Homework #7

Part 1 – Polynomial Fitting

Algorithm Details

The goal of part 1 was to create an algorithm that will fit an n order polynomial to a series of n+1 data points. The function was actually written to allow the fit of any order polynomial such that the polynomial order is less than the number of data points. The algorithm sets up and solves the following system in the following way:

n= order of the polynomial
m= number of (x,y) pairs
The polynomial that will be fit to the data is described by:

$$y = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_n \cdot x^n$$

The system of equations can be described by:

$$\begin{Bmatrix} y_0 \\ \vdots \\ y_m \end{Bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} \cdot \begin{Bmatrix} a_0 \\ \vdots \\ a_n \end{Bmatrix}$$

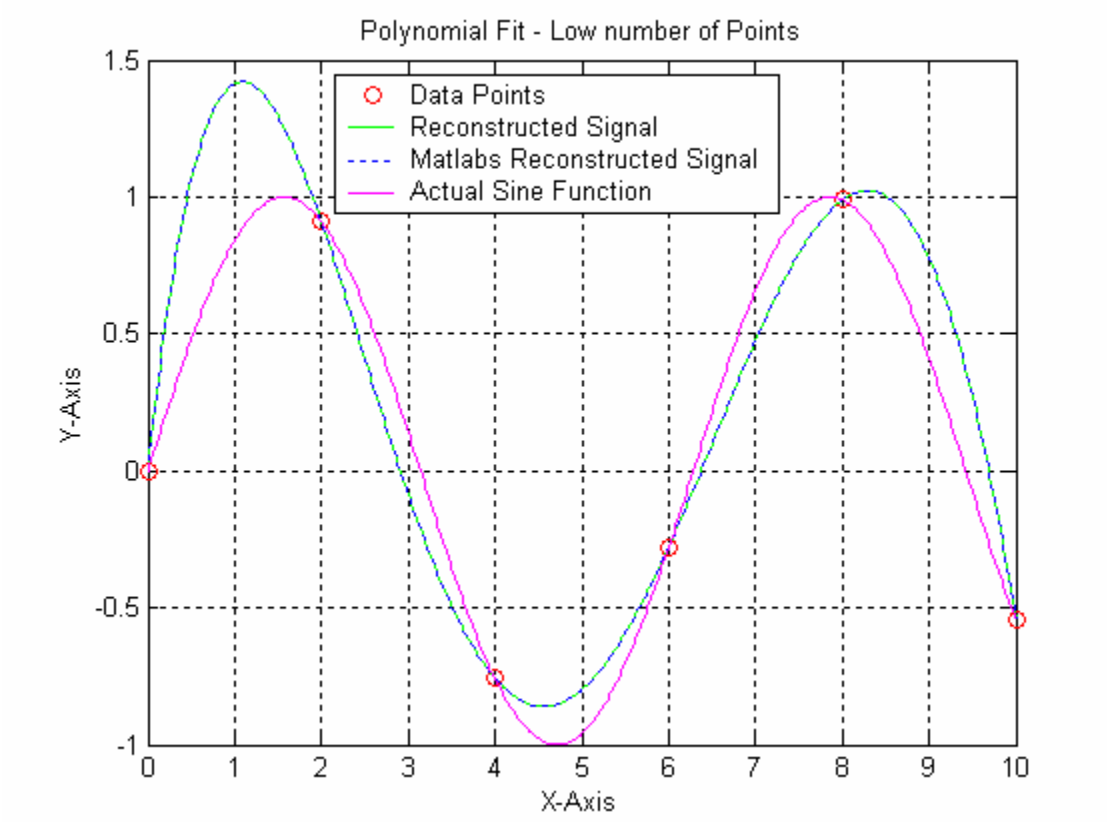
If m=n+1, then there is exactly one solution to the system of equations and the resulting polynomial will go exactly through each data point. In the case of m>n+1, the system is overconstrained, and Matlab applies a least squares solution to find the polynomial of order n that best fits the data. If n is greater than or equal to m, the algorithm will not run because not enough data points (and therefore not enough equations) are available to solve the system. The algorithm also displays the following warning message if not enough points are supplied. "Not enough data to fit model of order n. Provide more points or reduce model order such that # of data points > n."

The actual function code for the function *poly_morison* is included at the end of this report.

Testing the Algorithm

A sine function with a low number of points was used to evaluate the success of the algorithm. The following code was used.

```
>> x=[0:2:10];
>> y=sin(x);
>> x2=[0:.02:10];
>> y2=sin(x2);
>> n=5;
>> [a_comp]=poly_morison(x,y,n);
>> recon=polyval(flipud(a_comp),x2);
>> mat_po=polyfit(x,y,n);
>> recon_mat=polyval(mat_po,x2);
>> figure('color','white')
>> plot(x,y,'ro',x2,recon,'g',x2,recon_mat,'b:',x2,y2,'m')
>> grid on
>> legend('Data Points','Reconstructed Signal','Matlabs Reconstructed Signal','Actual
   Sine Function')
>> xlabel('X-Axis')
>> ylabel('Y-Axis')
>> title('Polynomial Fit - Low number of Points')
```

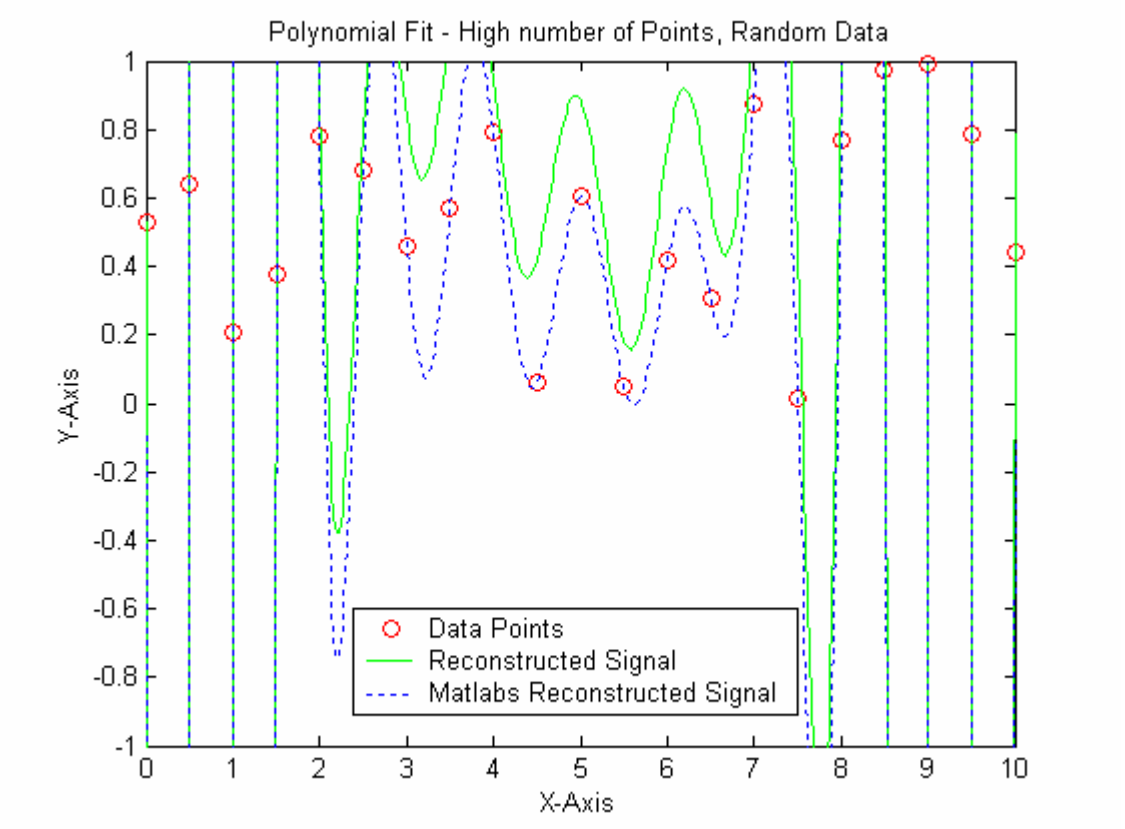


Inspection of the figure shows that the polynomial does indeed go through the data points, as should be the case for 6 (x,y) pairs and a 5^{th} order polynomial. It

generally approximates the true function also. Furthermore, Matlab's polyfit command yielded identical results.

As the number of points becomes large, the problem tends to become poorly conditioned due to raising x values to high powers in the matrix formulation. To investigate this phenomenon, the following code was used.

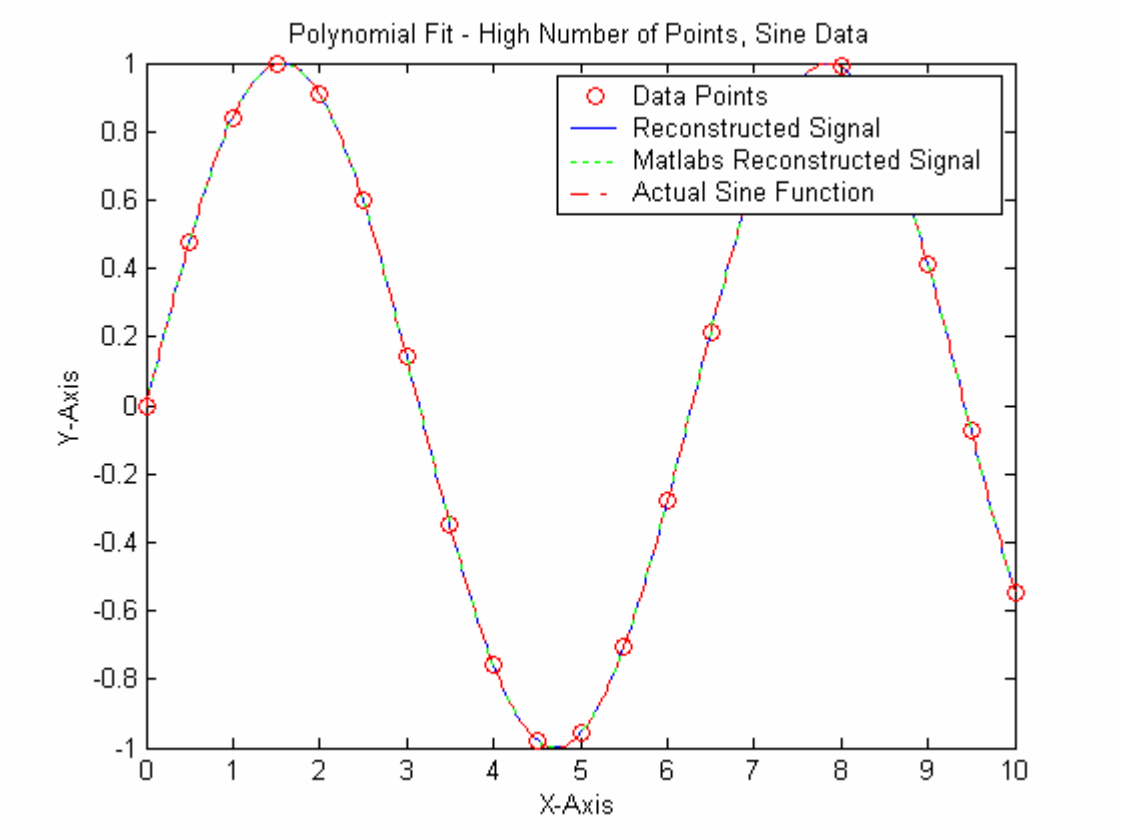
```
>> x=[0:.5:10];
>> y=rand(1,21);
>> [a_comp]=poly_morison(x,y,20);
Warning: Matrix is close to singular or badly scaled.
        Results may be inaccurate. RCOND = 1.022589e-027.
       In G:\GSF4610\HW7\poly_morison.m at line 51
>> x2=[0:.02:10];
>> recon=polyval(flipud(a_comp),x2);
>> mat_po=polyfit(x,y,20);
Warning: Polynomial is badly conditioned. Remove repeated data points
        or try centering and scaling as described in HELP POLYFIT.
>> recon_mat=polyval(mat_po,x2);
>> figure('color','white')
>> plot(x,y,'ro',x2,recon,'g',x2,recon_mat,'b:')
>> legend('Data Points','Reconstructed Signal','Matlabs Reconstructed Signal')
>> xlabel('X-Axis')
>> ylabel('Y-Axis')
>> title('Polynomial Fit - High number of Points, Random Data')
```



Here we can see that my polynomial does not go through the points as it should and a warning about the matrix being badly scaled has been reported in both the home made algorithm and Matlab polyfit function. Even with the warning, the polyfit command was still able to find a valid solution though. The accuracy error is a perfect example of the instability that occurs with a high model order and a high number of data points. Matlab's function also obviously has features that the home made algorithm does not to improve its result. This, however, was random data. In almost all cases, the data to be fit will have an underlying pattern. To investigate a high number of points of structured data, the sine function was again used. In this case, the same x values were used and the y values were calculated y=sin(x).

```
>> x=[0:.5:10];
>> x2=[0:.02:10];
>> y=sin(x);
>> [a_comp2]=poly_morison(x,y,20);
Warning: Matrix is close to singular or badly scaled.
        Results may be inaccurate. RCOND = 1.022589e-027.
> In G:\GSF4610\HW7\poly_morison.m at line 51
>> recon2=polyval(flipud(a_comp2),x2);
>> mat_po=polyfit(x,y,20);
Warning: Polynomial is badly conditioned. Remove repeated data points
        or try centering and scaling as described in HELP POLYFIT.
>> recon_mat=polyval(mat_po,x2);
>> figure('color','white')
>> plot(x,y,'ro',x2,recon2,'b',x2,recon_mat,'g:',x2,sin(x2),'r-.')
>> title('Polynomial Fit - High Number of Points, Sine Data')
>> xlabel('X-Axis')
>> ylabel('Y-Axis')
>> legend('Data Points','Reconstructed Signal','Matlabs Reconstructed Signal','Actual
    Sine Function')
```

Inspection of this figure (see next page) shows that even though a high number of points and a high model order is used, the results are far more reliable when the underlying data has some sort of continuity. Matlab has again reported a poorly scaled problem within the execution of *poly_morison*, but was able to find an accurate solution.



Part 2 – Cubic Spline Fitting

Algorithm Details

The goal of this algorithm is to fit a cubic spline to a series of data points. The spline is continuous in value, first, and second derivative at each knot (data point). The end conditions are handled by forcing the first derivative to zero. The algorithm takes as input a series of data points and returns the values of the coefficients of each third order polynomial needed to create the spline. The following linear system is created and solved within the function (spline_morison):

$$[A]\cdot\{x\}=\{b\}$$

A is a matrix containing coefficients for the various equations needed to generate the spline, x contains the coefficients of the polynomials that can be used to generate the spline, and b contains the values at each knot or zero, depending on the type of equation.

The equations for the value of the spline at each knot were built in the following sub-matrix form and then distributed into the overall matrix equation.

$$\begin{bmatrix} 1 & x_i & x_i^2 & x_i^3 \\ 1 & x_{i+1} & x_{i+1}^2 & x_{i+1}^3 \end{bmatrix} \cdot \begin{Bmatrix} a_i \\ b_i \\ c_i \\ d_i \end{Bmatrix} = \begin{Bmatrix} y_i \\ y_{i+1} \end{Bmatrix}$$

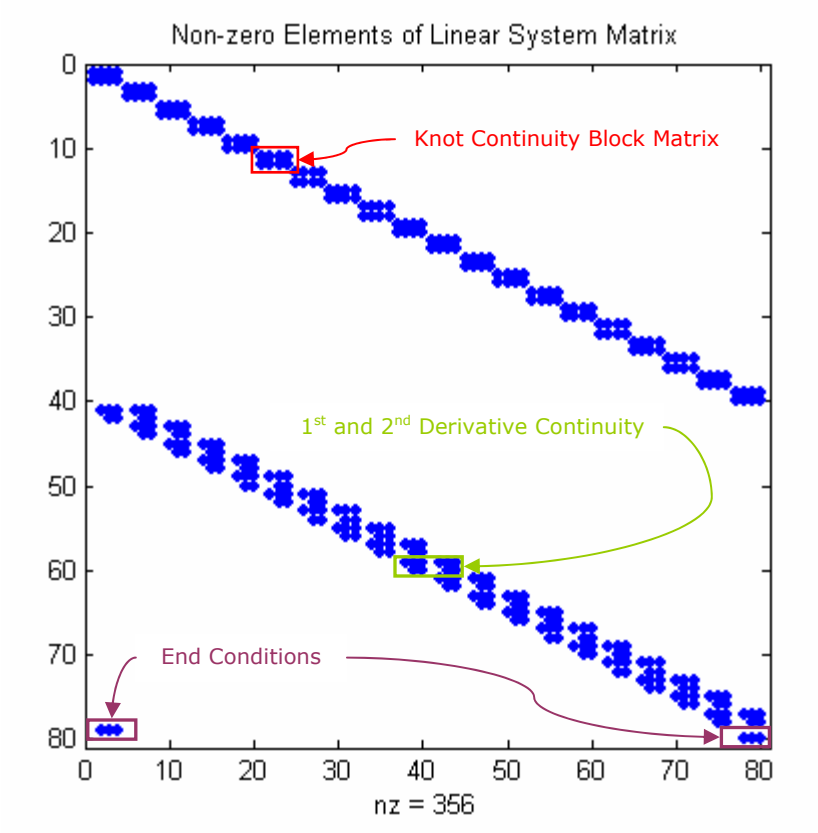
One block representative of this pattern is circled on figure showing the non-zero elements of the **A** matrix.

To match the first and second derivatives at each interior knot another type of submatrix was generated and distributed into the overall system matrices. The first row of this block forces matching of the first derivatives, while the second row causes the second derivatives to match. Again, its distribution into the overall **A** matrix is noted below.

$$\begin{bmatrix} 0 & 1 & 2\cdot x_i & 3\cdot x_i^2 & 0 & 1 & 2\cdot x_{i+1} & 3\cdot x_{i+1}^2 \\ 0 & 0 & 2 & 6\cdot x_i & 0 & 0 & 2 & 6\cdot x_{i+1} \end{bmatrix} \cdot \begin{Bmatrix} a_i \\ b_i \\ c_i \\ d_i \\ a_{i+1} \\ b_{i+1} \\ c_{i+!} \\ d_{i+1} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

Finally, the end conditions are set by another type of matrix block. This forces the first derivatives at each end to zero. Its dispersion into the overall **A** matrix is noted also.

$$\begin{bmatrix} 0 & 1 & 2\cdot x_0 & 3\cdot x_0^2 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 2\cdot x_n & 3\cdot x_n^2 \end{bmatrix} \cdot \begin{Bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ \vdots \\ a_n \\ b_n \\ c_n \\ d_n \end{Bmatrix}$$



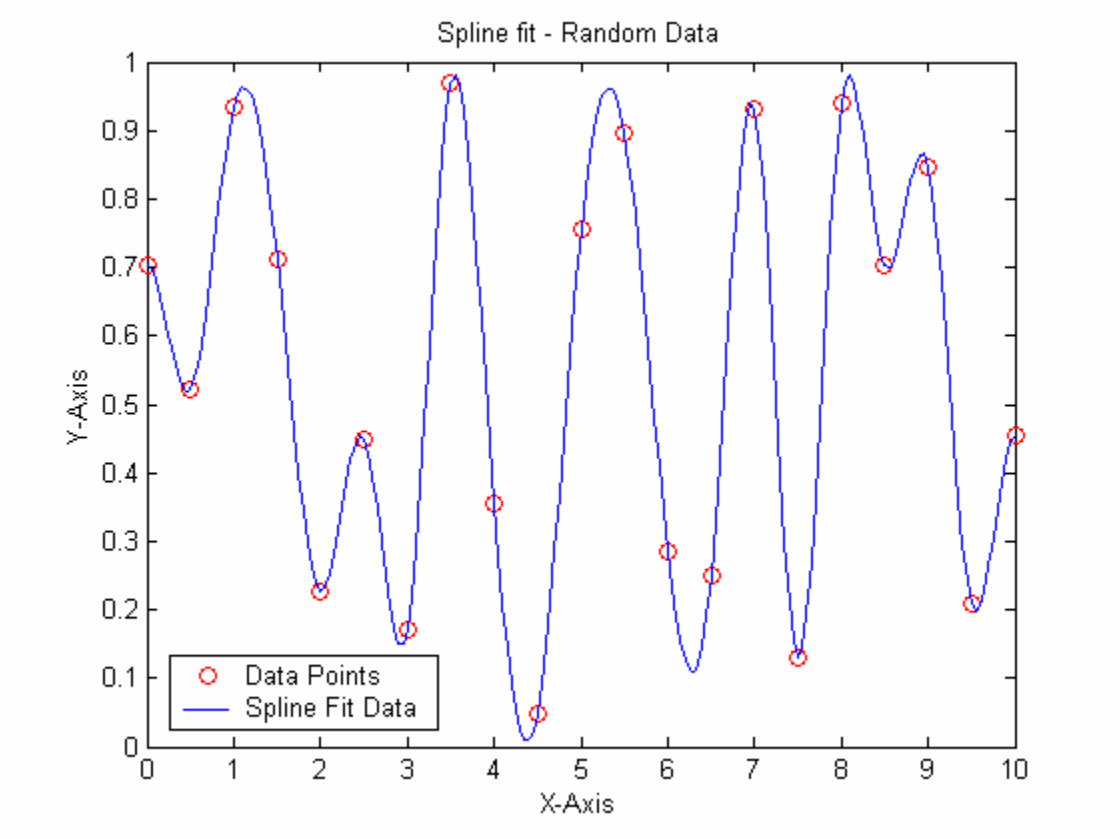
Testing the Algorithm

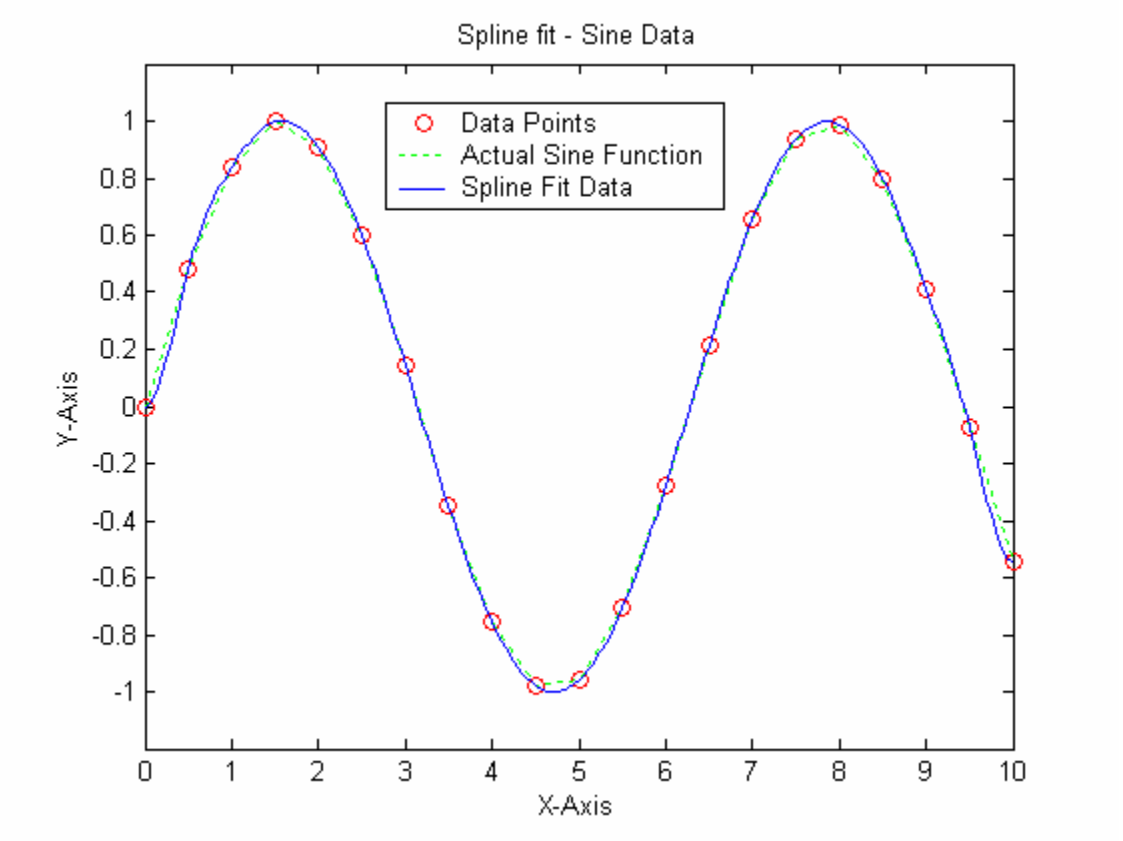
To make sure that the function correctly builds the linear system, the spy command was used. Since the example shown above was generated from a real test case it helps illustrate not only how the matrix was supposed to be put together, but also validates that in fact the matrix was built as prescribed.

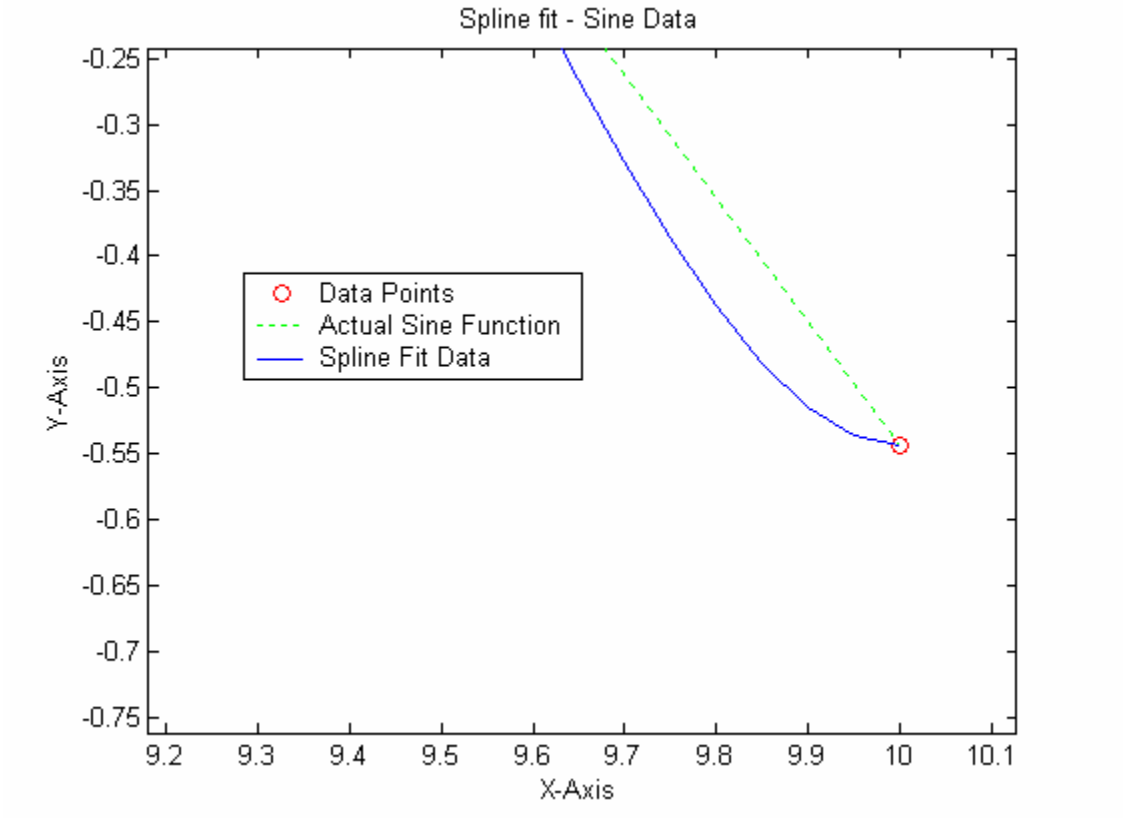
The algorithm was tested with both random and sinusoidal data with the following code.

```
>> x=[0:.5:10];
>> y=rand(length(x));
>> y2=sin(x);
>> y=rand(1,length(x));
>> [coeffs_r]=spline_morison(x,y);
>> [coeffs_s]=spline_morison(x,y2);
```

The attached m-file spline_test.m was used to create the following plots from the spline fitting results. Inspection of the random data results shows that the spline fit does indeed pass through all of the data points. Also, the derivatives are continuous at each data point, making very smooth transitions at each knot. The same is true of the sine function data. Also, the zoomed graph shows that the endpoint derivative is being forced to zero, while still passing through the data point. These results show that the function is indeed working correctly.




Spline fit - Sine Data
Data Points
Actual Sine Function
Spline Fit Data
Y-Axis
X-Axis
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6
7
8
9
10


Spline fit - Sine Data
Data Points
Actual Sine Function
Spline Fit Data
Y-Axis
X-Axis
-0.25
-0.3
-0.35
-0.4
-0.45
-0.5
-0.55
-0.6
-0.65
-0.7
-0.75
9.2
9.3
9.4
9.5
9.6
9.7
9.8
9.9
10
10.1

Matlab Code

```
function [a_comp]=poly_morison(x,y,n)
% Gus Morison
% MA 4160 - HW#7
% March 20, 2003
%
% This function takes a series of data points and fits a polynomial model to them
%
%           data points =>  (x0,y0), (x1,y1),(x2,y2), ... , (xm,ym)
%                model =>     a0 + a1*x + a2*x^2 + ... + an*x^n
% Syntax:
%      [a_comp]=poly_morison(x,y,n)
%
% Explanation of Arguments:
%  Input:
%      x - X coordinates of the ordered pairs (x,y)
%      y - Y coordinates of the ordered pairs (x,y)
%      n - order of the model to be fit
%
%  Output:
%     a_comp - the coefficients of the polynomial [a0,a1,...,an]

num_x=length(x);
[row_y,col_y]=size(y);
if col_y>row_y
   y=y';
end

num_y=max(col_y,row_y);

%-- Checking to make sure that the data and model order are compatible ----------------------
if num_x~=num_y
   disp('Number of X and Y coordinates does not match.  Function will terminate.')
   break
end

if num_x<=n
   en=num2str(n);
   otpt=strcat('Not enough data to fit model of order n=',en)
   disp(otpt)
   disp('Provide more points or reduce model order such that # of data points > n')
   disp('Function will terminate.')
   break
end


%-- Building the matrix of equations ----------------------------------------
Cap_X=ones(num_y,n+1);
for ii=1:num_y
 for jj=1:n
   Cap_X(ii,jj+1)=x(ii)^jj;
 end
end

%---- Solving the system for the coefficients ----------------------------------
a_comp=Cap_X\y;
```

```
function [coeffs]=spline_morison(x,y)
% Gus Morison
% MA 4160 - HW#7
% March 20, 2003
%
% This function takes a series of data points and cubic spline to them.
%   End conditons are handeled by forcing the first derivitive to zero.
%   First and Second derivitive continuity is set for each interior knot.
%
% Syntax:
%      [coeffs]=spline_morison(x,y)
%
% Explanation of Arguments:
%  Input:
%      x - X coordinates of the ordered pairs (x,y)
%      y - Y coordinates of the ordered pairs (x,y)
%
%  Output:
%     coeffs - the coefficients of the cubic polynomials between each knot
%
%                     coeffs=[a0,b0,c0,d0,a1,b1,c1,d1,..,an,bn,cn,dn]'
%
%


%-- Checking compatibility of the Data set ----------------------------------------------
num_x=length(x);
[row_y,col_y]=size(y);
if col_y>row_y
   y=y';
end

num_y=max(col_y,row_y);

if num_x~=num_y
   disp('Number of X and Y coordinates does not match.  Function will terminate.')
   break
end

n_eq=num_x-1;
v=n_eq*4;

%--------------- Building the Right Hand Side of the Eqn. --------------------------------
rhs=zeros(n_eq*4,1);
rhs(1)=y(1);
for ii=1:num_y-2
   rhs(2*ii:2*ii+1)=y(ii+1);
end
rhs(length(rhs)/2)=y(num_y);

%--------------- Building the A Matirx ---------------------------------------------------
A=zeros(n_eq*4,n_eq*4);
```

```
%--- The value equations ---------------------------------------
rae=1;
for jj=1:n_eq

   A(jj+(jj-1),rae:rae+3)=[1 x(jj) x(jj)^2 x(jj)^3];
   A(jj+(jj-1)+1,rae:rae+3)=[1 x(jj+1) x(jj+1)^2 x(jj+1)^3];
    rae=rae+4;
end

%--- Continuity (first and second derivative) at each knot ----------------------
pp=2;
for gg=(length(rhs)/2)+1:2:length(rhs)-2;

   A(gg:gg+1,(pp-2)*4+1:(pp-2)*4+8)=[0  1  2*x(pp) 3*x(pp)^2  0  -1  -2*x(pp) -
3*x(pp)^2;...
                          0  0   2     6*x(pp)    0   0    -2    -6*x(pp)];
   pp=pp+1;
end

%--- The end equations ----------------------------------------------
A(v-1,1:4)=[0  1  2*x(1) 3*x(1)^2];
A(v,v-3:v)=[0  1  2*x(num_x) 3*x(num_x)^2];

%--- Solving the system for the coefficients -------------------------
coeffs=A\rhs;
```

```
%spline_test.m


figure('color','white')
plot(x,y2,'ro',x,y2,'g:')
hold
n=length(x);
f_lim=n-1;
step=(x(2)-x(1))/10;
for ii=1:f_lim
    range=[x(ii):step:x(ii+1)];
 yy=polyval(flipud(coeffs_s((ii-1)*4+1:(ii-1)*4+4)),range);
 plot(range,yy)
end

hold

title('Spline fit - Sine Data')
xlabel('X-Axis')
ylabel('Y-Axis')
legend('Data Points','Actual Sine Function','Spline Fit Data')
```