Notes on Least Squares Cubic Splines

The documentation of the Spline Toolbox provides a one-line command for computing the cubic spline that best fits data in a least squares sense. Specifically,

\[ sp = \text{spline}(b,y(:)'/\text{spline}(b,\text{eye}(\text{length}(b)),x(:)')); \]

returns the cubic spline \( sp \) with breakpoints \( b \) that best fits the values \( y \) at the nodes \( x \) in a least squares sense. This is a complicated command that makes a sophisticated use of the underlying functions, so in these notes we’ll sort out what is being done and in the process learn more about cubic splines and least squares fits. We conclude with a function that facilitates computing a least squares cubic spline.

In MA5315 we have given our attention to the problem of interpolating discrete data with a breakpoint at each node. We begin by discussing this problem in a different way. The breakpoints are to be \( b_1, \ldots, b_m \). Like the fundamental Lagrangian interpolation polynomials, we define \( \phi_i(x) \) as a cubic spline with \( \phi_i(b_j) = \delta_{i,j} \), i.e., the spline has value 1 at \( b_i \) and 0 at all the other nodes. To see a basis function, we use the little program

\[
\begin{align*}
b & = 0:20; \\
M & = \text{eye}(\text{length}(b)); \\
xplot & = \text{linspace}(b(1),b(\text{end})); \\
\phi5 & = \text{spline}(b,M(5,:),xplot); \\
\text{plot}(xplot,\phi5,b,0,'ro')
\end{align*}
\]

to calculate \( \phi_5(x) \) for the specified breakpoints and plot it as Figure 1. This is pretty straightforward, but notice that we use row 5 of an identity matrix to get a row vector that is 0 except for entry 5 which is a 1. Also, we use the option in \text{spline} of returning values of the spline at points specified by the argument \( xplot \).

We can write a general spline \( S(x) \) with the given breakpoints as a linear combination of these basis functions,

\[
S(x) = c_1 \phi_1(x) + \ldots + c_i \phi_i(x) + \ldots + c_m \phi_m(x) \quad (1)
\]

If we wish to interpolate at the breakpoints only, then just as with the fundamental Lagrangian polynomials, we can use the properties of the basis functions to deduce that

\[
S(x_i) = y_i = c_1 0 + \ldots + c_i 1 + \ldots + c_m 0 = c_i
\]

for \( i = 1, \ldots, m \). However, if there are \( n \geq m \) nodes that are not necessarily breakpoints, we can say only that

\[
S(x_i) = y_i = c_1 \phi_1(x_i) + \ldots + c_m \phi_m(x_i)
\]
In obvious notation, the equations for \( i = 1, \ldots, n \) are
\[
Ac = y
\] (2)
where the \((i, j)\) entry of the matrix \( A \) is \( \phi_j(x_i) \). In MATLAB a system of linear equations \( A\mathbf{c} = \mathbf{y} \) with a square matrix can be solved with the backslash operator. The backslash can also be used when the matrix has more rows than columns. In this situation it is generally not possible to satisfy all the equations, so the system is solved in the sense of minimizing the sum of the squares of the residuals, a least squares solution. Accordingly, we can find the coefficients of the spline that fits the data as well as possible with \( \mathbf{c} = A\mathbf{y} \). Here it is assumed that \( \mathbf{y} \) is a column vector and the result \( \mathbf{c} \) is also a column vector. For our purposes it will be more convenient to have a row vector. One way to do this is to transpose the equation to get \( \mathbf{c}' = (A\mathbf{y})' = \mathbf{y}'/A' \). (Notice that the direction of the slash has been reversed.) That is easy enough, so we consider now how to compute \( A \) efficiently.

When we computed \( \phi_5(x) \) and in most of our use of \texttt{spline} in MA5315, we supplied a row vector of values to be interpolated at breakpoints, but the function allows a user to supply a matrix. It then computes a collection of splines, one for each row of the data matrix. In this way we can obtain all the basis functions at one time by using the identity matrix as argument:

\[
\Phi = \text{spline}(b, \text{eye}(\text{length}(b)));
\]
Each of these splines is computed by solving a set of linear equations. The matrix depends only on the breakpoints, so it is the same for each spline. As we’ll see when we take up solving linear systems, it is considerably cheaper to compute all the basis splines in this way rather than one after the other.

For a given set of nodes \( \mathbf{x} \) that might be a row or column vector, the result of \( \mathbf{x}(:)' \) is a row vector. With this we understand that the command

\[
\mathbf{M} = \text{spline}(b, \text{eye}(\text{length}(b)), \mathbf{x}(:)') ;
\]
evaluates all the basis functions for breakpoints \( \mathbf{b} \) at all the nodes \( \mathbf{x} \) and returns them in an array \( \mathbf{M} \). For instance, the first row is \( \phi_1(x_1), \phi_1(x_2), \ldots, \phi_1(x_n) \). In general, the \((i, j)\) entry of the matrix \( \mathbf{M} \) is \( \phi_i(x_j) \). This says that \( \mathbf{M} \) is the transpose of the matrix \( A \) in (2), which turns out to be convenient.

We now have all that we need to understand the command

\[
\mathbf{sp} = \text{spline}(b, \mathbf{y}(:)'/\text{spline}(b, \text{eye}(\text{length}(b)), \mathbf{x}(:)')) ;
\]
The portion \( \text{spline}(b, \text{eye}(\text{length}(b)), \mathbf{x}(:)') \) forms all the basis functions simultaneously, makes a row vector of the nodes, and evaluates all the basis functions at these nodes. After making the values \( \mathbf{y} \) a row vector with \( \mathbf{y}(:)' \), a slash operation is used to calculate the least squares solution of \( Ac = y \) in the form of a row vector \( \mathbf{c} \). The \texttt{spline} function is used again to form a spline (1) with breakpoints at \( \mathbf{b} \) and values \( \mathbf{c} \) there that provide the best least squares fit of the spline to the values \( \mathbf{y} \) at the nodes \( \mathbf{x} \). This spline can be evaluated with \texttt{ppval} wherever we like.
To illustrate the computation of a least squares spline, the following program specifies some breakpoints and then forms data to be fit using a smooth function plus some normally distributed random noise. The spline is computed with the command studied in these notes and then it is evaluated for plotting. It is also evaluated at the breakpoints so that the portions of the spline that are cubic polynomials are distinguished in the plot. The data is plotted to show how a least squares fit can smooth the effects of noise.

```matlab
b = linspace(0,2*pi,5);
x = linspace(0,2*pi,20);
y = cos(x) + 0.2*randn(size(x));
sp = spline(b,y(:)' / spline(b,eye(length(b)),x(:)'));
xplot = linspace(0,2*pi);
yplot = ppval(sp,xplot);
spb = ppval(sp,b);
plot(xplot,yplot,'b',x,y,'ro',b,spb,'ks')
legend('Spline','Data','Breakpoints','Location','SouthWest')
axis([-1 7 -1.5 1.5])
```

The result of this program is displayed as Figure 2.

It is convenient to have a function for computing a least squares cubic spline, so we have implemented this scheme in a function `lstsqrsp` that resembles `spline`. It is short enough to list:

```matlab
function output = lstsqrsp(b,x,y,xx)

% LSTSQRSP Least squares cubic spline data fit.
% PP = LSTSQRSP(B,X,Y) provides the piecewise polynomial form of the
% cubic spline with breakpoints B that best fits values Y at nodes X
% in a least squares sense, for use with the evaluator PPVAL. X and Y
% must be vectors of the same length.
%
% YY = LSTSQRSP(B,X,Y,XX) is the same as YY = PPVAL(SPLINE(B,X,Y),XX),
% thus providing, in YY, the values of the least squares fit at XX.
% For information regarding the size of YY see PPVAL.

if length(x) ~= length(y)
    error('X and Y must have the same length.')
end
x = x(:); y = y(:);
b = b(:); b = sort(b);
pp = spline(b,y/spline(b,eye(length(b)),x));
if nargin == 3, output = pp; else output = ppval(pp,xx); end
```
Figure 1: $\phi_5(x)$ for breakpoints 0:20.

Figure 2: A least squares spline fit.