# Tutorial on Curve Fitting for GIS

Wm Randolph Franklin
Rensselær Polytechnic Institute
Troy, NY, 12180 USA

*Phone:* +1 (518) 276-6077, *Fax:* +1 (518) 276-6261
*Internet:* wrf@ecse.rpi.edu

**Abstract**

This paper is an introduction to curves and splines for representing cartographic data. We consider why a curve might be better than a chain of points, what a good fit to the data means, the data characteristics, the desired operations on the data, the possible forms of equations, and the Bézier curve. For representing cartographic data, we recommend a parametric cubic spline. We consider the B-spline, the Catmull-Rom-Overhauser spline, producing points from the curve, and finding knots for the spline. We recommend the Douglas-Poiker line generalization algorithm for the latter.

## 1 Introduction

The question of data representation in science has been studied ever since observations were quantified. The issue is important since the proper formulation can make an impossible problem simple. Consider, for example the effect on geometry of Descartes' discovery of analytic geometry as an alternative to synthetic geometry. In cartography, representation questions are also important. Consider the implications of a polygon-based versus an edge-based data structure for a map.

In this paper we consider the representation of non-straight lines in cartography by curves instead of as chains of points. Since curves have been used in computer graphics and computer aided design (CAD) for a few decades, there is a body of knowledge to draw on, which is widely available. The well-known Bézier curves are described in most graphics and computer aided design (CAD) texts, such as Foley, van Dam, Feiner, and Hughes[FvDFH90], Faux and Pratt[FP81], and Choi[Cho91].

## 2 Why use curves?

Suppose that we need to represent non-straight line, such as a coastline or highway entrance ramp. Although using a chain of points, implicitly connected by straight line segments, is traditional, there are several reasons to use curves instead.

**Space** If the outline is smoothly curved, then representing it as a curve is much more compact. In the extreme case, if the outline is a perfect circle with 1 cm radius, representing it as a sequence of points such that the lines between them are always within 0.1 mm of the exact circle will require about 20 points. If we want one part in one million accuracy, then about 2000 points are necessary. (In general, if we space a point every $d$ along the circle, then the error is about $d^2/8$).

Although approximation by selecting some subset of the points seems universal, creating new points that are not in the given set is more efficient. When approximating a circle, if points slightly outside are created, then for any given number of points, the error is halved since the line between the points goes inside then outside the circle by equal distances. Therefore only 14, not 20, points are required for 0.1 mm accuracy, and only 1400, not 2000, points for one part in one million. If the points are spaced $d$ apart, then the error is about $d^2/16$.

Note that the cost of storing a large database rises in steps as the database size exceeds the available space on the current medium.

**Time** Data that require more space also require more time to read and process. Many computers with fast processors are, in practice, limited by the speed of the I/O. Many users would be better served by slower computers with faster, larger, peripherals.

**Looks** A piecewise straight line may look irregular, no matter how many points are used, unlike the original curve.

**Scalability** The appropriate number of points depends on the scale, but a curve is smooth at any resolution.

This is not to say that curves should always be used, but that often they may be a reasonable alternative to a chain of points.

## 3 Criteria for goodness of fit

Even when we use a curve, achieving an exact fit is often infeasible. Therefore, what constitutes an adequate approximation? Several criteria have been used. Since almost any method gives good results if you include enough terms, i.e., have enough degrees of freedom, the question is how different methods compare when using the same number of terms.

**Match N derivatives at the initial point** This would seem inappropriate since we are rarely interested in more than about two derivatives, and we are interested in the whole interval, rather than just the initial point. However, this is the criterion satisfied by a Taylor series, such as $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$

**Minimize the average error** This looks attractive, but it could produce a result that is mostly excellent, but contains a large narrow error spike. It is also hard to calculate.

**Minimize the maximum error (minimax)** This is the basis of the Chebyshev polynomial approximation to a function. It is more economical than the, more popular, Taylor approximation, i.e., a Chebyshev polynomial is usually a better approximation than a Taylor polynonial with the same number of degrees of freedom. However, a Chebyshev approximation can be harder to calculate, and might produce a result that, while always close, oscillates a lot.

**Variation minimization** This is a property that an approximation may, or may not, have. It means that the output curve should remain inside the convex hull of the input points. Also the output curve should not oscillate more than the input sequence of points. This is a desirable property in CAD, where the points are set by a designer merely to indicate the form of the curve. However, since we wish to interpolate the points, and produce a curve through them, or at least through some of them, then variation minimization is undesirable here.

Also, when interpolating elevation data, it *is* desirable that a peak or pit be inferred even though we have no data point there. Such a curve would go outside the convex hull of the data points.

The minimax technique looks the most attractive, since the excess oscillations do not occur in practice.

# 4 Characteristics of the data

We must also consider the form of the input data, since some interpolation techniques are incompatible with some certain classes of data. The are various possible data characteristics.

**Continuous and differentiable k times ($C^k$)** A continuous function need not be differentiable anywhere, and, if it is differentiable once, it may not be differentiable twice, and so on. The discovery of such functions in the nineteenth century forced calculus to be formalized better.

In industrial CAD, $C^2$ is desirable since otherwise consumers can see surface kinks, which are artifacts of the design process. However, the real world is not so smooth, and often not even continuous ($C^0$).

**Fractal** This is the opposite extreme case. However the real world is not really fractal either, if only because erosion is asymmetric. Mountain tops are sharper than ocean trenches.

The proper characterization of cartographical data is quite difficult. The recent concept of *wavelets* appears attractive, but a full solution is not yet in sight.

# 5 Desired operations

Choosing the best form of equation for a curve requires us to consider the operations that we wish to perform on it. Here are the most frequent operations.

**Produce points** on the curve, to plot it. This is easy for a chain of points, somewhat harder for a curve.

**Test** whether a point is on the curve, or determine to which side of the curve it lies. This might be easier for a curve than for a long chain, since then the point would need to be tested against every edge segment on the chain, unless we used an auxiliary data structure, such as a uniform grid, Franklin[FKN89, FNK+89].

Testing which side of the curve or chain a point is on is complicated in either case. A chain of points requires some auxilary data structure, such as trapezoids partitioning the plane. For a curve, see the paragraph on parametric curves in section 6.1 below.

**Rotate** and otherwise transform of the curve. This is easy for points, while the difficulty for a curve depends on the form of the equation.

**Calculate arc length.** We can't integrate most forms of curves exactly, but, since they are smooth, most approximation techniques work. The arc length of a chain of points is easy, but may be slower to calculate for a long chain than for the corresponding curve because of the number of square-root function calls.

**Calculate area.** This can be determined exactly for some curves; for the others, the above comments on approximations apply.

**Intersect two curves.** This is hard for both chains of points and curves. Intersecting two chains of points efficiently requires an auxiliary data structure, such as a uniform grid. Alternatively we might put boxes around the chains and subdivide. Intersecting two curves generally produces polynomial equations for the resulting point. One method uses resultants, described later in section 6.2.

# 6 Form of equation

Once we have decided to approximate a set of points by a curve, there is the question of the mathematical form of the equation. The perfect form would allow the preceeding operations to be performed efficiently; unfortunately no one equation form does them all well. The following sections describe three choices that we must make.

## 6.1 Explicit, implicit, or parametric?

**Explicit** e.g., $y = x^2$. Producing and testing points is easy. However the curve must be split into single-valued segments since one function produces only one $y$ value for each $x$. If the curve is rotated, then the segments must be redetermined. Therefore explicit curves are not used much where the data may be rotated.

**Implicit** e.g., $x^2+y^2 = 1$. It's easy to test whether a point is on such a curve, and the form of the equation stays the same if the curve is rotated, although the coefficients change. However, it's difficult to produce a sequence of points along the curve. Given one point on the curve, we must determine the tangent and curvature there. Then we step along the tangent for a distance inversely proportional to the curvature. This gives a new point that is probably close to, but almost certainly not on, the curve, so we must move the point onto the curve. Various nasty things can cause errors, so implicit curves are also uncommon.

**Parametric** e.g.,

$$x = x(t) = \frac{t^2 - 1}{t^2 + 1}, \quad y = y(t) = \frac{2t}{t^2 + 1}$$

which is a circle. $t$ is restricted to a certain range, usually 0 to 1, although here it's $[-\infty, \infty]$. It's easy to produce points on the curve by evaluating $x$ and $y$ at a sequence of $t$'s. Rotating the curve doesn't change the form of the equations, only the actual coefficients. However determining whether a point is on the curve is complicated. Given $(x, y)$, we must invert the equation for $y$ to find the value of $t$ that gives that $y$. There may be 0, 1, or more than 1, solution, even after we restrict to solutions in the legal interval for $t$. If 0, then this point cannot be on the curve. If 1, then we evaluate $x(t)$ to see if this is the given $x$. If not, then this point is not on this curve. If there is more than one $t$ that gives this $y$, then we find every corresponding $x$ to see if any match. It takes only one match for the point to be on the curve. In complicated cases, where the curve crosses itself, there may be two values of $t$ that produce the same point.

In spite of the difficulty of testing points against the curve, the ease of producing points and transforming the curve causes parametric curves to be the usual choice.

## 6.2 Conversions

Some of these forms can be converted into each other. An explicit equation is already also in the implicit and parametric form. A set of parametric polynomial equations may be converted to the implicit form by eliminating the parameter. However, the resulting implicit equation may have a much higher degree and may have extraneous pieces. In addition, we want only that part of the implicit equation that corresponds to the parameter's legal range. For example, to convert the parametric equation set $x = t^2 + t$, $y = t^3 + 1$ to an implicit equation in $x$ and $y$, we can use the method of resultants to consider each equation as a polynomial in $t$, the variable that we wish to eliminate.

$$\begin{aligned} t^2 + t - x &= 0 \\ t^3 + 0t^2 + 0t + (1 - y) &= 0 \end{aligned}$$

then form the coefficients into a $5 \times 5$ determinant, with each list of coefficients repeated several times. The first several rows each contain the coefficients of the first equation, each time shifted one column to the right. The unused entries are zero-filled. The remaining rows similarly contain the second equation's coefficients. The number of rows of each type is chosen to make the whole determinant square.

$$\begin{vmatrix} 1 & 1 & -x & 0 & 0 \\ 0 & 1 & 1 & -x & 0 \\ 0 & 0 & 1 & 1 & -x \\ 1 & 0 & 0 & 1-y & 0 \\ 0 & 1 & 0 & 0 & 1-y \end{vmatrix}$$

$$= x^3 - (3x + 1)(y - 1) - (y - 1)^2 = 0$$

The Groebner basis method is better than the resultant method for this purpose, in that it is faster and the resulting equation has no extraneous roots. However it is also too complicated to discuss here.

Turning an implicit equation into a parametric system of equations is possible sometimes. Turning an implicit or parametric equation into an explicit equation is usually impossible.

## 6.3 Polynomial, rational, or otherwise

If we wish to represent $(x, y)$ using parametric equations in $t$, what type of equations are best?

**Polynomial** e.g., $x(t) = 2t^2 - 5t^2 - 3t + 4$. This is the simplest and fastest. However parametric polynomials, or any degree, cannot represent a circle exactly. On the other hand, the approximation can be made as good as desired by using multiple curve segments as described below in section 8.

Even using only one cubic polynomial to represent one quarter of a circle has a maximum error of about only 1%.

**Rational** A rational expression is the *ratio* of two polynomials. As the example under parametric curves in section 6.1 above shows, a circle can be represented exactly as the ratio of two quadratic polynomials. However equally-spaced values of $t$ give very unequally-spaced points, i.e., $(dx/dt)^2 + (dy/dt)^2$ varies a lot.

Another advantage of rational curves is that they allow perspective transformations, while a perspective transformation of a polynomial curve is probably not exactly representable as a polynomial. Rational curves are used in CAD, but they are probably overkill in GIS.

**Other possibilities** such as transcendental functions exist. We can represent a circle as $(x(t) = \cos(t), y(t) = \sin(t))$ and points move at a constant speed as $t$ varies. However almost every operation on such equations is much more complicated and slower, so this option is rarely used.

Therefore, we recommend using polynomial equations.

## 6.4 Degree of equation

The next question is, how to approximate data that are too complicated for a simple curve, of say the third degree. The obvious answer is to raise the degree of the curve to give it more degrees of freedom. Considering an explicit curve, for simplicity, with a Lagrangian interpolation, we can exactly interpolate 4 points with a cubic, and, in general, $N + 1$ points with an $N$-degree curve. Indeed, given $(x_i, y_i)$, $i = 0..N$, we can calculate

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Then

$$y(x) = \sum_{i=0}^{N} y_i L_i(x)$$

This is an exact fit; the curve goes through the points. The problem is what the curve does between the points: it often oscillates wildly, with the size of the swing rising exponentially with the number of points. This brings us back to the minimization of variation criterion.

Another problem with high degree fits is that there are numerical roundoff problems in calculating the coefficients. This process involves evaluating an ill-conditioned determinant, whose value is inverse exponential in $N$.

Finally, the classification of higher degree curves is more complicated, most easily seen with implicit curves in 2-D. First-degree curves are straight lines.

Second-degree curves are conics. Already complications arise since some equations, such as $x^2 - y^2 = 1$, have two components, while others, such as $x^2 + y^2 = -1$, have no solution. Third-degree curves continue the progression in difficulty. They may cross themselves, and have isolated points. Whether or not an implicit cubic, such as $x^3 + xy + 2 = 0$, has a parametric form depends on the coefficients. Even determining the number of separate components is difficult.

For these reasons, the consensus tends to be not to use one high-degree curve, but rather to use several connected pieces of low-degree curves. The joins are smooth enough that the user cannot see them, which generally implies that the tangents and radii of curvature are continuous across the join. This requires that curves of at least third degree are used. Higher degrees are generally unnecessary. Therefore we recommend using segments of curves each of which are cubic parametric polynomials, or a *cubic spline*.

# 7 The Bézier curve

We have decided by now to use a sequence of cubic parametric polynomials to represent a curve. One such piece is often called a Bernstein-Bézier curve, or, simply, a Bézier curve, after Pierre Bézier who invented them to design auto bodies for Renault. The cubic Bézier curve in two dimensions consists of a parameter, $t$, ranging from 0 to 1, and two equations:

$$x(t) = \sum_{i=0}^{3} a_i t^i \qquad\qquad y(t) = \sum_{i=0}^{3} b_i t^i$$

Note that there are eight degrees of freedom, i.e., eight coefficients to determine: $a_i, b_i$, for $i = 0..3$. Since we are geographers, not human calculators, we need a better interface than merely explicitly supplying the coefficients.

One method is to specify the two endpoints of the curve, and its derivative $(dx/dt, dy/dt)$ at the ends. This supplies eight numbers to match the degrees of freedom. However another method is more convenient here, that is, defining a *control polygon*.

A control polygon is a quadrilateral, with vertices $P_0P_1P_2P_3$ that controls the shape of the curve. The curve starts at $P_0$ and ends at $P_3$. Its derivative at $P_0$ is 1/3 $P_0P_1$, i.e. pointing from $P_0$ to $P_1$ with magnitude one third of that line. Likewise its derivative at $P_3$ is 1/3 $P_2P_3$. The tangent in each case is the derivative, normalized to be of unit length. The important thing is the direction of each tangent, towards the adjacent control point. The 1/3 just makes the math easier. Note the four control points have eight degrees of freedom, which matches the desired number.

A general point on the curve, $P(t)$, is a linear combination, or weighted sum,

of the four control points:

$$P(t) = \sum_{i=0}^{3} w_i(t) P_i$$

The weights, $w_i(t)$ depend on the value of $t$ thus:

$$w_i(t) = \frac{3!}{i!(3-i)!} t^i (1-t)^{3-i}$$

$i!$ means $i$-factorial: 0! = 1! = 1, 2! = 2, 3! = 6.

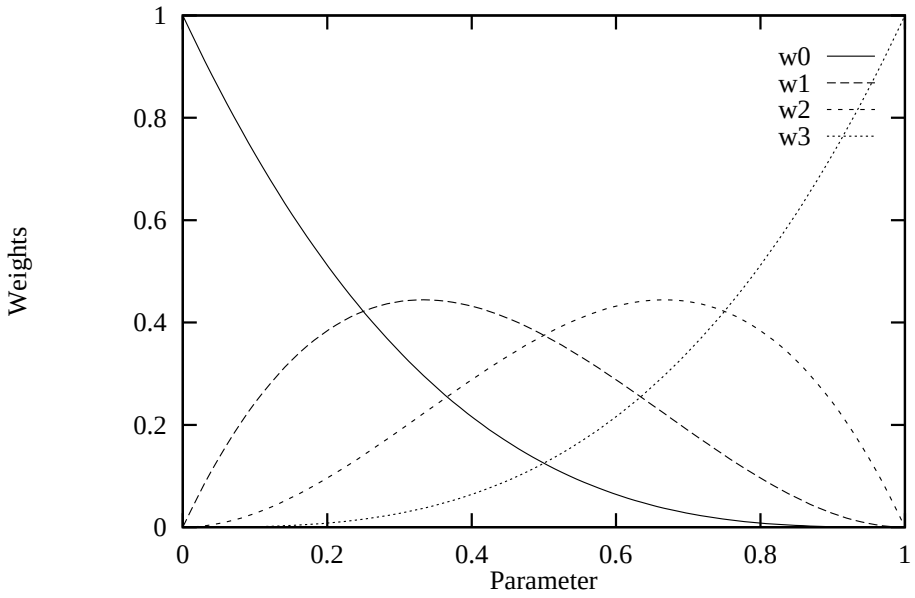


Figure 1: The Four Weight Functions for a Cubic Bezier Curve

Here are some sample points.

$$\begin{aligned} P(0) &= P_0 \\ P(0.1) &= 0.729P_0 + 0.243P_1 + 0.027P_2 + 0.001P_3 \\ P\left(\frac{1}{2}\right) &= \frac{1}{8}P_0 + \frac{3}{8}P_1 + \frac{3}{8}P_2 + \frac{1}{8}P_3 \end{aligned}$$

Since the four weights sum to one, the curve is always inside the convex hull of the four points. Therefore, if the control polygon is relatively flat, then so will be the curve. Figure 2 shows some control polygons and their corresponding Bézier curves.

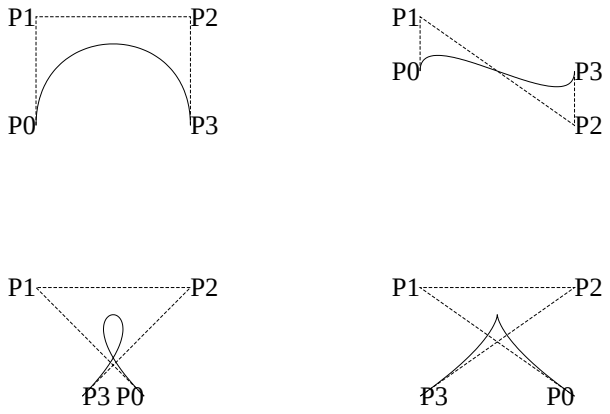


Figure 2: Some Bézier Control Polygons and Corresponding Curves

# 8 The B-spline

We wish to represent a complex curve as a *spline* composed of cubic polynomial parametric segments, or a *B-spline*, since using a single curve of higher degree has the disadvantages described above in section 6.4.

We'll define the curve with a sequence of $n+1$ *control points*, $P_0, P_1, \ldots, P_n$, with $n \geq 3$. The spline will have $n-2$ segments, $Q_3, Q_4, \ldots, Q_n$. Each segment will be affected by only four control points, and each control point will affect only four segments. Segment $Q_i$ will be affected by points $P_{i-3}, \ldots, P_i$, and control point $P_i$ will affect segments $Q_i, \ldots, Q_{i+3}$. This concept is called *local control*.

The parameter $t$ varies from 3 to $n+1$ in this simplest, or *uniform*, spline. In segment $Q_i$, $t$ is in the range $[i, i+1]$. A*knot* is where two curve segments join, and the *knot value* is the parameter value at the knot. Here the knots are at $3, 4, \ldots, n+1$ since the endpoints of the whole spline are also knots.

The spline has $C^2$ continuity since $(dx/dt, dy/dt)$ and $(d^2x/dt^2, d^2y/dt^2)$ are continuous at each knot. This is stricter than we actually need, since, for example the tangent would still be continuous if $(dx/dt, dy/dt)$ kept the same direction but changed its length across the knot. This could be caused by the parameterization of the curve, independently of the curve itself. Splines which have only this geometric continuity, or $G^k$, are called $\beta$-splines. They are more general since they have extra degrees of freedom, but are too complicated for our purpose.

Every point on a B-spline is the linear combination of four control points. If we define four weighting functions:

$$\begin{aligned} w_0(t) &= (1-t)^3/6 \\ w_1(t) &= (3t^3 - 6t^2 + 4)/6 \\ w_2(t) &= (-3t^3 + 3t^2 + 3t + 1)/6 \\ w_3(t) &= t^3/6 \end{aligned}$$

then

$$P(t) = \sum_{i=0}^{3} w_i(t - \lfloor t \rfloor) P_{\lfloor i+t-3 \rfloor}, \qquad 3 \le t \le n+1 \tag{1}$$

The B-spline does not go through any control points, even the end ones, unless some control points are superimposed or duplicated. A double control point reduces the continuity and the corresponding knot from $C^2$ to $C^1$. A triple control point reduces the continuity to $C^0$, i.e., the curve has a corner here, and it goes through the control point. The adjacent curve segments are straight lines in this case. There is a generalization of the B-spline, called a *non-uniform B-spline*, where the parameter does not vary equally from control point to control point. Here instead of having different knot values giving coincident control points, we duplicate the same knot value at some control points. This handles multiple control points better, but the weight functions are more complicated to calculate. Figure 3 shows an example of a B-spline on ten control points, together with the control polygon ($P_0P_1\ldots$) and the knots ($K_3K_4\ldots$).

# 9 The Catmull-Rom-Overhauser Spline

B-splines do not go through their control points, which is alright for a freeform designer, but less desirable when we wish to fit a spline to some data. The Catmull-Rom-Overhauser splines go through, or interpolate, their control points. In addition, the tangent to the spline at control point $P_i$ is in the direction $P_{i-1}P_{i+1}$. Thus each segment is defined by two points and two derivatives, which give the correct number of degrees of freedom. This spline is defined as follows.

$$\begin{aligned} w_0(t) &= (-t^3 + 2t^2 - t)/2 \\ w_1(t) &= (3t^3 - 5t^2 + 2)/2 \\ w_2(t) &= (-3t^3 + 4t^2 + t)/2 \\ w_3(t) &= (t^3 - t^2)/2 \end{aligned}$$

and then equation (1) is the same. Figure 4 shows an example on the same control polygon as in Figure 3. The knots are not shown since they are the same as the control points.

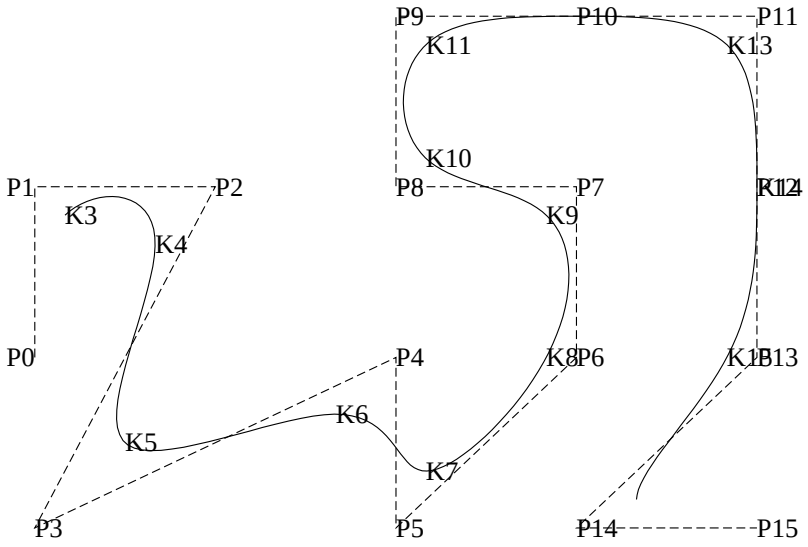


Figure 3: B-spline on Ten Control Points

# 10 Knot Selection

There is no simple answer to the question of how to pick the spline's knots, i.e., how to cut the complex curve into the cubic parametric pieces. An optimal solution, giving the best fit with the fewest degrees of freedom, is infeasible. If a designer is creating a curve with a CAD package, then he can always split a cubic into two new cubics with their control points. The first new cubic exactly matches the first half of the old cubic, and the second new cubic matches the second half of the old one. Then the designer can move their control points separately and make the spline more complicated.

However since we are more likely to be fitting a spline to existing data, the Douglas-Poiker line generalization algorithm, extended to curves, might be more appropriate. If the existing data is a chain of points, then the process might go as follows.

1. Initially represent the curve as a Catmull-Rom-Overhauser spline with the four control points as the two endpoints, and two points about 1/3 and 2/3 of the way along the chain.

2. Find the distance of every point from the chain.

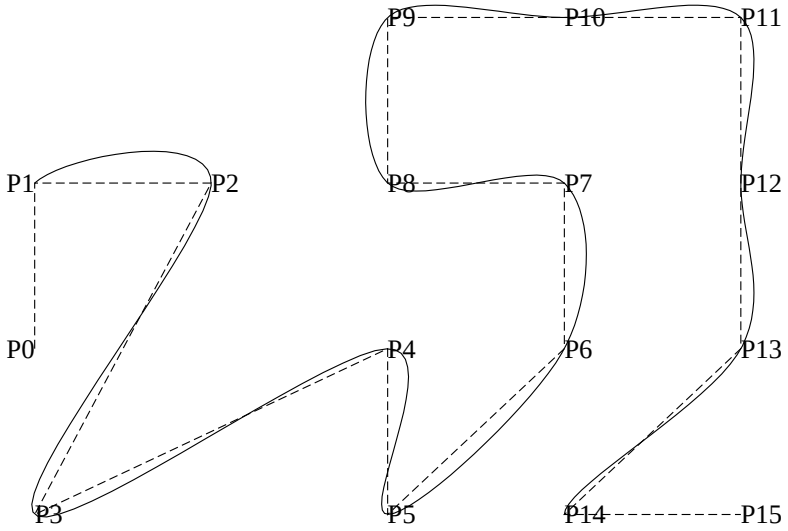


Figure 4: Catmull-Rom-Overhauser Spline On The Same Points

3. If the farthest point is farther than the tolerance from the chain, then make this point a knot and split the cubic curve there into two cubics.
4. Repeat the tolerance test on both spline parts, subdividing until the input chain is adequately represented as a spline.

The above is only a suggestion, and leaves room for research into at least the following questions.

- Is there a faster method of accuracy measurement than the distance of a point to the spline?
- Should we use a Bézier spline instead?
- Even with a Catmull-Rom spline, might we pick control points that are not in the input chain of points?

# 11 Producing points from the curve

To plot a curve, it is usually necessary to extract a sequence of points along it. For each value of the parameter $t$, we can substitute into the parametric

equations $x(t)$ and $y(t)$ to find $(x, y)$. The problems with this are, first, what values of $t$ to use, and, second, that there might be faster methods to find a large number of points.

## 11.1 Difference tables

If equally spaced values of $t$ are desired, the fastest way is to establish a *difference table*. E.g., to find $x(t)$ at $t = 0, 0.01, 0.02, \cdots$, we can do the following. Find the first four $x(t)$. Find the differences between their values, $\mathcal{D}_i = x_{i+1} - x_i$. Then find the second order differences, $\mathcal{D}_i^2 = \mathcal{D}_{i+1} - \mathcal{D}_i$, and, finally, the third order differences, $\mathcal{D}_i^3 = \mathcal{D}_{i+1}^2 - \mathcal{D}_i^2$. The third-order differences are constant for a cubic, so we can then sum backwards three times to get each new $x_i$. For instance, with $x(t) = 1000000t^3 - 50000t^2 + 300t + 10$, we get this table:

| $i$ | $t$ | $x_i$ | $\mathcal{D}_i$ | $\mathcal{D}_i^2$ | $\mathcal{D}_i^3$ |
|---|---|---|---|---|---|
| 0 | 0 | 10 | −1 | −4 | 6 |
| 1 | 0.01 | 9 | −5 | 2 | *6* |
| 2 | 0.02 | 4 | −3 | *8* | *6* |
| 3 | 0.03 | 1 | *5* | *14* | *6* |
| *4* | *0.04* | *6* | *19* | *20* | |
| *5* | *0.05* | *25* | *39* | | |
| *6* | *0.06* | *64* | | | |

The numbers in Roman type are calculated left-to-right from the given data, then the numbers *in italics* are calculated right-to-left from them. With this method, finding a new $x_i$ requires only three additions and no multiplications, so each new point requires only six additions and no multiplications. Here $\mathcal{D}t = 0.01$ to show that it need not be one.

What increment should we use for $t$ since the points may be very unequally spaced? (In the worst case, the curve might even backtrack on itself as $t$ increases, as does $x = t^3 - t$, $y = t^3 - t$. We'll ignore such badly parameterized curves.) We might always differentiate to get $(dx/dt, dy/dt)$ and then maximize that over the interval. However, it's probably adequate simply to use a small value for the increment.

One concern with difference tables is that any errors will also grow cubically. If $x_0$ were calculated as 10.1 instead of 10, then the third difference would be calculated as 5.9 instead of 6, which would be equivalent to changing the leading term of the polynomial by one part in 60.

## 11.2 Recursive subdivision

Alternatively, we might subdivide the curve recursively until each piece is small enough that it appears straight. E.g., we start with the curve for $t = 0, .., 1$, and its endpoints, $P_0, P_1$. Evaluate $P_{.5} = (x(.5), y(.5))$ and find its distance from the line $P_0P_1$. If $P_{.5}$ is too far away, then split the curve there and repeat on the two halves. Note that $P_{.5}$ may be close also if the curve bends to one side, crosses

over the straight line at the point, and then bends to the other side. Therefore, we should always do some subdivisions. In practice, subdividing for five levels to produce 31 points on the curve seems quite sufficient.

If we have the four control points drawn on a sheet of paper, then it is possible to find the midpoint of the curve by finding the midpoints of six line segments. By repeating this, we can sketch the curve by hand. More generally, to find $\boldsymbol{P}(t)$ given the control points $\boldsymbol{P}_0\boldsymbol{P}_1\boldsymbol{P}_2\boldsymbol{P}_3$, first we divide the control polygon's sides in the ratio $1-t:t$, so that $\boldsymbol{Q}_i = (1-t)\boldsymbol{P}_i + t\boldsymbol{P}_{i+1}$, for $i = 0, 1, 2$. Then we repeat, $\boldsymbol{R}_i = (1-t)\boldsymbol{Q}_i + t\boldsymbol{Q}_{i+1}$ for $i = 0, 1$, and, finally, $\boldsymbol{P}(t) = (1-t)\boldsymbol{R}_0 + t\boldsymbol{R}_1$.

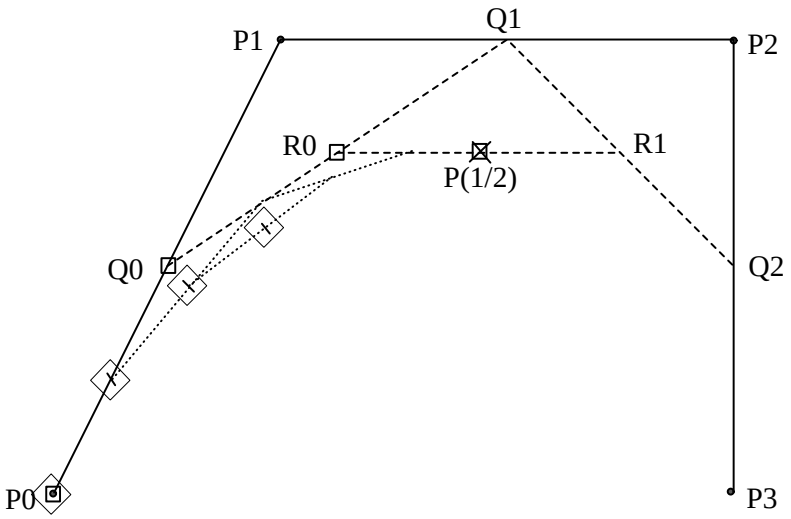


Figure 5: Recursive Construction of a Bezier Curve

Figure 5 shows finding the midpoint X at ($t = 1/2$). The original control points are labeled and shown by circles; the construction lines are dashed. A benefit of this process is that that it splits the Bézier curve into two parts at the midpoint, *and* produces the new control points of each half. In this case, the four control points of the first half are $\boldsymbol{P}_0\boldsymbol{Q}_0\boldsymbol{R}_0\boldsymbol{X}$ (marked by small boxes) and for the right half: $\boldsymbol{X}\boldsymbol{R}_1\boldsymbol{Q}_2\boldsymbol{P}_3$. The two smaller Bézier curves match the original curve exactly. However, these two new control polygons are flatter, so that continuing the process to about five levels produces a piece-wise linear representation of the curve. In Figure 5 we've also split the first half curve in half, using dotted construction lines, and marked the control points of the quarter-curve with diamonds.

## 12 Summary

We have seen various issues in the representation of cartographic lines by curves. If we decide that curves are advantageous, we must select a criterion for the goodness of fit, and know our data characteristics and the desired operations. We will probably choose a spline composed of parametric cubic polynomial curves to approximate the line. Each curve in the spline might be a Bézier polynomial, and they might be connected at knots into a perhaps a B-spline or Catmull-Rom-Overhauser spline. In either case, knot selection might be done with the Douglas-Poiker algorithm. Finally, we can easily return from the spline to a sequence of points by various methods.

## 13 Acknowledgements

Parts of this work were supported by NSF grant CCR-9102553, by the Directorate for Computer and Information Science and Engineering, NSF Grant CDA-8805910, and by the Gruppo Nazionale Informatica Matematica of the Italian National Research Council. This idea was suggested by David Douglas.

## References

[Cho91] Byong K. Choi. *Surface modeling for CAD/CAM*, volume 11 of *Advances in industrial engineering*. Elsevier, 1991.

[FKN89] Wm Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. Geometric computing and the uniform grid data technique. *Computer Aided Design*, 21(7):410–420, 1989.

[FNK+89] Wm Randolph Franklin, Chandrasekhar Narayanaswami, Mohan Kankanhalli, David Sun, Meng-Chu Zhou, and Peter YF Wu. Uniform grids: A technique for intersection detection on serial and parallel machines. In *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography*, pages 100–109, Baltimore, Maryland, 2-7 April 1989.

[FP81] I.D. Faux and M.J. Pratt. *Computational geometry for design and manufacture*. Mathematics and its applications. Halsted Press, 1981.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Systems Programming Series. Addison-Wesley, 2nd edition, 1990.

**Do not print this page (the page numbers would be wrong when the article was printed; this for your information only. Of course, if you give me the final page numbers, I can make the table of contents correct.**

# Contents



This file was formatted on January 14, 1993.