

- Combinations of scalar multiples of Hermite basis functions can be used to produce a curve, with the values of position and tangent at the endpoints controlled each by one of the functions.

Hermite representations have been employed by a number of researchers for approximation of digitized curves. Ichida [IKY77] used cubic Hermite polynomials for approximating scalar functions, using a fast one-pass technique. His method produces C^1 curves, using a least-squares technique. Chong [Cho80] describes an automatic curve-fitting algorithm, again producing C^1 curves. The interesting aspect of his system is that a linear approach is used (in general, the problem of least-squares piecewise polynomial approximation with variable knots is a non-linear optimization problem, with correspondingly expensive computations).

Perhaps one of the more interesting curve-fitting schemes has been developed by Plass and Stone [PS83]. We describe it in some detail, as Phoenix was influenced by some of the techniques they used. Their approach emphasizes minimization of the number of segments generated, sacrificing speed. They noted that the obvious method for accomplishing this goal is to try all possible combinations of knots, and use the one that optimizes the fit-versus-number-of-segments tradeoff – this currently requires an amount of time that is exponential to the number of points. Their approach is to use dynamic programming to search the knot space.

As in the other approaches, knots are chosen from the data points. As is done (optionally) in Phoenix, they fit a polygon (i.e., a linear spline) to the original points, reducing the set of candidate knots. They then scan this subset for corners (as does Phoenix), making such points knots.

From the remaining candidate knots they select the final set of knots by dynamic programming. In all the previous systems, the knot-finding preceeds the fit. In Plass and Stone's system, as well as Phoenix, the knots are produced by the fitting process. That is, they attempt to fit cubics to a number of combinations of knots, and the knots that result in the best fit become final knots in the representation. They use a fairly conventional least-squares parametric fitting method, subject to the constraint that they produce a G^1 curve. Like Grossman's method, and like the author's, the initial parameterization used is based on relative chord lengths. If the initial fit using this approximations is not good enough, they use Newton-Raphson interation to improve the parameterization, and then fit a cubic again. Both Grossman's method and the author's method take this approach.

2.4.5 Other Representations

A number of other representations have been utilized:

- Hyperbolic splines in tension ([Koz86], [Sch66], [Cli74]).
- Cubic splines in tension ([Cli74], [Dub87], and [Sch66]).
- Conic sections ([Boo79]).
- Conic splines ([Pav83]).
- Conic arcs and straight-line segments ([Alb74]).
- Circular arcs and straight-line segments ([Pie86a]).

2.5 Comparison and Evaluation

Any sort of comprehensive evaluation of this large a number of curve-fitting techniques is well beyond the scope of this thesis. Reeves [Ree81] discusses in some detail curve-fitting methods based on linear splines, cubic splines, and Hermite polynomials, but ignores B-spline and Bézier representations, and concludes that the Hermite representations are the best of the group.

We should note that all of these methods fail to be useful for achieving the author's goals in one way or another:

- Many of the systems merely fit a curve to a set of digitized points, and there is no provision for editing or changing the curve once it is fitted.
- All the systems but Plass and Stone's fail to enforce geometric continuity, instead restricting the continuity to first- or second-order parametric continuity.

- Many of the systems are too slow for interactive use (e.g. Plass and Stone's), or if they are fast enough, fail in that they fit poorly or for some other reason.
- Many are not automatic, requiring the user to manipulate control points, knots, or tension parameters, or some combination of these.

2.6 A Short Course in Bézier Curves

The curve representation that is used in Phoenix in approximating the digitized curve is the Bézier curve. Accordingly, we have included a rather detailed description of the formulation, properties, and algorithms associate with such curves.

2.6.1 Formulation of Bézier curves

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 [BFK84] and [Dav75]. Bézier curves are constructed using the Bernstein polynomials, thus such curves are often referred to as Bernstein-Bézier curves. The Bernstein polynomials of degree n are defined by

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

where $\binom{n}{i}$ is the binomial coefficient $\frac{n!}{(n-i)!i!}$. (For a an interesting discussion of such polynomials in terms of probability and urn models, see [Gol83].)

A Bézier curve of degree n is defined in terms of Bernstein polynomials:

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

where the V_i are the control points, and the sequence $V_0, \ldots, V_n$ of control points forms the control polygon of the segment. See Figure 2.8 for an example of a cubic Bézier curve.

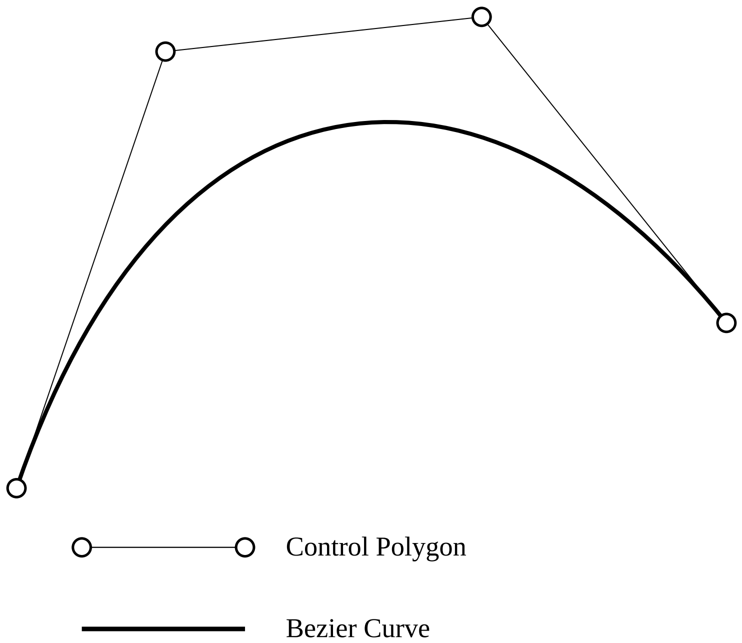


Figure 2.8: A single cubic Bézier segment.

2.6.2 Properties of Bézier curves

There are three important properties of Bézier curves:

$$\begin{aligned}\sum_{i=0}^{n} B_i^n(t) &= 1, \forall t \\ B_i^n(t) &\geq 0, \forall t \\ B_i^n(t) &= (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t)\end{aligned}$$

The first property, known as *partition of unity*, guarantees that the relationship between the curve and the control points is not affected by the orientation of the set of control points with respect to the coordinate system. That is, we may rotate, translate, or scale the set of control points, and the curve associated with them will retain its relationship (in terms of shape) with the control points. The second property, *positivity*, guarantees that the curve in the interval $[0,1]$ lies within the *convex hull* of the control polygon. This property is exploited in several places in the author's curve-fitting system.

The third property, *recursion*, is of interest because it leads to an elegant algorithm for evaluating $Q(t)$.

Two very important properties of Bernstein-Bézier curves are due to Descarte's Law of Signs. This requires a bit of background: Descartes considered the signs of the coefficients of polynomials, and found that the number of zeros (roots) of a polynomial is equal to the number of sign changes (between one coefficient and the next) in the polynomial, or differs from it by an even positive number. For a Bézier curve, the coefficients of the polynomial are the coordinates of the control points in the plane. In general, if we consider *any* straight line, the number of intersections of that line by edges of the control polygon will be greater than or equal to the number of intersections of that line with the curve associated with that control polygon (if the variation-diminishing property holds). In terms of roots of the polynomial, we count intersections with the horizontal axis. The characterization based on straight-line intersection is the so-called *variation-diminishing* property of Bézier curves. To understand this better, consider the straight line S from V_0 to V_n. The control polygon will then cross S at most $n-3$ times, and the curve will cross that line at most $n-3$ times, and will be closer to S than the control points. Thus, the curve varies less from a straight line than does the control polygon, and will be "smoother" than the control polygon. The Bézier curve can thus be thought of as a smooth approximation of the control polygon. This characteristic is exploited in several places in the author's system.

2.6.3 Algorithms for evaluating Bézier curves

Considering the equation

$$Q(t) = \sum_{i=0}^{n} V_i B_i^n(t),$$

an obvious algorithm for evaluating $Q(t)$ is:

Bezier$(V_0, \ldots, V_n, t)$

/* Return $Q(t)$ */

$Q \leftarrow (0, 0)$

for $i \leftarrow 0$ **to** n **do**

  $Q \leftarrow Q + V_i \binom{n}{i} t^i (1-t)^{n-i}$

endfor

return Q

This algorithm is somewhat expensive, due to the exponentiation operations and the combinatorial functions; fortunately, there exists an efficient (for low-degree curves) and elegant algorithm resulting from the recursion property, as we mentioned earlier. This algorithm is due to de Casteljau, and bears his name. Recall that property:

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

The algorithm:

Bezier($V_0, \ldots, V_n, t$)
/* Return $Q(t)$ */
for $i \leftarrow 0$ **to** n **do**
    $V_i^0 \leftarrow V_i$
endfor
for $j \leftarrow 1$ **to** n **do**
    for $i \leftarrow 0$ **to** $n-j$ **do**
        $V_i^j \leftarrow (1-t)V_i^{j-1} + tV_{i+1}^{j-1}$
    endfor
endfor
return V_0^n

This algorithm has an elegant and intuitive geometric interpretation (see Figure 2.9) : each of the points V_i^j lie on the line segment V_i^{j-1}, V_{i+1}^{j-1}, breaking that line segment into two parts of relative length t and $1-t$. Thus, one computes the point $Q(t)$ in the following manner: break each leg of the control polygon into two parts at a point t percent of the length of each leg. Then, connect each successive point so derived with a straight line segment. Apply this method recursively to these new line segments, until a single point is obtained. This point is $Q(t)$, and is labelled as V_0^3 in Figure 2.9.

2.6.4 Subdivision of Bézier curves

Referring to Figure 2.9, consider the auxiliary control points V_i^j. Points V_0, V_1, V_2, and V_3 are the control points of the original segment. Points V_0, V_0^1, V_0^2, and V_0^3 can be viewed as the control points for the curve Q between 0 and t; similarly, points V_0^3, V_1^2, V_2^1, and V_3 are the control points for the curve Q between t and 1. Thus, the de Casteljau algorithm can be used to find the control points of the Bézier curve in a subinterval of an existing

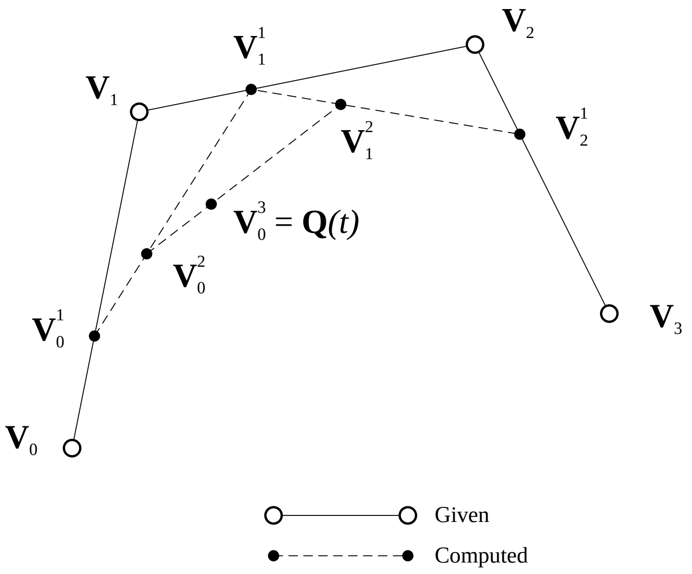


Figure 2.9: The de Casteljau algorithm for a cubic segment

Bézier curve. In general, for subdividing at a point t, $0 \leq t \leq 1$, the control points for the left half are given by

$$V_0^0, V_0^1, \ldots, V_0^n,$$

and the control points for the right half are given by

$$V_0^n, V_1^{n-1}, \ldots, V_n^0.$$

Note that the two control polygons are closer to the curve they define than the original control polygon; in fact, the subdivision process converges to the curve itself.

2.6.5 Derivatives of Bézier curves

The author's curve-fitting system enforces first-order geometric continuity (G^1). Recall that this means we must ensure that the tangent directions at the joints between segments match.

The k-th 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 $\Delta^k V_i = \Delta^k V_{i+1} - \Delta^{k-1} V_i$ ([Wat87], [BFK84]).

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 end 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 G^1 continuity at the joints, we will require that the second-to-last control point of one segment 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 Figure 2.10 for an example; the second control point of the right-hand segment (the one with control points W_i) may be located anywhere along the half-line labelled VW.

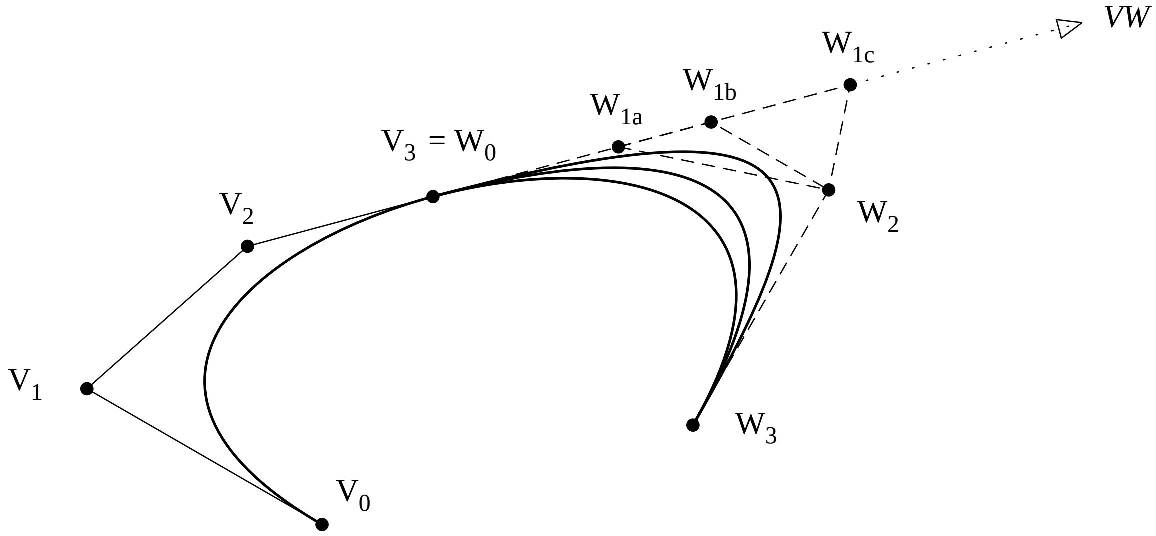


Figure 2.10: G^1 continuity condition for cubic Bézier curves

Chapter 3

Phoenix

3.1 Introduction

This section gives a high-level overview of Phoenix. The intention is to convey to the reader an understanding of the system as a whole – what the basic problems are, and the algorithm describing the basic function of the system. Following this are sections describing in detail the issues involved in each sub-problem, and the algorithms used in their solutions.

Recall the basic design paradigm we are trying to simulate: the designer begins by drawing in a "first attempt" at the curve. Then, he modifies the curve by repeatedly erasing and redrawing portions of the curve, until it conforms to his mind's-eye view of the curve. Figure 3.1 describes the basic method.

Were we *not* interested in maintaining an analytic representation of the curve, and instead only using the curve defined by the points actually drawn in by the designer, the algorithm of Figure 3.1 would suffice. Phoenix attempts to follow this algorithm as closely as possible – the overall structure remains the same, but with several added steps:

1. Pre-processing of the digitized curve, in order to :

    - Remove noise due to quantization and hand jitter.
    - Identify key features we wish to preserve, such as corners.

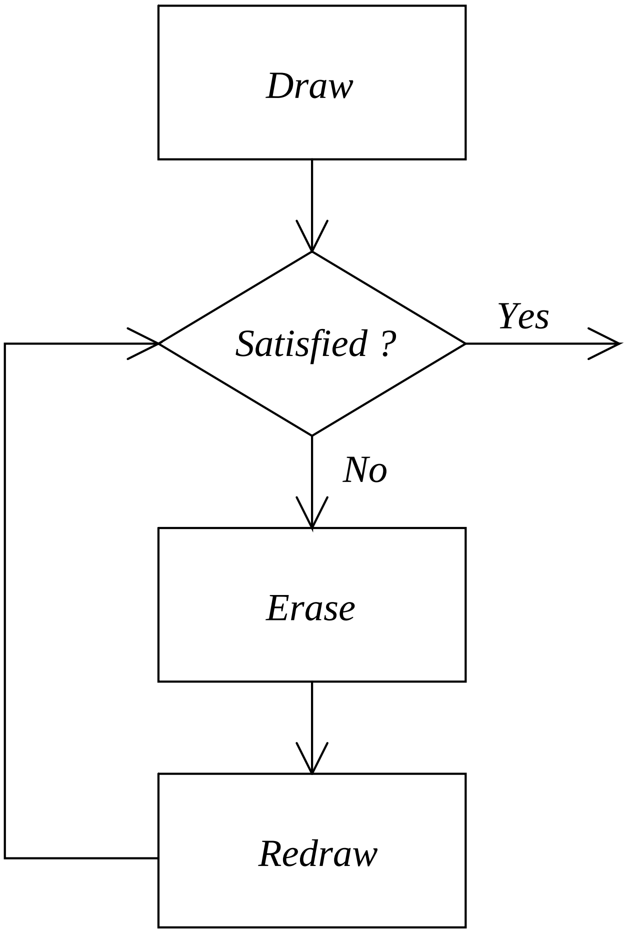


Figure 3.1: The basic drawing paradigm

- Reduce the number of points, without reducing the informational content of the curve, for reasons of efficiency.

2. Fitting an analytic curve to the processed digitized curve.

3. "Garbage collection" – an attempt to "coalesce" a number of small segments in the curve into one larger one, to reduce the size of the representation with little or no loss of detail.

See Figure 3.2 for a description of the curve-design process as implemented by the author. For reasons of clarity, steps not essential to understanding the system have been removed from this picture (such as the ability to "undo" a previous step, etc.). The next sections describe each step in detail.

3.2 The Pre-processing Steps

Input into design systems like the one described here is generally done with a digitizing device, such as a tablet and stylus, or a mouse. The author's implementation was done with the X Window System [SG86], and thus used a mouse. In either case, the output of a drawing action is a time-ordered sequence of (x, y) (integer) coordinates specifying the location of the drawing device. Several characteristics of such curves are important :

- The density of the point samples is inversely proportional to the speed at which the designer's hand moves, and directly proportional to the sampling rate – the points in a slowly-drawn curve will be much closer together than if the curve were drawn faster, and a high sampling rate yields a denser curve. In the case of digitizing tablet/stylus input, curves drawn very slowly may have long sequences of points with the same coordinates.See Figure 3.3 for a curve sampled at differing rates.

- The user's hand motion is a continuous phenomenon – the actual path taken by the drawing device consists of an infinite number of points; the location of each point being an infinitely-precise real-valued (x, y) coordinate pair. Sampling is a discrete process, and the samples are of limited precision (generally, they are given in integer values, or normalized fixed-precision real values). This introduces *quantization errors*, an inherent problem arising when sampling any continuous process.

- Noise, resulting from unintended "jitter" of the designer's hand, may appear in the digitized curve, especially if the sampling rate is high, and/or the curve is drawn slowly. That is, there may be "features" in the curve that are not part of the curve as the designer intends.

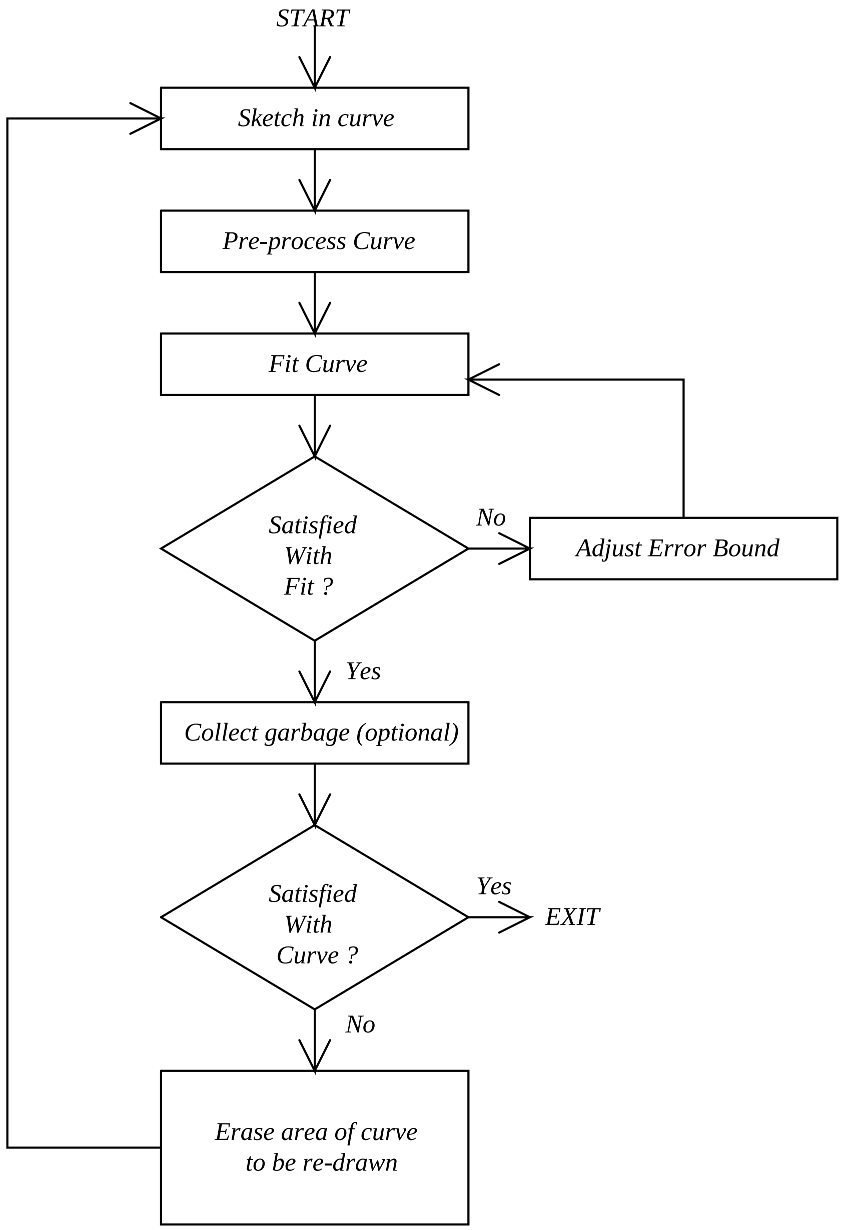


Figure 3.2: An overview of the design process

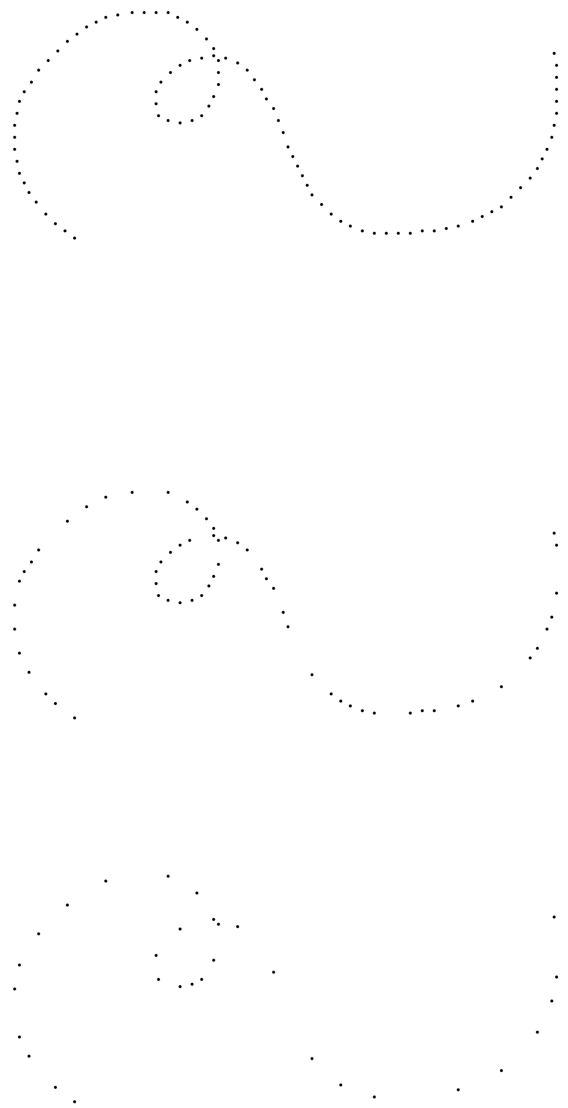


Figure 3.3: Dense and sparse sampling of curves

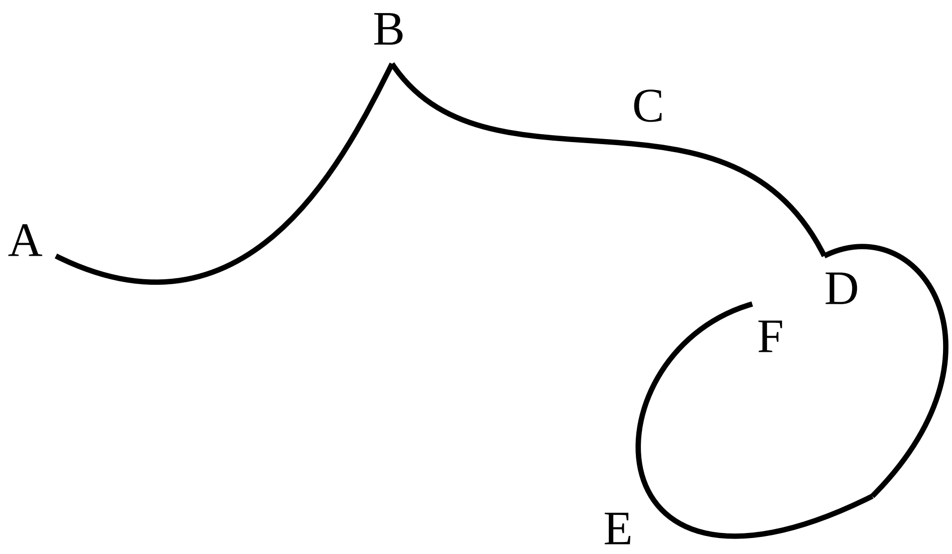


Figure 3.4: A Curve with some important features noted

Before we describe the pre-processing of the digitized curve, we discuss some general aspects of the representation of curves. Because we are dealing with the visual aspects of curve representation, we must consider what features of a curve are important to the perception of its shape. Reeves [Ree81] calls this “visual accuracy”.

Perceptually speaking, points of discontinuity are perhaps the most important features. The endpoints, where there is not even zero-order continuity, are obviously quite important to preserve. Points of higher-order discontinuity (in our case, we consider only first-order) within the curve, which we may call “cusps” or “corners”, are equally important; indeed, one may view a curve with such corners as a set of sub-curves, connected at the corners with zero-order continuity. The author’s system treats the entire curve in exactly this manner. In Figure 3.4, points A and F are endpoints, and B and D are cusps, or corners.

At this point, we have reduced a larger curve into some number of sub-curves; each of the sub-curves is relatively smooth. Regions of high curvature require more “information-per-unit-length" than do flatter regions. This implies that we must be careful to represent this denser information in our analytic curve. Finally, changes in the direction of curvature, that is, *inflection points*, are important features to reproduce. In Figure 3.4, point E represents a region of high curvature, and point C is an inflection point.

Finally, we note that there is a "minimal feature size", dependent on the size and resolution of the display device. Features below some minimum size cannot be adequately represented – for example, a tiny loop will appear as a "glob" of pixels (or as one pixel).

Summarizing, we have a number of problems to address:

- We would like to eliminate noise.
- We must deal with quantization error.
- We need to identify and preserve the visually important features.
- We wish to represent the shape with as few points as possible, for efficiency in the curve-fitting process.

These problems seem more complex than they are in practice. A description of the author's approach follows.

3.2.1 Removing points too close together

After the user has drawn in the curve, or redrawn a portion of the curve, we scan along the set of points, eliminating those points that are closer to the previous point by less than some constant δ. The value of δ is dependent on the size and resolution of the display device, and on the sampling rate in the case of a tablet and stylus (the end points are never eliminated in this process). This process has several effects:

- It eliminates much of the hand-jitter-induced noise, as these features are generally very small.
- It generally eliminates "features" below the minimum feature size.

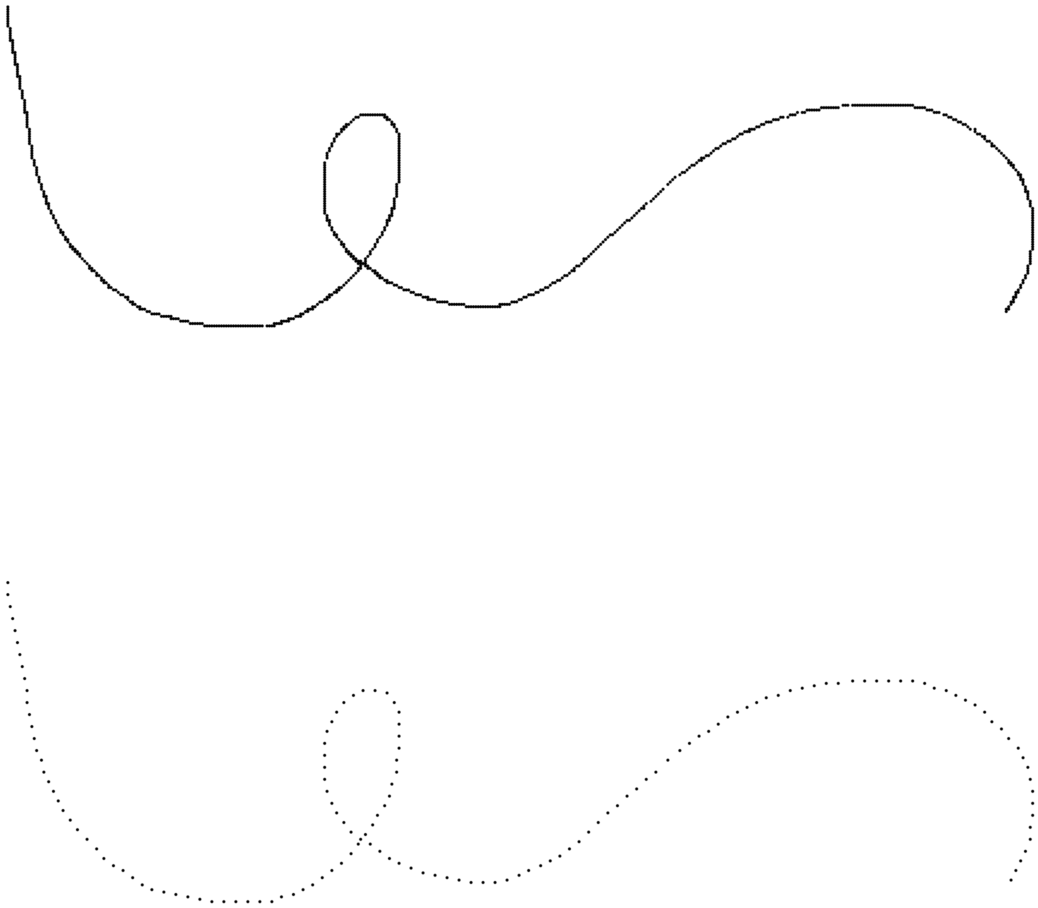


Figure 3.5: A digitized curve before and after removing too-close points

- It eliminates sequences of coincident points. (These may be generated by digitizing tablet systems.)
- It eliminates points that are so close to their neighbors that they add no additional information about the shape of the curve in their local neighborhood.
- It helps in the corner-finding process.

Figure 3.5 shows a curve before and after the the “extra” points are removed.

3.2.2 Finding corners

Once the thresholding process just described has been done, we attempt to find algorithmically those points that are corners or cusps. Freeman [Fre74] describes the use of *incremental curvature* to accomplish this task. Essentially, the algorithm is to consider the angle between the segments connecting a point with its predecessor and with its successor, and define as a corner any point whose angle so measure exceeds some pre-defined constant Φ. Plass and Stone [PS83] use the same method in their curve-fitting scheme, using an angle of 135 degrees as Φ. The choice of Φ is highly application-, implementation-, and user-dependent, but the author has found that only a small amount of experimentation is necessary to determine an appropriate threshold.

The reader may object that this step is performed *after* the too-close points are eliminated, as one may eliminate a potential corner in doing so. However, if the set of points is sparse to begin with, few or no points will be eliminated in the first step. If the set of points is rather dense, then any one of two or three points in the vicinity of the "true" corner may be used as the corner, and the others will be eliminated as part of the first step. Finally, if too-close points are *not* eliminated first, then one must consider the distance from the point currently considered to the points one looks at in determining the angle for corner-locating, and must be careful not to mistake "noisy" points for corners. As there is an inherent relationship between these two steps, and (again) the resolution- and application-dependent aspects, one must experiment to obtain a good balance.

An earlier implementation allowed the user to interactively specify the corners, but as a major goal of this thesis was to automate the fitting as much as possible, and in view of the satisfactory results obtained as described above, this step has been eliminated in the current implementation. The larger dots on the curve in Figure 3.6 are corners that have been automatically located by the system.

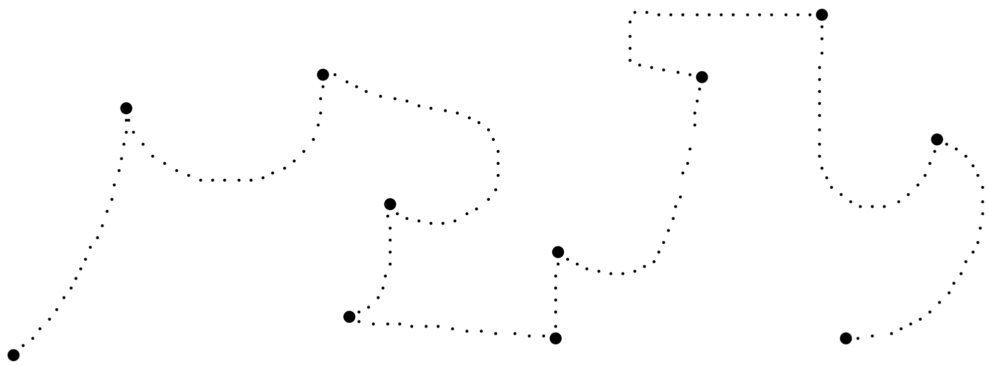


Figure 3.6: Automatically-located corners in a digitized curve

3.2.3 Removing Noise

As stated earlier, the data points in the digitized curve may be thought of as representing discrete samples over time of a continuous curve. The implications of this view were discussed briefly earlier. This general problem, and solutions to it, appear in a number of other disciplines, notably signal processing and image processing. In signal processing, one may be sampling an analog signal. Imprecision and errors of measurement, and natural noise in the original signal may show up as noise in the sampled signal, just as it does in our situation. Many sophisticated techniques have been developed to filter out the noise from a signal; one simple and effective technique, known as *convolution* has been used in Phoenix with good results. Here, we use discrete convolution.

The basic idea behind convolution is to run a filter across the data from one end to the other in such a manner as to spread out local variations to the area in the immediate neighborhood of the point currently under consideration. In our case, we consider the neighborhood to be the point immediately preceding and following the current point. While the mathematical background and justification for convolution is subtle and complex, the implementation of it in our situation is simple and relatively intuitive: we march along the curve, considering each point in turn. The current point's location is replaced by a weighted average of the locations of that point, and its neighbors. In our case, we use what is known as an approximation to a *Gaussian* filter.

In simple terms, the filter we use does the following for each pont P_i:

$$x_i' = (0.5)x_i + (0.25)x_{i-1} + (0.25)x_{i+1}$$
$$y_i' = (0.5)y_i + (0.25)y_{i-1} + (0.25)y_{i+1}$$

yielding a new set of points P_i'. After applying the above formulae to each point (except the corners and endpoints), take the curve P' to be our representation. This process is repeated a number of times, each time smoothing out effectively lower and lower frequency noise. The bound on the number of iterations is dependent on the amount of noise that is to be eliminated, and the distance between the successive original points. Phoenix sets the limit at about ten iterations. A particularly vivid example of the effect of this filtering can be seen in the next chapter (see Figure 4.3).

Phoenix can perform quite well without this filtering, as the algorithm has the characteristic of producing smooth curves even in the presence of noisy data (though filtering does often help the system generate a better-fitting curve with fewer segments). The performance of the fitting system depends heavily on its ability to derive accurate approximations to the slope of the curve, using the data points, as it looks in a small neighborhood around the point it is currently considering. Thus, even small amounts of noise can adversely affect the fitting, forcing the system to subdivide and try again, even if this would not be necessary were the tangent estimate better.

3.2.4 Reducing the number of points while retaining the shape

The final pre-processing step is to reduce the number of points in the digitized curve. Recall that the reasons for this are two-fold:

1. The number of points needed to adequately convey the shape information in regions of high curvature is greater than that needed for flatter areas.

2. The efficiency of the curve-fitting process is greatly affected by the number of points in the digitized curve.

The general problem can be stated in this manner: given a set of points $p_0, p_1, \ldots, p_{n-1}$ defining a curve C, find the minimal (or near-minimal) subset of these points $q_0, q_1, \ldots, q_{m-1}$ such that the curves described by each set of points are "sufficiently similar". This last statement suggests that there is a tradeoff between the error introduced by eliminating points, and the resulting increase in efficiency resulting from the elimination of points. Again, the optimal tradeoff point is highly application-, resolution-, implementation-, and user-dependent; additionally, the denser the original digitized curve, the greater the potential savings. Note that this problem can be stated equivalently: we wish to represent the true (continuous) curve as defined by the path taken by the designer's hand as a sequence of points, within some small error bound, by using the set of samples generated by the digitization process. This suggests that we use a linear-splines approach, and we have indeed found this to be a satisfactory method. As linear splines have already been discussed, we merely point out here that the author's system as implemented uses a recursive procedure similar to that used in [Ram72]. The specific algorithm used here is not of great importance, as the author implemented several different algorithms, and noticed no great differences in the appearance of the curves. The various available algorithms emphasize speed, minimization of the number of output points, or some optimal tradeoff of these two; the choice depends only on the implementation (on a sufficiently powerful machine, this step can be eliminated).

3.3 Fitting the analytic curve

This section describes the heart of the design system – the algorithm used to actually fit the analytic curve to the (now pre-processed) digitized curve. The approach taken by the author has been greatly influenced by two previously-published algorithms: the curve-fitting method described in [PS83], and the surface-fitting method described in [SBD86].

Plass and Stone's technique has been described earlier; here, we explain the relevant aspects of the method of Schmitt et al. (Their method is a 3-D surface-fitting method, but their algorithm works for 2-D curves as well.) Given a curve, or some portion of a curve, to which they wish to fit a single cubic Bézier segment, they determine the tangents at each end $\hat{t}_1$ and $\hat{t}_2$, and the distance (i.e., chord-length) d between the endpoints. The first and last control points are located at the curve endpoints, and the two inner control points are located each at points $d/3$ away from V_0 and V_3, in the $\hat{t}_1$ and $\hat{t}_2$ directions, respectively. This simple heuristic works surprisingly well. The fitting procedure is recursive: they attempt to fit a single cubic segment to the entire curve. If the maximum error is greater than some specified constant ϵ, then the digitized curve is divided in two at the point of greatest error, and the fitting process is repeated on each section. This procedure is repeated recursively until the entire curve has been fitted with some number of segments. As implied earlier, they too enforce G^1 continuity.

The author has adopted from this algorithm the recursive fitting idea, rather than using the dynamic programming approach of Plass and Stone, and the use of the heuristic. In general, a least-squares method is used for curve-fitting, but as the mathematics fail when only two points are to be fitted to, the Schmitt/Barsky/Du heuristic is used in these situations. Methods from Plass and Stone's approach that the author adopted will be described shortly. (Their method is notable in that in an attempt to produce a near-minimal number of segments, they use dynamic programming techniques to determine knots; see [Bel57] or [AHU74] for an explanation of this technique. Dynamic programming solutions often are expensive, though optimal; this may make them inappropriate for interactive systems.)

As described earlier, 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 $X - Y$ plane. In Phoenix, as well as Stone and Plass's system, "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 additional constraints of tangent vector direction considered, in X and Y simultaneously. As we are working with 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 accurately estimate 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 below), 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. Because the chord-length parameterization step is an approximation, there may be substantial room for improvement – if the initial fit exceeds the user-specified error, the simple approach would be to subdivide, and try again. But, it may just be that if we had a better estimate of t_k for each p_k, it might be possible to fit a single cubic segment to that area of the curve. This is highly desirable, as the fitting process is expensive, and we also wish to minimize the number of segments generated. The squared distance between a given point P and a point $Q(t)$ on a parametric curve Q is

$$d = \|P - Q(t)\|.$$

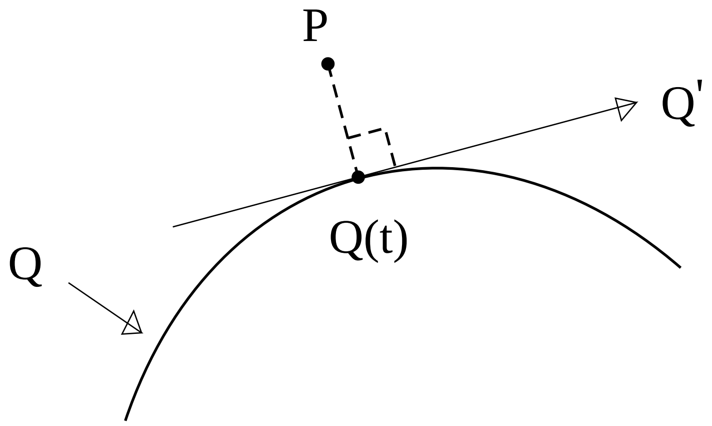


Figure 3.7: The distance from P to $Q(t)$.

Refer to Figure 3.7 for a diagram of this.

If we are interested in finding the parametric value t for such a point P (in order to improve our fitting step), we need to find t such that distance (squared) d is minimized. To do this, we need to solve

$$[Q(t) - P] \circ Q'(t) = 0 \tag{3.1}$$

for t.

This general approach was used by Plass and Stone; they noted that as they had an estimate for t already (from the chord-length parameterization), they could use Newton-Raphson iteration to find the root. The author has also found this method to be both fast and sufficiently accurate. It is discussed in the next section. This iteration can greatly improve the fit of the curve to the points, see Figure 3.8 for an example of the process.

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 we wish to approximate the points with is a cubic Bernstein-Bezier curve, and our fitting

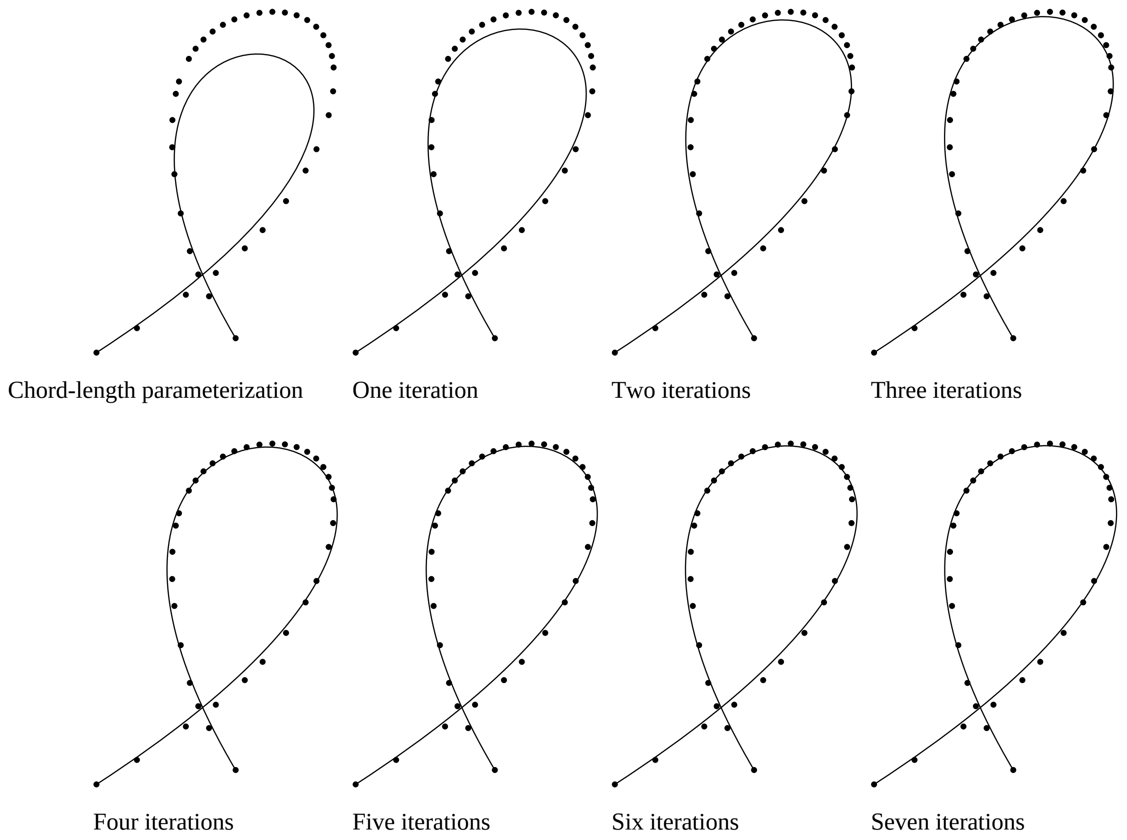


Figure 3.8: Use of the iterative process to improve the fitting of the cubic

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:

$$S = \sum_{i=1}^{n} [d_i - Q(u_i)]^2 \tag{3.2}$$

$$= \sum_{i=1}^{n} [d_i - Q(u_i)] \circ [d_i - Q(u_i)], \tag{3.3}$$

where

- d_i are the (x, y) coordinates of the given points.

- 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.

- $\hat{t}_1$ 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:

$$\frac{\partial S}{\partial \alpha_1} = 0 \tag{3.4}$$

$$\frac{\partial S}{\partial \alpha_2} = 0 \tag{3.5}$$

Expanding Equation 3.4:

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

Expanding the second term, we get:

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

Thus,

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

Rearranging, we get:

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

For convenience, define

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

Then,

$$\sum_{i=1}^{n} Q(u_i) \circ A_{i,1} = \sum_{i=1}^{n} d_i \circ A_{i,1} \tag{3.6}$$

Expanding $Q(u_i)$:

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

Equation 3.6 becomes:

$$(\sum_{i=1}^{n} A_{i,1}^2)\alpha_1 + (\sum_{i=1}^{n} A_{i,1} \circ A_{i,2})\alpha_2 = \sum_{i=1}^{n} (d_i - (V_0 B_0^3(u_i) + V_0 B_1^3(u_i) + V_3 B_2^3(u_i) + V_3 B_3^3(u_i))) \circ A_{i,1}$$

Similarly, for $\frac{\partial S}{\partial \alpha_2}$:

$$(\sum_{i=1}^{n} A_{i,1} \circ A_{i,2})\alpha_1 + (\sum_{i=1}^{n} A_{i,2}^2)\alpha_2 = \sum_{i=1}^{n} (d_i - (V_0 B_0^3(u_i) + V_0 B_1^3(u_i) + V_3 B_2^3(u_i) + V_3 B_3^3(u_i))) \circ A_{i,2}$$

If we represent the previous two equations by:

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

we "only" need to 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

$$\begin{aligned}
\mathcal{C} &= \begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \end{pmatrix} \\
\mathcal{X} &= \begin{pmatrix} X_1 \\ X_2 \end{pmatrix},
\end{aligned}$$

Then, using Cramer's Rule, the solution to

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

is:

$$\alpha_1 = \frac{det\begin{pmatrix} \mathcal{X} & C_2 \end{pmatrix}}{det\begin{pmatrix} C_1 & C_2 \end{pmatrix}}$$

$$\alpha_1 = \frac{det\begin{pmatrix} C_1 & \mathcal{X} \end{pmatrix}}{det\begin{pmatrix} C_1 & C_2 \end{pmatrix}}$$

Our algorithm that attempts to fit a single cubic Bézier segment to a set of points appears in Figure 3.9.

Two points require further explanation:

1. The method for determining the closeness of the fit.

2. The rule used to decide if iteration may help the fitting process.

Recall that our root-finder, when given a point P and a parametric curve Q, returns the parametric value of the point on Q that is closest to P. We can, using the deCasteljau algorithm, then determine the coordinates of this closest point, and measure the distance squared from the digitized point to the point on the fitted curve. By comparing the distance of each digitized point to their corresponding points on the fitted curve, we can determine if the maximum distance squared exceeds the fitting threshold set by the user. Because we must determine the new parametric value u for each digitized point in order to test the fit, we can perform the steps of reparameterizing each point (the iterative process described earlier) and testing the fit with one application of the algorithm for each point.

The rule used to decide if iteration might help is, of course, an heuristic. As noted earlier, iteration does indeed greatly improve the fitting, though the improvement has been empirically observed to decrease with each iteration.

Experimentation has shown that after approximately for to five iterations, the improvement is negligible, so we set a maximum number of iterations to around this level.

```
Fit-Cubic
/* Fit a single cubic to a set of points */
/* Return TRUE if single cubic can be fit within tolerance */
Compute t̂1 and t̂2, the tangents at the endpoints.
Compute an initial estimate of parametric value u for each digitized point.
Fit a single cubic to the points.
Compute the maximum distance between the points and the curve.
if the error is less than the user-specified threshold
  then return TRUE
endif
if the error is small enough to indicate that iteration may help
  then while the error exceeds the threshold and
             the iteration count is less than some predetermined value
             compute new paramtric values u for each data point
             fit a cubic, using the new u's
             if the error is less than the threshold
               then return TRUE
             endif
       endwhile
       return FALSE, as the error exceeds the threshold
else
    return FALSE, as no iteration is likely to help
endif
end
```

Figure 3.9: The algorithm for fitting a single cubic segment

Clearly, if the fit using the chord-length parameterization is quite poor, and an iteration or two does not bring the error much closer to the threshold, then we can assume that it is not worth it to continue to iterate. Thus, we can look at the slope of the line connecting the initial error value with the error value after one iteration, determine where this line intercepts the horizontal line defining the desired fit, and guess if further iteration will be fruitful. Additionally, it should be noted that if the error in fitting after the chord-length parameterization is sufficiently greater than the desired error, it would not even be worth one iteration, and we can use this as a quick check. These heuristics deserve a great deal more investigation than the author has had time to accomplish – the fitting process and subsequent fitting check are both computationally expensive, so a more careful and sophisticated heuristic, based on adequate experimentation, would be of great benefit in terms of optimizing the speed-versus-minimal-number-of-segments tradeoff.

3.4 The Root-Finders

This section describes two root-finding algorithms that are used in the author's system. The first, Newton-Raphson iteration, is used as we try to improve the parameterization of the points, in an attempt to get a better fit. The second, which we call the "approximate inversion" method, was developed by the author and Tony DeRose; it is a variant of the bisection method. It is used in the editing phase of a design session, when we must find the parameter value of an arbitrary point specified by the user.

The basic problem is this: given a parametric curve Q and a point P, both in the plane, find the point on the curve closest to P (i.e., find the parameter value t such that the distance from P to $Q(t)$ is a minimum). Our approach begins with the geometric observation that the line segment (whose length we wish to minimize) from P to $Q(t)$ is perpendicular to the tangent of the curve at $Q(t)$. See Figure 3.7 for an example.

Thus, the equation we wish to solve for t is:

$$[Q(t) - P] \circ Q'(t) = 0 \tag{3.7}$$

In our particular case, curve Q is a cubic Bernstein-Bézier polynomial, for which we know the coefficients – these are the control points:

$$Q(t) = \sum_{i=0}^{n} V_i B_i^n(t)$$

The tangent curve Q' for the above curve Q can also be represented in Bernstein-Bézier form:

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

We are dealing with cubics, so polynomial Q is of degree three, and Q' is of degree two. $Q - P$ is also of degree three, so the polynomial described by Equation 3.7 is (in general) of degree five. Thus, we can restate our problem as one of finding the roots of this fifth-degree polynomial. Note that a closed form expression for the roots does not exist, hence the necessity of the algorithm herein described. We can rewrite Equation 3.7 as a single polynomial of degree five: let

$$Q_1(t) = Q(t) - P \tag{3.8}$$

$$Q_2(t) = Q'(t) \tag{3.9}$$

Then, Equation 3.7 can be written:

$$Q_1(t) \circ Q_2(t) = 0 \tag{3.10}$$

3.4.1 Newton-Raphson Iteration

In the case where we already have an initial approximation to the root (the chord-length parameterization, as described earlier), we can use Newton-Raphson iteration to get a better estimate. The formula for Newton-Raphson iteration for solving $f(t) = 0$ is

$$t \leftarrow t - \frac{f(t)}{f\prime(t)}$$

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

$$\frac{Q_1(t) \circ Q_2(t)}{[Q_1(t) \circ Q_2(t)]'}$$

This method is used in the fitting process where we are trying to find better and better approximations to the root, and then refitting, in an attempt to find a better fit without subdividing. As stated earlier, the chord-length approximation constitutes an initial approximation to the root.

3.4.2 The Approximate Inversion Method

If there is no previous estimate for the root, as would be the case when we are trying to find the distance from an arbitrary point to an existing curve (and thus there is no reliable way of obtaining an initial root approximations), we must take a much more expensive approach, and solve directly for the root. Expanding each of the above: Equation 3.8:

$$\begin{aligned} Q_1(t) &= Q(t) - P \\ &= \sum_{i=0}^{n} V_i B_i^n(t) - P \\ &= \sum_{i=0}^{n} (V_i - P) B_i^n(t) \\ &= \sum_{i=0}^{n} c_i B_i^n(t) \end{aligned}$$

where $c_i = V_i - P$.

Equation 3.9:

$$\begin{aligned} Q_2(t) &= n \sum_{i=0}^{n-1} (V_{i+1} - V_i) B_i^{n-1} \\ &= \sum_{i=0}^{n-1} d_i B_i^{n-1} \end{aligned}$$

where $d_i = n(V_{i+1} - V_i)$.

Then, Equation 3.10 can be written as:

$$\begin{aligned} 0 &= Q_1(t) \circ Q_2(t) \\ &= \sum_{i=0}^{n} c_i B_i^n(t) \circ \sum_{j=0}^{n-1} d_j B_j^{n-1}(t) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} c_i \circ d_j B_i^n(t) B_j^{n-1}(t) \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} c_i \circ d_j \tbinom{n}{i}(1-t)^{n-i}t^i \tbinom{n-1}{j}(1-t)^{n-1-j}t^j \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} c_i \circ d_j \tbinom{n}{i}(1-t)^{n-i}t^i \tbinom{n-1}{j}(1-t)^{n-1-j}t^j \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} c_i \circ d_j \tbinom{n}{i}\tbinom{n-1}{j}(1-t)^{2n-i-j-1}t^{i+j} \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} c_i \circ d_j \tbinom{n}{i}\tbinom{n-1}{j}(1-t)^{(2n-1)-(i+j)}t^{i+j} \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} c_i \circ d_j \frac{\binom{n}{i}\binom{n-i}{j}}{\binom{2n-1}{i+j}} B_{i+j}^{2n-1}(t) \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} c_i \circ d_j z_{i,j} B_{i+j}^{2n-1}(t) \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-1} w_{i,j} B_{i+j}^{2n-1}(t),
\end{aligned}
$$

where $z_{i,j} = \frac{\binom{n}{i}\binom{n-i}{j}}{\binom{2n-1}{i+j}}$, and $w_{i,j} = c_i \circ d_j z_{i,j}$.

So, we now have a Bernstein-Bézier polynomial representation of our original equation, which we can now solve for its roots. The coefficients of the fifth-degree polynomial $Q_1(t) \circ Q_2(t) = 0$ are given by

$$\lambda_k = \sum_{i+j=k} w_{i,j}$$

The Bernstein-Bézier coefficients $(\lambda_0, \ldots, \lambda_{2n-1})$ have a geometric interpretation:

Suppose $c_0, \ldots c_k$ are the Bernstein-Bézier coefficients for a scalar polynomial $f(u)$. The graph of f is a parametric curve

$$Graph(u) = (u, f(u))$$

The question is now: what is the Bézier control polygon for *Graph*? That is, we seek control points $V_i = (x_i, y_i)$ in the plane that satisfy

$$(u, f(u)) = \sum_{i=0}^{n} V_i B_i^n(u),$$

The x and y components are independent, so the above vector-valued equation is really two scalar equations:

$$\begin{aligned} u &= \sum_{i=0}^{n} x_i B_i^n(u), \\ f(u) &= \sum_{i=0}^{n} y_i B_i^n(u) \end{aligned}$$

From the given form of f we know that y_i must be equal to c_i, so we know the y coordinates of the control polygon. What about the x coordinates? For them, we must find the representation of u in terms of the Bernstein polynomials. This is, fortunately, a well-known result [BFK84]:

$$u = \sum_{i=0}^{n} \frac{i}{n} B_i^n(u),$$

implying that $x_i = \frac{i}{n}$; that is, that the control points are equally spaced out along the interval $[0, 1]$.

For example, given the single cubic Q and point P as shown in Figure 3.7 the fifth-degree polynomial we must find the roots of is shown in Figure 3.10. The roots of the polynomial are exactly those values of t where the Bernstein-Bézier curve crosses the horizontal axis (we call the horizontal axis the $t - axis$) . We can exploit knowledge regarding the control polygon in order to help us find the roots:

- The number of times the control polygon crosses the horizontal axis is greater than or equal to the number of roots between 0 and 1.
- If there is a root in an interval, then the control polygon will cross the horizontal axis in that interval.
- There are no loops in the curve; in fact, this is like a scalar function in that it does not "turn back" on itself.

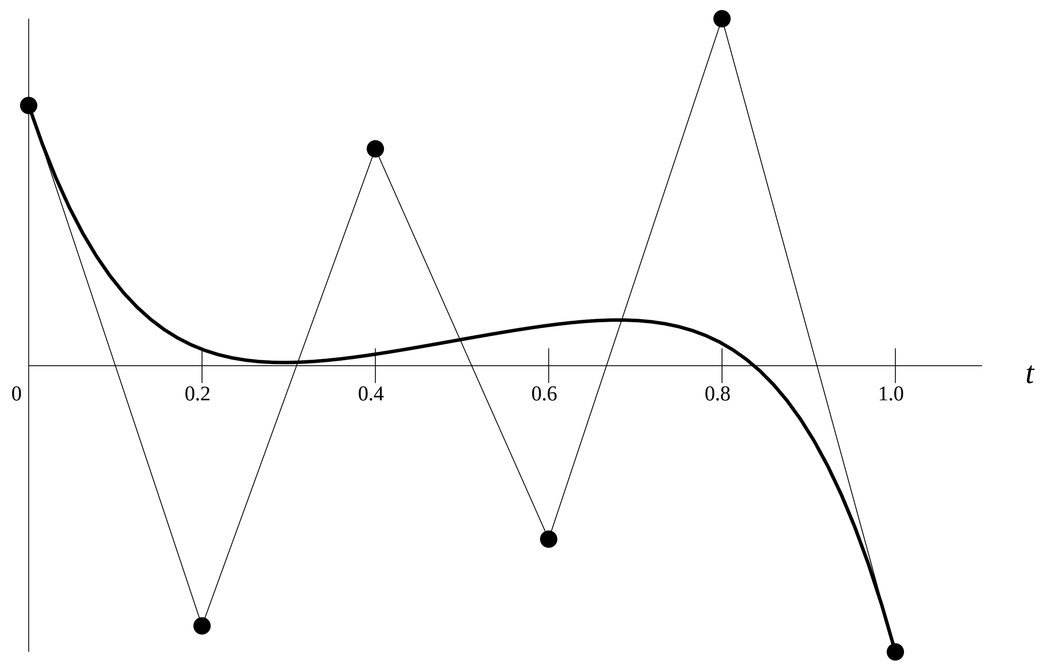


Figure 3.10: A fifth-degree polynomial in Bezier form

- If the control polygon is "flat enough", and crosses the horizontal axis just once, then the root can be approximated by computing the intersection of the horizontal axis with a chord from the the first control point to the last.

These observations lead to a simple recursive algorithm, which appears in Figure 3.11.

As stated earlier, we can stop the recursion when the control polygon of the current polynomial is "flat enough" – that is, when the control polygon is close enough to a line segment (and crosses the t-axis just once), then the root can be approximated by the point of intersection of the t-axis and a line segment drawn from the first to the last control points. Our problem now is to determine how flat "flat enough" is for a specified acceptable error. For a control polygon crossing just once, (see Figure 3.12 for an example), we can define a bounding box whose top and bottom are parallel to the line connecting the first and last control points, and whose ends are perpendicular to the top and bottom. Then,

```
Solve:
/* Find the root of the fifth-degree polynomial */
begin
    if the control polygon does not cross the t-axis then
        there are no roots, so return none,
    else
        if the control polygon is flat enough and crosses t-axis just once then
            return the t-intercept of chord from λ0 to λ5 as the root
    if the subdivision limit is reached, then
        return the midpoint as the root,
    else
        subdivide the polynomial at t = 0.5,
        Solve left half,
        Solve right half,
        return all solutions thereby generated.
end.
```

Figure 3.11: The approximate inversion algorithm.

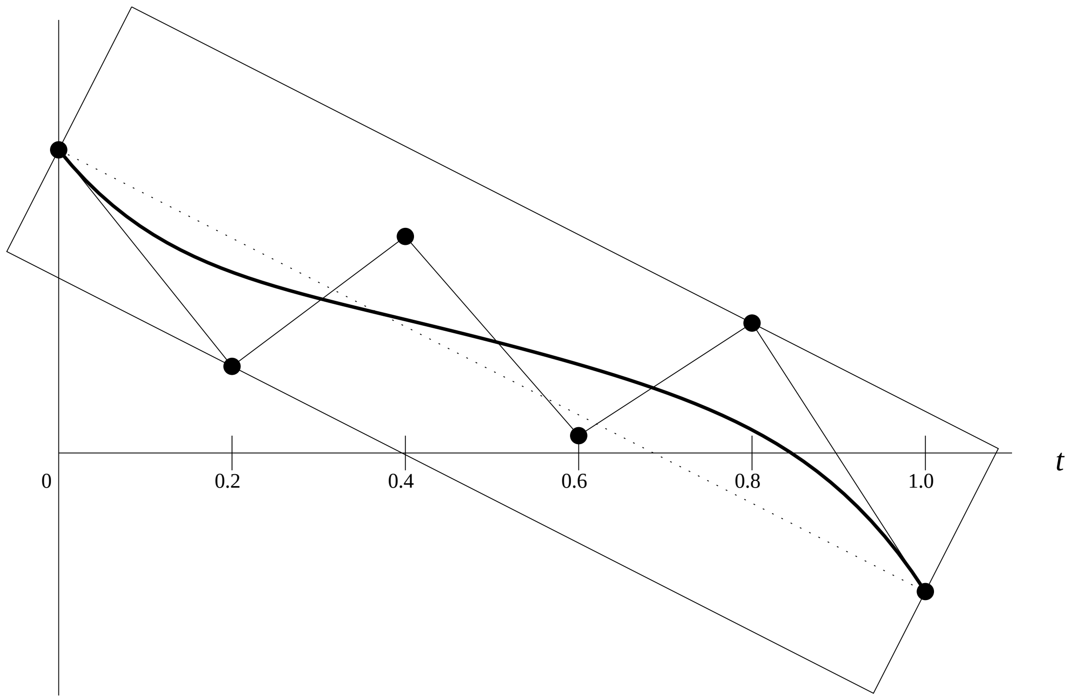


Figure 3.12: Bounding box for a fifth-degree polynomial

the error of the solution is then bounded by half the distance between the points at which the bounding box intercepts the t-axis (or one or both endpoints of the interval [0,1], if the intersection point(s) lie outside this interval). When the error of the solution is within the desired precision of the root approximation, we compute the intersection of the line from the first to last with the t-axis, and return.

Because we are doing binary subdivision of the fifth-degree polynomial, and because the t-values are bounded between 0 and 1, we can easily compute the relationship between the depth of the recursive subdivision and the desired error. At the top level (before any subdivision), the interval is $[0, 1]$ and the expected error is 0.5. Each binary subdivision therefore cuts the expected error in half, so we merely stop the recursion when we have reached a depth corresponding to the base-2 logarithm of the desired error bound. For example, if our desired precision is $\frac{1}{128}$, or $\frac{1}{2^7}$, we set our depth limit to 7 (assuming the "top level" is considered to have depth 1.)

3.5 Editing the curve

Once the designer has drawn in his initial curve, and the system has fitted the piecewise cubic Bézier curve to it, the curve may be modified if the designer so chooses. The modification may involve the addition or removal of features, or simply "adjustment" of portions of the curve. Recall that the author's intention was to simulate the "paper-and-pencil" approach to curve design; this section describes the author's solution to the "erase and redraw" phase of the process.

When the designer is "editing" the curve, the original points making up the digitized curve are gone, and the curve that appears on the screen is the result of evaluating the Bézier segments generated by the fitting process.

3.5.1 The Erasing Step

The first step the designer must take is to specify the region of the curve he wishes to modify. In the paper-and-pencil world, a designer would simply erase the old portion and then draw in a new curve section in its place, or vice versa. The author's approach was that the designer specify two points on the curve, the area between which will be redrawn.

Consider this situation in terms of the Bézier curve representation – the curve C is represented by a sequence $C = (Q_1, Q_1, \ldots, Q_n)$, where each Q_k is a cubic Bézier curve. The user specifies two points in the plane, say $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$. Thus, the first problem to be solved is to represent P_1 and P_2 in terms of the parametric curve representation. That is, we must locate segments Q_k and Q_l that P_1 and P_2 lie closest to (they may in fact be the *same* segment), and compute parameter values u_k and u_l, which represent P_1 and P_2 within Q_k and Q_l, respectively. See Figure 3.13 for a picture of this situation.

Fortunately, we already have the necessary tools to solve these problems. Recall the discussion of the root-finding methods – we may use the approximate inversion method described earlier: The solution was to compute the parameter value u of a point on a

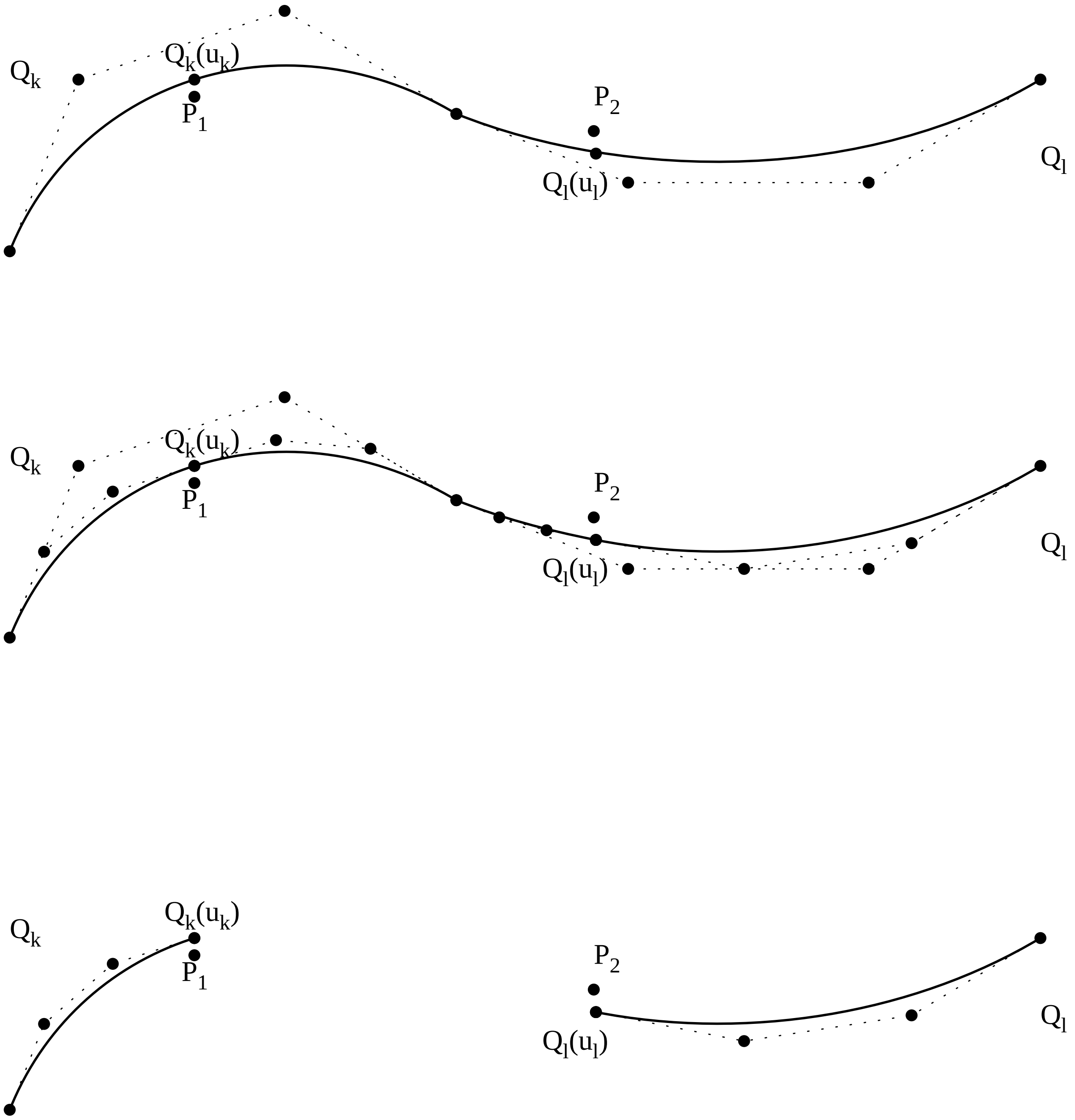


Figure 3.13: Subdividing a fitted curve in preparation for redrawing a section.

curve Q such that point $Q(u)$ was the closest point on Q to the digitized point P under consideration, and then measure the distance from $Q(u)$ to P. In the current situation, we have a point P and a number of segments $Q_1, Q_2, \ldots, Q(n)$. So, we can use the following algorithm to determine both the segment Q_k of curve C that P lies closest to, and the parameter value u for the point on Q_k lying closest to P:

> **Locate**$(P, Q_1, Q_2, \ldots, Q_n)$
> /* Return the location of P */
> **begin**
>   **foreach** Q in $(Q_1, Q_2, \ldots, Q_n)$ **do**
> $u_k \leftarrow$ location of P in Q
>   **end foreach**
>   mindist $\leftarrow =$ distance$(Q_1(u_1),\ P)$
>   **for** $k \leftarrow 2$ to n **do**
>   **if** P is within bounding box of Q_k **then**
>   **if** mindist $\leftarrow min(mindist, distance(Q_k, P))$
> $u \leftarrow u_k$
> $whichsegment \leftarrow Q_k$
>   **return** $(u, whichsegment)$
> **end**

If we apply this procedure to P_1 and P_2, we will have successfully identified the segments that they are closest to, and the parametric values u_k and u_l of the points on the sub-curves Q_k and Q_l closest to P_1 and P_2, respectively, thereby representing the points in terms of the piecewise Bézier representation.

Notice that $Q_k(u_k)$ and $Q_l(u_l)$ represent arbitrary points in curve C. If We simply "wipe out" the curve between these two points, we would be left with "dangling" segments. So, the general problem is this: given a cubic segment (in our case, the ones the user has located his points in), how may we represent some arbitrary portion of the curve in terms of another cubic Bézier segment? The solution to this was given in Section 3.1.4 – this is simply the de Casteljau subdivision algorithm. That is, we subdivide the segments Q_k and Q_l at points u_k and u_l, respectively. Recall that this does not change the curve. Then, the curve between the origin and $Q_k(u_k)$, and the curve between $Q_l(u_l)$ and the terminus are now represented by sequences of cubic Bézier segments, consistent with our intended representation scheme. We can then "erase" the region between the points specified by the designer by deleting them from the sequence. Our original curve C is now represented by the sequence $C = (Q_1, Q_2, \ldots, Q_k, Q_l, Q_{l+1}, \ldots, Q_n)$, where Q_k and Q_l are not connected (the space between them being the erased portion of the curve C). See Figure 3.13 for an example of the entire process.

3.5.2 Drawing in the new curve

Now, the designer can redraw the portion of the curve "erased" in the manner just described. Consistent with the goal that all curve specification and modification be accomplished via sketching, the same algorithms are used for this step as those used in sketching the initial curve. The user, after selecting the section of the curve he wishes to redraw, redraws that section as he sees fit. This new digitized subcurve is fit with a piecewise Bézier curve in almost exactly the same manner as was the entire original curve. The only difference is that this new fitted curve be smoothly integrated with the existing segments on either side of the fitted curve. As we are always maintaining G^1 continuity, we must be sure to maintain continuous unit tangent vector across both joints. So, instead of computing the tangents at either end of the new curve, we simply use the tangents defined by the segments immediately adjacent to the ends of the new curve. This causes the appropriate sets of control points at the joints to be collinear, and thereby maintains G^1 continuity.

3.6 Garbage Collection

Recall that one important goal of the author's curve design system is to minimize the number of segments representing the curve. Now, consider what happens in the typical design process – the designer has drawn in his curve, then spent perhaps many hours editing/changing it in the manner described earlier. The repeated subdivision and re-drawing steps can potentially have the effect of increasing the number of segments in the representation, and these segments may be quite small and/or relatively flat may be the case that a sufficiently similar (from a perceptual standpoint) curve may be representable with many fewer segments – one large segment may adequately represent nearly the same curve as a series of small segments.

So, the general problem may be stated in this manner: given a sequence of Bézier segments representing some curve, how can we derive a smaller sequence of segments that represent the same curve with a minimal amount of change to the curve? As far as the author has been able to determine, this problem has received little or no attention in the research community. There may be some relation to the problems of degree reduction and degree elevation, which have received some attention, but the author has not investigated this possibility.

The approach taken by the author is simple – given a sequence of cubic Bézier segments $S_1, S_2, \ldots, S_n$, we generate a new set of points by evaluating each polynomial S_k at some rate relative to the estimated length of the current segment , yielding a digitized curve. Note that we must retain information regarding which points are corners, etc. Then, we can simply apply the same curve-fitting algorithm that we use for fitting to user-drawn curves.

This approach was implemented as a first attempt at a solution to the garbage-collection problem, and the various parameters influencing the tradeoff between segment reduction and changes in the shape of the curve have been arrived at through experimentation. Much more work must go into this approach for it to be considered completely satisfactory. The current implementation is adequate, but a more thorough investigation is certainly called for.

Chapter 4

Evaluation

The purpose of this chapter is to provide the reader with some concrete examples of Phoenix's operation, to provide quantitative evaluations of the performance of the algorithms on typical digitized curves, to show the system's applicability to fitting curves to digitized fonts, and to provide a subjective evaluation of the approaches taken.

4.1 The Usefulness of Iteration to Improve Parameterization

As stated in the previous chapter, Phoenix, as well as Grossman's [Gro70] and Stone and Plass's system [PS83], use chord-length parameterization as an initial approximation for the fitting step, and then successively refine parameter values for each of the points before trying to refit. The intention is that better parameter values for the points will result in a better fit. Recall Figure 3.8, and the obvious improvement in fit as the parameterization improved. The figure appears to show that the first two or three iterations improve the resulting fit significantly, and that the successive improvement tapers off quickly thereafter. Figure 4.1 show a number of typical curve sections that may be successfully fit with one cubic Bézier segment, and the curves that were fit to them after six iterations. The decrease in the sum of squared errors between the fitted curve and the points, and the decrease in the maximum squared error is shown in Figure 4.2. The correlation between the error decreases of each type should not be surprising, intuitively speaking – one would

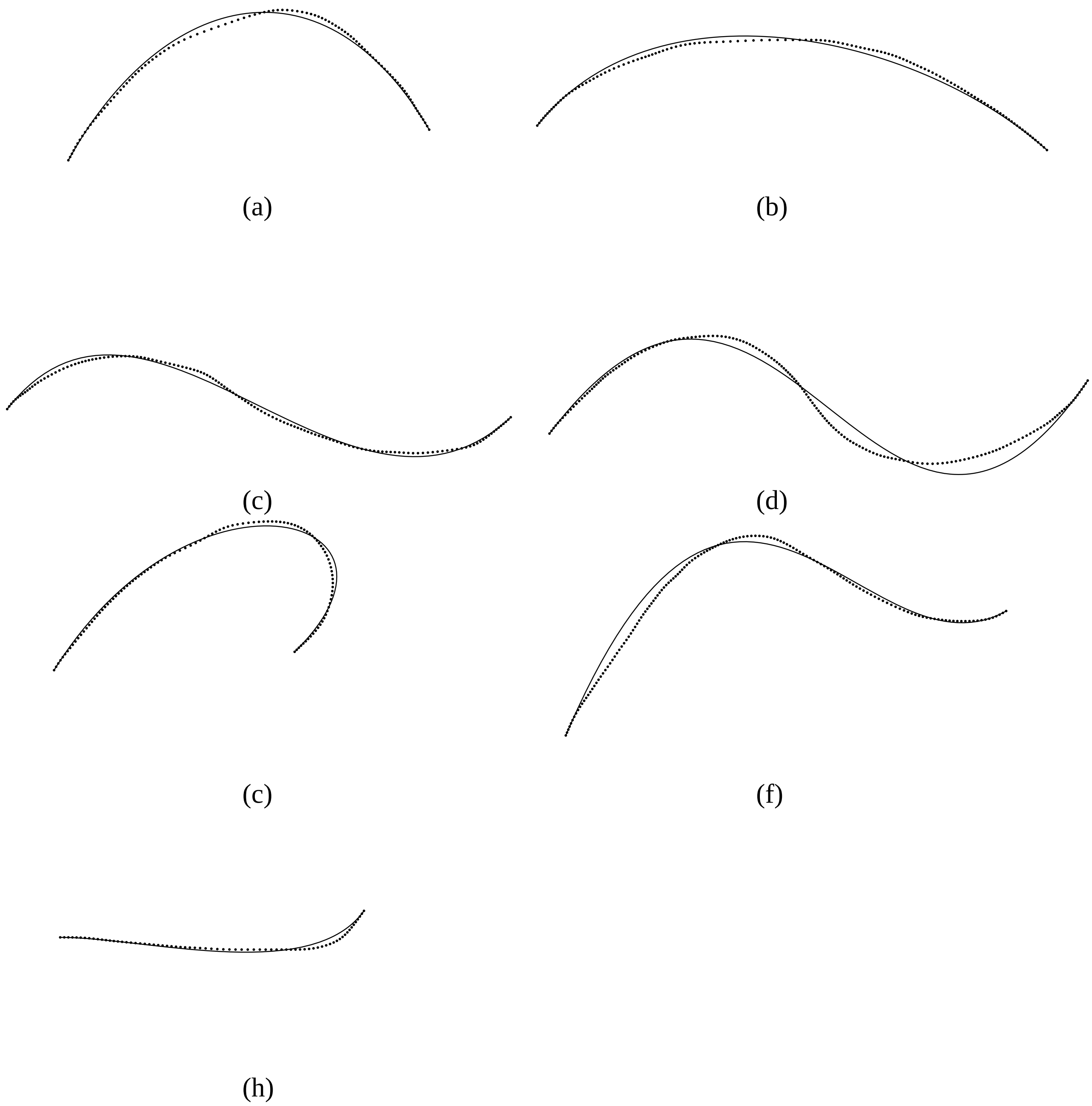


Figure 4.1: Sample sketched curves, and their fitted curves

expect that the maximum error would decrease as the total error decreased. This fact is quite useful – recall that the fitting method used is intended to minimize the sum of squared errors, but that the fitting criterion for a segment is the maximum squared error; the correlation between the two helps justify this approach. The reader may object that the author has chosen for these examples a number of quite "well-behaved" curves – they all have shapes that could be reasonably fit with one Bézier segment. This was intentional – iteration is useless on curves that cannot, by the nature of their shapes (e.g., two or more loops, more than two inflection points, etc.), be fitted with a single segment. Further, while iteration will decrease the sum of squared errors in such a curve, frequently the maximum error will increase, or will alternately increase and decrease. Generally, the sum of squared errors for an impossible-to-fit curve (by one segment, that is) will be several orders of magnitude higher (on the initial fit) than it will be for a curve that might possibly fit (depending on the error tolerance) with one segment. This phenomenon helps us decide whether to attempt iteration, or to give up immediately and subdivide the curve, and refit the smaller and relatively better-behaved sub-curves.

4.2 Font Fitting

One of the most important applications of curve design and fitting systems today is related to font design – the creation and manipulation of type faces. Analytic representations of type faces are important because they can be scaled up and down without changing appearance, their smoothness can be controlled, and they are relatively compact (these features are in marked contrast to bitmap representations, another common representation).

The author's system was intended for use in interactive curve design, but can also be applied to fitting analytic curves to digitized curves of different origins. Fitting curves to digitized or bitmap-defined fonts is discussed here. We present an example of the performance of the system on one letter from a common font set.

The original digitized curve describing the outline of one character is shown in Fig-

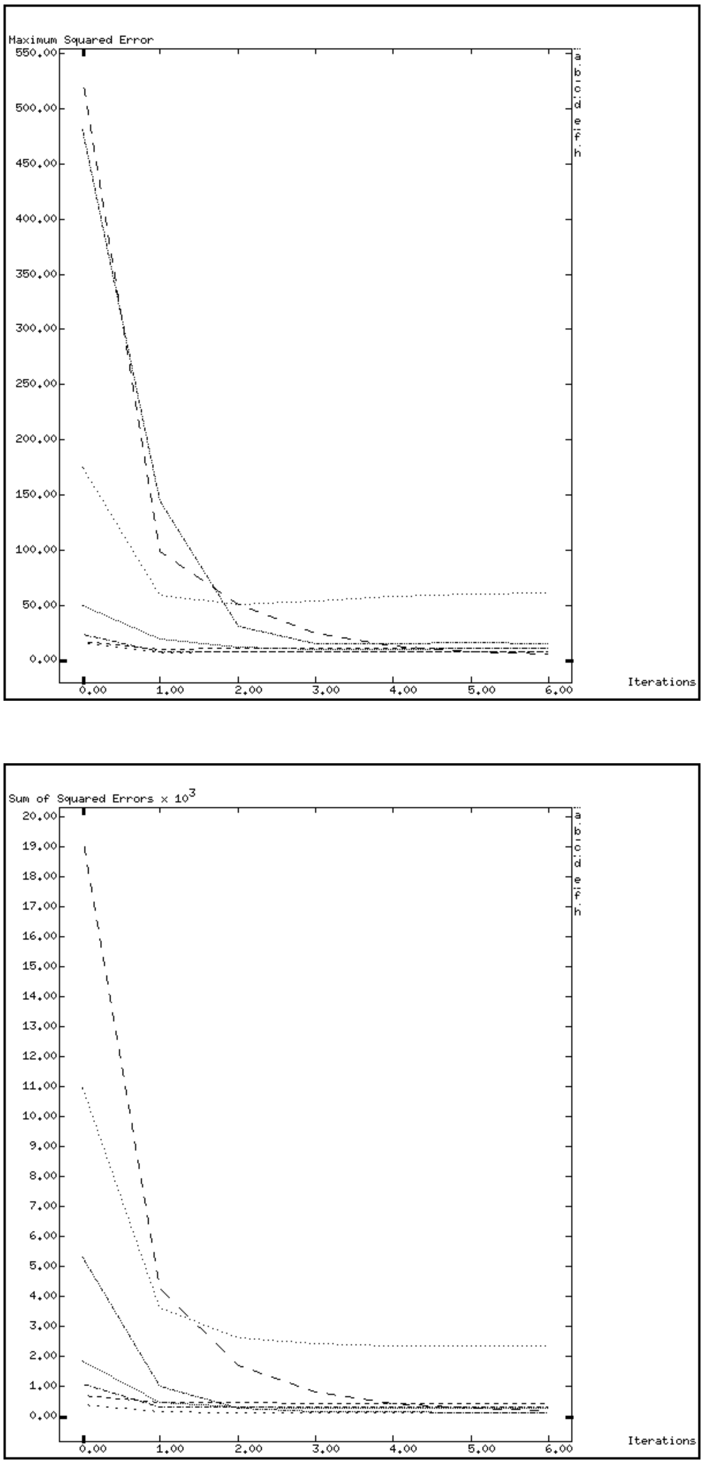


Figure 4.2: Sum of squared errors and maximum errors as a function of iterations

Figure 4.3: The original data, and after filtering.

ure 4.3, in its original form, and after filtering by convolution using a kernel of width three. In this example, one can clearly see how filtering makes the estimation of tangents much more accurate.

An example of the results of fitting a curve to the sampled data, showing the filtered points, control polygons, and control points can be seen in Figure 4.4. The 452 original points have been fitted with 21 cubic Bézier segments, with a maximum error of less than 2 units. This particular level of fit was chosen for this figure so that the control polygons could easily be indentified. Figure 4.2 shows a curve fit with a maximum error of less than one unit, using 29 segments. These same data were fit with varying degrees of maximum error, in order to see how the execution time and number of segments generated as the fit varied. The results of this testing appears in Figure 4.6.

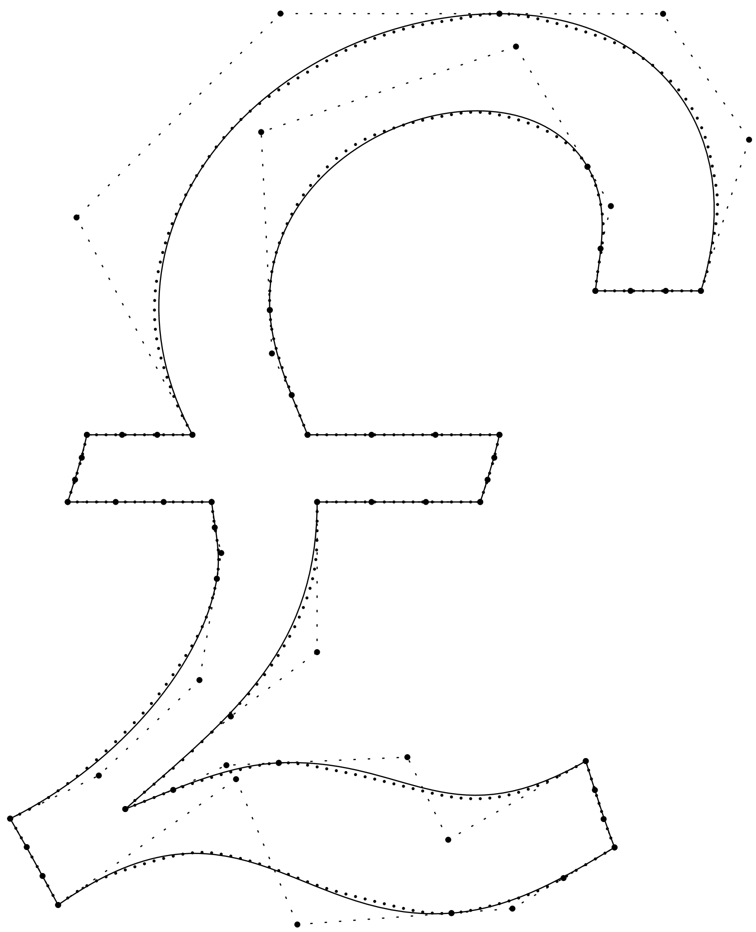


Figure 4.4: Analytic curve fitted to digitized font, showing the control polygons.

4.3 Performance Evaluation

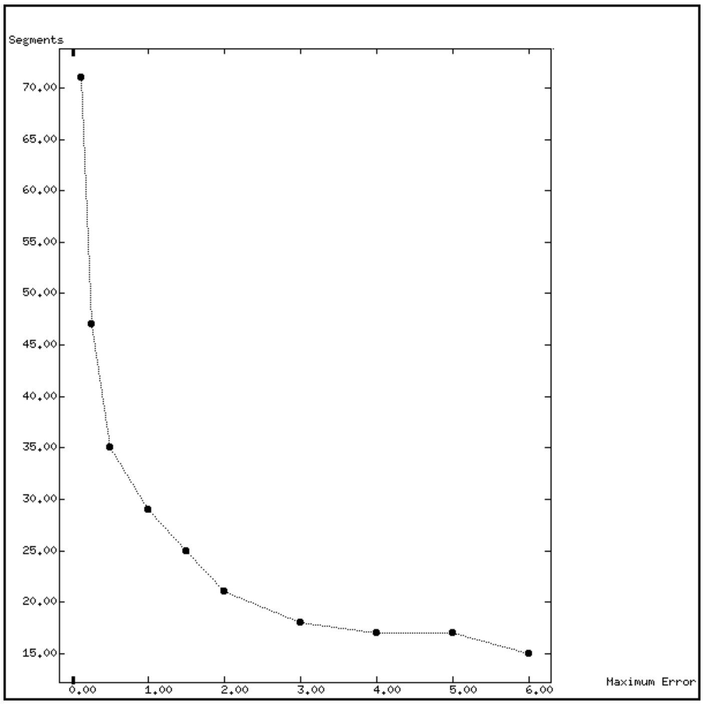
This section presents a performance evaluation of the fitting algorithms. The intent is to show how the system performs, in terms of speed and in terms of numbers of segments generated in fitting a curve, as we vary the closeness of fit.

The test data consists of two sets. The first set, shown in Figure 4.7, consists of eight hand-sketched curves. Each of these is intended to represent, in the extreme, some particular characteristic curve, to show how the system performed under worst-case conditions. It is expected that a designer would generally not be generating such curves. The number

Figure 4.5: Analytic curve fitted to digitized font with negligible error

of segments generated in the fitting process, and the CPU seconds taken, is presented as a graph in Figure 4.8. The second set, shown in Figure 4.9, consists of eight much simpler curves, which are intended to represent more typical curves that a designer may want to fit. Graphs showing the execution time and number of segments generated as a function of the maximum error are shown in Figure 4.10 and Figure 4.11

These performance figures, as well as those for the font fitting data in the previous section, were generated by the author's testbed design system, which was an X Window System (Version 10, Release 4) application program, written in the **C** programming lan-



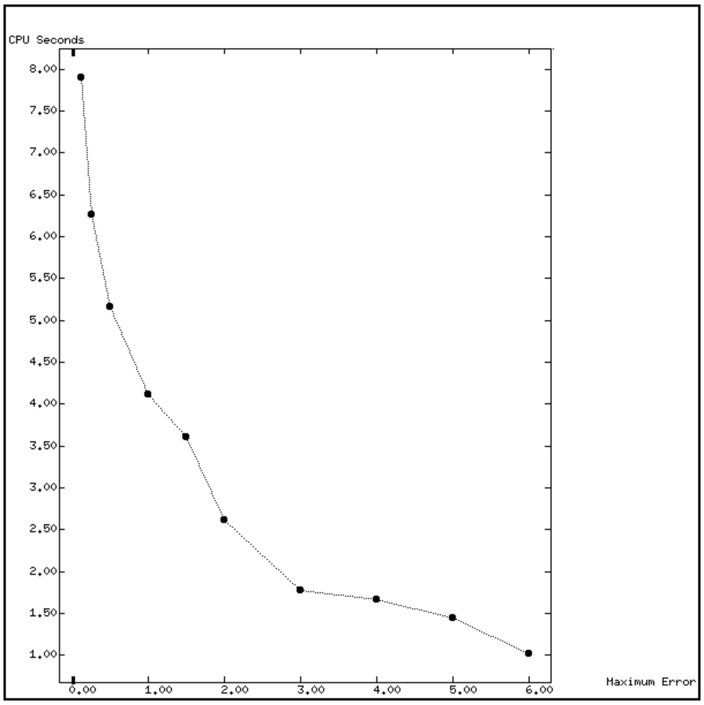


Figure 4.6: Execution time and number of segments generated as a function of maximum error, for the example font.

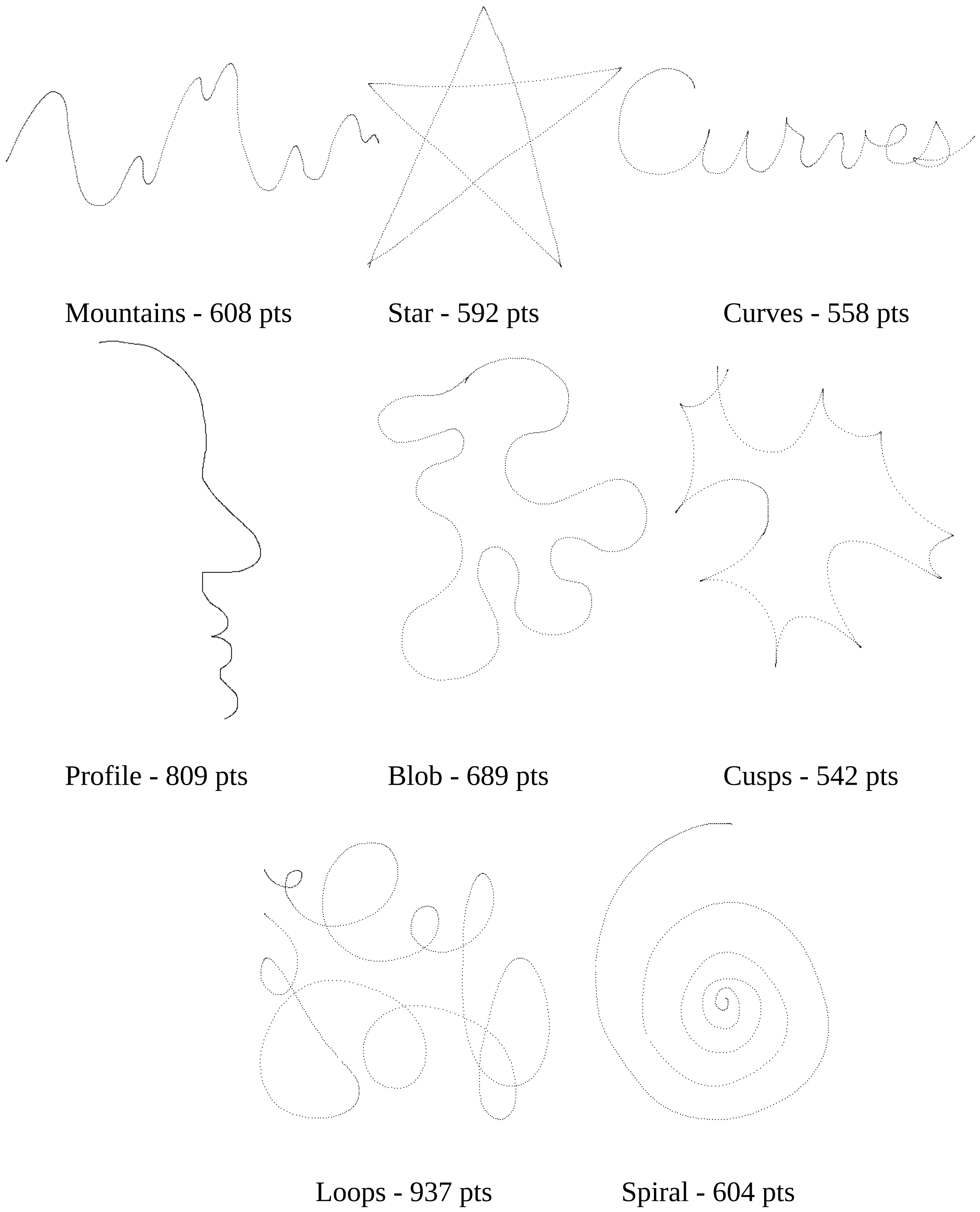


Figure 4.7: Complex shapes used in testing

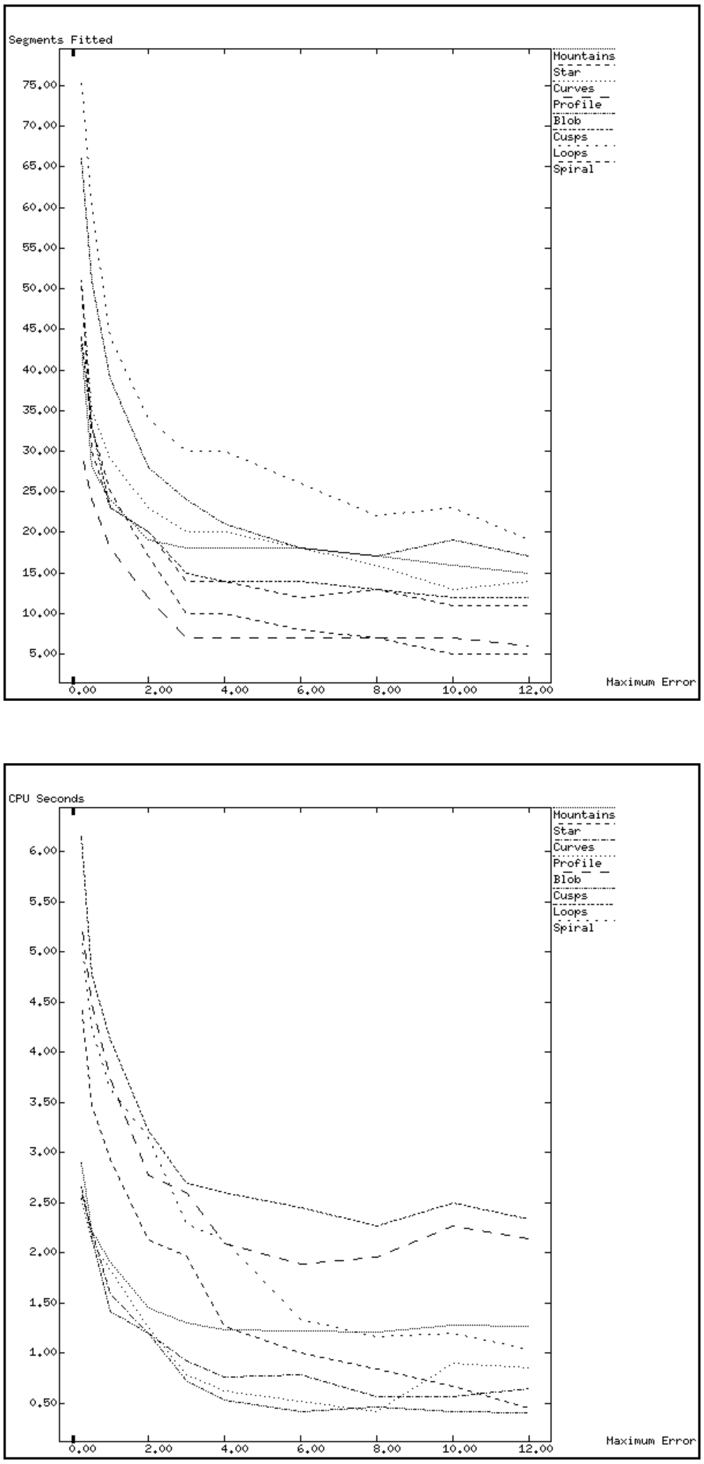


Figure 4.8: Execution time and number of segments generated as a function of maximum error, for the complex shapes.

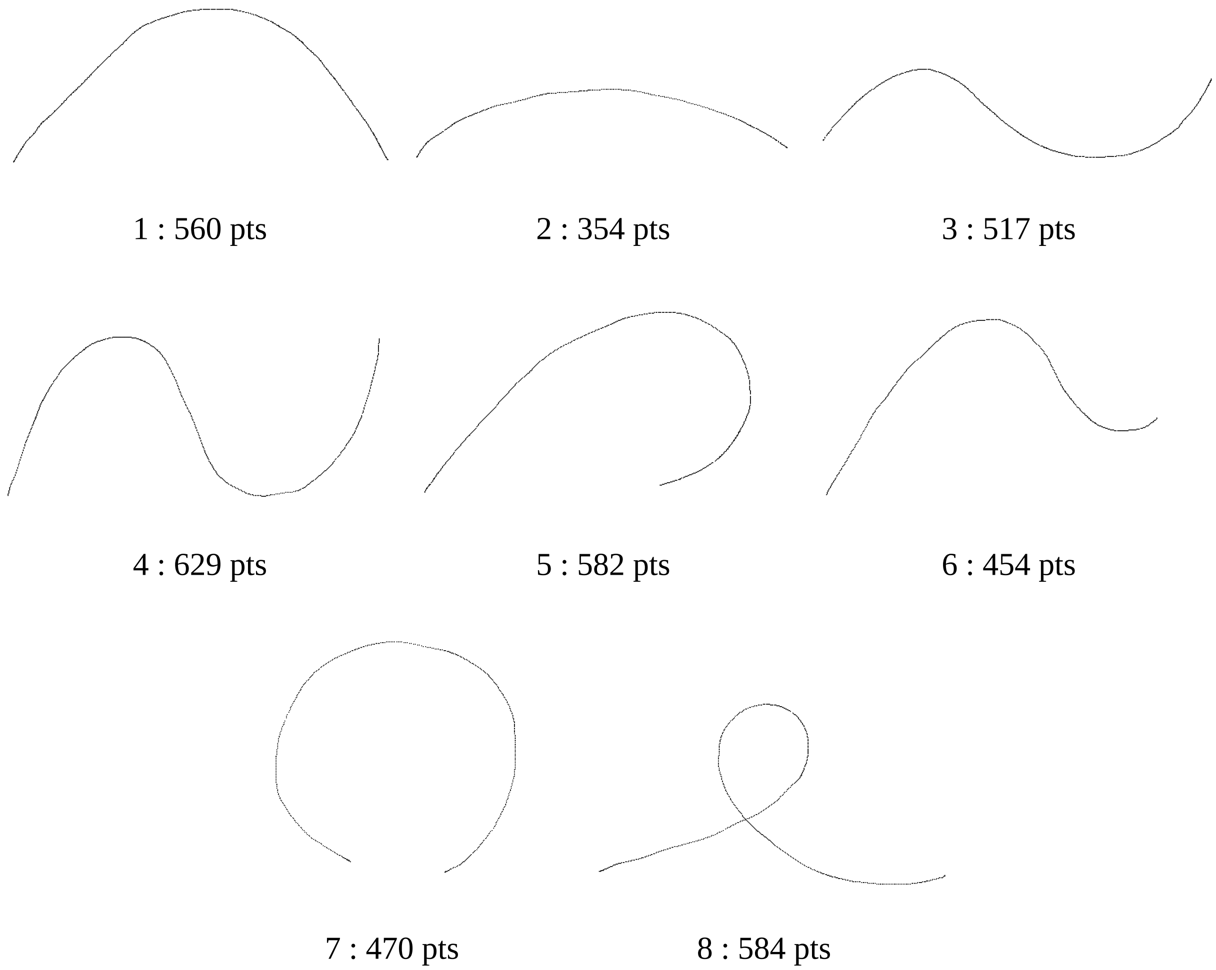


Figure 4.9: Simple shapes used in testing

guage, and which ran on a Digital Equipment Corporation VAX 8650 under Ultrix 1.2. For these tests, the pre-processing steps were as follows:

- Removal of too-close points:
  - Points two pixels or less apart were removed for the complex and simple curve examples.
  - No points were removed for the font fitting.
- Filtering:

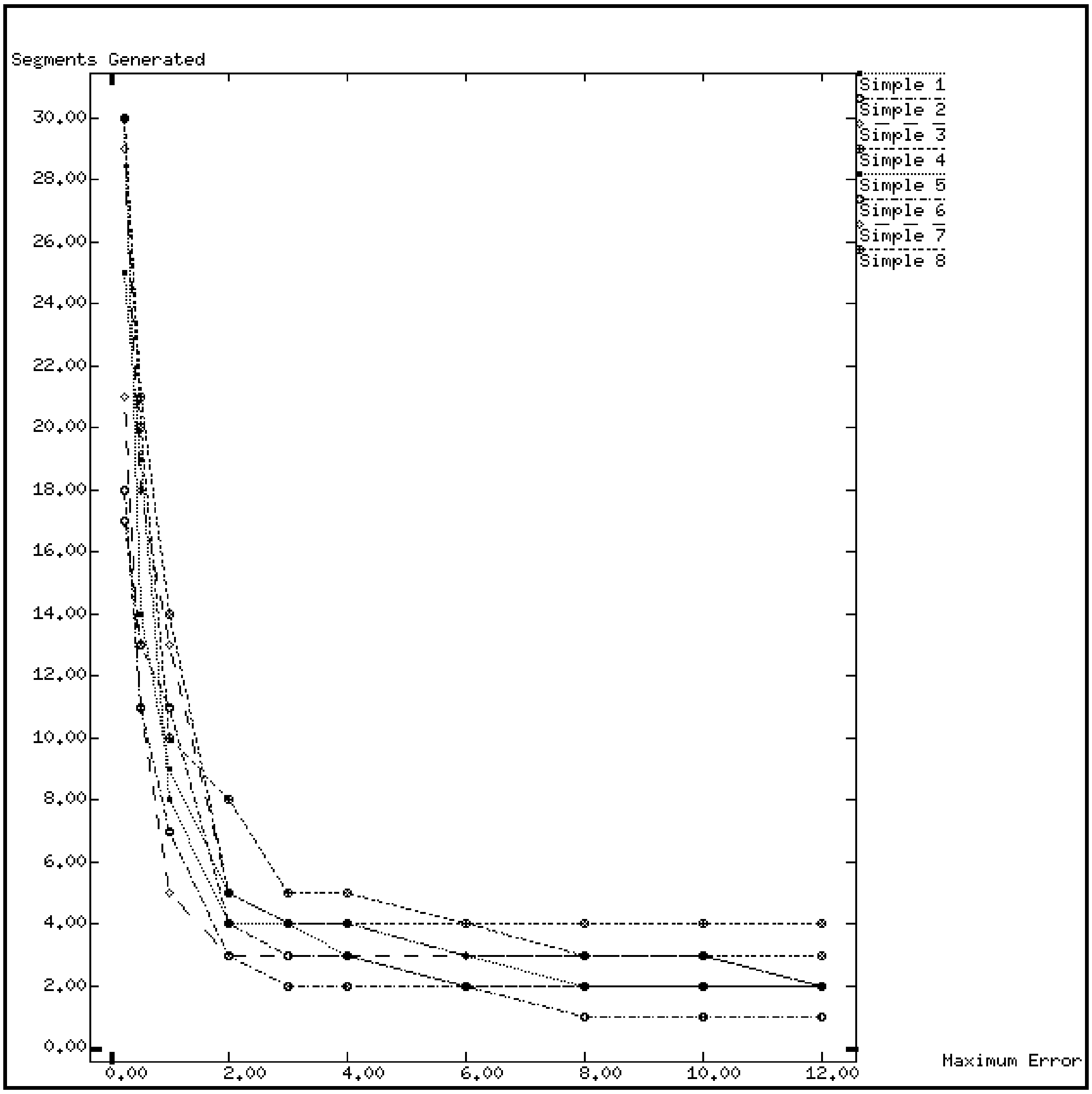


Figure 4.10: Number of segments generated as a function of maximum error, for the simple shapes

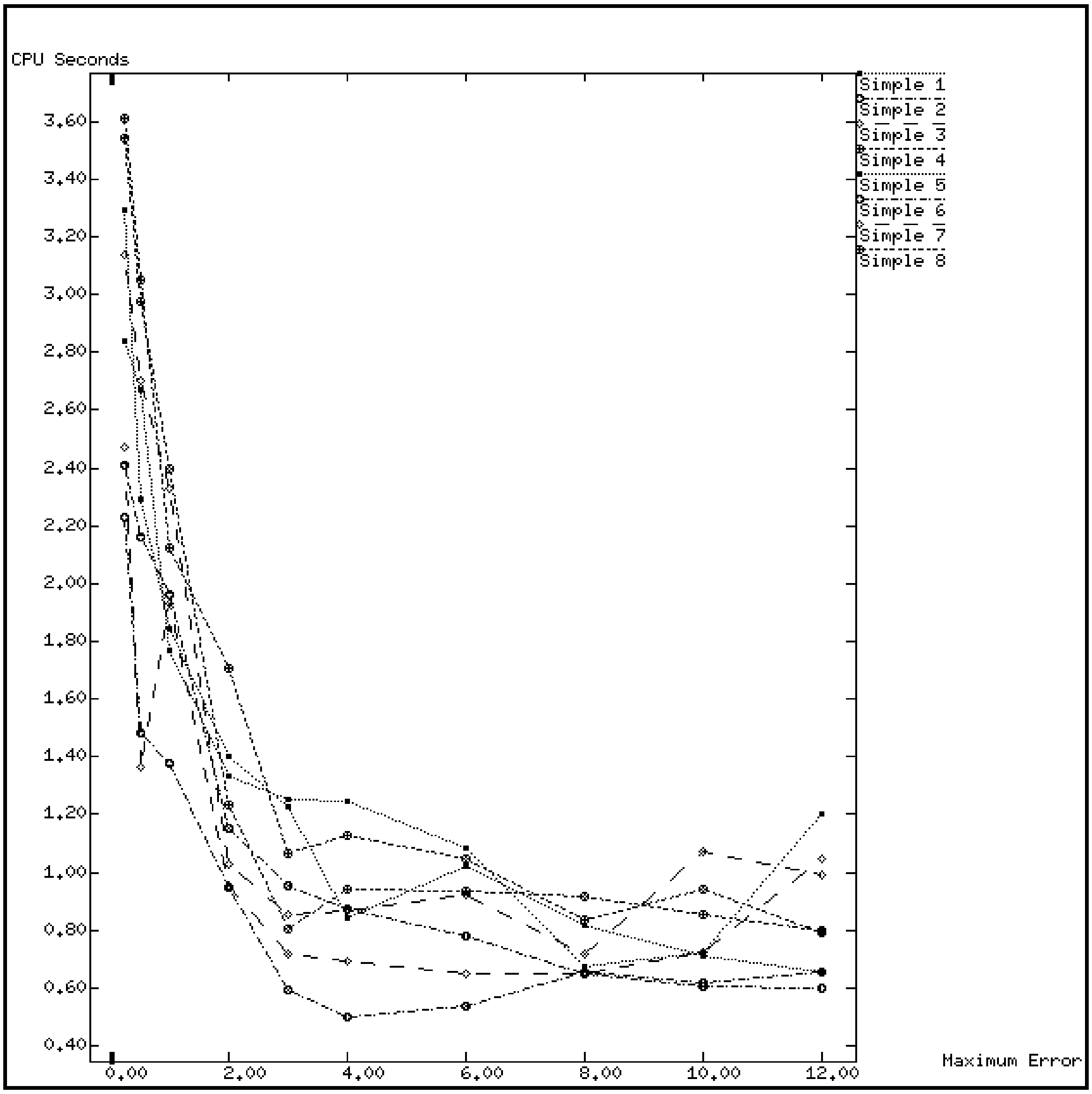


Figure 4.11: Execution time as a function of maximum error, for the simple shapes

  - Six iterations for the complex and simple curves.
  - Ten iterations for the font fitting examples.
- Linear spline fitting: Not used in any of the testing.

These parameters were set to put the system in a worst-case situation – they were set to make the system work harder than necessary (with respect to the visual aspect of the results). In the case of font fitting, as it is not an interactive design problem, we are not as concerned with response times as we are with minimizing the number of segments generated and accurately fitting the curve. In spite of this, the execution times for fitting this particular font character were quite short, especially for the less extreme cases. It should be noted that, visually speaking, a fit with maximum error of less than one or two pixels is unnecessary, as the resulting curve is almost indistinguishable from one fit at a maximum error of two or three pixels, respectively. The same may be said for the complex and simple curve examples.

Phoenix was created as a testbed for different algorithms and approaches, and was not intended to be of production software quality. The implementation was, therefore, extremely inefficient – no code optimization was used, very few macros were used, many procedure calls passed large numbers of parameters, etc. Additionally, the design is excessively modular, to allow for easy replacement of algorithms, and much debugging code was inserted. Because of these characteristics, the execution times should be viewed relative to each other only. The author feels that a more efficient and careful implementation, the use of an optimizing compiler, and the use of pre-processing parameters that further reduce the number of points could decrease the execution times by a factor of four to eight. Further, it should be noted that the execution time in general is approximately equally divided between actually generating the cubic segment for a section of the curve, and measuring the fit. Reeves' [Ree81] suggestion that one measure the error on only a portion of the points in a section would also help to significantly reduce the execution time, particularly in the first several recursive fitting attempts on a complex curve. In these situations, it is clearly unnecessary to measure the error on hundreds of points if there is not a chance for a fit at that particular recursive call to the fitting procedure. One could from the measuring routine immediately after encountering the first point exceeding the desired error bound, but as we subdivide at the point of greatest error, we cannot so exit. If we were to examine, say, every other point on such a curve, then we would cut the measuring time in half. We would not then be guaranteed of finding the point of greatest error, but as curves exhibit "locality of points", we would generally find an immediate neighbor of the point of greatest error, which would in virtually all cases be satisfactory.

The graphs of the execution times and numbers of segments generated as a function of the accuracy of fit, for the font as well as the complex and simple sketched curves, show quite similar and predictable behavior – for extremely tight fits (errors of one pixel or less), large numbers of segments may be generated, using three to ten CPU seconds. By about one- to three-pixel maximum errors, the execution times and numbers of segments has dropped precipitously. At maximum errors of greater than four to six pixels, there is generally little reduction in execution time or number of segments. In some cases, the execution time can be seen to increase. The reason for this phenomenon is as follows – if the shape of a section of a curve can be represented by a single cubic Bézier segment, then the initial chord-length-parameterization-based fit will generally have a maximum error of a few tens of pixels. Larger errors generally means that some significant feature of a curve (such as a loop) has not been represented by the cubic segment. Thus, irrespective of the fitting parameter, there is some minimum number of segments required to represent the shape (in a visually intuitive sense) of the curve. When we reach this minimum number of segments for a particular curve, we have reached a plateau. We will "fall off" the plateau and generate fewer segments only when the fitting parameter is so large that major features of the sketched curve are not represented in the fitted curve. The author's experience is that fits with maximum errors from one to ten pixels constitute the useful range (for the resolution of the devices used by this implementation). A fit of one pixel or so is visually indistinguishible from the original curve, a fit of about three pixels smooths out hand jitter and other undesirable "wobbliness" from the curve (and is about the most generally useful setting), and fits with maximum error of six or more pixels tend to turn the sketched curve into one more resembling a similar curve that would be generated by a more conventional "control-point based" system.

In spite of the previous disclaimer regarding the efficiency of the author's implementation, the response times for fitting the curves were in almost all cases (with the exceptions of the very complex curves being fitted with a maximum error of 0.5 pixels or less), fast enough for satisfactory use in an interactive system – the "real time" response was generally less than three or four seconds (in the case of tight fits on complex curves), and virtually instantaneous for many of the test cases. This has also been the author's experience when using the system in situations other than performance-testing.

In general, the most computationally expensive operation in a design session is the initial fitting of the entire curve. Subsequent redrawing and refitting of the curve generally would involve smaller numbers of points, and is likely to be less complex. Thus, the refitting process involved with editing the curve is quite fast, making it easy for the designer to make many changes to the curve in a short period of time.

Chapter 5

Conclusions and Future Work

This chapter summarizes the work and its significance, and provides some ideas for future research in this area.

5.1 Conclusions

Overall, the goals set out at the beginning of the research have been met: the fitting is fast enough for interactive use, and it is easy to make changes to the curve in an intuitive fashion, with predictable results.

In addition to its usefulness as a basis for an interactive curve design scheme, the fitting algorithm presented here is also useful for static curve representation, both for arbitrary curves and shapes, and for fitting analytic curves to bitmapped fonts.

As suggested in a number of places in this thesis, the approach of a recursive fitting technique is inherently inefficient in that a good deal of work may be done in attempting to fit a single cubic Bézier segment to a set of points which have a shape that cannot be represented by a single segment, or which cannot be fitted within the error bound specified by the user. However, the test cases shown in the previous chapter, and the author's experience using the system, suggest that the algorithm generally converges to an adequate fit quickly enough to be used interactively. A positive aspect of the recursive approach is that each successive call to the fitting routine attempts to fit a smaller number of points (one-half, on average) which have a simpler shape than that at the previous level.

The approach taken to editing a fitted curve has had mixed results. It can be considered successful in that it is easy to use and has predictable results, but it is in some cases "overkill". If one wants to make changes in the shape of the curve in some region, it is a quite nice technique; however, if one is satisfied with the overall shape of a region, but merely would like to "pull it over this way a bit", having to re-draw it seems like the wrong technique. The author thus feels that his approach is not a complete replacement of the traditional systems, but as a supplement or alternative, which may be part of a more complete design system.

5.2 Future Work

The general problem of fitting curves to data has been studied for many years – draftsmen and their splines, numerical analysts and the interpolation and approximation of functions, CAD systems, and all the rest, curve design is still underdeveloped with respect to the state of the art in computational geometry. The author knows of only a handful of systems that allow the user to interactively design geometrically continuous curves. Only recently has commercial software been available that provides the designer, artist, or draftsman with reasonably powerful and intuitive design capabilities (Adobe Illustrator, Gargoyle, and a few others come to mind). Clearly, much work can be done in this area.

With respect to a system (or sub-system) based on the approach taken by the author's research, there is room for more work in the future as well. As far as the author knows, only his system and that of Plass and Stone's fit geometrically continuous curves, and both of these only fit G^1 curves. G^2 curves could be fit to digitized curves using the same approach by using fifth-degree B'ezier curves – this would neccessitate a procedure for estimating curvature, as well as the tangent, at the joints between segments. Otherwise, the form of the solution would be the same, with a few more terms in the equations to be solved in the various steps of the algorithm. Conceptually, this sounds simple, but the performance of the system depends heavily on good estimates of tangents and curvature. Having two such approximations involved would place a heavier burden on the implementor of the system. This may be well worth investigating.

As was already mentioned, the author feels that the redrawing approach to changing the curve is good for some types of changes. However, to be more satisfactory, there should be other methods of changing the curve. One idea that should be investigated is motivated by the desire to shield the user from the control points representing the curve: The user could point to two points on the curve, indicating the endpoints of the section he wishes to change, as is done in the current system. The curve could be subdivided at these points, as is currently done. Then rather than erasing the section, the user could point to a spot on the curve, and pull on that point, making the curve move in the direction he tugs. This could be accomplished in the following manner: the system could again subdivide the curve at the point the user specifies (recall that this does not in itself change the curve). Then, as the user pulls on the curve (this point is now a control point at the junction of two segments), the control point moves as well. In order to maintain G^1 continuity, we must move not only that particular point, but the control points on either side as well, so that they all remain co-linear. The rest of the "triads" of control points that are associated with the section to be changed could be made to move in a manner dependent on their distance along the curve to the point the user is pulling on. The rules for moving the rest of these triads are obviously heuristics; for example, one could move the sets of three points in a direction parallel to that the user is moving the point he specified some percentage of the distance, based on how far away they are. If the percentages dropped off quickly with the distance, the curve would tend to behave as if it were highly elastic; if the percentages dropped off very little, the whole curve in the area specified would tend to "bulge" in or out in the direction the user pulled. This drop-off rate could be controlled by a single parameter specified by the user.

Another alternative for curve editing would be to make the control points themselves available to the user at his explicit request. He then could take the traditional approach of moving control points to change the curve. One would then have to make the decision of how to treat the continuity conditions at the joints.

The author feels that a good system might make use of all three of these types of curve-editing, allowing the user to interact with the system in whatever manner he feels is most appropriate. These methods can be made completely independent, so that they would not interact in non-intuitive ways.

As was mentioned in Chapter 3, the "garbage-collection" phase of the system was not well-developed. Clearly, after much editing of the type now implemented, many small segments may be underlying a section of a curve that could possibly be represented by a smaller number of Bézier segments. The current approach, that of sampling the segments at a rather fine resolution, and then refitting the subsequent digitized curve does work fairly well, and is not too inefficient. It also has a good intuitive feel, and fits in rather elegantly with the philosophy and design of the system in general. However, it would be an interesting topic to attempt to solve the more general problem of (approximately) representing a piecewise Bézier curve with successively smaller numbers of segments, using only the mathematical representation.

A user-interface issue involved with this system is that of making it easy for the user to get exactly the fit he wants. Currently, through a little experience, the user can interactively specify the fit he wants, and the system will generate a fitted curve in a predicatable manner. If the user is unhappy with the fit, he can "undo" the fit, specify a new fitting parameter, and re-fit the curve. This is a bit awkward. Ideally, the user should be able to turn a dial or move a slider, and have the system display a fitted curve that fits tighter or looser as the user changes the position of the control device. This may, in fact, be possible with a better implementation on a sufficiently fast machine. One could use in this case two fitting techniques – as the user changes the fit by adjusting the dial or slider, the system could use a small subset of the points to fit (subsetting them by using the linear spline fitting technique, for example). Then, when he releases the controller, a more precise fitting could be done, using more points. The author has found that the shape of a curve fitted by using a smaller number of points does not differ so radically from a curve fitted to the full set of points as to make this approach unacceptable.

A "hidden agenda" of this thesis was to test out design ideas in two dimensions, to see if they were feasible, as a precursor to trying out a similar approach for a three-dimensional design system. With appropriate 3-D input and output devices, such as have been developed recently [Fol87], one could (conceptually) extend this fitting idea. The approach has been relatively successful in 2-D; it would be interesting to see if similar success could be accomplished in 3-D.

Bibliography

[AA56] F. Attneave and M. D. Arnoult. The quantitative study of shape and pattern perception. *Psychological Bulletin*, 53:273–288, 1956.

[AHU74] A.V. Aho, J.E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Aki70] Hiroshi Akima. A new method of interpolation and smooth curve fitting based on local procedures. *JACM*, 17(4):589–602, October 1970.

[Alb74] Antonio Albano. Representation of digitized contours in terms of conic arcs and straight-line segments. *Computer Graphics and Image Processing*, 3:23–33, 1974.

[Bar81] Brian A. Barsky. *The Beta-Spline: A Local Representation Based on Shape Parameters and Fundamental Geometric Measures.* PhD thesis, University of Utah, 1981.

[BBB87] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to the Use of Splines in Computer Graphics.* Morgan Kaufman Publishers, Inc., 1987.

[Bel57] R. Bellman. *Dynamic Programming.* University Press, 1957.

[BFK84] Wolfgang Boehm, Gerald Farin, and J. Kahman. A survey of curve and surface methods in cagd. *Computer-Aided Geometric Design*, 1:1–60, 1984.

[Boe86] Wolfgang Boehm. Curvature continuous curves and surfaces. *Computer-Aided Design*, 18(2):105–106, March 1986.

[Boo79] Fred L. Bookstein. Fitting conic sections to scattered data. *Computer Graphics and Image Processing*, 9:56–71, 1979.

[Can71] A. Cantoni. Optimal curve-fitting with piecewise linear functions. *IEEE Transactions on Computers*, C-20:59–67, 1971.

[Cd72] S. D. Conte and C. deBoor. *Elementary Numerical Analysis.* McGraw-Hill, 1972.

[Cha74] George M Chaiken. An algorithm for high-speed curve generation. *Computer Graphics and Image Processing*, 3:346–349, 1974.

[Cho80] Won L Chong. Automatic curve fitting using an adaptive local algorithm. *ACM Transactions on Mathematical Software*, 6(1):45–57, March 1980.

[Cli74] A. K. Cline. Scalar- and planar-valued curve fitting using splines under tension. *Communications of the ACM*, 17:218–223, 1974.

[Cox71] M. G. Cox. Curve fitting with piecewise polynomials. *J. Inst. Maths. Applics.*, 8:36–52, 1971.

[Dav75] P. J. Davis. *Interpolation and Approximation.* Dover Publications, New York, 1975.

[DeB78] C. DeBoor. *A Practical Guide to Splines.* Springer-Verlag, 1978.

[DeR85] Anthony D. DeRose. *Geometric Continuity: A Paramterization Independent Measure of Continuity of Computer Aided Geometric Design.* PhD thesis, University of California, Berkeley, 1985.

[Die82] P. Dierckx. Algorithms for smoothing data with periodic and parametric splines. *Computer Graphics and Image Processing*, 20:171–184, 1982.

[Dub87] R. Peter Dube. Preliminary specification of spline curves. *IEEE Transactions of Computers*, C-28(4):286–290, April 1987.

[Ear85] R. A. Earnshaw. *A Review of Curve Drawing Algorithms*, pages 290–301. Springer-Verlag, 1985.

[EM77] T. M. R. Ellis and D. H. McLain. Algorithm 514 – a new method of cubic curve fitting using local data. *ACM Transactions on Mathematical Software*, 3(2):175–178, June 1977.

[FB85] Alain Fournier and Brian A. Barsky. Geometric continuity with interpolating bezier curves. In *Graphics Interface '85*, pages 337–341, 1985.

[FG69] Herbert Freeman and Jeremy M. Glass. On the quantization of line-drawing data. *IEEE Transactions of Systems Science and Cybernetics*, SSC-5(1):70–79, January 1969.

[FJ84] David R. Ferguson and Alan K. Jones. *Parametric Curve Fitting With Shape Control.* Engineering Technology Applications Technical Report ETA-TR-20, Boeing Computer Services, August 1984.

[Fol86] Thomas A. Foley. Scattered data interpolation and approximation with error bounds. *Computer Aided Geometric Design*, 3:163–177, 1986.

[Fol87] James D. Foley. Interfaces for advanced computing. *Scientific American*, 257(4):126–135, 1987.

[FP79] I.D. Faux and M. J. Pratt. *Computation Geometry for Design and Manufacture.* Ellis Horwood, 1979.

[Fre74] Herbert Freeman. Computer processing of line-drawing images. *Computing Surveys*, 6(1):57–97, March 1974.

[Fre79] Herbert Freeman. Use of incremental curvature for describing and analyzing two-dimensional shape. In *Proceedings of PRIP*, pages 437–444, 1979.

[FW66] A.H. Fowler and C.W. Wilson. *Cubic Spline: A Curve Fitting Routine*. Report Y-1400, Union Carbide Corporation, 1966.

[Gil78] W. K. Giloi. *Interactive Computer Graphics*. Prentice-Hall, 1978.

[Glu62] B. Gluss. A line segment curve-fitting algorithm related to optimal encoding of information. *Information Control*, 5:261–267, 1962.

[Gol83] Ronald N. Goldman. An urnful of blending functions. *IEEE Computer Graphics and Applications*, 49–54, October 1983.

[Gro70] M. Grossman. Parametric curve fitting. *The Computer Journal*, 14(2):169–172, 1970.

[Her76] Christopher F. Herot. Graphical input through machine recognition of sketches. In *Proceedings of SIGGRAPH '76*, pages 97–102, 1976.

[HN82] Koichi Harada and Eihachiro Nakamae. Application of the bezier curve to data interpolation. *Computer-Aided Design*, 14(1):55–59, January 1982.

[IK66] E. Isaacson and H. B. Keller. *Analysis of Numerical Methods*. Wiley, 1966.

[IKY77] Kozo Ichida, Takeshi Kiyono, and Fujiichi Yoshimoto. Curve fitting by a one-pass method with a piecewise cubic polynomial. *ACM Transactions on Mathematical Software*, 3(2):164–174, June 1977.

[Koz86] Jernej Kozak. Shape preserving approximation. *Computers in Industry*, 7:435–440, 1986.

[LP81] O. Lozover and K. Preiss. Automatic generation of a cubic b-spline representation for a general digitized curve. In J. L. Encarancao, editor, *Eurographics '81*, pages 119–126, North-Holland Publishing Company, 1981.

[McC70] D. J. McConalogue. A quasi-intrinsic scheme for passing a smooth curve through a discrete set of points. *The Computer Journal*, 13(4):11–15, November 1970.

[McL83] Harry W. McLaughlin. Shape-preserving planar interpolation: an algorigthm. *IEEE Computer Graphics and Applications*, 58–67, May/June 1983.

[Mon70] U. Montanari. A note on minimal length polygonal approximation to a digitized contour. *Communications of the ACM*, 13:41–47, 1970.

[Pav83] Theo. Pavlidis. Curve fitting with conic splines. *ACM Transactions on Graphics*, 2:1–31, 1983.

[Pie86a] L. Piegl. Curve fitting algorithm for rough cutting. *Computer-Aided Design*, 18(2):79–82, March 1986.

[Pie86b] L. Piegl. A geometric investigation of the rational bezier scheme in computer-aided design. *Computers in Industry*, 7:401–410, 1986.

[Pie86c] L Piegl. Representation of rational bezier curves and surfaces by recursive algorithms. *Computer-Aided Design*, 18(7):361–366, September 1986.

[Pie87] L Piegl. Interactive data interpolation by rational bezier curves. *IEEE Computer Graphics and Applications*, 45–58, April 1987.

[PS79] Theo. Pavlidis and D. J. Sakrison. Applications of a simple statistical model for curves. In *Proceedings of PRIP*, pages 599–603, 1979.

[PS83] Michael Plass and Maureen Stone. Curve-fitting with piecewise parametric cubics. *Computer Graphics*, 17(3):229–239, July 1983.

[Ram72] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1:244–256, 1972.

[Ree81] William T. Reeves. *Quantitative Representations of Complex Dynamic Shapes for Motion Analysis*. PhD thesis, University of Toronto, 1981.

[Rei67] Christian H. Reinsch. Smoothing by spline functions. *Numerische Mathematik*, 10:177–183, 1967.

[Ric64] John R. Rice. *The Approximation of Functions, Volume 1*. Addison-Wesley, 1964.

[Ric78] John R. Rice. Algorithm 512 – adapt, adaptive smooth curve fitting. *ACM Transactions on Mathematical Software*, 4(1):82–94, March 1978.

[Rie75] R. F. Riesenfeld. On chaiken's algorithm. *Computer Graphics and Image Processing*, 4:304–310, 1975.

[Rou80] John A. Roulier. Constrained interpolation. *SIAM J. Sci. Stat. Comput.*, 1(3):333–344, September 1980.

[SBD86] F. J. M Schmitt, Brian A. Barsky, and Wen-Hui Du. An adaptive subdivision method for surface-fitting from sampled data. In *Proceedings of SIGGRAPH '86*, pages 179–188, 1986.

[Sch66] D. G. Schweikert. An interpolation curve using a spline in tension. *J. Math. Phys.*, 45:312–317, 1966.

[SCH72] J. Sklansky, R. L. Chazin, and B. J. Hansen. Minimum perimeter polygons of digitized silhouettes. *IEEE Transactions on Computers*, C-23:1355–1364, 1972.

[SDPO81] P. Suetens, P. Dierckx, R. Piessens, and A. Oosterlinck. A semiautomatic digitization method and the use of spline functions in processing line drawings. *Computer Graphics and Image Processing*, 15:390–400, 1981.

[SG81] J. Sklansky and V. Gonzalez. Fast polygonal approximation of digitized curves. In *Proceedings of PRIP*, pages 604–609, 1981.

[SG86] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Sto61] H. Stone. Approximation of curves by line segments. *Math. Comp.*, 15:40–47, 1961.

[VPV87] Christine Vercken, Christine Potier, and Sylvie Vignes. Spline curve fitting for an interactive design environment. In *Theoretical Foundations of Computer Graphics and CAD*, NATO, 1987.

[WAG77] Sheng-Chuan Wu, John F. Abel, and Donald P. Greenburg. An interactive computer graphics approach to surface representaion. *Communications of the ACM*, 20(10):703–712, October 1977.

[Wat87] Mark Allen Watkins. *Degree Reduction of Bezier Curves and Surfaces.* Master's thesis, University of Utah, 1987.

[Wil78] Charles M. Williams. An efficient algorithm of the piecewise linear approximation of planar curves. *Computer Graphics and Image Processing*, 8:286–293, 1978.

[Yam78] Fujio Yamaguchi. A new curve fitting method using a crt computer display. *Computer Graphics and Image Processing*, 425–437, 1978.

[YKC*86] Mark Yang, Chong-Kyo Kim, Kuo-Young Cheng, Chung-Chin Yang, and S. S Liu. Automatic curve fitting with quadratic b-spline functions and its applications of computer-aided animation. *Computer Vision, Graphics, and Image Processing*, 33:346–363, 1986.