Spline Tutorial Notes

Alvy Ray Smith
Computer Graphics Project
Lucasfilm Ltd

Technical Memo No. 77
8 May 1983

Introduction

There are many different types of splines, but there are only a few which need be mastered for computer animation. In preparation for these notes, I looked through all the computer graphics texts on my shelf (six of the well-known ones) and found that none of them present splines as simply as I and my colleagues know them. They burden the reader with splines which I believe to be of only historical interest (natural spline) while failing to present one of the most useful splines for computer animation (cardinal spline). In all but one text, the convenient 4×4 matrix formulation of cubic splines is not mentioned. So the purpose of these notes is to present two very powerful classes of cubic splines—the cardinal and the beta splines—for computer animation and simple 4×4 matrix realizations of them. The Catmull-Rom spline and the B-spline, representing the two classes respectively, will be paid particular attention. The addition of simple parameters to the matrices for these two special cases form the defining matrices for the two general classes.

What Are Splines?

A *spline* is a piecewise polynomial satisfying continuity conditions between the pieces. We will study piecewise cubic polynomials which are continuous in the first derivative between pieces or in both the first and second derivative. These are called *cubic splines*, and we will henceforth assume cubic splines in these notes. One of the two classes of splines we present will be continuous in the first derivative only; the other will be continuous in both the first and second derivative. Since polynomials are continuous in all derivatives, it must be the joints between the pieces in a spline where continuity is a question. These points are called *knots*, or *ducks*.

The original spline was not mathematical but a real draftsman's tool consisting of a flexible strip of metal or wood and several heavy pieces which anchored the flexible strip to the drafting table. The strip curved depending on where these heavy "ducks" were placed, providing the draftsman with a guide for drawing a general class of curves, more general than provided by a set of French curves.

The first mathematical splines were models of this real spline, but the math has now evolved beyond the constraints of reality to the point where modern splines bear little resemblance to the mechanical predecessor. They share with it the notion of graceful curves generated from a small set of discrete points.

The "natural" cubic spline is a model of the mechanical spline. Many texts belabor their readers with the mathematics of the natural spline. It is historically interesting, but it suffers from a fatal flaw so far as computer graphics is concerned: It is global. Consider a mechanical spline which has been bent into a curve by placing its ducks on the drafting table. If the draftsman now moves one of the ducks to a new location, the entire spline changes shape. A local movement of a duck causes global changes in the curve. We shall be interested here in *local* splines only. A movement of a knot (what the mathematical equivalent of a duck is usually called) causes a change in the curve only in the local neighborhood of the knot.

How Splines Are Used in Computer Animation

Splines are used in three principal ways in computer animation:

1. To define spatial shapes—eg, the shape of an object in two or three dimensions. The knots correspond to points on the surface of the object.
2. To define the path of an object through space. The knots correspond to points on the path.
3. To define eases—the velocity of movement along a given path. An "ease in" has slow velocity at the beginning of a path, high at the end. An "ease out" is the reverse. We use "ease" here as a generalization of these two main forms.

In general, splines are used for modeling (cf 1 above) and to pass smooth space-time curves through parameters known at sparse times—namely keyframe times (cf 2 and 3 above).

A polynomial—hence a spline—is a function of one variable. In computer animation, the one variable is frequently time. Suppose, for example, we want to move a cube through time and have been given the x locations of one of its corners at several keyframe times. We wish to generate the corresponding x locations for inbetween frames. If we simply did linear interpolation between x's, the cube would make discontinuous changes in direction when animated. This is where splines shine; the x values are splined together. Because of the continuity property of a spline, there will be no discontinuities when the cube is animated. Most of the early computer animation programs used linear interpolation instead of splines, one of the reasons their output had a characteristic "mathematical" look. Of course, "mathematical" in this unhappy context really means "low-order mathematical".

Points usually have more than one coordinate, of course. So to spline points together means to spline their x coordinates together, their y coordinates together, and their z coordinates together. Thus it is customary to represent splining of points by presenting a solution for one coordinate, say x, and assuming the same solution for y and z.

The Two Classes of Splines

Interpolation splines are those which pass through their knots. The earliest splines were all interpolating since the mechanical spline from which they were derived (arguably) "interpolated" its ducks. The splines of a newer class, the *approximating splines*, approach but do not intersect their knots, which in this case are sometimes called *control points*. By analogy, the knots for interpolating splines are sometimes referred to as control points also. So the two classes of splines we are interested in here are local interpolating splines and local approximating splines.

The interpolating splines have first-order continuity (continuity of the first derivative at knots) and approximately splines have second-order continuity (continuity of the first and second derivative at knots). They are both, of course, infinitely differentiable at points which are not knots. The approximating splines, in a sense, trade off the desirable feature of having the knots on the curve for another desirable feature, increased grace from second-order continuity. We shall present a very useful local interpolation spline then generalize it. Then we shall do the same for local approximating splines. The generalization in both cases allows us to handle a shape parameter called "tension". In all cases, a spline can be generated from its control points by use of a 4×4 matrix.

The Classic Interpolating Spline

A local interpolating spline used for many years by graphicists has gone under several names: the *cardinal* spline, the *Catmull-Rom* spline [CATROM], or the *Overhauser* spline [OVRHSR]. It has obviously been reinvented several times. I shall call the generalized class (see below) "cardinal" splines and the most popular special case "Catmull-Rom" for the authors of the first paper I read containing it [CATROM][1]. I owe the generalization to a chapter of an unpublished textbook by Jim Clark (issued as a technical memo [CLARKJ]).

The cubic Catmull-Rom Spline may be specified with the following 4x4 matrix:

$$\mathbf{C} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

This matrix is used—as are all which we shall present here—to generate a spline curve as follows: Given a list of *x*-coordinates, and a parameter *u* which will take us along the spline connecting (or approximately connecting) one coordinate x_0 to the next x_1 as the parameter is varied from 0 to 1, a new *x*-coordinate is obtained from each value of *u* from the four nearest given *x*-coordinates (two behind, two ahead, along the curve) by $U\mathbf{C}X^T$, where

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

and

---

[1] This paper introduced a very general method of generalizing classes of splines. The so-called Catmull-Rom spline is just one of the many possible results of applying this method.

$$X = \begin{bmatrix} x_{-1} & x_o & x_1 & x_2 \end{bmatrix}.$$

Recall that the y-coordinates and z-coordinates of a given set of points to be interpolated would be treated similarly, as would any other parameter to be smoothly interpolated through an animation—eg, the angle of rotation about the z-axis. The Appendix contains C code realizing this scheme. This routine will handle all splines presented in these notes. It is simple, which is one of the messages of these notes. These splines are so simple and beautiful they should be taught right along side other basic functions such as sine and cosine.

The Cardinal Splines

A shape parameter called *tension* causes a spline to bend more sharply; it increases the magnitude of the tangent vector at the knots. The cubic cardinal spline generalization of the Catmull-Rom spline adds tension with the parameter labeled a in the following defining matrix:

$$\mathbf{C}_a = \begin{bmatrix} -a & 2-a & a-2 & a \\ 2a & a-3 & 3-2a & -a \\ -a & 0 & a & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Clearly, $\mathbf{C} = \mathbf{C}_{.5}$. Another popular value of a is 1 which, of course, has sharper bends than the Catmull-Rom but a simple integer matrix

The four curves defined by

$$c_0(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} C_0,$$

$$c_1(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} C_1,$$

$$c_2(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} C_2,$$

$$c_3(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} C_3,$$

where C_i is the ith column vector of matrix $\mathbf{C}_a$, are called the *basis segments* for the cubic cardinal spline, and the four curves drawn as one (see Figure CBASIS) are called the *basis function* for the cubic cardinal spline. A spline can be thought of as the curve $X(u)$ resulting from the sum of copies of the basis function positioned at each knot and weighed by its magnitude. This is the same as reconstruction of a function from its samples by a filter equal to the basis function. So the basis functions for splines turn out to be interesting filter functions for antialiasing. All splines discussed here have associated basis functions which may be derived similarly.

The parameter a has a simple interpolation which explains the use of the word "tension". The matrix above is used to generate, for each four consecutive points, the part of the spline curve between the middle two points. Suppose these four points are called x_{-1}, x_0, x_1, and x_2. Consider the tangent at points x_0; for cardinal splines it is parallel to the vector $x_{-1}x_1$. Similarly the tangent at point x_1 is parallel to the vector x_0x_2. The magnitudes of these two tangents are proportional to the lengths of the two vectors, respectively, and the constant of proportionality is a.

The cubic cardinal matrix may be easily derived from the two tangent constraints above and the fact that the curve must pass through x_0 and x_1. Since it is to be a cubic polynomial, the curve segment between these two points may be expressed as

$$X(u) = Au^3 + Bu^2 + Cu + D,$$

or

$$X(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} A & B & C & D \end{bmatrix}^T.$$

Similar functions for Y (u) and Z (u) are also required or course. Notice that

$$X'(u) = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} \begin{bmatrix} A & B & C & D \end{bmatrix}^T.$$

Our four constraints may be expressed by the following equations:

$$X(0) = x_0$$

$$X(1) = x_1$$

$$X'(0) = a(x_1 - x_{-1})$$

$$X'(1) = a(x_2 - x_0)$$

Thus, if s_0 and s_1 represent the slopes, or tangents, at points x_0 and x_1, respectively, then

$$\begin{bmatrix} x_0 \\ x_1 \\ s_0 \\ s_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}.$$

Let $\mathbf{M}_1$ represent the matrix in this equation. Also it is true that

$$\begin{bmatrix} x_0 \\ x_1 \\ s_0 \\ s_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & 0 & a & 0 \\ 0 & -a & 0 & a \end{bmatrix} \begin{bmatrix} x_{-1} \\ x_0 \\ x_1 \\ x_2 \end{bmatrix}.$$

Let $\mathbf{M}_2$ represent the matrix in this equation. From these two matrix equations it is easy to derive the relationship

$$\mathbf{C}_a = \mathbf{M}_1^{-1} \mathbf{M}_2.$$

The Classic Approximating Spline

The best-known approximating spline is the cubic B-spline which can be specified with the following matrix:

$$\mathbf{B} = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

It is used just as was the cardinal spline matrix above. Its basis segments are generated just as for the cardinal spline. In brief, they are given by

$$b_i = UB_i, \ 0 \le i < 4.$$

The B_i are the column vectors of **B.** Its basis function, the concatenation of the basis segments, is shown in figure BBASIS. This is another particularly interesting function for an antialiasing filter.

Besides having second-order continuity, the B-spline has another valuable property: It lies within the convex hull of its control points. It cannot get any "wilder" than its control points, or the polygon defined by its control points. Interpolating splines are not so nice. They can have kinks and wild fluctuations. All generalizations below of the B-spline share this convex-hull property.

Tensioned B-Splines

Tom Duff has generalized the B-splines to tensioned B-splines (paper in preparation, Lucasfilm) by analogy with the generalization of Catmull-Rom splines to cardinal splines. The tensioned cubic B-splines may be specified with the following matrix:

$$\mathbf{B}_a = \frac{1}{6}\begin{bmatrix} -a & 12-9a & 9a-12 & a \\ 3a & 12a-18 & 18-15a & 0 \\ -3a & 0 & 3a & 0 \\ a & 6-2a & a & 0 \end{bmatrix}$$

Clearly, $\mathbf{B} = \mathbf{B}_1$.

The Beta Splines

Brian Barsky [BARSKY] has generalized the B-spline even further by adding not only tension but also what he calls *bias*. Bias may be described as the tendency for a spline to bunch more to the left than to the right, or vice versa. More precisely, it is the ratio of the velocity right of a knot to the velocity left of the knot. Its action is best described with pictures (Figure BIAS). The biased and tensioned B-spline is called the Beta-spline, or β-spline. The cubic β-spline may be specified with the following matrix:

$$\mathbf{B}_{\beta_1,\beta_2} = \frac{1}{\delta}\begin{bmatrix} -2\beta_1^3 & 2\left(\beta_2 + \beta_1^3 + \beta_1^2 + \beta_1\right) & -2\left(\beta_2 + \beta_1^2 + \beta_1 + 1\right) & 2 \\ 6\beta_1^3 & -3\left(\beta_2 + 2\beta_1^3 + 2\beta_1^2\right) & 3\left(\beta_2 + 2\beta_1^2\right) & 0 \\ -6\beta_1^3 & 6\left(\beta_1^3 - \beta_1\right) & 6\beta_1 & 0 \\ 2\beta_1^3 & \beta_2 + 4\left(\beta_1^2 + \beta_1\right) & 2 & 0 \end{bmatrix}$$

where

$$\delta = \beta_2 + 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + 2 .$$

Clearly, $\mathbf{B} = \mathbf{B}_{1,0}$. Not so clearly, $\mathbf{B}_a$ can be shown to be a special case of $\mathbf{B}_{\beta_1,\beta_2}$ where

$$\beta_1 = 1$$

and

$$\beta_2 = 12\frac{1-a}{a} .$$

The Duff formulation of tension is slightly more convenient to use than the Barsky formulation because of a nicer range for the tension parameter. For example, the results of varying *a* from 0 to 1 are obtained by varying β_2 from ∞ to 0.

Knot Spacing

We have assumed throughout this discussion that the knots are uniformly spaced; in fact, they are assumed to be integers. Parameter *u* varying from 0 to 1 takes the spline from one integer knot to the next. This is reflected in the code in the Appendix. The code also works for knots of multiplicity greater than 1, where *multiplicity* is the number of times a knot is counted, or the number of times it appears consecutively in the list of knots provided as an argument to the routine. That is, we assume either unit spacing of knots or zero spacing in the case of multiplicities greater than 1.

For cubic splines, the first and last knots in a list used only for starting and stopping conditions (since only the middle two knots of each consecutive four are connected by application of the spline matrix to the four knots). If the spline is not to be a closed curve—it is not to be a *cyclic* spline—then it is frequently adequate to double the first and last knots, to give them multiplicity 2. That is, the first and second knot are made identical, and the last and next-to-last knot are made identical. To make a cyclic spline, simply append the last knot to the first of the list and append the (original) first two knots to the end.

The effect of a knot with multiplicity greater than 1 is to cause an approximating spline to approach the knot more closely. Thus it is like a local tension control but unfortunately is not subtle (but see The Future below). For interpolating splines, which already pass through the knots, increased multiplicity causes loops, or kinks, at the knots.

To use nonuniform spacing of knots would imply that the basis function changes shape as it is translated along the parameter *u* axis—ie, that a *nonuniform basis* is being used. Since the shape of the spline would certainly change with a different knot spacing, knot spacing might be considered to be a shape control. One method for assigning nonuniformly spaced knots separates them by distances proportional to the distances separating the values at the knots. So tightly bunched data is assigned to closely spaced knots. This technique [RIESEN] does not make dramatic changes in spline shape, relative that obtained with a uniform spacing, unless the data is tightly bunched. That this is true can be seen by considering what happens if the data is so close it actually coalesces into one point. Then the proportional spacing method of nonuniform basis places the corresponding knots atop one another to get, equivalently, a single knot of multiplicity 1. We have already seen that this causes the spline to approach its knot or to kink. The more general problem of assigning nonuniform spacing to knots is a large one. It is related to the difficult problem of sampling in perspective, where the perspective mapping causes nonuniform spacing between samples.

On Using the Program

The code of the Appendix generate all the splines in these notes. For convenient use in animation, however, there is one further consideration which the user may want to make. The code provided returns a constant number of points be-

tween knots. What is typically desired in animation are points evenly spaced along the length of the spline. Analytically, this is a difficult problem. An approximation which may be satisfactory in many cases assumes the spline is accurately represented by the polygon formed from the line segments joining the points returned between the knots. Then a cumulative length can be determined from the individual lengths of these line segments and equally spaced points marked off along this length.

Words We Have Not Used

Topics we have not covered which relate to splines—or rather we have not *had* to cover—include Bezier curves, Hermite interpolation, parabolic blending, Bernstein basis, and natural splines. On the other hand, a very interesting topic which we did not cover here is the generalization of splines to surface patches, a topic which is covered in several texts and which is perhaps more appropriate in the context of modeling that animation.

A Case in Point

The "Genesis Demo" for *Star Trek II: The Wrath of Khan* which we executed for Paramount featured a flyby of a planet by an imaginary spacecraft. Its path as a very complex spiraling move about the planet with several direction changes for dramatic impact and story purposes. We modeled it with a 6-dimensional cubic spline—ie, six parameters (three for position and three for orientation) were splined with the techniques outlined here.

The Future

One of the next advances in splines will be announced at SIGGRAPH 83. Brian Barsky and John Beatty have generated the β-spline in such a way that the shape parameters, bias and tension, are local properties of the spline just as is shape [BARBTY]. Throughout these notes, tension has been a global property, a fact inconsistent with the insistence of shape locality.

Acknowledgement

I first became aware of the 4×4 formulation of splines from an appendix in Ed Catmull's thesis [CATMUL]. These notes have benefited a great deal from discussions I have had with Tom Duff.

References

[BARBTY]   Brian A Barsky and John C Beatty, *Local Control of Bias and Tension in Beta-Splines*, SIGGRAPH 83, Detroit, Jul 25-29, 1983. [Also in *Varying the Betas in Beta-Splines*, Report No UCB/CSD 83/112, Computer Science Division (EECS), University of California, Berkeley, California, Mar 1983.]

[BARSKY]   Brian A Barsky, *The Beta-Spline: A Local Representation Based on Shape Parameters and Fundamental Geometric Measures*, PhD dissertation, Department of Computer Science, University of Utah, Salt Lake City, Dec 1981.

[CATMUL] Edwin Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, PhD dissertation, Department of Computer Science, University of Utah, Salt Lake City, Dec 1974.

[CATROM] Edwin Catmull and Raphael Rom, *A Class of Local Interpolating Splines*, **Computer Aided Geometric Design**, edited by Robert E Barnhill and Richard F Riesenfeld, Academic Press, San Francisco, 1974, 317-326.

[CLARKJ] James H Clark, *Parametric Curves, Surfaces and Volumes in Computer Graphics and Computer-Aided Geometric Design*, Technical Report 221, Computer Systems Laboratory, Stanford University, Palo Alto, California, Nov 1981.

[DEBOOR] Carl de Boor, **A Practical Guide to Splines**, Springer-Verlag, New York, 1978.

[OVRHSR] J A Brewer and D C Anderson, *Visual Interaction with Overhauser Curves and Surfaces*, **Computer Graphics** (SIGGRAPH 77 Proceedings), Vol 11, No 2, San Jose, California, Jul 20-22, 1977, 132-137.

[RIESEN] Richard Riesenfeld, *Applications of B-spline Approximation to Geometric Problems of Computer-Aided Design*, PhD dissertation, Department of Computer Science, University of Utah, Salt Lake City, Mar 1973.

APPENDIX: Spline Program

```
/*  CubicSpline.c—Generates a cubic spline through given list of (x,y,z)
        triples.
    Given: Number of knots n (including two endpoint knots),
        the list of knots as (x,y,z) triples,
        the grain = number of line segments desired between knots,
        the type of spline (BETA, CARDINAL),
        the parameters for tension and bias.
    Returned: (n - 3) * grain + 1 points along spline, including the knots.
        The two endpoints (used for end conditions only)
        and any points between them and their nearest neighbor
        knots are NOT returned.
    CubicSpline() returns pointer to list of points.
        (NULL is returned for error.)
*/


#define BETA 1
#define CARDINAL 2
#define NULL 0
#define PNTLMT 4096     /* max. no. of points which may be returned */
#define GRNCNT 256      /* max. no. of line segments between knots */


typedef struct { double x,y,z; } point;
typedef double matrix [4][4];


static point Spline[PNTLMT];  /* points returned here /*
```

```
point *CubicSpline(n, knots, grain, type, tension, bias)
    point *knots;
    int n, grain, type;
    double tension, bias;
{
    register point *s, *k0, *km1, *k2;
    int 1, j, last;
    double alpha[GRNCNT], matrix();
    matrix m;

    if( n<3 || grain<0 || grain>GRNCNT ) return(NULL);
    last = (n-3)*grain + 1;
    if( last>PNTLMT ) return(NULL);
    if( type!=BETA && type!=CARDINAL ) return(NULL);
    if( type==BETA ) {
        if( bias==0. && tension==0.) return(NULL);
        GetBetaMatrix(bias, tension, m);
    } else GetCardinalMatrix(tension, m);
    for(i=0; i<grain; i++) alpha[i] = ((double )i)/grain;
    s = Spline;
    km1 = knots;
    k0 = km1+1; k1 = k0+1; k2 = k1+1;
    for(i=1; i<n-1; i++) {
        for(j=0; j<grain; j++) {
            s->x = Matrix(km1->x, k0->x, k1->x, k2->x, alpha[j], m);
            s->y = Matrix(km1->y, k0->y, k1->y, k2->y, alpha[j], m);
            (s++)->z = Matrix(km1->z, k0->z, k1->z, k2->z, alpha[j], m);
        }
        k0++; km1++; k1++; k2++;
    }
    return(Spline);
}

GetBetaMatrix(b0, b1, m)
    double b0, b1;
    matrix m;
}
    register i, j;
    double d, b2, b3;

    b2 = b0*b0;
    b3 = b0*b2;
    m[0][0] = -2.*b3;
    m[0][1] = 2.*(b1+b3+b2+b0);
    m[0][2] = -2.*(b1+b2+b0+1.);
```

```
    m[1][0] = 6.*b3;
    m[1][1] = -3.*(b1+2.*b3+2.*b2);
    m[1][2] = 3.*(b1+2.*b2);
    m[2][0] = -6.*b3;
    m[2][1] = 6.*(b3-b0);
    m[2][2] = 6.*b0;
    m[3][0] = 2.*b3;
    m[3][1] = b1+4.*(b2+b0);
    m[0][3] = m[3][2] = 2.;
    m[1][3] = m[2][3] = m[3][3] = 0.;
    d = 1./(b1+2.*b3+4.*b2+4.*b0+2.);
    for(i=0; i<4; i++) for(j=0; j<4; j++) m[i][j] *= d;
}

GetCardinalMatrix(a, m)
    double a;
    matrix m;
{
    m[0][1] = 2.-a;
    m[0][2] = a-2.;
    m[1][0] = 2.*a;
    m[1][1] = a-3.;
    m[1][2] = 3.-2*a;
    m[3][1] = 1.;
    m[0][3] = m[2][2] = a;
    m[0][0] = m[1][3] = m[2][0] = -a;
    m[2][1] = m[2][3] = m[3][0] = m[3][2] = m[3][3] = 0.;
}

double Matrix(a, b, c, d, alpha, m)
    double a, b, c, d, alpha;
    matrix m;
{
  double p0, p1, p2, p3;

  p0 = m[0][0]*a + m[0][1]*b + m[0][2]*c + m[0][3]*d;
  p1 = m[1][0]*a + m[1][1]*b + m[1][2]*c + m[1][3]*d;
  p2 = m[2][0]*a + m[2][1]*b + m[2][2]*c + m[2][3]*d;
  p3 = m[3][0]*a + m[3][1]*b + m[3][2]*c + m[3][3]*d;
  return( p3+alpha*(p2+alpha*(p1+alpha*p0)) );
}
```
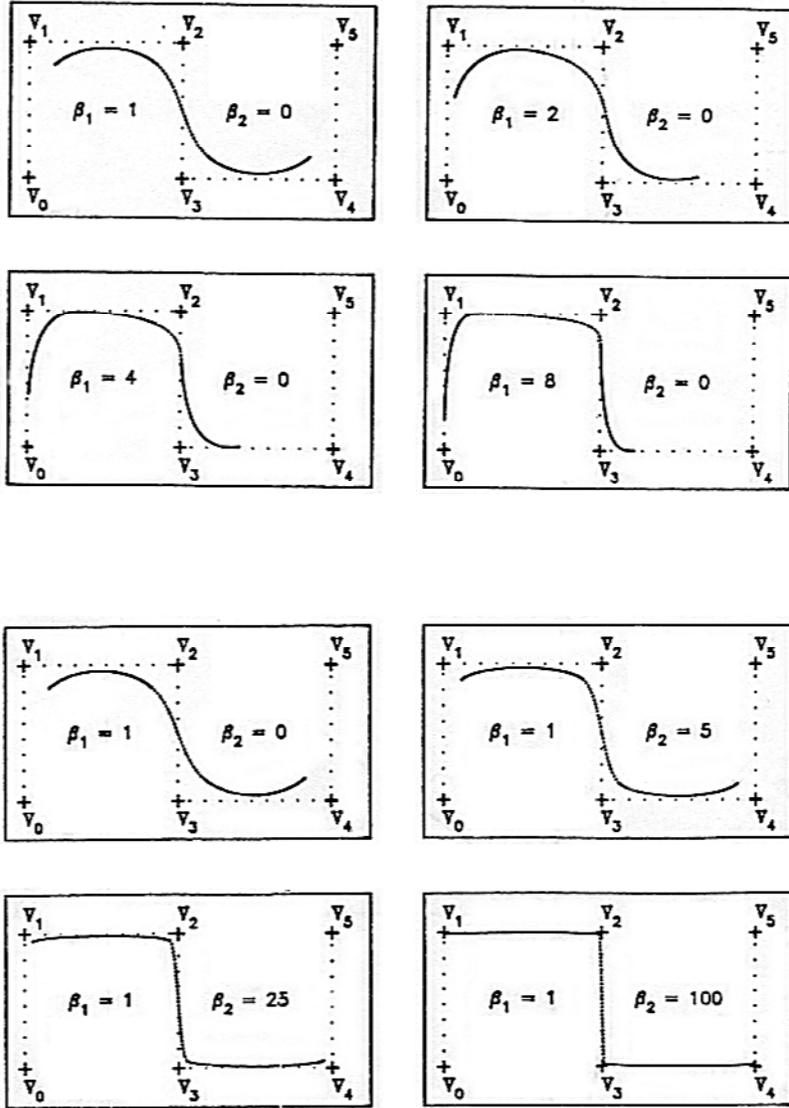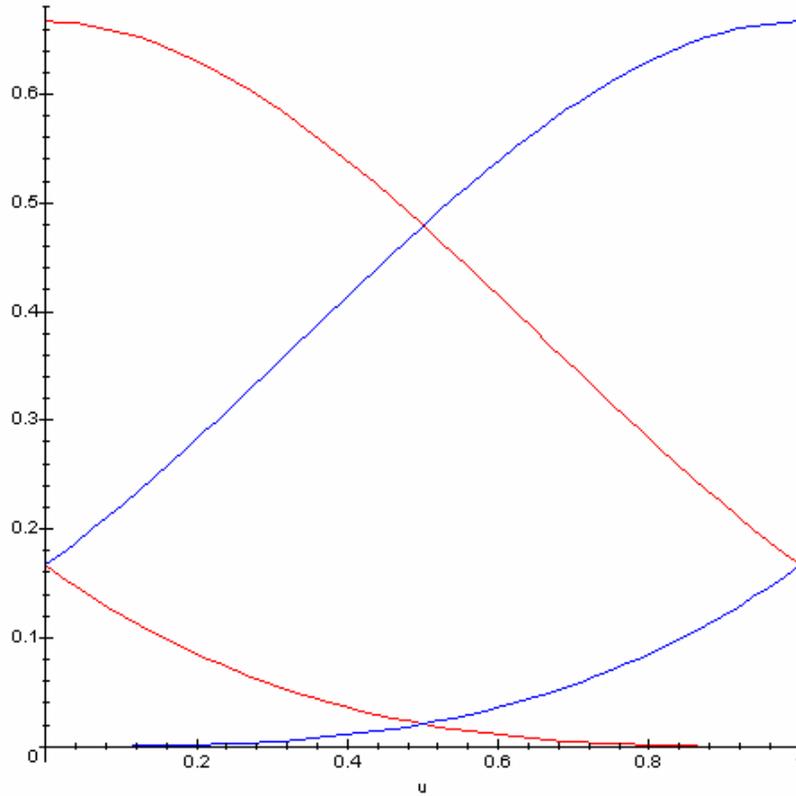
Figure BIAS.

Adapted from [BARBTY]

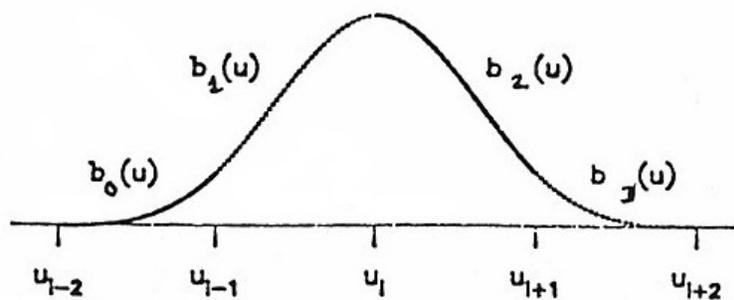[Above, b_0 and b_1 red, b_2 and b_3 blue. Plotted in Maple.]
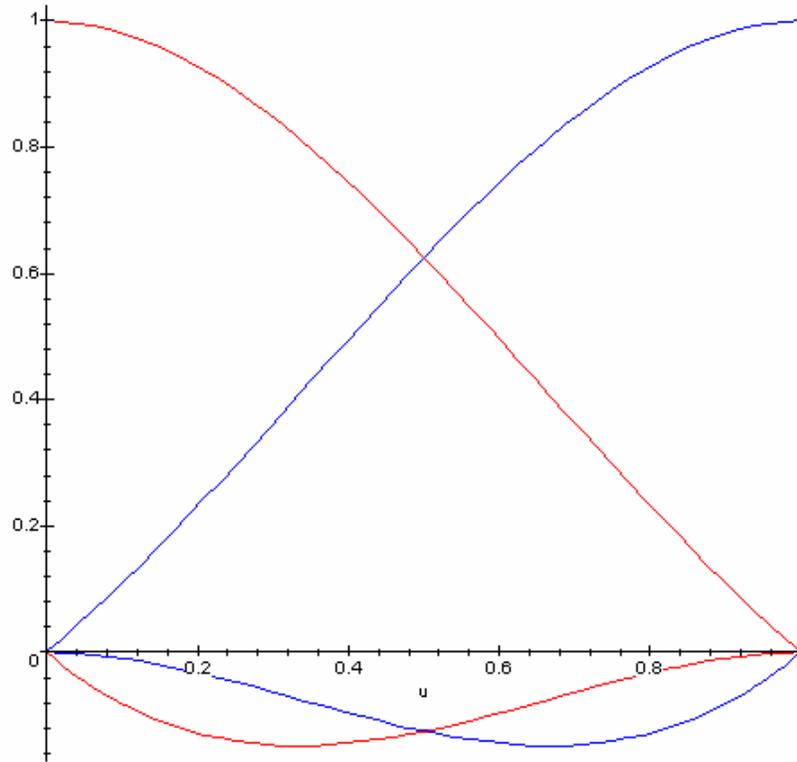
Figure BBASIS[2].

(Adapted from [BARBTY])

---

[2] [The function labels are reversed left to right. So b_0 and b_3 labels should be swapped; b_1 and b_2 labels too. See the new figures above, added Nov 1999.]

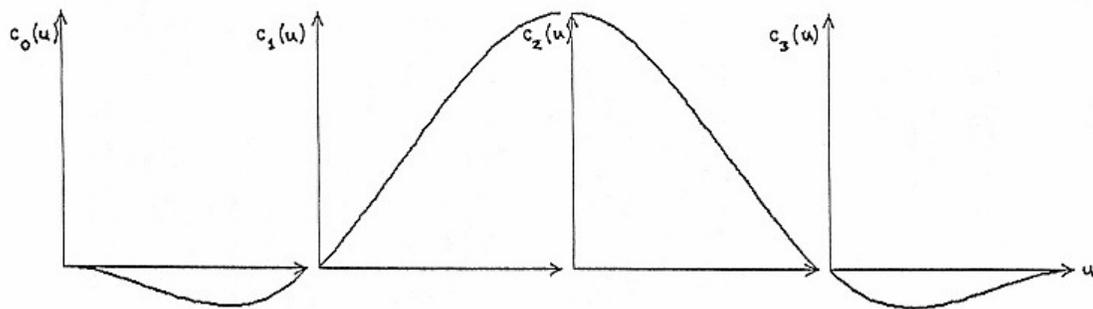[Above, c_0 and c_1 red, c_2 and c_3 blue. Plotted in Maple.]

Figure CBASIS.

(Adapted from [CLARKJ])

The Cardinal spline basis functions[3]. The functions have been illustrated together in this way to demonstrate the continuity they share.

---

[3] [The function labels are reversed left to right. So c_0 and c_3 labels should be swapped; c_1 and c_2 labels too. See the new figures above, added Nov 1999.]