Piecewise Polynomial Interpolation

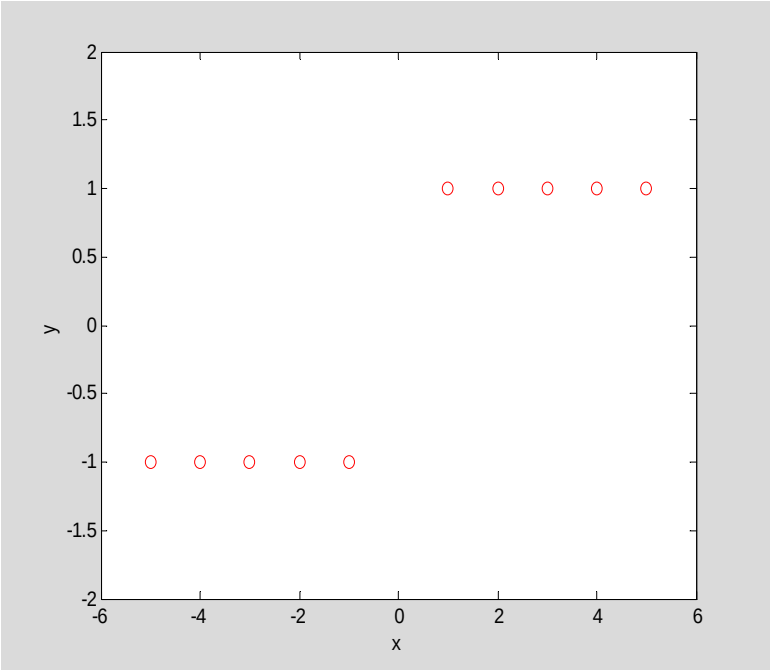
We have examined piecewise linear approximations as a way of providing estimates for a function over large intervals. Generalizing such methods so as to use higher degree interpolants provides one of the fundamental tools in computer aided design, among other areas.

Piecewise Polynomial Interpolation – An Example

Interpolation generally involves replacing a complicated function by a simpler approximation (usually a polynomial), constructed to pass through a given set of values of the underlying function. However, in some circumstances we simply have a data set of points in the plane and we wish to draw a smooth curve through these points. There is no particular underlying function that we wish to approximate. Rather, we are searching for a curve through the data that looks "right". In such cases, ordinary polynomial interpolation or linear piecewise interpolation may not produce the most desirable results.

To take a rather simple example, suppose we have the points $(-5,-1),(-4,-1),\ldots(-1,-1)$ and in addition the points $(1,1),(2,1),\ldots,(5,1)$, shown below:

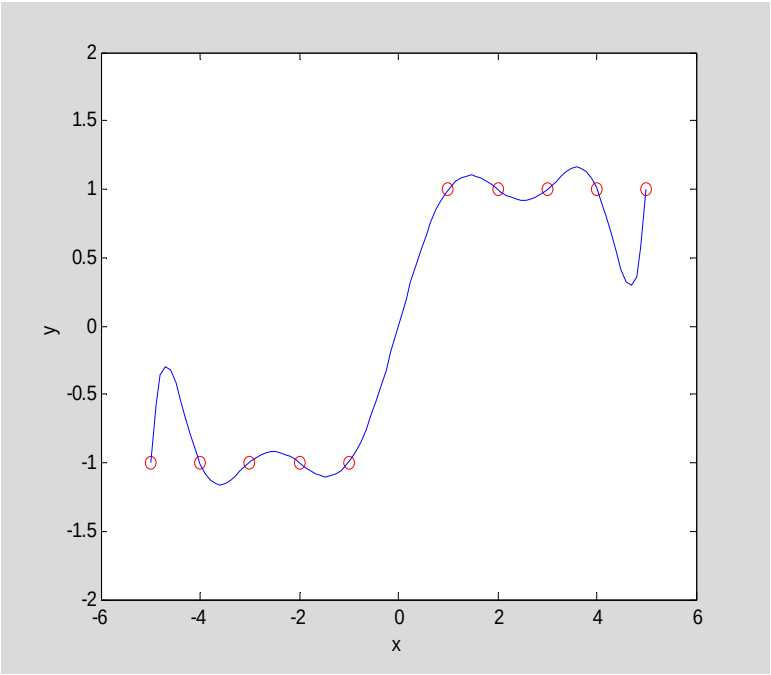
```
clf; x=[-5:-1,1:5]; y=[-ones(1,5),ones(1,5)];
plot(x,y,'or'); axis([-6 6 -2 2]); xlabel('x'); ylabel('y')
```



How would we construct a curve joining these points? If we use all 10 points to construct a degree 9 interpolating polynomial through the points we obtain the graph shown below.

```
p=polyfit(x,y,9);
x1=linspace(-5,5,100); y1=polyval(p,x1);
```

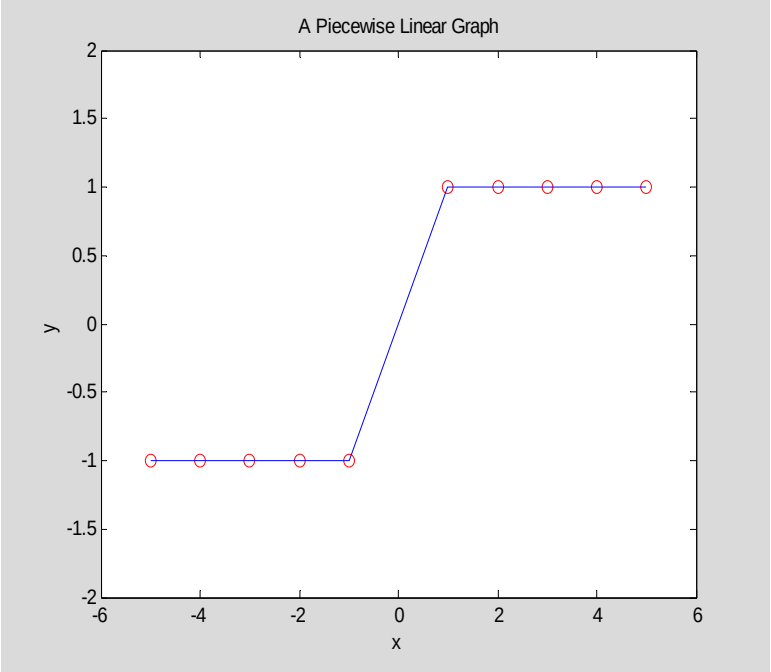
```
hold on;
plot(x1,y1)
```



This would hardly seem the sort of curve we might visualize connecting the points. As is often the case, high degree polynomials can have extraneous wiggles that make them unsuitable for this sort problem. In technical terms, high degree polynomials do a poor job of "shape preservation."

A more reasonable approach is to create a curve using piecewise polynomial interpolants. The simplest such method is to join successive points by straight line segments. The resulting curve is called piecewise linear. In fact, if we ask Matlab to join the original points with a line, it carries out precisely that construction, at least visually.

```
clf; plot(x,y,'or',x,y); axis([-6 6 -2 2])
xlabel('x'), ylabel('y'); title('A Piecewise Linear Graph');
```



Such interpolation always preserves shapes, and produces a continuous graph, since each piece begins where the last one ended. However, the graph is usually not smooth, meaning that it does not have a tangent direction at every point. In this example, the smoothness breaks down at the points $(-1,-1)$ and $(1,1)$. Smoothness is usually a desirable feature, so the piecewise linear structure, while simple to draw and compute, is not generally a satisfactory solution.

To obtain a smooth structure for the above graph we would like to have the middle segment from $(-1,-1)$ to $(1,1)$ join the two straight line segments on either end so as to have slope zero at both $x=-1$ and $x=1$. In other words, instead of a straight line in the middle, we want the graph of some polynomial $p(x)$ such that $p(-1)=-1$, $p'(-1)=0$, $p(1)=1$, and $p'(1)=0$. These are four conditions to satisfy, and the polynomial of minimal degree that has four coefficients that could (possibly) be adjusted to satisfy these conditions would be a cubic. Thus, our interpolant in the interval $[-1,1]$ would be a cubic polynomial $p(x)$, satisfying the four conditions just stated. How do we find it?

We can proceed using the method of undetermined coefficients, but we can simplify the algebra substantially by using a representation for $p(x)$ not in powers of x, but in powers of $(x+1)$. This is just the Taylor expansion of the polynomial with center $x=-1$. Thus, we write

$$p(x)=a(x+1)^3+b(x+1)^2+c(x+1)+d\,.$$

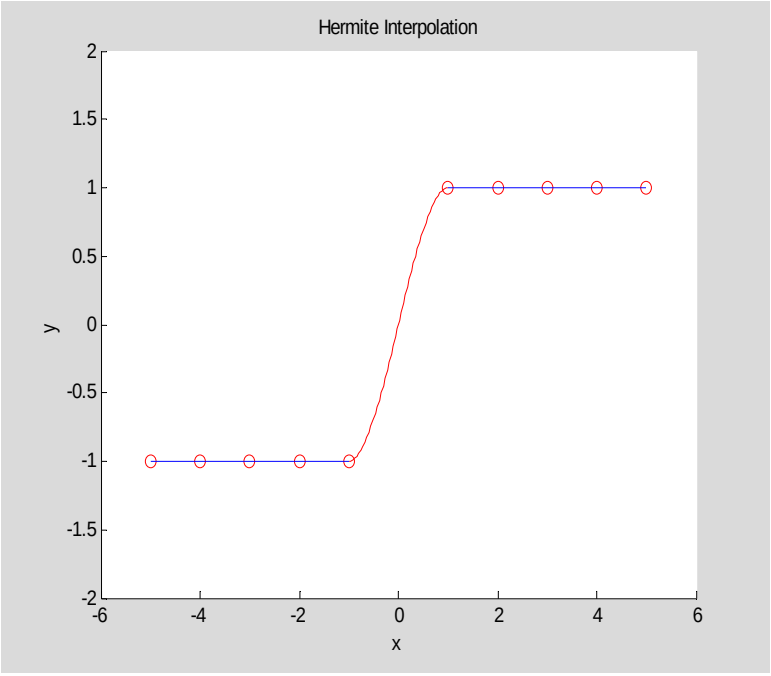
We know that $d=p(-1)$ and $c=p'(-1)$ so these values are then $d=-1$ and $c=0$. We then have $p(x)=a(x+1)^3+b(x+1)^2-1$. We now impose the requirements that $p(1)=1$ and $p'(1)=0$. This gives the two equations

$$4a+2b=1$$
$$3a+b=0$$,

which has the unique solution $a=-0.5, b=1.5$. Thus, the polynomial is $p(x)=-0.5(x+1)^3+1.5(x+1)^2-1$. There is no need to expand this polynomial to get the standard representation. If we let $p=[-.5, 1.5, 0, -1]$, we can evaluate the polynomial at any input x using the command `polyval(p,x+1)`.

We graph this polynomial in the interval $[-1,1]$, with line segments connecting the data points in the remaining intervals. We obtain the following graph:

```
clf; hold on
plot(x,y,'or');   % plot data points
plot(x(1:5),y(1:5),'b');  % connect first five points with straight lines
plot(x(6:10),y(6:10),'b');  % connect last five points with straight lines
p=[-.5 1.5 0 -1];  % defne cubic interpolating polynomial
x1=linspace(-1,1,50); y1=polyval(p,x1+1); % plotting points in [-1 1]
plot(x1,y1,'r'); axis([-6 6 -2 2])
xlabel('x');ylabel('y'); title('Hermite Interpolation')
```



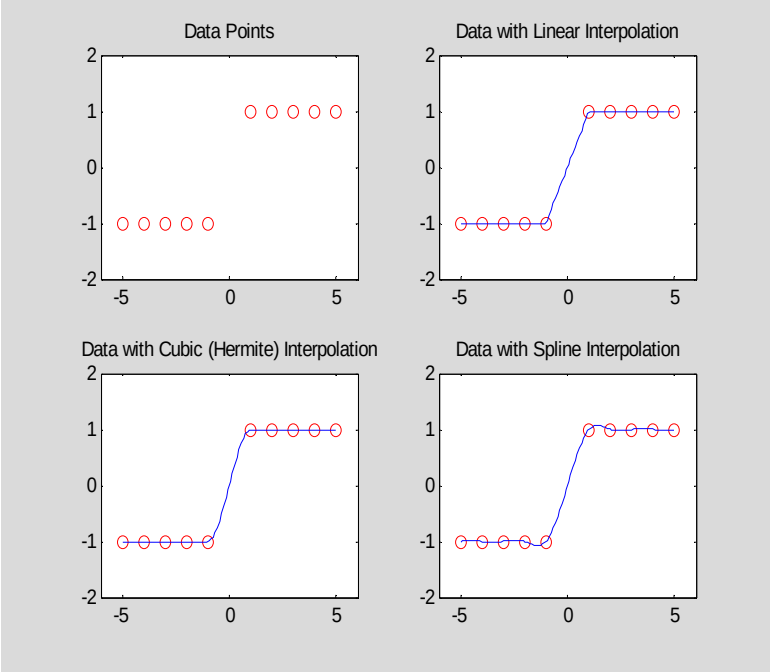
The interpolant construction that we have just described is called **Hermite or cubic interpolation**. Given two points (x_0, y_0) and (x_1, y_1), and additional numbers y_0' and y_1',

this produces a (unique) cubic polynomial $p(x)$ such that $p(x_0) = y_0$, $p(x_1) = y_1$, with the values y_0' and y_1' specifying the slopes: $p'(x_0) = y_0'$ and $p'(x_1) = y_1'$.

Piecewise Polynomial Interpolation in Matlab

Matlab has a builtin command, `interp1`, for computing piecewise polynomial interpolations. (The "1" indicates that this is one-variable interpolation. There are `interp2` and `interp3` commands as well. In fact, there is an entire toolbox devoted to this area.) This command requires as inputs vectors *x* and *y*, which contain the *x* and *y* coordinates of the data points. We then usually give a list of input values *xx* at which we wish to compute the interpolated values. For example, if we want to graph the resulting function over a large interval we would enter for *xx* a list of values selected uniformly over the interval. Finally, we generally specify the method of interpolation. The relevant choices are '`linear`' (default, if no method is specified),'`cubic`' (which does Hermite interpolation), and '`spline`', which we will discuss in more detail below. Let's illustrate.

```
subplot(2,2,1);
plot(x,y,'or'); axis([-6 6 -2 2]);
title('Data Points')
xx=linspace(-5,5,100);
yy=interp1(x,y,xx,'linear');
subplot(2,2,2)
plot(x,y,'or',xx,yy); axis([-6 6 -2 2]);
title('Data with Linear Interpolation')
yy=interp1(x,y,xx,'cubic');
subplot(2,2,3);
plot(x,y,'or',xx,yy); axis([-6 6 -2 2]);
title('Data with Cubic (Hermite) Interpolation')
yy=interp1(x,y,xx,'spline');
subplot(2,2,4);
plot(x,y,'or',xx,yy); axis([-6 6 -2 2]);
title('Data with Spline Interpolation')
```



The spline interpolation appears to have some additional wiggle that in this case is probably less desirable than the result produced by cubic interpolation. Note, that as described above, cubic interpolation requires information about the slope of the curve at the interpolation points. In fact, in this problem we do not have such information and matlab's cubic interpolation scheme makes some reasonable assignments for those numbers based on the data points. In this case, judging by the results, the choices seem perfectly reasonable.

It is perhaps instructive to understand what the command `interp1` actually computes. In fact, if we omit the input points, denoted above by *xx*, we can view the piecewise polynomial structure that interp1 computes.

```
pp=interp1(x,y,'cubic','pp')  % pp stands for piecewise polynomial
```

```
pp =
      form: 'pp'
    breaks: [-5 -4 -3 -2 -1 1 2 3 4 5]
     coefs: [9x4 double]
    pieces: 9
     order: 4
       dim: 1
    orient: 'first'
```

The output is a Matlab structure, which is a mult-dimensional array with named fields. The important fields are the list of `breaks`, which show the intervals on which the various interpolation polynomials are defined, and the `coefs` field. The latter field

shows the coefficients of the interpolation polynomial in each interval. Let's look at that field in more detail.

```
pp.coefs
```

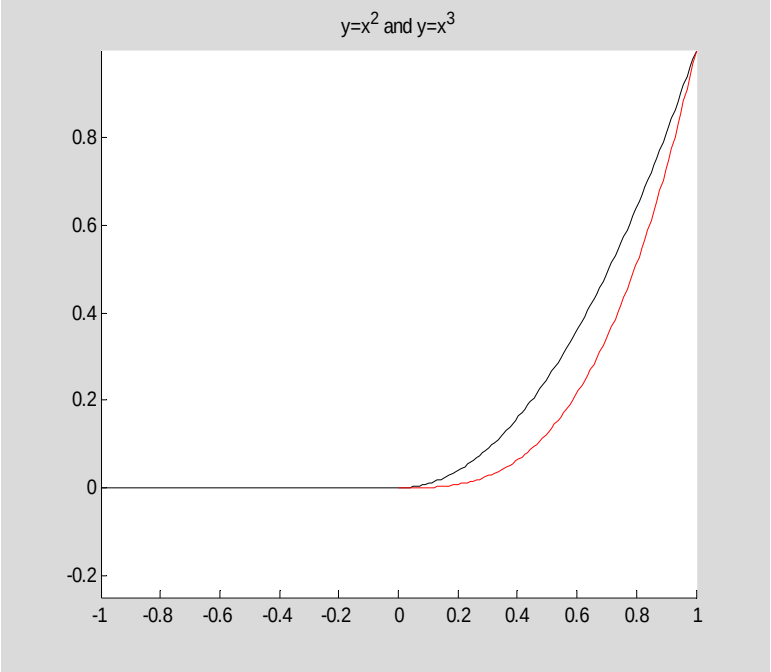
```
ans =
         0         0         0   -1.0000
         0         0         0   -1.0000
         0         0         0   -1.0000
         0         0         0   -1.0000
   -0.5000    1.5000         0   -1.0000
         0         0         0    1.0000
         0         0         0    1.0000
         0         0         0    1.0000
         0         0         0    1.0000
```

Each row of the 9 x 4 matrix holds the coefficients for one of the interpolating polynomials, arranged in order of decreasing degree. Except for the middle polynomial, all are constants. Notice that the middle polynomial has the coefficients that we had computed earlier, so that implicitly the polynomial is being expressed in powers of $(x+1)$, as we had done. In fact, in each row the coefficients refer to the expansion of the polynomial in powers $(x-x_i)$, where x_i is the interpolation point at the beginning of the interval for which that polynomial is the interpolant.

Splines -Generalities

In using Hermite (cubic) interpolation we match slopes at the endpoints of the interpolation intervals to produce a smooth graph. However, continuity of the derivative which is obtained by matching first derivatives at the break points, does not produce a curve that is as visually appealing as possible. For example, consider how the two graphs $y=x^2$ and $y=x^3$ for $x \geq 0$ blend with the negative x-axis at the origin.

```
 clf; hold on;
xx=linspace(0,1,100);y2=xx.^2; y3=xx.^3;
plot([-1 0], [0 0], 'k', xx,y2, 'k', xx, y3,'r');
axis([-1 1 -.25 1]); title('y=x^2 and y=x^3');
```



Although both graphs produce a smooth transition with the horizontal segment to the left, the cubic curve gives a more pleasing, smoother transition. Mathematically, the latter transition not only has a continuous first derivative, but continuous second derivative as well, both of these being zero at the origin. For the quadratic curve the second derivative at the origin is two. Geometrically, this results in a discontinuity in the curvature as we make the transition from the quadratic to the line. This discontinuity is subtle, but visually we have no doubt as to which of the two transitions is "smoother."

The cubic spline interpolation method produces cubic polynomials in each interval, such that the first and second derivatives match each other at the endpoints. Let's sketch how this is accomplished. We have interpolation intervals $[x_0, x_1]$, $[x_1, x_2]$, …, $[x_{n-1}, x_n]$. We want to construct *n* cubic polynomials $p_0(x), p_2(x), \ldots, p_{n-1}(x)$ which pass through the given *y* values at each interpolation point, i.e. satisfy

$$\begin{aligned} p_i(x_i) &= y_i \\ p_i(x_{i+1}) &= y_{i+1} \end{aligned}, \tag{1.1}$$

for $i = 0,1,\cdots,n-1$. In addition, we require that at the interior break points $x_1, x_2, \ldots, x_{n-1}$ we have

$$\begin{aligned} p_i'(x_{i+1}) &= p_{i+1}'(x_{i+1}) \\ p_i''(x_{i+1}) &= p_{i+1}''(x_{i+1}) \end{aligned}, \tag{1.2}$$

for $i = 0,1,\cdots,n-2$. Notice, that these conditions do not involve specifying the actual values of the first or second derivatives at the break points.

Can conditions (1.1) and (1.2) be satisfied? Using the method of undetermined coefficients we can show that it is reasonable to expect that, with some additional

specifications, a unique family of cubic polynomials exists that satisfies both conditions. Namely, write the polynomial $p_i(x)$ as

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

We have a total of $4n$ coefficients that have to be determined. Equations (1.1) give $2n$ linear equations for these unknown coefficients. Equations (1.2) give an additional $2(n-1) = 2n-2$ equations, so that we have $4n-2$ linear equations. This leaves two degrees of freedom in the system. Generically, we can impose two additional linear restrictions in order to achieve a unique solution. One way in which this is done is to declare that $p_0''(x_0) = p_{n-1}''(x_n) = 0$. Note that these values are not constrained by any of the other conditions. This construction is called the natural cubic spline.

Matlab's implementation of the spline construction in the function `interp1` uses a different condition, the so-called "not-a-knot" condition. This imposes that $a_0 = a_1$ and $a_{n-2} = a_{n-1}$. Essentially, it says that in the first two intervals and the last two, the two polynomials exhibit continuity through the 3rd derivative. In effect, there is a unique 3rd degree polynomial throughout these pairs of intervals so the two break points ("knots" in the technical literature) x_1 and x_{n-1} are no longer acting as knots where two distinct interpolants meet.

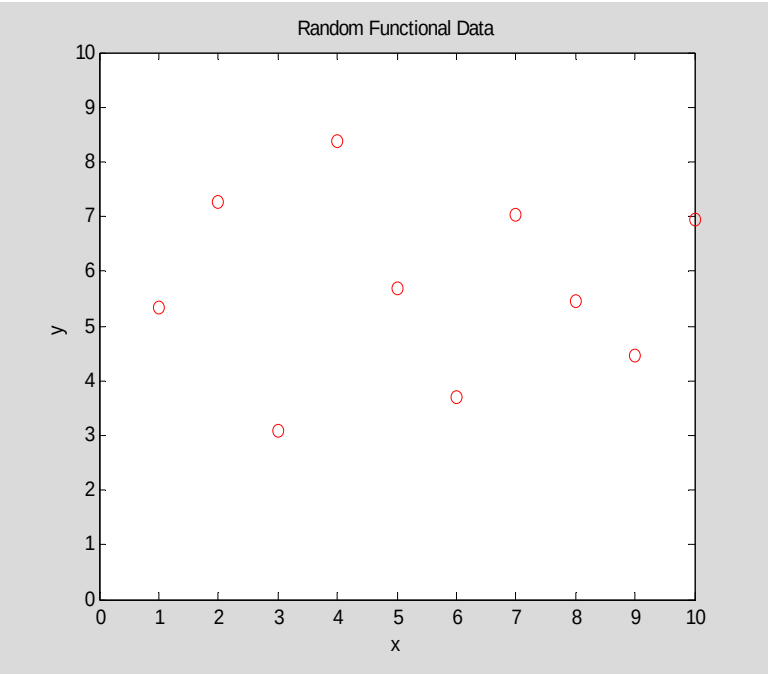
The above discussion by no means constitutes a proof that even with the extra conditions we can produce the requisite polynomials. We must actually examine the resulting linear system and determine that it has a unique solution. The details of this are somewhat involved. The reader can consult the textbook. In a nutshell, the strategy is to assume that the functions $p_0, p_1, \ldots, p_{n-1}$ exist and have derivatives $y_0', y_1', \ldots, y_n'$, at $x_0, x_1, \ldots, x_n$ which match at the interior break points. If we can find the values of these derivatives, we could use Hermite interpolation to produce each cubic polynomial from the requirement (1.1) and the two slopes at the end points x_i and x_{i+1}. The main part of the calculation is to show that the continuity requirement of the **second derivative,** plus the two extra conditions imposed, lead to a unique set of values for the slopes $y_0', y_1', \ldots, y_n'$. Moreover, this set of values can be computed very quickly (in $O(n)$ steps), so the entire construction of the spline also has the same computational complexity, rather than the $O(n^3)$ one might have expected from general considerations of solving a system of $4n$ linear equations.

Splines – an Example

Let's examine how well the various interpolation schemes perform in a setting where we have to join a set of data points with a smooth graph. We take our *x* values to be the integers 1, 2, 3, …, 10, and *y* values randomly drawn between 0 and 10. We first display the points.

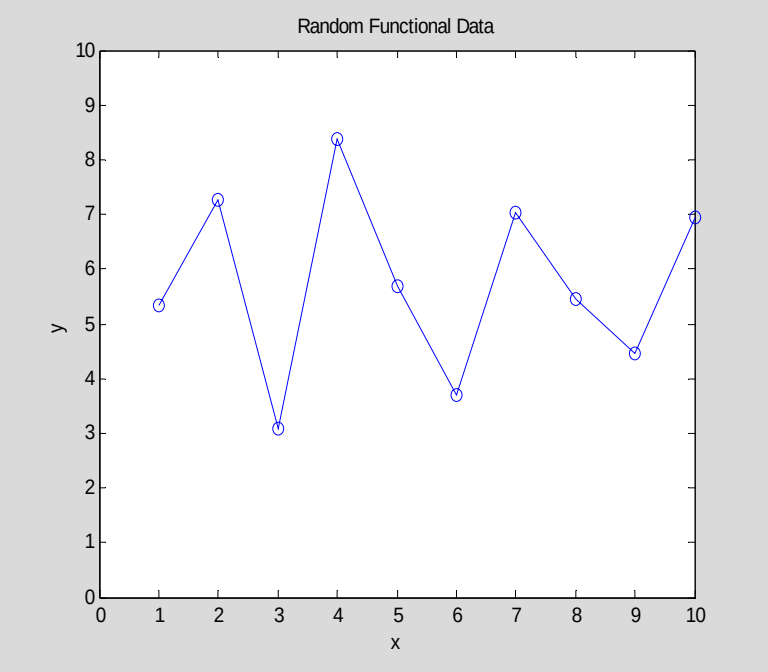
```
clf;
x=1:10; y=10*rand(1,10);  % generate break points and values
plot(x,y,'or');
```

```
axis([0 10 0 10])
title('Random Functional Data'); xlabel('x');ylabel('y');
```



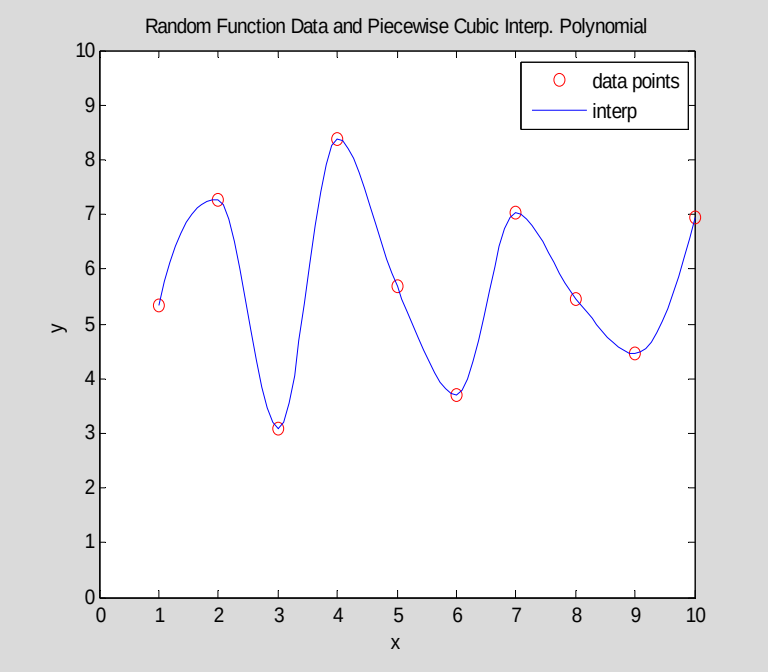
We can join the points from left to right via straight lines, which is equivalent to using the 'linear' option with interp1.

```
plot(x,y,'o-')
axis([0 10 0 10])
title('Random Functional Data'); xlabel('x');ylabel('y');
```



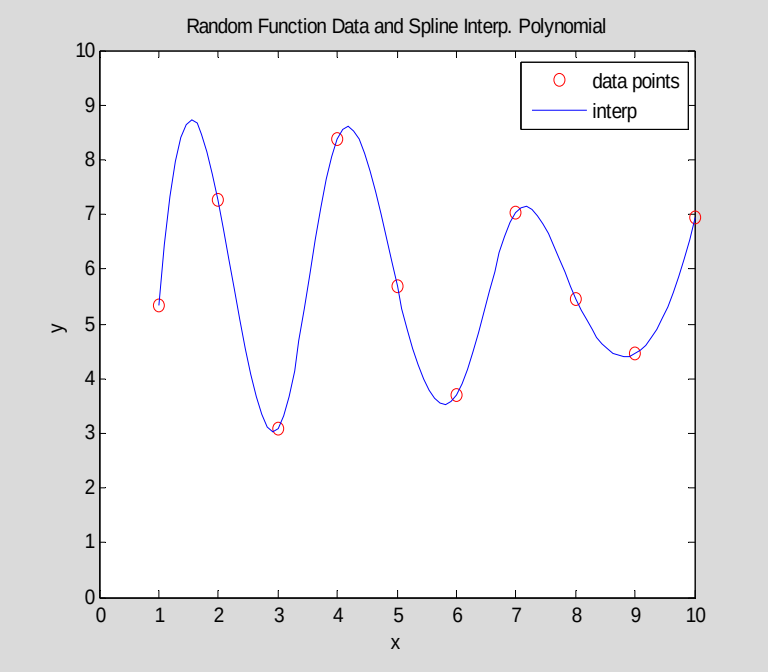
The piecewise linear interpolation perserves the shape of the data, but is not smooth. Let's examine how well we can do with cubic interpolation through the same points.

```
clf;
xx=linspace(1,10, 100); %divide plotting interval into subintervals
yp=interp1(x,y, xx, 'cubic');  % compute y values of the interp at
points xx
plot(x,y,'or', xx,yp,'b');
title('Random Function Data and Piecewise Cubic Interp. Polynomial');
legend('data points','interp')
axis([0 10 0 10]);xlabel('x');ylabel('y');
```



This is does a good job of preserving the shape of the data and is relatively smooth. Let's examine the spline produced by `interp1`.

```
clf;
yp=interp1(x,y, xx, 'spline');  % compute y values of the interp at points xx
plot(x,y,'or', xx,yp,'b');
title('Random Function Data and Spline Interp. Polynomial');
legend('data points','interp')
axis([0 10 0 10]);xlabel('x');ylabel('y');
```



The curve definitely has a smoother appearance than in the previous example. However, it does seem to "overshoot" in several regions, particularly near the beginning. There is no quantitative way to decide which of the last two graphs is "better". The choice would involve both esthetics and some understanding of possible patterns that the data might be describing.

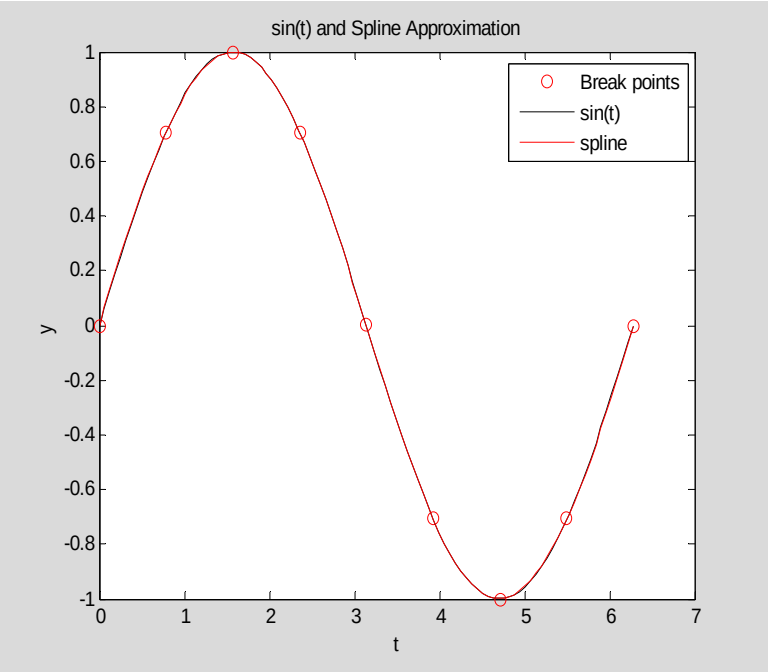
Parametric Splines

Suppose we want to use splines to draw a circle. Since splines are just functions and a circle is not the graph of a function it might not seem possible to do this. We can however resort to parametric plots. In this case the circle (say of radius one and centered at the origin) can be parametrized by $x = \cos t$, $y = \sin t$, $0 \le t \le 2\pi$. If we approximate $\cos t$ with a spline approximation $p_c(t)$ and similarly approximate $\sin t$ with another spline $p_s(t)$ then setting $x = p_c(t)$ and $y = p_s(t)$ ought to give a parametric curve that is close to a circle.

First let's examine how well we can approximate the cosine and sine functions on $[0, 2\pi]$ using a spline with eight subintervals constructed by dividing the interval from 0 to 2π into segments of length $\pi/4$. Notice that this construction uses only very easily computed values of the trigonometric functions.

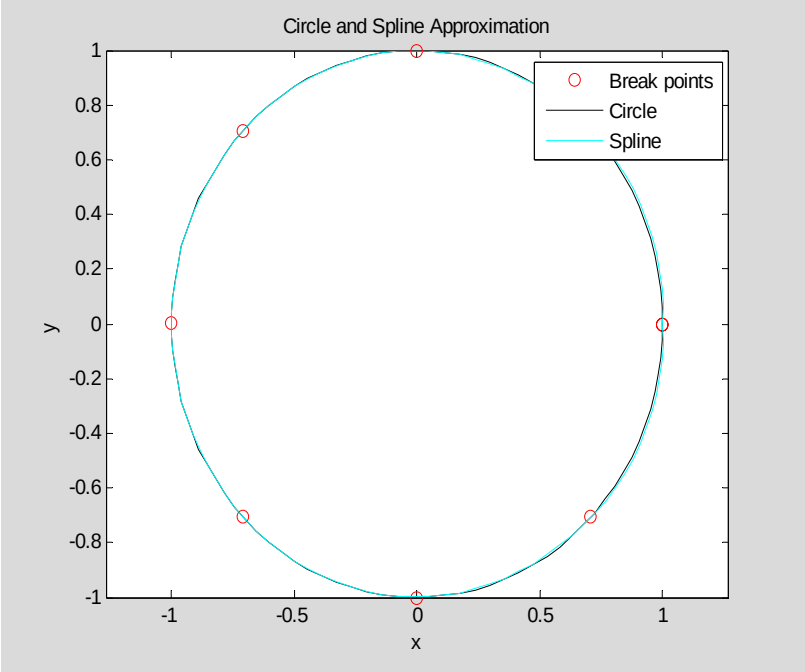
```
clf;
t=0:pi/4:2*pi;
ys=sin(t);  % sine values to interpolate
tt=linspace(0,2*pi,100);
ps=interp1(t,ys,tt,'spline'); % interpolated values at points tt
```

```
plot(t,ys,'or', tt, sin(tt), 'k', tt, ps, 'r');
legend('Break points', 'sin(t)','spline')
title('sin(t) and Spline Approximation');
xlabel('t'); ylabel('y');
```



The agreement is quite good. In fact, only one curve is visible because the actual sine curve is mostly overdrawn by the spline plot. One obtains a similar result for the cosine function. We can put them together to obtain a parametric plot approximating a circle to very high accuracy.

```
clf;
yc=cos(t);
pc=interp1(t,yc,tt, 'spline');
plot(yc,ys, 'or', cos(tt),sin(tt), 'k', pc,ps, 'c')
title('Circle and Spline Approximation')
legend('Break points','Circle','Spline')
xlabel('x');ylabel('y');
axis equal
```



What is the value of this representation over the plot of a circle using the sine and cosine functions? There are two principal advantages. First, if we wish to store the picture of the circle, instead of having to store the coordinates of the all the plotted points, we have only to store the small number of coefficients for the polynomials and break point data for the spline. A suitably programmed grahics rendering device can then reproduce the circle. Second, the rendering does not require evaluating complicated functions such as sine and cosine. To generate the points on the curve we are simply evaluating polynomials, which can be done very quickly. Thus, this type of representation lends itself to computer animation, where graphics must be rendered in real time. In practice, parametric splines are not efficient enough for this purpose, and other parametric plotting methods are used. See the article http://en.wikipedia.org/wiki/NURBS in Wikipedia on a general class of methods of this variety.