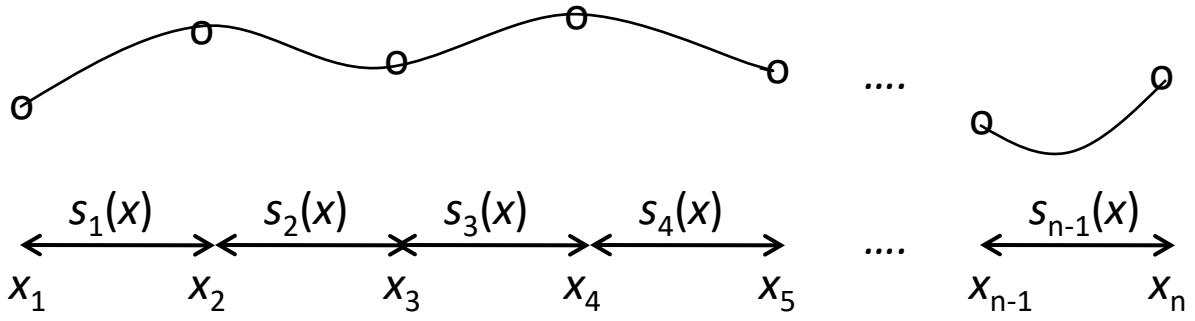
Splines:

Another form of interpolation.

Instead of using a single function (as in polynomial interpolation) separate functions are used between each pair of data points.

Name comes from flexible *splines* used in drafting to draw a smooth curve point through a series of points (though linear splines won't look much like this, cubic splines will).

Conventions: n points (x_1 through x_n), $n - 1$ intervals
$s_i(x)$ used between x_i and x_{i+1}
$y_i = y$ value at x_i (note: text uses $f(x_i)$, f_i)

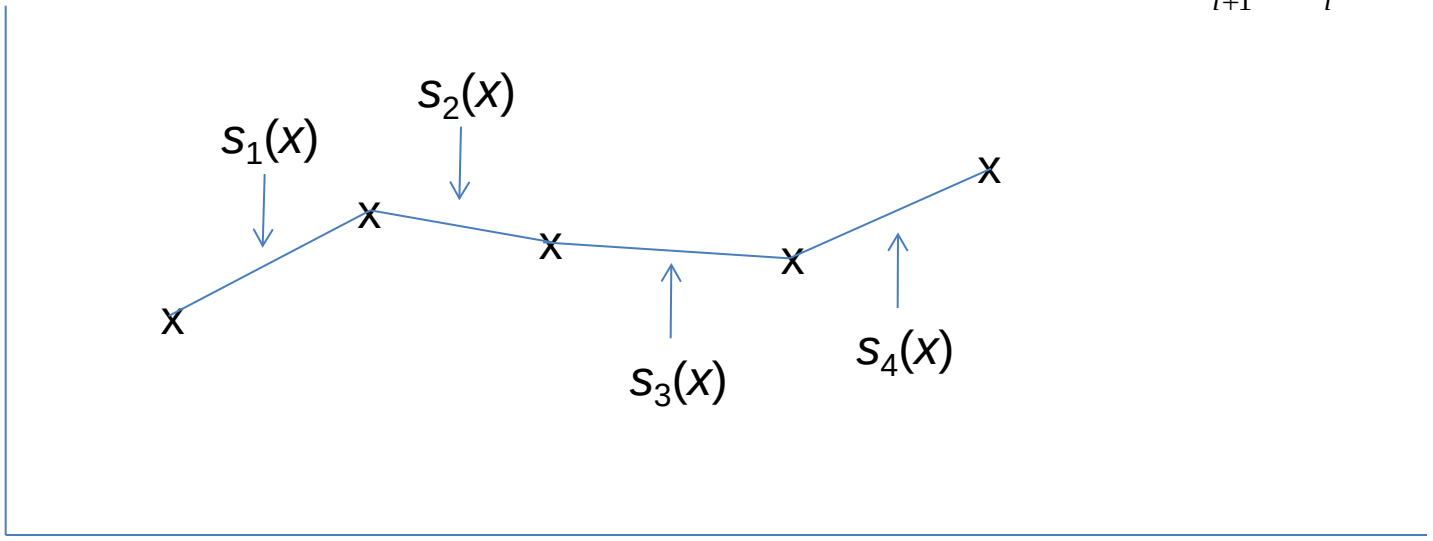


Linear Splines:

Straight lines used between data points $s_i(x) = a_i + b_i(x - x_i)$

Same thing as piecewise linear interpolation $a_i = y_i \quad b_i = \dfrac{y_{i+1} - y_i}{x_{i+1} - x_i}$

$$s_i(x) = y_i + (x - x_i)\frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$



Matlab:

yout = interp1 (x, y, xin, 'linear');
% or yout = linearSpline(x,y,xin); see linearSplineExample.m
x, y = data points
xin = x values of interest
yout = y values of linear spline at a values of interest

Each interval involves two variables (a, b)
There are therefore a total of $2(n-1)$ variables
Each $s_i(x)$ must pass through (x_i, y_i) and (x_{i+1}, y_{i+1}):

$$s_i(x_i) = y_i \quad s_i(x_{i+1}) = y_{i+1}$$

There are therefore $2(n\text{-}1)$ equations.

In practice it is not necessary to work out all of the a's and b's
The concept will make more sense with quadratic and cubic splines

In order to find the value of a linear spline for a given x it is only necessary to
i) Locate the correct interval
ii) Perform linear interpolation between the interval end points

```
function [ yout ] = linearSpline( x, y, xin )

yout = zeros(size(xin));
for k = 1 : length(xin)

   xval = xin(k); % for convenience

   lo = 1; hi = length(x);
   if xval < x(lo) || xval > x(hi); error ('x value out of range'); end;
   while hi ~= lo + 1
      m = round((lo + hi) / 2);
      if x(m) <= xval
         lo = m;
      else
         hi = m;
      end
   end

   yout(k) = y(lo) + (xval - x(lo))*((y(hi)-y(lo))/(x(hi) - x(lo))) ;

end

end
```

Quadratic Splines:

Same basic idea but with quadratics between data points

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2$$

Each interval involves three variables (a, b, c)
There are therefore a total of $3(n - 1)$ variables

Each $s_i(x)$ must pass through (x_i, y_i) and (x_{i+1}, y_{i+1}):

$$s_i(x_i) = y_i \quad s_i(x_{i+1}) = y_{i+1}$$

This gives $2(n - 1)$ equations.

We require that the first derivative be continuous at all "knots" (points at which we switch from one $s_i(x)$ to the next):

$$s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}) \quad \text{for } 1 \le i \le n - 2$$

This gives a further $n - 2$ equations.

The remaining required equation is obtained by making an arbitrary assumption (e.g. that $c_1 = 0$, which equates to saying that $s_1(x)$ will be a straight line.

Eq'n 1: $s_i(x_i) = y_i$

$$a_i + b_i(x_i - x_i) + c_i(x_i - x_i)^2 = y_i$$

$$a_i = y_i$$

Eq'n 2: $s_i(x_{i+1}) = s_{i+1}(x_{i+1})$

simplifies to

$$b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 = y_{i+1} - y_i$$

Eq'n 3: $s_i'(x_{i+1}) = s_{i+1}'(x_{i+1})$

$$b_i + 2c_i(x_{i+1} - x_i) = b_{i+1} + 2c_{i+1}(x_{i+1} - x_{i+1})$$

$$b_i - b_{i+1} + 2c_i(x_{i+1} - x_i) = 0$$

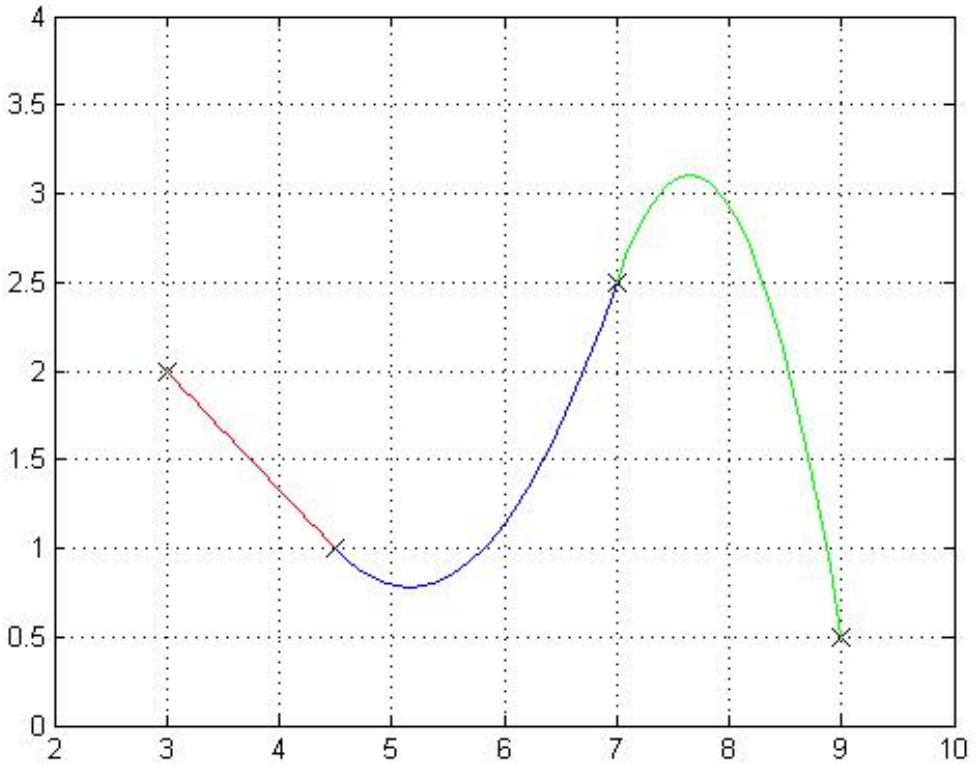
E'qn 1 gives the values of all a_i.

Eq'n 2 and eq'n 3 plus $c_1 = 0$ give 2(n – 1) equations which can be solved to give the values of all b_i and c_i.

Example:

x	3.0	4.5	7.0	9.0
y	2.0	1.0	2.5	0.5

< see quadSplineExample.m >



Coefficients for s1, s2, s3:

a	b	c
2.0000	-0.6667	0
1.0000	-0.6667	0.5067
2.5000	3.3867	-1.4333

< Try making c3 = 0 instead of making c1 = 0 >

< Compare with interpolating polynomial >

< quadSplineExample.m >

Quadratic splines are not supported in Matlab.

Typically not used except as education examples.

Cubic splines are generally preferred (better and not computationally demanding)

Cubic Splines:

Same idea but with cubics between data points

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Each interval involves four variables (*a*, *b*, *c*, *d*)
There are therefore a total of 4(*n* – 1) variables

Each $s_i(x)$ must pass through (x_i, y_i) and (x_{i+1}, y_{i+1}):

$$s_i(x_i) = y_i \quad s_i(x_{i+1}) = y_{i+1}$$

This gives 2(*n* - 1) equations.

We require that the first and second derivatives be continuous at all "knots" :

$$s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}) \quad s_i''(x_{i+1}) = s_{i+1}''(x_{i+1}) \quad \text{for } 1 \le i \le n-2$$

This gives a further 2(*n* – 2) equations.

The remaining two equations required to define the spline can be obtained in a number of ways. The fact that two assumptions are required is good because it makes for symmetry.

“Natural” end condition:

Second derivative at both ends is zero:

$$s_1''(x_1) = s_{n-1}''(x_n) = 0$$

Called “natural” because this is what one gets when a draftsman’s spline is bent around a number of pins.

“Clamped” end condition:

First derivative at both ends is defined:

$$s_1'(x_1) = k_s \quad s_{n-1}'(x_n) = k_e$$

Called “clamped” because this is corresponds to clamping the two ends of a draftsman’s spline at selected angles.

“Not a Knot” end condition:

Same cubic used for first and second intervals
Same cubic used for last and second to last intervals

The second and second last points cease to be “knots” (hence the name).

Number of functions reduced to $n - 3$
Number of variables reduced to $4(n - 3)$
Number of knots reduced to $n - 4$

The start and end points for each function give $2(n - 3)$ equations.

Continuity of the first and second derivatives at the knots gives $2(n – 4)$ equations.

The fact that the first and last functions must pass through an intermediate point gives a further two equations.

In all there are $2(n - 3) + 2(n - 4) + 2 = 4(n - 3)$ equations in $4(n - 3)$ unknowns

Cubic Spline Calculations:

Summary of steps (for details see text p 369-371)

Values of all a_i's obtained using

$$a_i = y_i$$

Values of all c_i's obtained by solving a tridiagonal matrix.

Values of all b_i's obtained using

$$b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{x_{i+1} - x_i}{3}\left(2c_i + c_{i+1}\right)$$

Values of all d_i's obtained using

$$d_i = \frac{c_{i+1} - c_i}{3\left(x_{i+1} - x_i\right)}$$

Cubic Splines in Matlab:

Commands:

yout = interp1 (x, y, xin, 'spline'); OR

yout = spline (x, y, xin);

x, *y* = data points, *xin* = *x* values of interest,
yout = values of the cubic spline at the *x* values of interest

By default the "not a knot" end condition is used.

For *spline* only (this doesn't work with *interp*1) : if the *y* vector is exactly two elements longer than the *x* vector the "clamped" end condition is used (with the first and last *y* values acting as the initial and final slopes).

The "natural" end condition is not supported (see text p 383 for a DIY version).

interp1 (x, y, xin, 'pchip') or interp1 (x, y, xin, 'cubic') gives *piecewise cubic Hermite interpolation*. This variation on the basic cubic spline reduces oscillations and can be a better choice when the data is not smooth (see text p378).

< Use splines with data points generated using Runge's function >

< Show the effects of clamping >

< splineExamples.m >