Joining the dots

Mike Liardet **introduces the handy technique of cubic splines for drawing accurate and smooth curves through a series of points.**

We have all seen the well-known children's puzzle that involves joining up the dots in a picture in order to uncover an underlying drawing. There is a computer equivalent to this puzzle which requires a sequence of points, plotted in a display area, to be connected by a smooth line. Although this may sound a somewhat esoteric problem, it is surprising that there are several well-known software applications that require an efficient solution to it.

Look no further than Windows itself. The handling of scalable fonts involves the manipulation of a sequence of points which define the shape of the character — a smooth curve is required through these points for all possible character sizes. Computer Aided Design needs the odd carefully crafted line as well. Most CAD applications allow the user to define areas or other arbitrary shapes by a sequence of plotted points, which must then be connected by an appropriate curve. Charting software also needs to offer the option of drawing a smooth line through some points in a graph, alongside the other options for pie charts, bar graphs and so on.

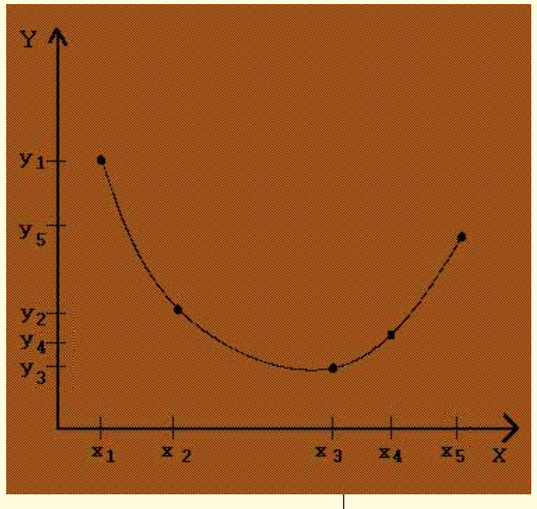


***Fig 1** A smooth curve drawn through five points. Here the points are neatly ordered and evenly spaced*

In this month's column we will introduce a powerful technique, called cubic splines, which is used for drawing extremely accurate and smooth curves through a sequence of points.

To get the flavour of the problem area, *Fig 1* shows the output for the fairly simple task of drawing a curve through just five points. In this case, the five points are neatly ordered so that the x coordinate of each is greater than its predecessor. The points are also fairly evenly spaced, and this makes for a simpler problem than if the points are all over the place.

Points and polynomials

Readers with some basic maths training will know that, in theory, given a sequence of N points we can devise an equation involving an N-1 degree polynomial. When this is drawn as a graph, it passes through each of the points. (A polynomial is an expression in the form

$a_1 + a_2x^1 \ldots + a_Nx^{N-1}$

For more information see last month's Low Level.)

For our five-point problem this would mean working with the equation

$y = a_1 + a_2x^1 + a_3x^2 + a_4x^3 + a_5x^4$

and attempting to find values for

a_1, a_2, a_3, a_4 and a_5

that enable it to go through each point. In this case it is not too difficult: we plug the values for the (x_1,y_1) coordinates for x and y in the equation and end up with one linear equation with the five "a" unknowns. The remaining four points provide four more linear equations, and, as we saw last month's Low Level, there is a simple method (Gaussian elimination) that can solve these equations directly.

Unfortunately, this technique does not work very well when applied to a large number of points. Firstly, the Gaussian elimination part of the method slows down markedly as it is asked to handle more than a handful of unknowns, and also the resulting polynomials can be difficult to compute. For example, a polynomial equation that can thread its way through 100 points will contain a sub-expression x^{99}. Even for small values of x an attempt to calculate this will cause an arithmetic overflow error in most programming languages.

The basic idea behind cubic splines is to dispense with the one large unwieldy polynomial that goes through all the points, and replace it with a number of simpler polynomials that are individually responsible for drawing a line only between an adjacent pair of points. Not surprisingly, given the name of the technique, we use cubic equations. The word "spline", by the way, is the name of a flexible draughting tool which is often used for solving the problem by hand.

To apply the cubic spline technique to the problem given in *Fig 1* we need to devise four cubic equations to handle each of the four segments, as shown in *Fig 2*. In order to draw the spline it is first necessary to calculate the values of the a,

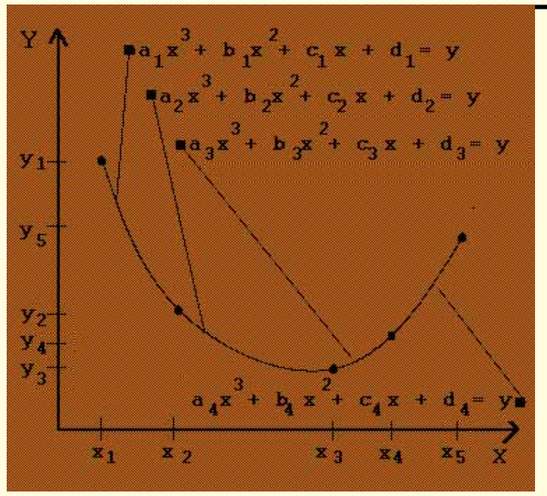
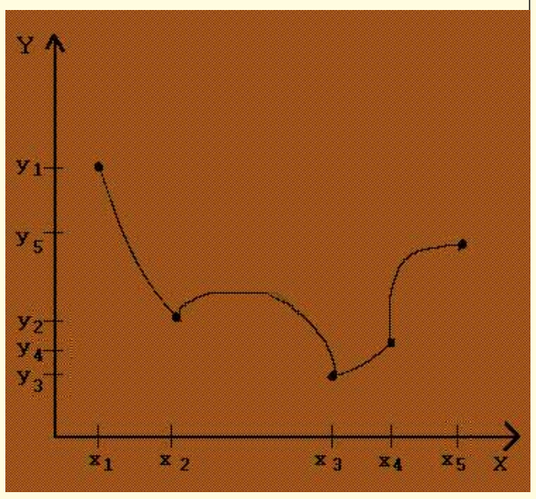


Fig 2 (left) *The cubic spline method uses four cubic polynomial equations to draw the curve; each equation handles its own segment of the curve*

Fig 3 (below) *If the slope of adjacent lines are not equal at the meeting point, each segment is joined by a smooth line, but there are discontinuities where they meet*



b, c and d coefficients in each equation — 16 unknowns in total. Looking at each cubic polynomial in turn, we can see that in each case it must pass through two points, its start point and end point. This gives us two equations for each polynomial and eight equations in all. But we need 16 equations to solve for the 16 unknown, so clearly this is not enough. The equations are under-constrained and we need to come up with some extra conditions in order to solve them.

It is not unreasonable to ask that the slope of the line at the end of a segment should be the same as the slope of the next line at the beginning. Without this requirement we could have two successive lines meeting at a point in such a way that the end result does not look smooth at all *(Fig 3)*. With the five-point problem, setting slopes equal for adjacent polynomials gives us another three equations, and 11 equations in total, but this is still not enough.

See the join?

We can further constrain the problem by attempting to make the join between adjacent segments even smoother. We do this by demanding that the rate of change of slopes (known as the second derivatives) are also equal. This provides another three equations — 14 in total, but still two short. We obtain the last two equations by adding the simple requirement that the second derivatives at the first and last points are zero.

Of course, we want to be able to apply the cubic spline technique to any number of points, and not just the five we have been looking at so far. *Fig 4* shows how the equations can be set up for a problem involving N points. Readers unfamiliar with elementary calculus can take it on trust that the slope of a cubic polynomial equation

$a x^3 + b x^2 + c x + d = y$

is given by

$3 a x^2 + 2 b x + c = y'$

This is known as the first derivative. The second derivative, which is also used in these equations, is given by

$6 a x + 2 b = y''$

The unknowns in these equations are the “a”s, “b”s, “c”s and “d”s. All the “x”s and “y”s are known, as these are given by the points which are to be joined by the spline. All the equations are linear and so in theory we could solve them by using the usual Gaussian elimination. Once the values of the unknowns are determined it is fairly easy to draw the curve as a sequence of line segments, with each line created by its own polynomial equation.

For a reasonable-size problem Gaussian elimination is too slow. For example, a cubic spline joining 26 points would require the solution of over 100 equations with over 100 unknowns, a task which would take a considerable time on most computers.

Fortunately, with the aid of a bit of equation manipulation it is possible to make the equation solution process a lot quicker. Part of the technique involves the introduction of some new unknowns, p_1 to p_N, which represent the second derivatives at each point. Notice that, once these “p” values are known, all the other unknowns can be worked out almost immediately. We already know p_1 and p_N (they are both 0) and *Fig 5* shows how we can derive a system of equations for the other “p”s. A close inspection of the last equation in *Fig 5* reveals that we have a tri-diagonal system, and although these are linear equations which could be solved by the Gauss technique, there is a much faster method for tri-diagonal equations, which was given in last month’s Low Level.

Visual Basic in practice

Having got the theory out of the way, it is fairly easy to develop a Visual Basic program that can draw cubic splines *(Fig 6)*. Interaction with the program is very simple. The points to be joined are given by clicking with the mouse in the large picture box. A small dot is plotted to show the

Fig 4

The complete set of equations used to find the cubic spline coefficients

The spline i passes through the points (x_i,y_i) and (x_{i+1},y_{i+1})...

$a_i x_i^3 + b_i x_i^2 + c_i x_i + d_i = y_i$ (1) (for i = 1..n-1)

$a_i x_{i+1}^3 + b_i x_{i+1}^2 + c_i x_{i+1} + d_i = y_{i+1}$ (2) (for i = 1..n-1)

The second derivatives of the i-1 and ith splines are equal at the point (x_i,y_i). With the second derivative at (x_i,y_i) denoted as p_i we have...

$6 a_i x_i + 2 b_i = p_i$ (3) (for i = 1..n-1)

$6 a_i x_{i+1} + 2 b_i = p_{i+1}$ (4) (for i = 1..n-1)

The first derivatives of the i-1and ith splines are equal at the point (x_i,y_i)...

$3 a_{i-1} x_i^2 + 2 b_{i-1} x_i + c_{i-1} = 3 a_i x_i^2 + 2 b_i x_i + c_i$ (5) (for i = 2..n-1)

The unknowns are $a_1..a_{n-1}$, $b_1..b_{n-1}$, $c_1..c_{n-1}$, $d_1..d_{n-1}$, $p_1..p_n$ ie there are 5n-4 in total. (1) to (5) give us 5n-6 equations, so two more equations are needed to solve (1) to (5). For example we can specify that the second derivative at the first and last points is zero...

$0 = p_1$ (6)

$0 = p_n$ (7)

Fig 5

Solving for the “p”s.The preparation work done here derives an equation where the only unknowns are “p”s. It is easy to program a fast and efficient solution to this equation

We will eliminate a_i, b_i, c_i and d_i from these equations. First eliminate d_i immediately by replacing (1) and (2) with (A) = (2) - (1)

$a_i(x_{i+1}^3 - x_i^3) + b_i(x_{i+1}^2 - x_i^2) + c_i(x_{i+1} - x_i) = y_{i+1} - y_i$ (A) (for i = 1 .. n-1)

Solve for a_i and b_i using (3) and (4). To solve for a_i we use (4) - (3) ...

$6 a_i(x_{i+1} - x_i) = p_{i+1} - p_i$ ...

$a_i = (p_{i+1} - p_i) / 6 (x_{i+1} - x_i)$ (for i = 1 .. n-1)

To solve for b_i we use x_{i+1}(3) - x_i(4) ...

$2 b_i x_{i+1} - 2 b_i x_i = p_i x_{i+1} - p_{i+1} x_i$ ...

$b_i = (p_i x_{i+1} - p_{i+1} x_i) / 2 (x_{i+1} - x_i)$ (for i = 1 .. n-1)

We can now solve for c_i by substituting for a_i and b_i in (A) ...

$(x_{i+1}^3 - x_i^3)(p_{i+1} - p_i) / 6 (x_{i+1} - x_i) +$

$(x_{i+1}^2 - x_i^2)(p_i x_{i+1} - p_{i+1} x_i) / 2 (x_{i+1} - x_i) +$

$(x_{i+1} - x_i) c_i = y_{i+1} - y_i$.

Using identities $r^3-s^3 = (r - s)(r^2 + rs + s^2)$ and $r^2 - s^2 = (r - s)(r + s))$ we obtain...

$(x_{i+1}^2 + x_{i+1} x_i + x_i^2)(p_{i+1} - p_i) / 6 +$

$(x_{i+1} + x_i)(p_i x_{i+1} - p_{i+1} x_i) / 2 +$

$(x_{i+1} - x_i) c_i = y_{i+1} - y_i$

Simplifying and rearranging we eventually get...

$c_i = (y_{i+1} - y_i) / (x_{i+1} - x_i) +$

$(- x_{i+1}^2 + 2 x_{i+1} x_i + 2 x_i^2) p_{i+1} / 6 (x_{i+1} - x_i) +$

$(- 2 x_{i+1}^2 - 2 x_{i+1} x_i + x_i^2) p_i / 6 (x_{i+1} - x_i)$ (for i = 1 .. n-1)

We can now substitute for a_i, a_{i-1}, b_i, b_{i-1}, c_i and c_{i-1} in (5). Working with left and right hand sides of (5) separately...

RHS (5) = $3 a_i x_i^2 + 2 b_i x_i + c_i$ = (after substitutions and much rearrangement) =

$(y_{i+1} - y_i) / (x_{i+1} - x_i) + (- x_{i+1} + x_i) p_{i+1} / 6 + (- x_{i+1} + x_i) p_i / 3$

LHS (5) = $3 a_{i-1} x_i^2 + 2 b_{i-1} x_i + c_{i-1}$ = (after substitutions and much rearrangement) =

$(y_i - y_{i-1}) / (x_i - x_{i-1}) + (x_i - x_{i-1}) p_i / 3 + (x_i - x_{i-1}) p_{i-1} / 6$

Now LHS (5) = RHS (5) so...

$(y_i - y_{i-1}) / (x_i - x_{i-1}) + (x_i - x_{i-1}) p_i / 3 + (x_i - x_{i-1}) p_{i-1} / 6 =$

$(y_{i+1} - y_i) / (x_{i+1} - x_i) + (- x_{i+1} + x_i) p_{i+1} / 6 + (- x_{i+1} + x_i) p_i / 3$

After further rearrangement.....

$p_{i-1} (x_i - x_{i-1}) (x_{i+1} - x_i) / 2 (x_{i+1} - x_{i-1}) + (x_{i+1} - x_i) p_i / 6 +$

$(x_{i+1} - x_i) p_{i+1} (x_{i+1} - x_i) / 12 (x_{i+1} - x_{i-1}) =$

$[(y_{i+1} - y_i) / (x_{i+1} - x_i) - (y_i - y_{i-1}) / (x_i - x_{i-1})] (x_{i+1} - x_i) / 2 (x_{i+1} - x_{i-1})$

position of each point. There are a couple of simple editing command buttons which can be used to clear the picture box or remove the last point entered, but the meat of the program lies in the spline-plotting commands — “Spline 1”, “ Spline 2” and “Spline - Gauss”.

The routines that underly two of these buttons are shown in *Fig 7*. “Spline 1” solves the simpler problem, where the points are given in the correct X order (there is no validation in it so it crashes if you try to run it with unordered points). Given the coordinates of the points, it first calls the routine MakeSpline. In effect, this uses the tri-diagonal equation-solving technique on the equation derived in *Fig 5* to calculate the the “p” values. These are returned to this routine in the array ys(). Once the “p”s are known, the routine plots the spline one segment at a time, working out the “a”, “b”, “c” and “d” values for each cubic polynomial by calling the CalcABCD routine. The service routines for working out the “p”s, “a”s, “b”s, “c” and “d”s are shown in *Fig 8*.

The “Spline 2” routine is similar to “Spline 1” but it can cope with points in any order *(Fig 9)*. Instead of using one set of cubic polynomials that plot y values in terms of x , it uses two sets which plot y values in terms of “d” and x values in terms of “d”. The “d” values are obtained by summing the overall distance to reach a particular point in the plot, summing the length of the straight lines between each successive point. The accumulated distances to each of the key points are calculated at the start of the routine, using the well-known Pythagoras formula to calculate the distance between two points. Of course, distance always gets greater as each point is passed, no matter what its direction, and so each individual spline works correctly. We also know that the plotted curve will go through each of the key points, since we have the correct X values and Y values calculated at the ends of each segment.

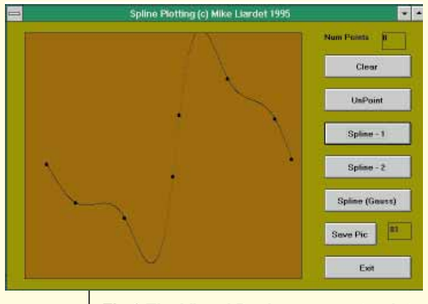


***Fig 6** The Visual Basic program to plot cubic splines through an arbitrary number of points*

Fig 7

The key subroutines that service the command buttons Spline 1 and Spline 2

```
Sub cmdSpline1_Click ()
‘Calcs and draws a spline by solving on second derivs first
‘Quick! - assumes X increasing - no error checks
Dim i As Integer
Dim AI As Single, BI As Single, CI As Single, DI As Single
Dim X As Single

    MakeSpline Val(labNumKnots), xk(), yk(), ys()
  ‘For each segment between knots...
    pic.DrawWidth = LINE_WIDTH
    For i = 1 To Val(labNumKnots) - 1
        DoEvents
      ‘Calc its cubic coeffs...
        CalcABCD i, xk(), yk(), ys(), AI, BI, CI, DI
      ‘Plot for each half pixel
        For X = xk(i) To xk(i + 1) Step .5
            pic.PSet (X, AI * X ^ 3 + BI * X ^ 2 + CI * X + DI)
            DoEvents
        Next X
    Next i
    Exit SubEnd Sub

Sub cmdSpline2_Click ()
‘Modification to Spline1 which can handle curves where x is not always
increasing
Dim A As Single, B As Single, C As Single
Dim i As Integer
Dim AX As Single, BX As Single, CX As Single, DX As Single
Dim AY As Single, BY As Single, CY As Single, DY As Single
Dim t As Single, tt As Single
Dim XX As Single, YY As Single

  ‘Calculate accumulated distances between knots
    ds(1) = 0
    For i = 2 To labNumKnots
        ds(i) = ds(i - 1) + Sqr((xk(i) - xk(i - 1)) ^ 2 + (yk(i) - yk(i - 1))
^ 2)
    Next i

  ‘Make 2nd derivs for each knot, x coords and y coords separately
    MakeSpline Val(labNumKnots), ds(), xk(), xs()
    MakeSpline Val(labNumKnots), ds(), yk(), ys()

    pic.DrawWidth = LINE_WIDTH
  ‘For each segment between knots...
    For i = 1 To labNumKnots - 1
      ‘Calc its cubic coeffs...
        CalcABCD i, ds(), xk(), xs(), AX, BX, CX, DX
        CalcABCD i, ds(), yk(), ys(), AY, BY, CY, DY
      ‘Plot for each half pixel
        For t = ds(i) To ds(i + 1) Step .5
            ‘t = tt / (ds(i + 1) - ds(i)) ‘makes t into range 0 to 1
            XX = DX + CX * t + BX * t ^ 2 + AX * t ^ 3
            YY = DY + CY * t + BY * t ^ 2 + AY * t ^ 3
            pic.PSet (XX, YY)
            DoEvents
        Next t
    Next i                                                    End Sub
```


Fig 8

Service routines for the two main command buttons

```
Sub MakeSpline (N As Integer, xp() As Single, yp() As Single, p() As Single)
‘Given N knots (xp(1),yp(1))..(xp(N),yp(N)) calculate
‘2nd derivs of cubic spline at each point, into P(1)..P(N)
Dim i As Integer
Static d(2 To MAX_KNOTS_M1) As Single
Static u(2 To MAX_KNOTS_M2) As Single
Static w(2 To MAX_KNOTS_M1) As Single


‘Set up arrays to represent the tri-diagonal matrix
  ‘d() contains values on the main diagonal...
    For i = 2 To N - 1
        d(i) = (xp(i + 1) - xp(i - 1)) / 3
    Next i
  ‘u() contains values of diagonal immediately above/below it...
    For i = 2 To N - 2
        u(i) = (xp(i + 1) - xp(i)) / 6
    Next i

‘Calc w(), the RHS values...
    For i = 2 To N - 1
        w(i) = (yp(i + 1) - yp(i)) / (xp(i + 1) - xp(i)) - (yp(i) - yp(i - 1))
/ (xp(i) - xp(i - 1))
    Next i

‘2nd deriv of first and last knot is zero
    p(1) = 0
    p(N) = 0

    For i = 2 To N - 2
        w(i + 1) = w(i + 1) - w(i) * u(i) / d(i)
        d(i + 1) = d(i + 1) - u(i) * u(i) / d(i)
    Next i

    For i = N - 1 To 2 Step -1
        p(i) = (w(i) - u(i) * p(i + 1)) / d(i)
    Next i

End Sub

Sub CalcABCD (i As Integer, xp() As Single, yp() As Single, p() As Single, A
As Single, B As Single, C As Single, d As Single)
‘Calculate A, B, C, D coeffs for cubic polynomial between knots at i to i+1,
‘given (xp(),yp()) coords of knots and p() 2nd deriv at each knot
‘There are 4 eqtns to solve..
‘First two come from the fact that cubic passes through knots at i and i+1
‘and next two from definition of p() as second deriv at these knots..
‘   A xp(i)^3   + B xp(i)^2   + C xp(i)   + D = yp(i)
‘   A xp(i+1)^3 + B xp(i+1)^2 + C xp(i+1) + D = yp(i+1)
‘ 6 A xp(i)   + 2 B                        = p(i)
‘ 6 A xp(i+1) + 2 B                         =p(i+1)
‘Use last two eqtns to solve for A and B first...
    LinEq2 6 * xp(i), 2, -p(i), 6 * xp(i + 1), 2, -p(i + 1), A, B
‘Now feed A and B into first two and solve for C and D...
    LinEq2 xp(i), 1, A * xp(i) ^ 3 + B * xp(i) ^ 2 - yp(i), xp(i + 1), 1, A *
xp(i + 1) ^ 3 + B * xp(i + 1) ^ 2 - yp(i + 1), C, d

End Sub
```

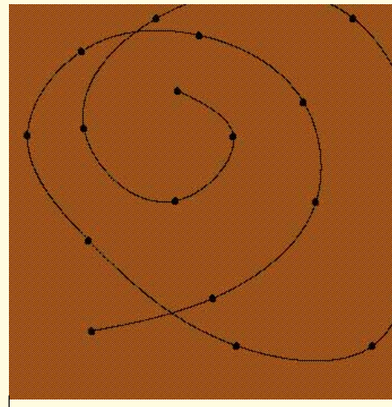


***Fig 9** An adaptation of the basic spline technique enables a curve to be plotted where the points are given in any order*

Smoothly does it

The plotting part of this routine ties the two cubic splines together, producing a smooth curve on the display. With this little dodge the points can be in any order. It also copes well with the situation where some points are very close together and others are more distant. The “Spline 1” method can be fooled under these circumstances.

Lastly, the Visual Basic program contains a “Spline - Gauss” command. This solves the *Fig 4* equations in the obvious fashion, using Gaussian elimination. This routine was developed without any need to understand the analysis we went though to derive equations for the “p” values in *Fig 5*.

The price that is paid for this simplicity is a slower solution process. The method also provides a useful check that the other Spline commands are working correctly, as the Gauss command should produce an identical curve to Spline 1 although it is calculated by a very different technique.

PCW Cover Disk

The full code for this month’s Low Level is on the cover disk given with this issue of *Personal Computer World.*

PCW Contacts

Mike Liardet is a freelance programmer and writer. He can be contacted via the *PCW* Editorial office or on email as **mliardet@cix.compulink.co.uk**