The Basic Curves and Surfaces of Computer Aided Geometric Design

Colm Mulcahy [*]

Introduction

Computer Aided Geometric Design (CAGD) plays a major role in the design of cars, airplanes, and submarines, as well as in many modern manufacturing processes. The mathematics behind CAGD is also indispensable in computer graphics.

We demonstrate the use of Maple V Release 3 as an educational tool in the construction, plotting and manipulation of the basic curves and surfaces of CAGD. This can be done using a bare minimum of Maple, hence anybody who knows a little linear algebra and multivariate calculus can be introduced to this important material. Maple's numerical and symbolic capabilities take the drudgery out of computing with the formulae, as well as providing immediate visual access to the resulting shapes.

Topics which can easily be explored in this way include: polynomial and parametric interpolation, least squares and Bezier curves, Hermite and natural cubic splines, tensor product surfaces, lofting and Coons surfaces, B-splines, B-spline curves and surfaces, interpolation with B-spline curves, least squares B-spline methods, and NURBS (non-uniform rational B-splines). We provide examples of many of these constructions, to give the general flavour of the subject. We concentrate on planar curves—the extension to space curves is routine. The emphasis throughout is on (piecewise) polynomial methods, and their rational counterparts.

There are essential aspects of CAGD, which can also be investigated with the help of Maple, that we will not have time to touch on, such as recipes for drawing the curves and surfaces, important and illuminating connections with projective geometry, numerical analysis considerations, advanced spline algorithms, and differential geometry. See [1, 2, 3, 4] for further information.

[*]Department Of Mathematics, Spelman College, PO Box 373, 350 Spelman Lane, Atlanta, GA 30314, USA; email: colm@auc.edu; URL: http://www.auc.edu/~ colm

Polynomial interpolation

The central idea of interpolation is to find a polynomial which goes through prescribed data points $(x_i, y_i), 0 \leq i \leq n$. Maple's `interp` command uses Newton interpolation to find the unique $y = f(x)$ of degree less than or equal to n—one less than the number of points—which does the job.

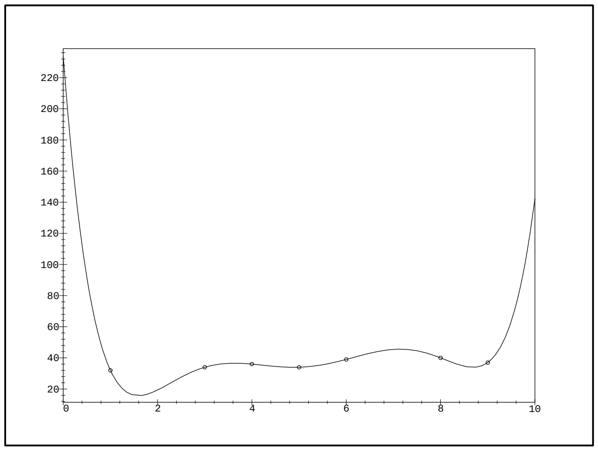
We try this for the seven data points $(1, 32)$, $(3, 34)$, $(4, 36)$, $(5, 34)$, $(6, 39)$, $(8, 40)$, and $(9, 37)$:

```
>  xx:=[1, 3, 4, 5, 6, 8, 9]:
>  yy:=[32, 34, 36, 34, 39, 40, 37]:
>  f:=interp(xx,yy,x);
```

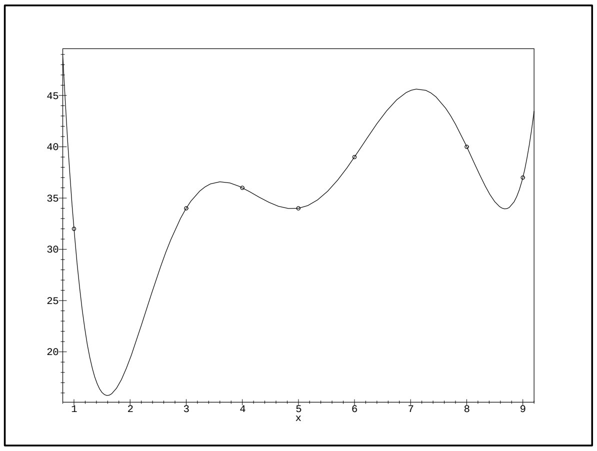
$$f := \frac{767}{20160}x^6 - \frac{379}{320}x^5 + \frac{41657}{2880}x^4 - \frac{84127}{960}x^3 + \frac{393091}{1440}x^2 - \frac{48097}{120}x + \frac{1639}{7}$$

While we get no hint as to how the answer was arrived at, the mathematics involved is elementary and well known (see [3, Chapter 6] or [4, p. 57]). Let's *look at* this interpolant, along with the data points. For politeness sake we plot from 0 to 10.

```
>  pts:=j->(xx[j],yy[j]): n:=6:
>  points:=[seq(pts(j),j=1..n+1)]:
>  POINTS:=plot(points, style=point,
>        symbol=circle, axes=boxed):
>  INTERPOL:=plot(f,x=0..10):
>  with(plots): display({POINTS,INTERPOL});
```



Right away, we see that although polynomial interpolants do the job they are assigned, they do strange things outside of the range of data points. They certainly cannot be trusted for extrapolation purposes! Plotting from 0.8 to 9.2 instead yields:



This highlights another undesirable aspect of our approach so far: the inevitable wiggling of high degree polynomials *within* the range of interest. In addition, the x coordinates need to be distinct.

For an orientation free approach, we could start with distinct $t_0, t_1, \ldots t_n$, and obtain a *parametric* curve $(f(t), g(t))$ such that $(f(t_i), g(t_i)) = (x_i, y_i)$ for all i, by taking $f(t)$ and $g(t)$ to be the unique polynomials of degree less than or equal to n such that $f(t_i) = x_i$ and $g(t_i) = y_i$ for all i. For fixed (x_i, y_i), different t_i's result in different curves, some of them relatively wiggle-free [4, pp. 201-2].

Bezier Curves

Instead of insisting on interpolation, we could settle for some sort of approximation to our data. We might ask for a "best fit" polynomial of lower degree. If we seek a cubic, in the case of seven points, then unless we are lucky, we are not going to get an exact fit. The method of least squares is one popular technique used in such circumstances—it can be explored using Maple's `leastsquare` command.

Bezier curves provide another alternative. We start with points $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$—note the switch to vector notation here—and end up with a parametric curve which is polynomial of degree less than or equal to n in each slot.

The *Bezier curve* associated with *control points* $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$ is defined for $0 \le t \le 1$ by

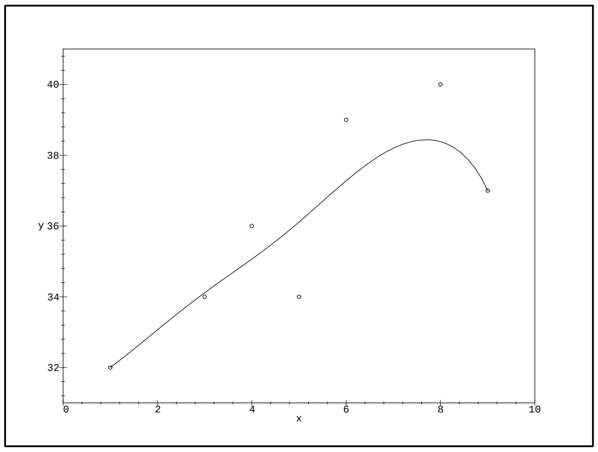
$$\mathbf{B}(t) = \sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) t^i (1-t)^{n-i} \mathbf{P}_i$$

The functions attached to the points here are known as the Bernstein basis functions.

Cubic Bezier curves use four control points, and the basis functions $(1-t)^3, 3(1-t)^2 t, 3(1-t)t^2, t^3$. These curves have many applications, sometimes in the equivalent Hermite formulation, specified in terms of just two control points with specified tangent vectors there [3, pp. 91]. Large curves can be built up by stringing many such segments together, with tangent continuity at the joins [3, pp. Chapter 8]). These cubic curves are also the basis of font design in **PostScript** [3, p. 130]). Consequently, every page in this journal is bursting at the seams with Bezier curves!

Let's look at the Bezier curve determined by our original data points, utilizing the `binomial` command. We replot the points over an extended range, making some labelling adjustments on account of the vector origins of `pts` and the desired form of the points $\mathbf{P}_i$'s. We take the linear combination of control points with the help of `evalm`.

```
> n:=6:  P:= i-> [pts(i+1)]:
> POINTSE:=plot(points, x=0..10, y=31..41,
>    style=point,symbol=circle,axes=boxed):
> brn:= (i,n,t) ->
>    binomial(n,i) * t^i * (1-t)^(n-i):
> bez:= t -> evalm
>     (sum('brn(i,n,t)*P(i)', 'i'=0..n)):
> BEZ:=plot([ bez[1], bez[2], t=0..1 ]):
> display({POINTSE,BEZ});
```



The resulting picture illustrates some standard properties of Bezier curves. For instance, Bezier curves always lie within the convex hull of their control points, since on $[0,1]$ the Bernstein basis functions are clearly non-negative and sum to 1. Maple can verify the latter claim symbolically—even if we switch from n to the unspecified nn:

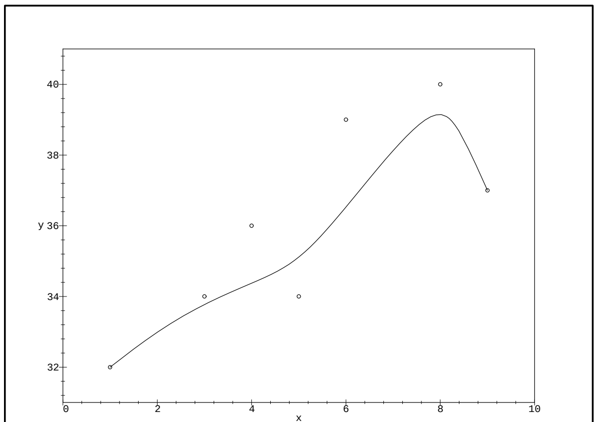
```
>  simplify(sum('brn(i,nn,t)','i'=0..nn));
                         1
```

As the picture suggests, generally only the first and last control points are interpolated. The intermediate control points influence the curve's shape in a different way, acting more like magnets. There are various ways to adjust the influence of the control points. One could repeat some points, i.e., list them more than once, but increasing the number of points also increases the degree of the resulting curve. Another restriction inherent to the Bezier approach is the fact that the curves change totally as soon as one control point is moved.

Let's broaden our horizons, and not restrict ourselves to polynomials. Given points $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$, and corresponding "weights" $w_0, w_1, \ldots, w_n$, the associated *rational Bezier curve* is defined on $[0,1]$ by

$$\mathbf{R}(t) = \frac{\sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array}\right) t^i(1-t)^{n-i} w_i \mathbf{P}_i}{\sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array}\right) t^i(1-t)^{n-i} w_i}$$

It is an easy matter to modify the definition of `bez` above to accommodate the weights and the new denominator.



The above plot shows the rational curve determined by our seven data points, taking w_3, and w_5 to be 5, and the other w_i's to be 1. Note the curve's newfound affection for the fourth and sixth points.

If the weights are all equal then the rational Bezier curve reduces to the ordinary Bezier curve, since the denominator simplifies to the common weight. Hence only select control points can be "emphasized." Readers who are curious about the effects of zero or negative weights can experiment for themselves at this point.

An important application of rational quadratic Bezier curves is to the construction of (bits of) conic sections—including circles, ellipses and hyperbolas—without resorting to trigonometric or hyperbolic functions [3, pp. 244-8] [4, pp. 155-8].

In our brief tour of Bezier curves, we got more than a glimpse of things to come. From now on all curves will be of the form $\mathbf{C}(t) = \sum_{i=0}^{n} f_i(t)\mathbf{P}_i$, for control points $\mathbf{P}_i$, and basis functions $f_i(t)$. The interpolation functions considered earlier can also be realized in this way where the $f_i(t)$ are the well known Lagrange functions [3, Chapter 6]. In the polynomial Bezier case, the $f_i(t)$'s are the Bernstein basis functions. It will often happen that $\sum_{i=0}^{n} f_i(t) = 1$ as well, at least on some parameter interval, which ties in with the convex hull property mentioned above when the basis functions are also non-negative.

Elementary Surface Patches

We can use the interpolation and Bezier constructions already discussed to come up with two broad classes of surface patches, so-called tensor product surfaces and lofting surfaces. By a surface patch we mean a function of two parameters u, v, plotted over some rectangle in the u, v plane, taking values in three space. Surfaces of the type $z = f(x,y)$ can be realized this way, as $\mathbf{S}(u,v) = [u, v, (f,u,v)]$.

First we consider a *tensor product surface patch* built up from cubic Beziers. We start with a grid of sixteen control points $\mathbf{P}_{i,j}$ $(0 \le i, j \le 3)$, and then consider the Bernstein functions $\frac{3!}{i!(3-i)!}u^i(1-u)^{3-i}$, and $\frac{3!}{j!(3-j)!}v^j(1-v)^{3-j}$, used in the definitions of the cubic Bezier curves controlled by $\mathbf{P}_{0,0}$, $\mathbf{P}_{1,0}$, $\mathbf{P}_{2,0}$, $\mathbf{P}_{3,0}$, and $\mathbf{P}_{0,0}$, $\mathbf{P}_{0,1}$, $\mathbf{P}_{0,2}$, $\mathbf{P}_{0,3}$, respectively.

We use the set of all possible products of these one variable functions as a basis for our surface: for $u, v \in [0,1]$, define $\mathbf{S}(u,v)$ to be

$$\sum_{i,j=0}^{3} \left(\begin{array}{c} 3 \\ i \end{array}\right) \left(\begin{array}{c} 3 \\ j \end{array}\right) u^i(1-u)^{3-i}v^i(1-v)^{3-i}\mathbf{P}_{i,j}$$

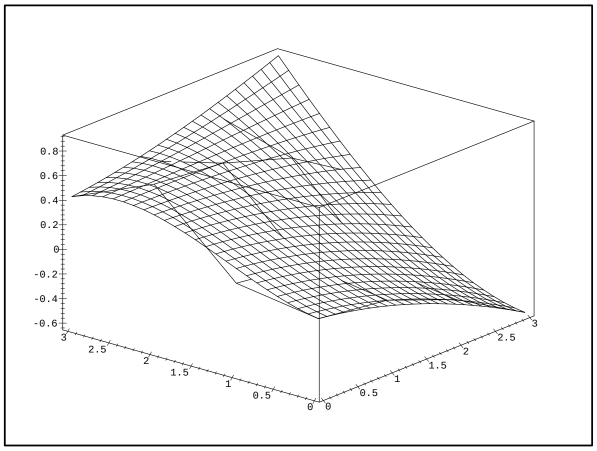
This surface interpolates the corner points $\mathbf{P}_{0,0}$, $\mathbf{P}_{0,3}, \mathbf{P}_{3,0}$, and $\mathbf{P}_{3,3}$—its precise shape may be varied by altering these and the other control points.

One way to get control points is to sample a surface, say $z = -\sin(\frac{3x^2-4y^3}{40})$, on a regular 4×4 grid. We construct and save the mesh these points determine, to aid in visualizing them later.

```
>  P:=(x,y)->[x,y,-sin((3*x^2-4*y^3)/40)]:
>  GRID:=plot3d(P(x,y), x=0..1, y=0..1,
>     grid=[4,4], axes=box, color=black,
>     style=wireframe, thickness=2):
```

We compute and plot the surface, using advance hindsight to chose a suitable viewing angle.

```
>  sur:=evalm(sum(sum('brn(i,3,u)*brn(j,3,v)
>                *P(i,j)','j'=0..3),'i'=0..3)):
>  SUR:=plot3d([sur[1],sur[2],sur[3]],
>     u=0..1, v=0..1, shading=none):
>  display({GRID,SUR},orientation=[-140,60]);
```

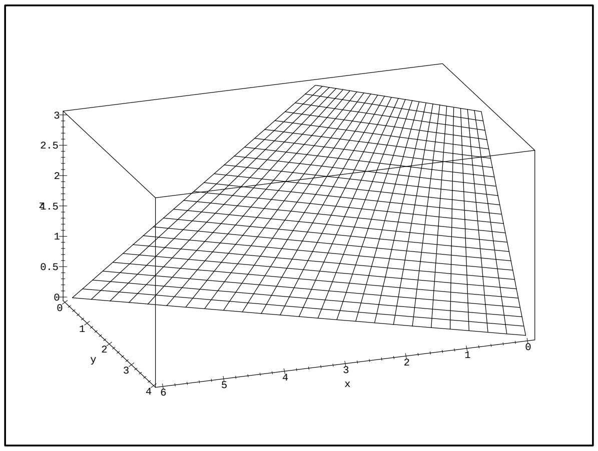


A simpler example of a tensor product surface patch is the *bilinear surface patch*, which also turns out to be the easiest example of our second class of surface patches. We start with linear Beziers, i.e., straight lines, connecting four points $\mathbf{Q_{0,0}}$, $\mathbf{Q_{1,0}}$, $\mathbf{Q_{1,1}}$, $\mathbf{Q_{0,1}}$ (in that order), yielding:

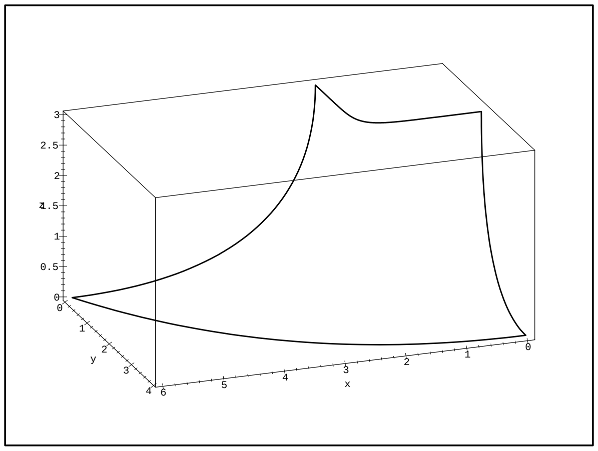
$$\begin{aligned}\mathbf{Bil}(u,v) &= (1-u)(1-v)\mathbf{Q_{0,0}} + (1-u)v\mathbf{Q_{0,1}} \\ &\quad +u(1-v)\mathbf{Q_{1,0}} + uv\mathbf{Q_{1,1}}\end{aligned}$$

for $u, v \in [0,1]$. This is the simplest surface joining four points. It is a ruled surface, and interpolates the lines which connect the given points.

If we take $\mathbf{Q_{0,0}} = (6,0,0)$, $\mathbf{Q_{1,0}} = (2,0,3)$, $\mathbf{Q_{1,1}} = (0,2,3)$, and $\mathbf{Q_{0,1}} = (0,4,0)$ we get:



Now suppose we have four curves connecting the points: say $\mathbf{C_{u,0}}$ and $\mathbf{C_{u,1}}$ joining $\mathbf{Q_{0,0}}$ to $\mathbf{Q_{1,0}}$ and $\mathbf{Q_{0,1}}$ to $\mathbf{Q_{1,1}}$ respectively, and $\mathbf{C_{0,v}}$ and $\mathbf{C_{1,v}}$ joining $\mathbf{Q_{1,0}}$ to $\mathbf{Q_{1,1}}$ and $\mathbf{Q_{0,0}}$ to $\mathbf{Q_{0,0}}$ respectively, where each parameter is in the interval $[0,1]$. These determine a frame, and now we seek a surface patch which has these as its boundary. Here is such a frame, drawn using `spacecurve` and `display`:



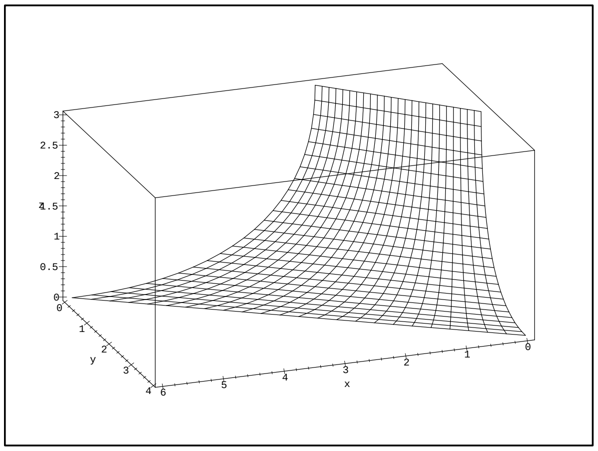
Here $\mathbf{C_{0,v}}$ is the Bezier curve $(1-v)^2\mathbf{Q_{0,0}} + 2(1-v)v(4,4,0) + v^2\mathbf{Q_{0,1}}$; $\mathbf{C_{1,v}}$ is a quarter of the intersection of the cylinder $x^4 + y^4 = 2^4$ with the plane $z = 3$, i.e., $\mathbf{C_{1,v}} = (2\cos^{\frac{1}{4}}(\frac{\pi v}{2}), 2\sin^{\frac{1}{4}}(\frac{\pi v}{2}), 3)$;

$\mathbf{C_{u,0}} = (1-u)^2\mathbf{Q_{0,0}} + 2(1-u)u(2,0,0) + u^2\mathbf{Q_{1,0}}$, and $\mathbf{C_{u,1}} = (1-u)^2\mathbf{Q_{0,1}}+2(1-u)u(0,2,0)+u^2\mathbf{Q_{1,1}}$. The parametrization of $\mathbf{C_{1,v}}$ provoked us to set `numpoints` equal to 270 for that spacecurve plot.

We can string a hammock between the curves $\mathbf{C_{u,0}}$ and $\mathbf{C_{u,1}}$ by lofting in the v direction with

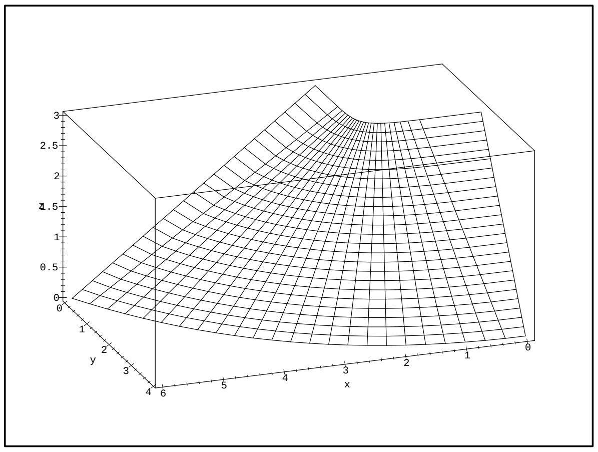
$$\mathbf{Loft_v}(u,v) = (1-v)\mathbf{C_{u,0}} + v\mathbf{C_{u,1}},$$

for $u, v \in [0,1]$. This linearly interpolates between corresponding points on each curve.



We can also loft in the u direction with

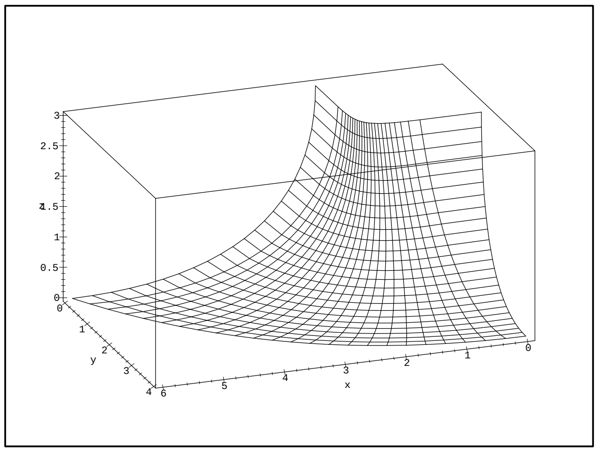
$$\mathbf{Loft_u}(u,v) = (1-u)\mathbf{C_{0,v}} + u\mathbf{C_{1,v}},$$



The *Coons surface* construction gives a surface patch which interpolates all four of the space curves. Its parametric equation is given by:

$$\mathbf{Coons}(u,v) = \mathbf{Loft_u}(u,v)+\mathbf{Loft_v}(u,v)-\mathbf{Bil}(u,v)$$

for $u, v \in [0,1]$. Our curves yield:



That this patch does indeed interpolate the four ingredient curves can be verified by hand—or we can let Maple do the work for us. For instance, with `Bil`, `Loft_u` and `Loft_v` defined as above, and `Coons := evalm(Loft_u + Loft_v - Bil)`, then `evalf(subs(v=0,eval(Coons)))` does indeed yield $(6(1-u)^2) + 4u(1-u) + 2u^2, 0, 3u^2) = \mathbf{C_{u,0}}$.

(We can alleviate the clutter of curves at the top of the $\mathbf{Loft_u}$ and **Coons** plots, by re-parametrizing the curve $\mathbf{C_{1,v}}$ so as to spread out more evenly the points obtained by uniformly sampling $v \in [0,1]$. One possibility is to replace each v in the definition of $\mathbf{C_{1,v}}$ by $\frac{\arctan(8v-4)+\arctan(4))}{2\arctan(4)}$.)

Designer surfaces for engineering purposes may be obtained by stitching together lofting, Coons, and Bezier patches—or the more general B-spline surfaces or NURBS patches discussed later—subject to stringent matching conditions across the joins.

B-splines

B-splines are a class of functions made up of pieces of polynomials, joined together in some fashion. We start by choosing an $m+1$-tuple $T = [t_0, t_1, \ldots, t_m]$ of non-decreasing real numbers, which is called the *knot vector.* We then define the *B-splines $B_{i,j}$ of order j* recursively, as follows. Set

$$B_{i,1}(t) = \begin{cases} 1 & \text{on } [t_i, t_{i+1}) \\ 0 & \text{elsewhere} \end{cases}$$

for $0 \le i \le m-1$ (with $B_{m-1,1}(t) = 1$ on $[t_{m-1}, t_m]$), and for any $j \le m$, we define

$$B_{i,j}(t) = w_{i,j}(t)B_{i,j-1}(t)+(1-w_{i+1,j}(t))B_{i+1,j-1}(t)$$

for $0 \le i \le m-j$, where the $w_{i,j}$ are given by

$$w_{i,j}(t) = \begin{cases} \frac{(t-t_i)}{(t_{i+j-1}-t_i)} & \text{if } t_{i+j-1} \neq t_i \\ 0 & \text{otherwise} \end{cases}$$

Given T and $k \le m$, we thus get $m-k+1$ piecewise polynomials $B_{i,k}(t)$ of degree at most $k-1$. The pieces which constitute each function join up at the knots, where they exhibit varying degrees of smoothness.

Order one B-splines $B_{i,1}(t)$ are step functions, order two B-splines $B_{i,2}(t)$ are zig-zag linears, order three B-splines $B_{i,3}(t)$ are piecewise quadratics, and so on.

It is difficult for the average reader to find much comfort or inspiration in the above formulae the first time around. We need examples and pictures—and fortunately Maple comes to the rescue. Our first example turns out to be an old friend.

We force Maple to start counting the array elements at index 0 to facilitate the standard notation for the definition of the B-splines.

```
> m:=7:  T:=array(0..m, [0,0,0,0,1,1,1,1]):
> w:=(i,j,t)->
>     if     T[i+j-1]=T[i]  then  0
>     else  (t-T[i])/(T[i+j-1]-T[i])  fi:
```

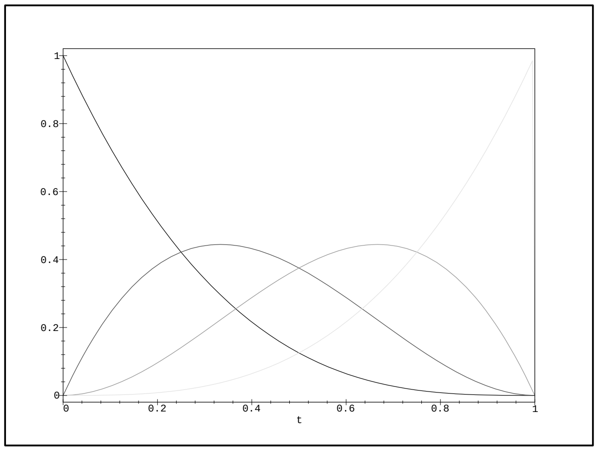
We start the ball rolling by defining step functions $S(a,b,t)$ on intervals of the form $[a,b)$ with the help of Maple's `Heaviside` command.

```
> S:=(a,b,t)->Heaviside(t-a)-Heaviside(t-b)
> B:=(i,j,t)->
>  if j=1 then S(T[i],T[i+1],t)
>  else
>     w(i,j,t)*B(i,j-1,t) +
>        (1-w(i+1,j,t))*B(i+1,j-1,t) fi:
```

(Note: $S(a,b,t)$ is 1 on $[a,b)$ if $a<b$, and 0 everywhere if $a=b$. We are being sloppy here and ignoring possible problems at the right most knot value).

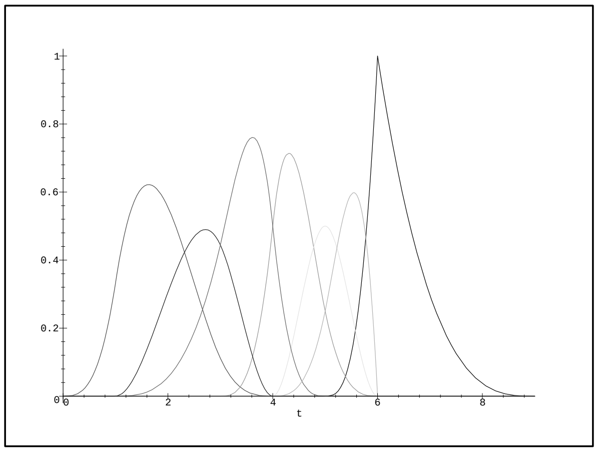
Maple is primed and ready to compute the B-splines: let's examine the cubics, i.e., the $B_{i,4}(t)$'s. There should be $7-4+1=4$ of them:

```
> k:=4: curves:= {seq(B(i,k,t),i=0..(m-k)}:
> plot(curves, t=0..t.m, axes=boxed);
```



Here we have four polynomials (no pieces) defined on $[0,1)$. They are in fact the Bernstein functions $(1-t)^3$, $3(1-t)^2t$, $3(1-t)t^2$, t^3, from earlier. This can be verified by *not* suppressing the output of the `curves` definition above, and ignoring the clutter due to the Heaviside functions.

For our second example, we use the new knot vector $[0,1,1,3,4,4,5,6,6,6,9]$ of length 11, and again plot the resulting B-splines. Changing m and T above, and re-executing the code, yields seven piecewise cubic functions on the interval $[0,9]$. In principle each comes in six pieces, switching at $t = 1,3,4,5,$ and 6:



The multiplicity of a given knot (i.e., how often it is repeated in the knot vector) tells us a lot about the way the pieces of the B-splines join up there: at a knot of multiplicity ℓ, each $B_{i,k}$ is at least $k-\ell-1$

times continuously differentiable [4, p. 174]), [3, p. 169]). If $k - \ell - 1 = -1$, this is to be interpreted as a potential discontinuity.

Thus, the spike in the seventh function $B_{6,4}$ in the last plot is not totally unexpected. At that point, $t = 6$, the B-splines are only guaranteed to be $4 - 3 - 1$ times continuously differentiable, i.e., continuous but not necessarily differentiable. Similar considerations apply at $t = 1$ and at $t = 4$.

The seven B-spline functions are visibly non-negative on $[0, 9]$, and a plot of their sum suggests that they sum to 1 on the subinterval $[3, 6]$. (These are key properties which will soon prompt us to define B-spline curves.) We can take advantage of Maple's symbolic prowess to verify that they sum to 1 here if we utilize the `assume` facility to restrict attention to those t between 3 and 6:

```
>  assume(t>3,t<6);
>  simplify(sum('B(i,k,t)','i'=0..(m-k)}:
```

$$1$$

This should work under the assumption that $3 \le t \le 6$ (or at least $3 \le t < 6$ in view of the sloppiness commented on earlier) but Maple takes additional nudging to get that totally right. In general, the correct subinterval over which to sum the $m - k + 1$ B-splines is $[t_{k-1}, t_{m-k+1}]$.

B-spline Curves and Surfaces

We have seen how Bernstein functions are special cases of B-splines, and we already know how to build parametric curves from the Bernstein functions. It should therefore come as no surprise that general B-splines can be made to play a similar role.

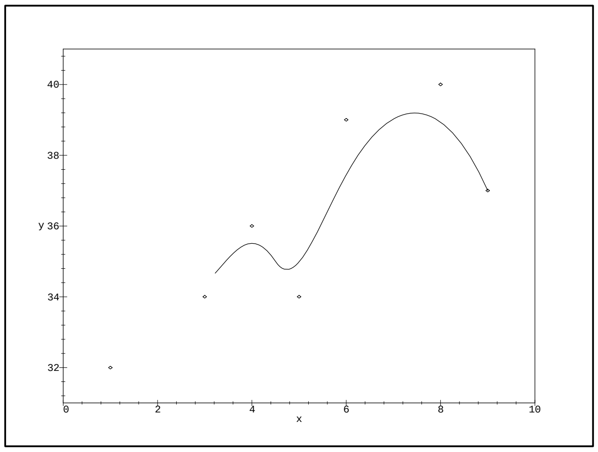
Given a knot vector $T = [t_0, t_1, \ldots, t_m]$, and the corresponding $m - k + 1$ B-splines $B_{i,k}(t)$ of order k, for some fixed $k \le m$, then if we also have $m - k + 1$ control points $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_{m-k}$, we can put everything together to get a *B-spline curve of degree* $k - 1$ *defined on* $[t_{k-1}, t_{m-k+1}]$:

$$\mathbf{C}(t) = \sum_{i=0}^{m-k+1} B_{i,k}(t)\mathbf{P}_i$$

If $T = [0, 0, 0, 0, 1, 1, 1, 1]$ and $k = 4$, we see that $\mathbf{C}(t)$ is just a cubic Bezier curve; indeed any Bezier curve can be realized in a similar way.

Let's look at a cubic B-spline curve based on the knot vector $[0, 1, 1, 3, 4, 4, 5, 6, 6, 6, 9]$ from before. Since we'll need $m - k + 1 = 10 - 4 + 1 = 7$ control points, we use our seven data points again.

```
>  bsp:=evalm(sum('B(i,k,t)*[P.i]',
>                     'i'=0..(m-k) )):
>  BSP1:=plot([ bsp[1], bsp[2],
>                t = T[k-1]..T[m-k+1] ]):
>  display({POINTS,BSP1});
```



This picture raises further questions. Why is $(9, 37)$ interpolated? How smooth is this piecewise cubic curve? Where, on the curve itself, are the various segments joined up? How do the knots and the control points influence the shape of the curve? Could we change the knots so that $(1, 32)$ doesn't get "left out in the cold"? Is there a way to actually interpolate all seven points with a piecewise cubic B-spline curve? These are important questions, and we take a look at *some* of them in the last section, **An Educational Experience**.

Maple has built in `spline` and `bspline` libraries, containing very general routines, which even allow for symbolic knots. However, we do not explore them here, in keeping with our philosophy of using a minimal amout of Maple, while trying to understand how the mathematical pieces fit together.

We finish this section by generalizing the tensor product bicubic Bezier surface patch definition from earlier, to get B-spline surface patches.

We start with knot vectors $U = [u_0, u_1, \ldots, u_{m_u}]$, $V = [v_0, v_1, \ldots, v_{m_v}]$, of possibly different lengths, and two curve orders k_u, k_v. Given a rectangular mesh of $(m_u - k_u + 1) \times (m_v - k_v + 1)$ control points

$\mathbf{P}_{i,j}$, we can then define

$$\mathbf{S}(u,v)=\sum_{i=0}^{m_u-k_u+1}\sum_{j=0}^{m_v-k_v+1}B_{i,k_u}(u)B_{j,k_v}(v)\mathbf{P}_{i,j}$$

for $(u,v)\in[u_{k_u-1},u_{m_u-k_u+1}]\times[v_{k_v-1},v_{m_v-k_v+1}]$.

When both knots vectors are $[0,0,0,0,1,1,1,1]$, and both orders are 4, we get a bicubic Bezier surface patch as seen before.

It should now be fairly obvious how to generate more exotic examples of surface patches using Maple.

Non-Uniform Rational B-Splines

Non-uniform rational B-Splines (NURBS) refers to curve and surface generation using not-necessarily uniform (equally spaced) knots, in conjunction with the rational function approach already encountered in our study of Bezier curves. This leads to increased flexibility in design. We give one example: the representation of an entire circle as a rational quadratic B-spline curve.

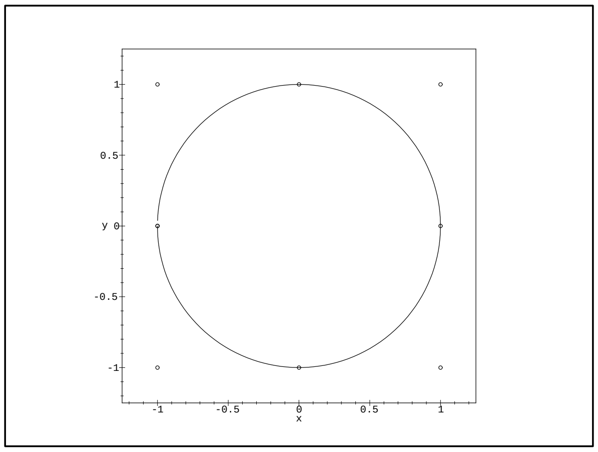
Our goal is to plot a rational B-spline curve:

$$\mathbf{R}(t)=\frac{\sum_{i=0}^{m-k+1}w_iB_{i,k}(t)\mathbf{P}_i}{\sum_{i=0}^{m-k+1}w_iB_{i,k}(t)},$$

given weights w_i and control points $\mathbf{P}_i$. We use the knot vector $[0,0,0,1,1,2,2,3,3,4,4,4]$ of length 12, set $k=3$, and get the recursively defined $B(i,k,t)$'s ready for action.

For our control points we choose $m-k+1=11-3+1=9$ points around the perimeter of a square centered at the origin, and plot these with a little room to spare. Note that $\mathbf{P}_0$ and $\mathbf{P}_8$ are the same. For the weights, we attach 1 at the five(!) corners of the square, and $\sin(\pi/4)=\frac{1}{\sqrt{2}}$ elsewhere.

```
>  P.0:=(-1, 0): P.1:=(-1,-1): P.2:=( 0,-1):
>  P.3:=( 1,-1): P.4:=( 1, 0): P.5:=( 1, 1):
>  P.6:=( 0, 1): P.7:=(-1, 1): P.8:=(-1, 0):
>  sq:=[P.(0..(m-k)]:  s:=1/sqrt(2):
>  SQ:=plot(sq, style=point, symbol=circle,
>      x=-1.25..1.25, y=-1.25..1.25,
>      scaling=constrained, axes=boxed):
>  ww:=array(0..8, [1,s,1,s,1,s,1,s,1]):
>  div:=sum('ww[i]*B(i,k,t)', 'i'=0..(m-k)):
>  circ:=evalm(sum('ww[i]*B(i,k,t)*[P.i]/div',
>                  'i'=0..(m-k) )):
>  CIRC:=plot([circ[1],circ[2],
>          t=T[k-1]..T[m-k+1]],numpoints=200):
>  display({SQ,CIRC});
```



We upped `numpoints` to 200, from the default 50, in an attempt to mask the annoying gap above $(-1,0)$. Our curve is composed of four quarter circles—each of which is a rational quadratic Bezier curve. One advantage of the NURBS approach is clear: we get the entire circle as a single curve.

An Educational Experience

Our final explorations bring out some key properties of B-spline curves, as well as addressing questions such as "where are the knots on the curve?", "how do the knots influence the curve?" and "how do the control points influence the curve?"

We look at a new, quadratic B-spline curve, and mark in the knots, so that we can see the different polynomial segments which make up the curve.

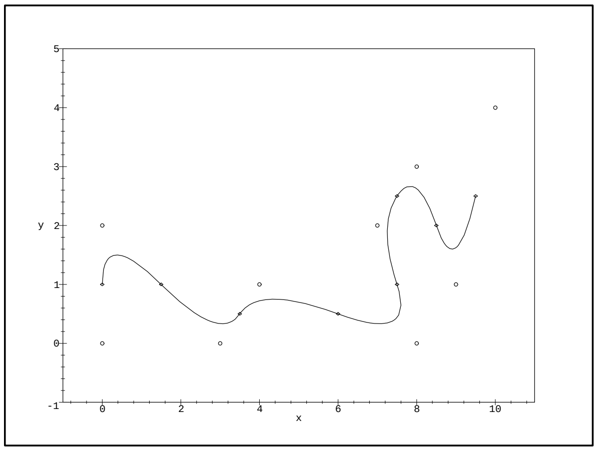
```
>  m:=11:  T:=array(0..m,
>               [0,1,2,3,4,5,6,7,8,9,10,11]):
```

Execute the `w:=(i,j,t)->`, `B:=(i,j,t)->` code, and pick $m-k+1=11-3+1=9$ control points.

```
>  P.0:=(0,0): P.1:=(0,2): P.2:=(3,0):
>  P.3:=(4,1): P.4:=(8,0): P.5:=(7,2):
>  P.6:=(8,3): P.7:=(9,1): P.8:=(10,4):
>  k:=3:  newpts:=[P.(0..(m-k))]:
>  NEWPTS:=plot(newpts, style=point,
>         x=-1..11, y=-1..5,
>         symbol=circle, axes=boxed):
```

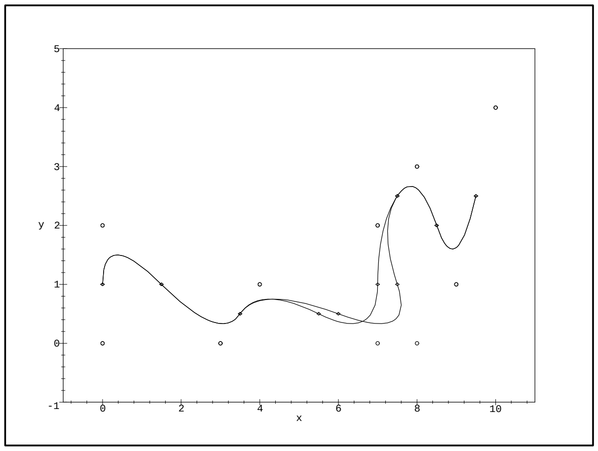
We plot on $[t_{k-1},t_{m-k+1}]$, in which there are $(m-k+1)-(k-1)+1=m-2k+3$ knots, i.e., the 8 knots from $t_2=2$ to $t_9=9$ inclusive. We evaluate the B-spline curve at these, and mark in the corresponding points.

```
> edu:=evalm(sum('B(i,k,t)*[P.i]',
>                  'i'=0..(m-k) ):
> EDU:=plot( [edu[1],edu[2],
>            t=T[k-1]..T[m-k+1]] ):
> knotsx:=seq( eval(subs(t=T[i],edu[1])),
>                i=(k-1)..(m-k+1) ):
> knotsy:=seq( eval(subs(t=T[i],edu[2])),
>                i=(k-1)..(m-k+1) ):
> klist:=j->(knotsx[j],knotsy[j]):
> knots:=[seq(klist(j),j=i..(m-2*k+3))]:
> KNOTS:=plot(knots,style=point,symbol=box):
> display({NEWPTS,KNOTS,EDU});
```



The curve appears to interpolate the midpoints of the lines joining consecutive control points, and these lines seem to be tangent to the curve there. These observations hold for any quadratic B-spline curve derived from a uniform knot vector.

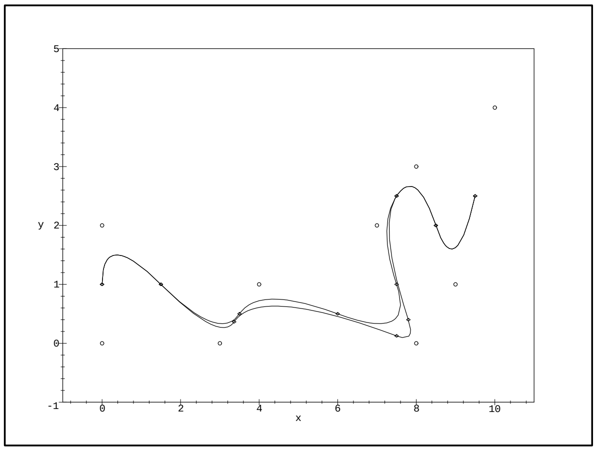
We now alter one of the control points, say $\mathbf{P}_4$, from $(8,0)$ to $(7,0)$. What changes? We combine the new and the old plots using `display`.



Evidently, altering one control point results in changes in three curve segments. In general, each control point on a B-spline curve of order k effects at most k segments of the curve: $\mathbf{P}_i$ influences only the segments between t_i and t_{i+k} [4, p. 176]. This important property of B-spline curves is known as "local control," and is one reason they are used so much in design: curve modifications can be made locally without changing other parts already deemed satisfactory.

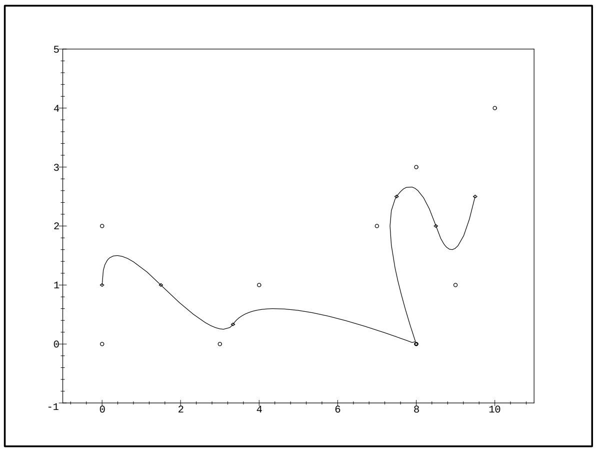
A complimentary fact is that each segment of such curves is controlled by at most k of the $\mathbf{P}_i$'s: the segment between t_j and t_{j+1} is completely determined by $\mathbf{P}_{j-k+1}, \ldots, \mathbf{P}_j$ [4, p. 176].

Next, we entertain the possibility of fiddling with the knots. Looking back at the original, before we altered a control point, we see that the knots $t = 5$ and $t = 6$ correspond to points marked on the curve just before and just after it swings by the control point $(8,0)$, going from left to right. Now we change $t_5 = 5$ to 5.75, and see what happens.



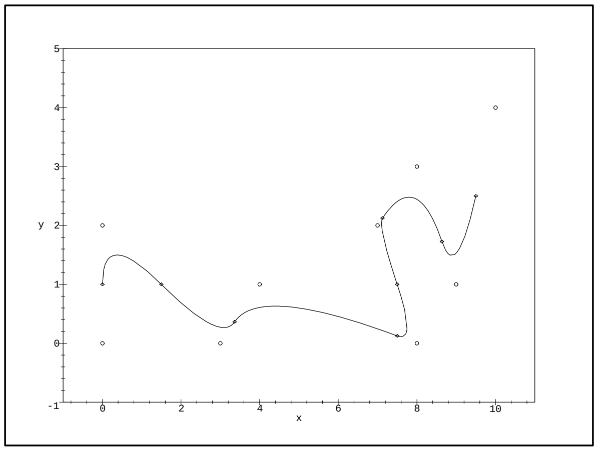
Only two curve segments on either side of the altered knot changed! Again, there is a general principle at work here: when we alter one knot in a B-spline curve of order k, we change at most $k-1$ segments of the curve on either side of that knot.

Another thing worth noting here: the two knots which are closer together seem to be dragging the curve towards the control point $(8,0)$. If this is so, we can surely guess what happens when we change the knot vector to $T=[0,1,2,3,4,6,6,7,8,9,10,11]$.



The double knot at $t=6$ resulted in interpolation of one of the control points. Furthermore, the curve has $k-\ell-1=3-2-1=0$ continuous derivatives at this point, as the picture suggests.

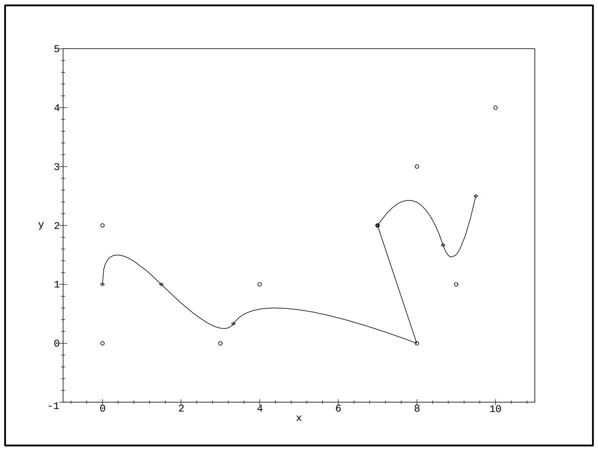
What effect would a triple knot have? First, let's have two knots sneak up on $t=6$ from different sides: try $T=[0,1,2,3,4,5.75,6,6.25,8,9,10,11]$.



The curve seems to be straining to interpolate two control points, with $t=6$ torn between the two points.

What happens when we push all the way, and use $T=[0,1,2,3,4,6,6,6,8,9,10,11]$?

Our final plot suggests that $(8,0)$ and $(7,2)$ are both interpolated—but if this is true, which point on the curve now corresponds to $t=6$?



The picture is in fact wrong: in reality the curve jumps from just before $(8,0)$ to $(7,2)$, as t reaches 6 from the left. It's hard to see at this scale, but $(7,2)$ is marked as a knot (square) and a control point (circle), whereas $(8,0)$ is only marked as a control point. There is actually a whopping big discontinuity here!

Maple shouldn't have joined these two control points with a straight line, but then Maple's default way of plotting is to find a bunch of correct points and then join the dots, effectively assuming continuity.

In spite of all the danger of believing the first computer generated picture we see, all is not lost: the truth of the matter is revealed upon changing the `plot[style]` option to `points`. Maple makes a graceful recovery given half a chance.

Conclusion

We have shown how Maple can be of assistance in exploring the principal geometric objects of study in CAGD (Computer Aided Geometric Design).

Concepts from just outside the standard undergraduate mathematics curriculum, together with a little Maple, go a long way in the construction of the shapes which are fundamental to many engineering design processes.

We have only scratched the surface here, and we believe that there is a great deal of potential for further Maple exploration in this area.

Acknowledgments

The author acknowledges the generous support of *Boeing Computer Services*, in the planning and running of an undergraduate seminar on CAGD for mathematics majors at Spelman College over the academic years 1993-5. It was in this forum that the bulk of the current materials was developed. In particular, Dr. Dave Ferguson of BCS was always happy to share his expertise at the drop of a hat.

The author would like to thank his Spelman colleague Dr. Sylvia Bozeman for unwittingly starting him down this road, Dr. Jeffrey Ehme for many helpful discussions, and the numerous seminar participants for their patience and perseverance while we grappled with these concepts together.

This manuscript was made possible thanks to support from the *W.K. Kellogg Foundation*, through Spelman College's Center for Scientific Applications of Mathematics (CSAM).

The referees made sensible suggestions which lead to significant strengthenings of this paper, for which the author is most grateful.

P.S. Editor Ross Taylor provided much appreciated PostScript advice in the assembly process.

References

[1] Richard Bartels, John Beatty & Brian Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling* Morgan Kaufman (1987) [also published in France as *Mathématiques et Cao, Volume 6, B-Splines* and *Mathématiques et Cao, Volume 7, Beta-Splines*, Hermès (1988), translated by Pierre Bézier]

[2] Carl de Boor, *B(asic)-Spline Basics* in *Fundamental Developments Of Computer-Aided Geometric Modeling* (Les Piegl, editor), Academic Press (1993)

[3] Gerald Farin, *Curves And Surfaces For CAGD (A Practical Guide)*, Academic Press, 3rd Edition (1993)

[4] Josef Hoschek & Dieter Lasser, *(Fundamentals of) Computer Aided Geometric Design*, A.K. Peters (1993).

Biographical Sketch

The author earned the B.Sc. and M.Sc. degrees in Mathematics from University College Dublin, in Ireland. His Ph.D. thesis, written both east and west of there under the direction and guidance of Alex F.T.W. Rosenberg of Cornell University, concerned an abstract setting for the algebraic reduced theory of higher level forms over fields.

Recently his mathematical interests have broadened to include computational algebra (Groebner bases), general algebraic codology, CAGD, image processing, computer graphics, and wavelets. He likes Matlab and music, though not necessarily in that order.

He actively encourages culinary minded readers to email him reliable, authentically Thai recipes for Massaman Curry.