>

Drawing a Bezier Cubic Spline Through Four Data Points

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2005 as GuruGram #59.
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

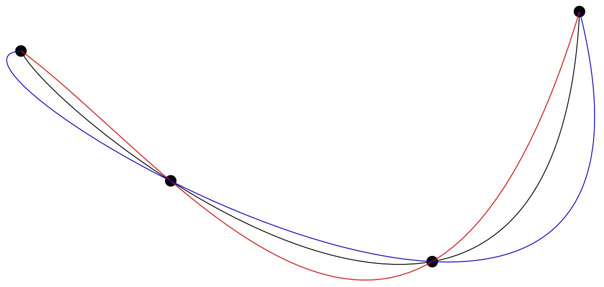
Bezier **Cubic Splines** are an excellent and preferred method to draw the smooth continuous curves often found in typography, **CAD/CAM**, and graphics in general.

Among their many advantages is a **very sparse data set** allowing a mere **eight** values (or four **x,y** points) to **completely** define a full and carefully controlled and device independent curve. Many tutorials and examples are now present in our **Cubic Spline** Library. A brief and useful intro **appears here**. The fundamental math behind Cubic Splines **appears here**.

A normal and typical Bezier cubic spline accepts four data points of **x0,y0**, **x1,y1**, **x2,y2**, and **x3,y3**. It turns out that **x0,y0** and **x3,y3** define the **end points** of the curve, while **x1,y1** and **x2,y2** establish the **initial and final slope** and the "enthusiasm" or "tension" with which the curve enters or leaves the final points. These interior control points are normally distant from the actual final curve.

We might instead like to fit a Bezier Cubic Spline to four data points, all of which are **on** the curve. New point variables of **x4,y4**, and **x5,y5** might be introduced as these **on-curve points**. The underlying math problem would then be to relate or **transform** the on-curve points **x4,y4**, and **x5,y5** to the standard control points of **x1,y1**, and **x2,y2**.

It turns out you can draw an infinite number of cubic spline curves through four data points, depending upon your choice of **t1** and **t2** for your intermediate point locations...



While the black curve looks "best", the red one and the blue one are both "too loopey" in some areas and "too straight" in others. The dilemma is to pick the "best" or "most aesthetic" solution. While such terms are hard to quantify, they most likely would take place with the **shortest** possible cubic spline curve. Such a curve would be the "most efficient" as well.

Optimizing the length of a cubic spline might involve some horrendous math and repeated converging approximations. Instead, we will use a **apportioned chords** approximation that is far simpler and seems to give acceptable results...

APPORTIONED CHORDS —

An approximation to the "best" cubic spline four point curve fit. Straight lines C1, C2, and C3 are drawn between the points and their lenthts are calculated.

The t values for the inside points are then calculated as t1 = C1/(C1 + C2 + C3) and as t2 = (C1 + C2)/(C1 + C2 + C3).

Some Utility Code

A greatly improved replacement for our earlier four-point code can be found as **IMBZ4P01.PSL**. As with most of our utilities, this is written in raw **PostScript**, makes optional use of my **Gonzo Utilities**, and is used to create **standard ASCII text files** sent to **Acrobat Distiller**. With the latter acting as a **General Purpose Host Based PostScript Interpreter**.

What follows can be best understood by having **IMBZ4P01.PSL** viewable as an open window. **Basis Functions** can enormously simplify use and understanding of cubic splines. Any point on the curve can be expressed as...

$$x(t) = x_0B0(t) + x_1B1(t) + x_2B2(t) + x_3B3(t)$$

Separating our knowns and our unknowns...

$$x4 - x_0B0(t1) - x_3B3(t1) = x_1B1(t1) + x_2B2(t1)$$

$$x5 - x_0B0(t2) - x_3B3(t2) = x_1B1(t2) + x_2B2(t2)$$

Or, in English, "The **x4** value we need at **t1** is made from known contributions of **x0** and **x3** and as yet unknown contributions of **x1** and **x2**". Similarly "The **x5** value we need at **t2** is made from known contributions of **x0** and **x3** and as yet unknown contributions of **x1** and **x2**". We thus have two plain old algebraic equations in two unknowns of **x0** and **x1**.

A similar set of two equations can be written and solved for the y values. Our first order of business should be creating an equation solver sub-utility...

```
% Linear equation solver utility for ai + bj = c and di + ej = f

/solvexy {/ff exch store                  % grab data values
          /ee exch store
          /dd exch store
          /cc exch store
          /bb exch store
          /aa exch store

          cc aa dd div ff mul sub         % find j
          bb aa ee mul dd div sub  div
          /jj exch store

          /ii cc bb jj mul sub aa div store  % find i

          ii jj } store                   % return to stack
```

This works by scaling the second equation by **a/d** and subtracting it from the first one to produce variable **j**. Variable **i** is then found by back substitution.

We can now start our actual 4-point plotting code...

```
/bez4pts1 {/y3 exch store                 % grab data
        /x3 exch store
        /y5 exch store                    % strange numbering
        /x5 exch store
        /y4 exch store
        /x4 exch store
        /y0 exch store
        /x0 exch store

        /c1 x4 x0 sub dup mul  y4 y0      % find chord lengths
        sub dup mul add sqrt store
        /c2 x5 x4 sub dup mul  y5 y4
        sub dup mul add sqrt store
        /c3 x3 x5 sub dup mul  y3 y5
        sub dup mul add sqrt store
```

The chords are simply the **vector** sum of the x axis and y axis differences between the data points. Continuing…

```
    /t1 c1 dup c2 add c3 add div store   % guess "best" t
    /t2 c1 c2 add dup c3 add div store

    /b0 {1 exch sub dup dup mul mul} store  % basis functions
    /b1 {dup 1 exch sub dup mul mul 3 mul} store
    /b2 {dup 1 exch sub exch dup mul mul 3 mul} store
    /b3 {dup dup mul mul} store

    t1 b1 t1 b2 x4 x0 t1 b0 mul sub      % transform x1 and x2
    x3 t1 b3 mul sub t2 b1 t2 b2 x5
    x0 t2 b0 mul sub x3 t2 b3 mul sub
    solvexy /x2 exch store /x1 exch store

    t1 b1 t1 b2 y4 y0 t1 b0 mul sub      % transform  y1 and y2
    y3 t1 b3 mul sub t2 b1 t2 b2 y5
    y0 t2 b0 mul sub y3 t2 b3 mul  sub
    solvexy  /y2 exch store /y1 exch store

    x0 y0 moveto                        % and draw the curve
    x1 y1 x2 y2 x3 y3 curveto
  } def
```

t1 and **t2** are found by chord apportioning. The basis functions are standard definitions per **this tutorial**. The transforms to get from the on-curve points to the off curve control points may look a little obtuse, but they are nothing but the equations in the last aqua box above. Once the two linear equations in two unknowns are found, they are sent to the equation solver sub-utility. First to find **x1** and **x2**, and second to find **y1** and **y2**.

Finally, a plain old **PostScript curveto** is used to generate the actual Bezier Cubic Spline through four points.

While chord apportionment seems "good enough" for most uses, you can further optimize for shortest spline by subdividing the chords or going to multi-pass schemes. **IMBZ4P01.PSL** has been updated with a new example and demo.

For More Help

Additional info on cubic splines can be found on our **Cubic Spline** library page. As are many dozens of examples of Bezier cubic spline techniques.

Additional consulting services are available per our **Infopack** services and on a contract or an hourly basis. Additional **GuruGrams** are found **here**.

Further **GuruGrams** await your ongoing support as a **Synergetics Partner**.