Curves I: Bezier Curves

The origins of the work leading to what we now refer to as Bezier curves is somewhat unclear because early work was done individuals employed in automobile design, and so advances were not made public immediately to maintain competitive advantage. Prior to the development of mathematical techniques such as these, automobile manufacturer's found it difficult to reproduce at adequate precision in fabrication the curved contours rendered by their designers in drawings using curved templates and similar drawing aids. Once the contours were able to be described mathematically, scaling the shapes up to actual size for fabrication became straightforward.

The basic geometric idea behind Bezier curves was apparently developed by Paul de Casteljau beginning in 1959 while working for Citroen. In the mid-1960's, P. Bezier of the Renault company approached the same problem from a somewhat different direction, developing the methods of curve generation which now bear his name. Because it turns out that de Casteljau's method gives a geometric insight into the principles behind Bezier curves, we start here with a short description of his method. However, in practice, one would not use the de Casteljau method directly, but rather, the formulas it leads to, giving Bezier curves.

De Casteljau's Algorithm

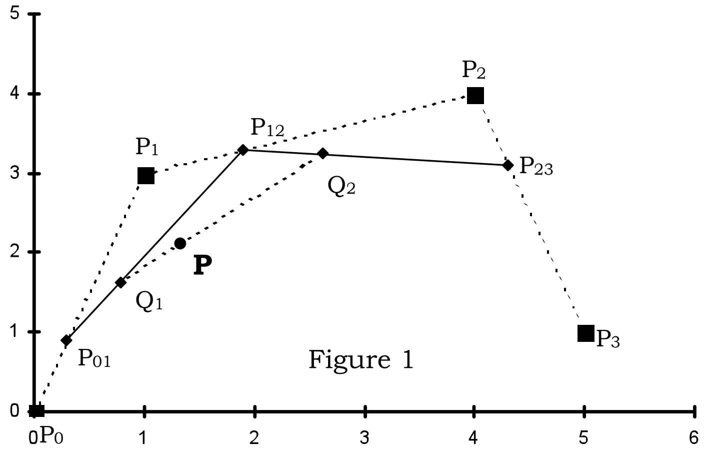

Figure 1

De Casteljau devised the conceptual approach to curve generation illustrated with a four point example in Figure 1 to the right. The desired curve here is to be determined by the four **control points** P_0 at (0, 0), P_1 at (1, 3), P_2 at (4, 4) and P_3 at (5, 1). De Casteljau started by adopting a parametric formulation for the x- and y-coordinates of points on the curve to be generated, where t = 0 would give the initial point on the curve, and t = 1 would give the final point on the curve. Then, the actual points on the curve were generated by repeating the following procedure for a succession of values of t between 0 and 1 (Figure 1 is based on a value of t of 0.3, for example).

Mark off points on each of the line segments joining the original control points a fraction t of the distance from the first to the second points of each pair. Thus, in Figure 1, since t = 0.3, point P_{01} is 30% of the way from point P_0 to point P_1, point P_{12} is 30% of the way from P_1 to point P_2, and point P_{23} is 30% of the way from point P_2 to point P_3. Draw a polyline through these new points. This polyline will have one less line segment than the polyline through the original set of control points. Now, repeat the process on this polyline. Thus, in Figure 1, point Q_1 is 30% of the way from point P_{01} to point P_{12} and point Q_2 is 30% of the way from point P_{12} to point P_{23}. Now the polyline joining these (only two) new points is just a single line segment, and the point **P** is calculated at the point 30% of the way from point Q_1 to point Q_2. This is the point on the final curve corresponding to t = 0.3.

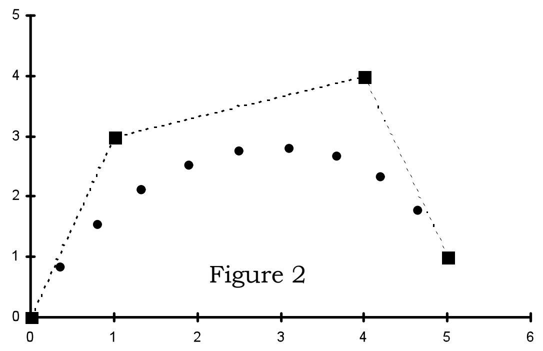
Figure 2

The process is easily generalized to situations with any number of initial control points. For any value of t, each cycle of the algorithm replaces a polyline with a new polyline having one less line segment, and the algorithm cycles until a polyline with just one line segment is obtained, on which a new point for the curve is calculated. The coordinates of all these new points in the algorithm are easily computed using the parametric form of the equation of the straight line.

Repeating this process for the four control points in Figure 1, for t = 0.1, 0.2, 0.3, 0.4, ... 0.8, 0.9 gives the points shown in Figure 2. It is easy to see that these points are forming a gently arching curve from P_1 to P_4. In practice, one would just generate enough such points that joining the points by straight line segments will produce the appearance of a curve.

You can see several characteristics of the curve that is generated in this manner. First, the curve will be contained entirely within the so-called **convex hull** of the original points -- that region you would get by stretching an elastic band around the original points. The curve cannot twist outside of this region, because the intermediate line segments generated always fall within this region. Thus, quite severe control of the curve has been established. The cost is that the curve no longer actually goes through all of the original points. As you could see with some experimentation, the location of the original points determines the detailed shape of the curve, but not as points through which the curve passes in general. This is why we adopt the term **control points** to refer to these original points -- they control the shape of the curve in a more general sense. If you follow the logic of the method through mentally for t = 0 and t = 1, you'll see that these values of t generate the first and last of the control points exactly, so this curve will pass through the first and last of the set of control points for sure.

The description above of De Casteljau's algorithm was intentionally made very visual and geometric to demonstrate the strategy being adopted. It is relatively easy to reduce this recursive algorithm to a set of very straightfoward computational formulas, so in practice, one would never go through the lengthy process of generating successive sets of polylines as described.

Bezier Curves

The curve resulting from De Casteljau's algorithm is more commonly known as a **Bezier curve**, named after the French researcher who is credited with developing the mathematical formulas needed to implement the algorithm in a computationally efficient manner in the early 1960s.

The process is not difficult, but it will involve extensive use of subscripts and so to avoid unnecessary and frustrating confusion, we will have to be quite careful with our use of notation. The control points will be numbered with indices starting from 0. The symbols p_k will represent either the x-coordinate or the y-coordinate of the k^{th} control point (or indeed, the z-coordinate, since this process works perfectly fine in three dimensions as well). We'll use bracketted superscripts to indicate the cycle of the recursive procedure being discussed, when necessary to make the distinction.

So, the whole process starts with the L + 1 control points, $p_0, p_1, p_2, \ldots, p_L$. (Notice that if L is the index of the final control point -- L = 3 in Figure 1 -- then since we've started counting from zero, there must be L + 1 control points -- 3 + 1 = 4 control points in Figure 1.) In the first cycle, we generate a new set of L points, using the formulas

$$p_k^{(1)} = (1-t)\cdot p_k + t\cdot p_{k+1}, \qquad k = 0,1,2,\ldots,L-1 \qquad \text{(CURV1-1)}$$

This result comes from the parametric equation of these line segments. Remember, the p's here stand for x's or for y's (or for z's). Notice that when t = 0, $p_k^{(1)} = p_k$, the first point on the line segment, and if t = 1, $p_k^{(1)} = p_{k+1}$, the last point on the line segment. Values of t between 0 and 1 give points intermediate on the line segment joining P_k to P_{k+1}.

Now, to do the second cycle of De Casteljau's algorithm, we just repeat (CURV1-1) but on the points $p_k^{(1)}$ to get points $p_k^{(2)}$:

$$\begin{aligned} p_k^{(2)} &= (1-t)\cdot p_k^{(1)} + t\cdot p_{k+1}^{(1)} \\ &= (1-t)\cdot\left[(1-t)\cdot p_k + t\cdot p_{k+1}\right] + t\cdot\left[(1-t)\cdot p_{k+1} + t\cdot p_{k+2}\right] \\ &= (1-t)^2\cdot p_k + 2t(1-t)\cdot p_{k+1} + t^2\cdot p_{k+2}, \qquad k = 0,1,2,\ldots,L-2, \end{aligned} \qquad \text{(CURV1-2a)}$$

The first line above is just (CURV1-1) but upping the superscript indices by 1 throughout. The second line is obtained by substituting from (CURV1-1) for the $p^{(1)}$'s in terms of the original p's. The third line is just a bit of algebraic simplification. Formulas (CURV1-2a) gives the coordinates of the points generated by the second cycle of De Casteljau's algorithm in terms of the coordinates of the original control points. Notice that each term in this formula consists of the product of a coordinate of an original control point and an expression involving t. This expression involving t can be viewed as a sort of "weight factor" -- they always have a value between zero and one as well -- measuring how much influence that particular control point has on the position of the curve for the current value of t. These expressions involving t do not depend at all on the positions of the control points, and so will be the same for any shape of curve whatsoever. As a result, they've been given special symbols. For example, we would rewrite (CURV1-2a) as[1]

$$p_k^{(2)} = B_0^{(2)}\cdot p_k + B_1^{(2)}\cdot p_{k+1} + B_2^{(2)}\cdot p_{k+2} \qquad \text{(CURV1-2b)}$$

where

$$B_0^{(2)} = (1 - t)^2, \quad B_1^{(2)} = 2t(1-t), \quad \text{and,} \quad B_2^{(2)} = t^2 \qquad \text{(CURV1-2c)}$$

This same process can now be repeated by substituting (CURV1-2a) into (CURV1-1) to get formulas for the coordinates of points $p_0^{(3)}$ through $p_{L-3}^{(3)}$; then substitute those formulas into (CURV1-1) to get formulas for coordinates of points $p_0^{(4)}$ through $p_{L-4}^{(4)}$, and so on, until finally after a total of L-1 cycles, we get down to the final single segment polyline, and the point

$$p^{(L)} = \mathbf{p} = \sum_{k=0}^{L} B_k^{(L)}\cdot p_k \qquad \text{(CURV1-3)}$$

on the Bezier curve corresponding to the present value of t. Remember, here p_k stands for one of the coordinates of the kth control point, the $B_k^{(L)}$ are a set of expressions depending only on the value of t, and the result of this formula is to produce the corresponding coordinate (x, y, or z) of the point on the Bezier curve corresponding to the actual value of t in use. If we now can come up with a way of computing the values of the $B_k^{(L)}$, it will be very easy to compute coordinates for points along the Bezier curve from formula (CURV1-3).

If you do a few more cycles of the process illustrated by formulas (CURV1-1) and (CURV1-2a,b) above, you will eventually notice that the $B_k^{(L)}(t)$ functions are the terms you get when you expand the binomial power $[(1 - t) + t]^L$. This means specifically that

$$B_k^{(L)} = \frac{L!}{k!\,(L-k)!}(1-t)^{L-k}t^k, \qquad k = 0, 1, 2, \ldots L, \qquad \text{(CURV1-4)}$$

where L! = L(L - 1)(L - 2) ... (2)(1), the product of the first L whole numbers, the so-called "factorial" of L. For quite small values of L, the values of this rather ugly looking fraction involving factorials can be read from the corresponding row of *Pascal's Triangle*:

[1] The symbol "B" here could stand for "Bezier" in honor of the developer of this methodology, or "Binomial", because as noted below, they also arise from the expansion of a power of a binomial, or for "Berstein", because they are also called Bernstein polynomials. Some people even call them the Bernstein-Bezier polynomials, though this is rather discriminatory against the important contributions of George Binomial to mathematics in general.☺

$$\begin{array}{lccccccccccccc}
L = 0: & & & & & & & 1 & & & & & & \\
L = 1: & & & & & & 1 & & 1 & & & & & \\
L = 2: & & & & & 1 & & 2 & & 1 & & & & \\
L = 3: & & & & 1 & & 3 & & 3 & & 1 & & & \\
L = 4: & & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
L = 5: & & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
L = 6: & 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array} \qquad \text{(CURV1-5)}$$

Here, each row begins and ends with 1's, and the elements in one row are the sum of the two elements in the row just above to its left and right. To get the formulas for the points on a Bezier curve for a particular value of L, just substitute the numbers from left to right of row L of this table for the fractional coefficient the formulas (CURV1-4).

Example
Write down the working formulas, (CURV1-3), for a four-control-point Bezier curve, and then use those formulas to compute the (x, y) coordinates of point **P** in Figure 1 (corresponding to t = 0.3).

Solution
The example in Figure 1 has L = 3. Using the "L = 3" row of table (CURV1-5), we first get:

$$\begin{array}{ll}
(k = 0) & B_0^{(3)} = 1\bullet(1 - t)^3\, t^0 = (1 - t)^3 \\
(k = 1) & B_1^{(3)} = 3\bullet(1 - t)^2\, t^1 = 3t(1 - t)^2 \\
(k = 2) & B_2^{(3)} = 3\bullet(1 - t)^1\, t^2 = 3t^2\,(1 - t)^1 \\
(k = 3) & B_3^{(3)} = 1\bullet(1 - t)^0\, t^3 = t^3
\end{array} \qquad \text{(CURV1-6)}$$

Thus, formula (CURV1-3) becomes, for this case,

$$\mathbf{p} = (1 - t)^3\, \mathbf{p}_0 + 3t(1 - t)^2\, \mathbf{p}_1 + 3t^2\,(1 - t)\, \mathbf{p}_2 + t^3\, \mathbf{p}_3 \qquad \text{(CURV1-7)}$$

and this formula will generate the Bezier curve for any collection of four control points as the value of t is varied from 0 to 1. In the specific example illustrated in Figure 1, the actual coordinates of the four control points are (0, 0), (1, 3), (4, 4), and (5, 1) in order, and so for t = 0.3, we get the point with coordinates:

$$\begin{aligned}
x &= (1 - 0.3)^3 \bullet x_0 + 3(0.3)(1 - 0.3)^2 \bullet x_1 + 3(0.3^2\,)(1 - 0.3)\bullet x_2 + 0.3^3\, x_3 \\
&= (1 - 0.3)^3 \bullet 0 + 3(0.3)(1 - 0.3)^2 \bullet 1 + 3(0.3^2\,)(1 - 0.3)\bullet 4 + 0.3^3\, 5 \\
&= 1.332
\end{aligned}$$

and

$$\begin{aligned}
y &= (1 - 0.3)^3 \bullet y_0 + 3(0.3)(1 - 0.3)^2 \bullet y_1 + 3(0.3^2\,)(1 - 0.3)\bullet y_2 + 0.3^3\, y_3 \\
&= (1 - 0.3)^3 \bullet 0 + 3(0.3)(1 - 0.3)^2 \bullet 3 + 3(0.3^2\,)(1 - 0.3)\bullet 4 + 0.3^3\, 1 \\
&= 2.106
\end{aligned}$$

Thus, for t = 0.3, we get a point on the Bezier curve having (x, y) coordinates of (1.332, 2.106). Substituting other values of t into formula (CURV1-7) will generate sequences of points as illustrated in Figure 2.

◆◆◆

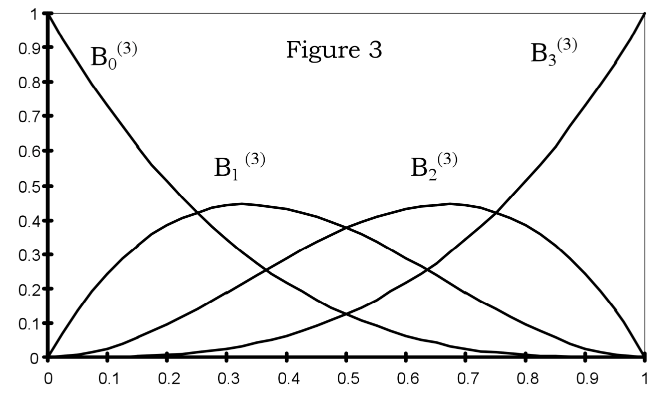

Figure 3

The **Bernstein Polynomials**, $B_k^{(L)}$, act like "weight factors" in formula (CURV1-3). A graph of these functions for the case L = 3 is shown to the right in Figure 3. Here values of the Bernstein polynomials are plotted along the vertical axis, against values of t along the horizontal axis. Notice that when t = 0, $B_0^{(3)} = 1$, but the remaining three B's are all zero. Thus, when substituted into formula (CURV1-7), only the coordinates of point #0 survive, and so the point on the Bezier curve for t = 0 is just exactly the first control point. As t increases from 0 to say, 0.2, the value of $B_0^{(3)}$ begins to decrease from 1, and the values of the other three B's increase -- especially, $B_1^{(3)}$ early on, so that formula (CURV1-7) will give coordinates that are still quite similar to the coordinates of point #0, but are gaining an increasing contribution from the coordinates of points 1, and to a lesser extent, from the coordinates of points 2 and 3. Around about t = 0.3, $B_1^{(3)}$ is the largest in value of the four B's, and so as t increases through the interval near 0.3, the points on the Bezier curve will be closest to the control point p_1. As t approaches 0.7, $B_2^{(3)}$ becomes the biggest of the B's in value, and so the Bezier curve will be tending to swing past control point p_2. Finally, for values of t greater than about 0.8, $B_3^{(3)}$ becomes dominant in terms of value, and the Bezier curve heads towards control point p_3, eventually ending at that point when t = 1. From this diagram, you see that the coordinates of points on the Bezier curve are "blends" of the coordinates of the control points, with the weighting of each control point varying according to the values of the B's as t varies from 0 to 1. This is why people often refer functions like the Bernstein polynomials here as **blending functions**.

While it may not be completely obvious from the graph in Figure 3, it is not difficult to demonstrate that for a specific value of t, the sum of all the blending functions is exactly 1. This means that the blending functions really do represent in detail the degree of influence each control point has on the position of the Bezier curve for any value of t.

Matrix Form of Bezier Curve Formulas

The description of Bezier curves given above was intended to illustrate intuitive ideas in the definition of these curves, but the formulas given so far are not all that useful from a computational or algorithmic point of view. It is possible to reorganize the formulas (CURV1-3) and (CURV1-4) into a matrix form, which then can be used in conjunction with the matrix methods described in other documents to carry out geometric and viewing transformations. We will illustrate the approach using the L = 3 case first, and then briefly describe a "recipe" for obtaining the necessary matrices for the general value of L. The formulas given here assume that the (x, y, z)-coordinates of points are to be written as rows of a matrix (rather than columns).

First, notice that formula (CURV1-7) can be written formally as a product of a row matrix (of the Bernstein polynomials) and the matrix of the coordinates of the control points:

$$\mathbf{p} = \mathbf{B} \bullet \mathbf{P} = \begin{bmatrix} B_0^{(L)} & B_1^{(L)} & \cdots & B_L^{(L)} \end{bmatrix} \bullet \begin{bmatrix} P_0 \\ P_1 \\ \vdots \\ P_L \end{bmatrix} = \begin{bmatrix} B_0^{(L)} & B_1^{(L)} & \cdots & B_L^{(L)} \end{bmatrix} \bullet \begin{bmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_L & y_L & z_L & 1 \end{bmatrix} \tag{CURV1-8}$$

The rightmost 4 x 4 matrix here is the matrix of coordinates of the L + 1 control points and the row matrix multiplying it from the left is a 1 x 4 matrix whose elements are formulas for the blending functions. The result of this product is a 1 x 4 matrix giving the coordinates of a point on the Bezier curve.

The important observation here is that the 1 x 4 matrix of blending functions does not depend upon the location of the control points at all -- all that information is in the right-hand matrix. Since this matrix of coordinates of control points is "exposed" on the right, carrying out a geometric transformation of a Bezier curve (translation, scaling, rotation, reflection, etc.), we simply need to transform the control points accordingly (that is, multiply (CURV1-8) from the right by the appropriate sequence of transformation matrices). This means that all of the previous work we did with images consisting of points joined by straight line segments now applies equally well to images consisting of points, straight line segments and Bezier curves (and indeed, most of the "curve-types" discussed in this document). You would still organize your program around a master array of coordinates of points. Some of these points would be endpoints of straight line segments, some could be control points for Bezier curve segments. Any geometric or other matrix transformation required would still be applied to the matrix of coordinates. However, now, instead of then just having a line information array indicating which pairs of points need be joined by straight line segments to produce the image, you'd need to have a somewhat more general informational structure, which would contain not only information on which pairs of points need to be joined to form straight line segments in the image, but which sequences of points should be used as control points to generate Bezier curves. Then, in addition to the usual *Lineto()* function that generates the image of a straight line segment from the coordinates of the two endpoints, we would need to develop a function perhaps called *Bezier()* which would implement formula (CURV1-8) given the coordinates of the L+1 control points. This would be as generic a function as the *lineto()* function. (This is exactly the approach taken by MS-Windows in generating TrueType font character shapes, which are combinations of straight line segments and curved segments.)

This generic function *Bezier()* would have to have a way of generating the values of the elements of the 1 x (L+1) matrix $[B_0^{(L)}, B_1^{(L)}, \ldots, B_L^{(L)}]$. If only a few very small, values of L were to be allowed, it would probably be easiest to simply hard-code formulas of the sort (CURV1-6) for the allowed values of L. If the routine is to allow any value of L in principle, one could code the more generic formula (CURV1-4), though some care would have to be taken to avoid excessive amounts of redundant computations in evaluating powers of t and (1 - t).

We'll describe one more way of generating this matrix $[B_0^{(L)}, B_1^{(L)}, \ldots, B_L^{(L)}]$ in the general case, by way of an easily extendable demonstration. The first thing to notice is that the blending functions, $B_j^{(L)}$ can be written as Lth degree polynomials in t. For example, multiplying out the expressions in (CURV1-6) gives:

$$\begin{aligned} B_0^{(3)} &= (1-t)^3 &&= 1 - 3t + 3t^2 - 1t^3 \\ B_1^{(3)} &= 3t(1-t)^2 = 3(t - 2t^2 + t^3) &&= 0 + 3t - 6t^2 + 3t^3 \\ B_2^{(3)} &= 3t^2(1-t)^1 = 3(t^2 - t^3) &&= 0 + 0t + 3t^2 + 3t^3 \\ B_3^{(3)} &= t^3 &&= 0 + 0t + 0t^2 + 1t^3 \end{aligned} \qquad \text{(CURV1-9)}$$

Based on this, form the following (L+1) x (L+1) matrix:

(i) write the rows of Pascal's triangle as columns, with the first row becoming the last column, the second row of the triangle becoming the second-last column of the matrix, and so forth, and write the entries starting at the bottom of the column. Fill in empty elements with zeros. In the case of L = 3, we have L + 1 = 4, and so get the following 4 x 4 matrix::

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

(ii) now, use the elements of the first column as multipliers onto the columns one at a time, from left to right. Thus, for the L = 3 case, we multiply the first column by 1, the second by 3, the third by 3 and the fourth by 1 to get

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 \\ 3 & 6 & 3 & 0 \\ 1 & 3 & 3 & 1 \end{bmatrix}$$

(iii)finally, starting with the first element of the first column, alternate + and - signs down the column, and then across each row. For the present example, this give:

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \qquad \text{(CURV1-10a)}$$

If we now define the matrix

$$\mathbf{T} = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \qquad \text{(CURV1-10b)}$$

a row matrix of powers of t, written from lowest to highest power, right-to-left, you can easily verify that

$$[B_0^{(L)}, B_1^{(L)}, \ldots, B_L^{(L)}] = \mathbf{T} \bullet \mathbf{B} \qquad \text{(CURV1-11)}$$

Those of you uncomfortable with this recipe description can verify that the elements of this matrix **B** are given by the formula

$$B_{k,j} = (-1)^{k-j} \frac{k!}{j! \cdot (k-j)!} \cdot \frac{L!}{k! \cdot (L-k)!} = (-1)^{k-j} {}_kC_j \; {}_LC_j \qquad \text{(CURV1-12)}$$

where ${}_NC_R$ are the so-called binomial coefficients arising in many computational methods.

What makes (CURV1-11) useful is that the matrix **B** is easily coded for general values of L, and the bulk of the arithmetic involved in evaluating points on a Bezier curve has become isolated in the evaluation of the elements of the matrix **T** for various values of t. In a program requiring repeated generation of many Bezier curves, one could initially generate and store a multi-row version of **T**, each row corresponding to one of a succession of values of t between 0 and 1 inclusive, and with enough columns to accommodate the highest order Bezier curve to be constructed. Then, every time a Bezier curve segment was to be generated, this pre-calculated **T** could be multiplied onto a generated **B** as in (CURV1-11), the result of which is then fed into (CURV1-8) to produce the points on the curve without having to recalculate powers of t again at all.

This completes our description of the computational aspects of working with Bezier curves. They have some very valuable properties, the most important of which is that the resulting curve is always confined within the convex hull of the control points. Further, they have a so-called variation-diminishing property -- the Bezier curve has less twist to it than the polyline formed by the control points. Bezier curves always pass through the first and last control points, and their directions at the two end control points are the same as the line segments joining the first two, and the last two control points. Away from the first and last control points, the curve runs smoothly towards each of the control points in general. If the polyline determine by the control points intersects itself, so will the Bezier curve. The formulas and computations required to generate the Bezier curve are not very complicated, and can be organized to be quite efficient arithmetically.

The Bezier curve approach has at least two disadvantages of some significance however. The most obvious is that the Bezier curve will still involve high order polynomials if one needs to work with a large number of control points, and this can result in computational inefficiency and undesirable features in the shape of the curve generated, even though the curve is confined within the convex hull of the control points. Also, the formula (CURV1-8) means that every single point on the Bezier curve (except for the two endpoints) are influenced by the location of every single control point. As a result, shifting a single control point (in an attempt to refine the shape of a Bezier curve) can cause the entire curve to change shape. Granted that the influence of a control point is probably only noticeable over part of the entire Bezier curve, this still can make the fine-tuning of a design frustrating -- a little like trying to flatten out the rolls in an improperly installed carpet: you flatten a hump in one place only to have it show up someplace else.

These two defects may be most bothersome when a large number of control points are involved, but they can also cause problems when on a few control points are present and we would like to create a curve with a pronounced twist to it. And example is seen in Figure 4 to the right, in which at attempt is being made to get a sort of double humped shape with just eight control points. The bends in the polyline based on the eight control points tend to get so smoothed out by the Bezier formula that the two humps are not very pronounced at all. In fact, even shifting some of the higher control points upwards even a considerable distance doesn't change the shape of the curve very much, though the whole thing will shift upwards to some extent. One way to get the two humps to show up more distinctly would be to add quite a few additional control points outlining the desired shape more densely. This, of course, would lead to a much higher order Bezier formula, with the problems that that causes. Instead, to deal with such cases, a variety of methods have been developed for piecing the entire curve together from pieces generated by low-order polynomial functions, each under the influence of just a small subset of the control points.

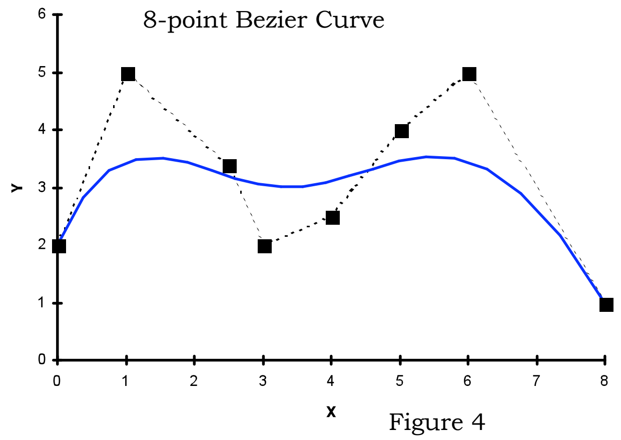

Figure 4

Figure 4 comes from a chart in an Excel workbook called BEZIER8.XLS, included with this document. You can change the coordinates of the control points in that workbook to see how the shape of the curve will change as a result. Although the spreadsheet is set up specifically to handle only an 8-point problem, you should be able to modify it to handle other numbers of control points without too much trouble.