Analysis of Interpolating Polynomials: Hermite vs. Spline

By: Adam Headley 24 November 2014

Objective:

Considering the function $f(x) = \cos(\pi(x^3 + x))$ on the interval $[0,2]$ with the number points $n = \left(\frac{2}{h}\right) + 1$, a free cubic spline polynomial over step size h can be computed by using Algorithm 3.4 from Burden's Numerical Analysis. Choosing the points $x_0 = 0, x_1 = h,\ x_2 = 2h,\ \ldots,\ x_{n-1} = 2 - h, x_n = 2$. Given n points, an interval in a vector $x = (x_0, \ldots, x_n)$, a set of corresponding constants a from function outputs $f(x) = a = (a_0, \ldots, a_n)$, the program outputs vectors with corresponding coefficients b, c, and d for the spline polynomials

$$s(x) = s_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

with natural boundary conditions at the ends of our interval, corresponding to the second derivative of $f(x)$ at the end points being set to 0.

In the second part of this program the function $f(x)$ will be interpolated with a Hermite polynomial. This type of interpolation involves the use of the functions derivatives at the n points.

Program: Interpolating Spline Polynomial

The script, functions, and graphs were produced in MATLAB. The first code block is a function M-file is named **fspline.m**, then the script for which **fspline.m** is used to display five graphs that show the function $f(x)$ (blue), the spline (green), and the absolute error (red). The second function M-file is a function that produces a Hermite interpolating polynomial named **fhermite.m**. The results of both M-file functions are then compared.

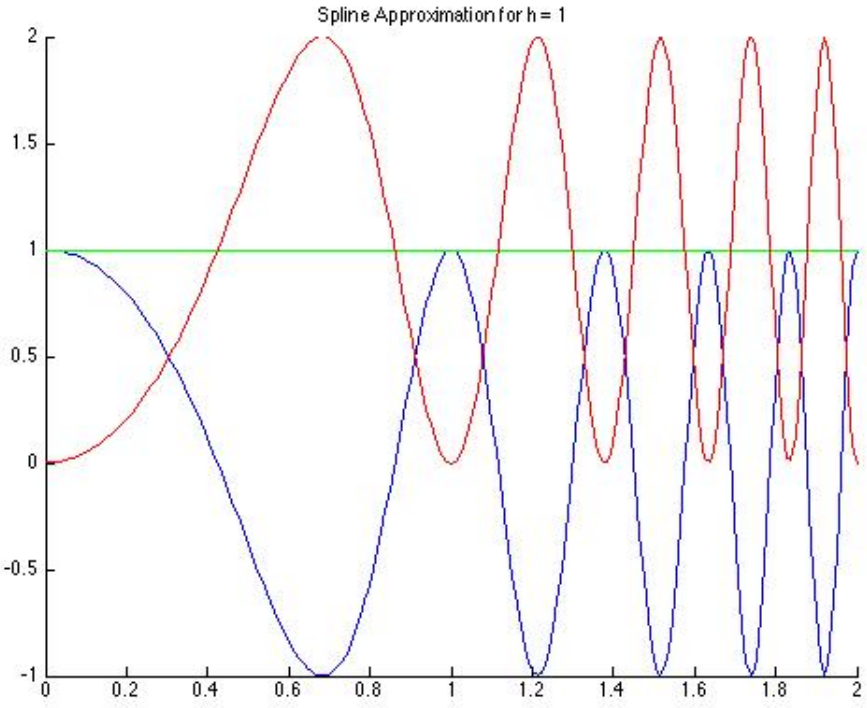
The code for **fspline.m** outputs the coefficients of the spline polynomials from n points with an adjustable step size h and the values of the function at those points by solving a decomposed triangular system of matrices. From these coefficients the script produces piecewise cubic splines connecting the points.

It is appropriate to use free boundary conditions in this case, instead of clamped boundary conditions because of the behavior of the function $f(x) = \cos(\pi(x^3 + x))$. This function's quickest oscillation exists around the endpoint $x = 2$, producing a

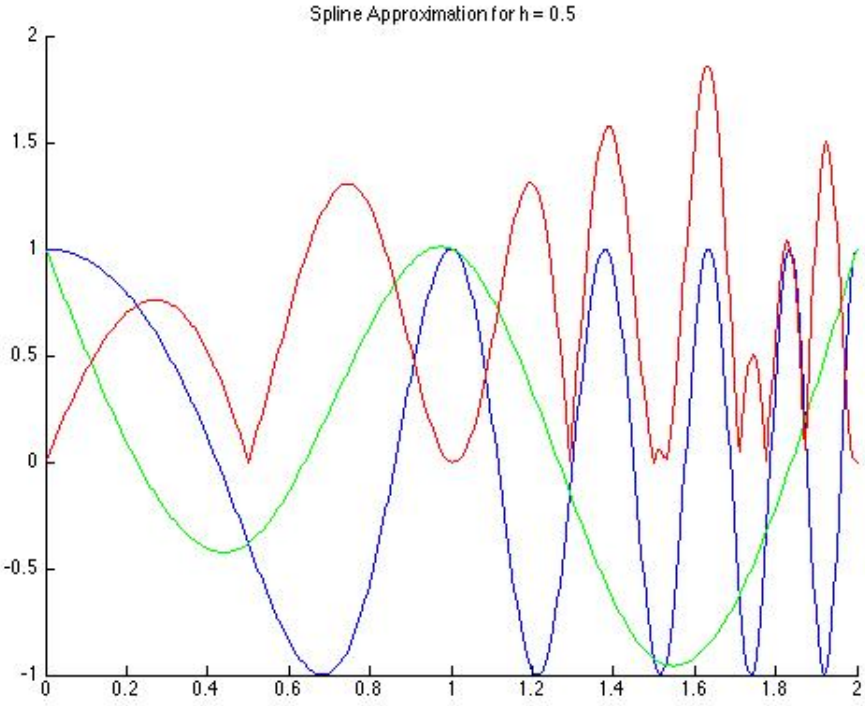
major source of error for the interpolating spline polynomial. Since the first derivatives at the end points are not predestined and the second derivative set to 0 at the end points, the spline has the self-determination it needs to account for slope and concavity as it approaches $x = 2$.

The following code only shows the script for $h = 1$. To see the other step sizes use the exact same code, only changing the initial value for h. The a values below are constants from function outputs $f(x_j) = a_j$, and the b, c, and d values are the unknown coefficients of the interpolating spline polynomials.

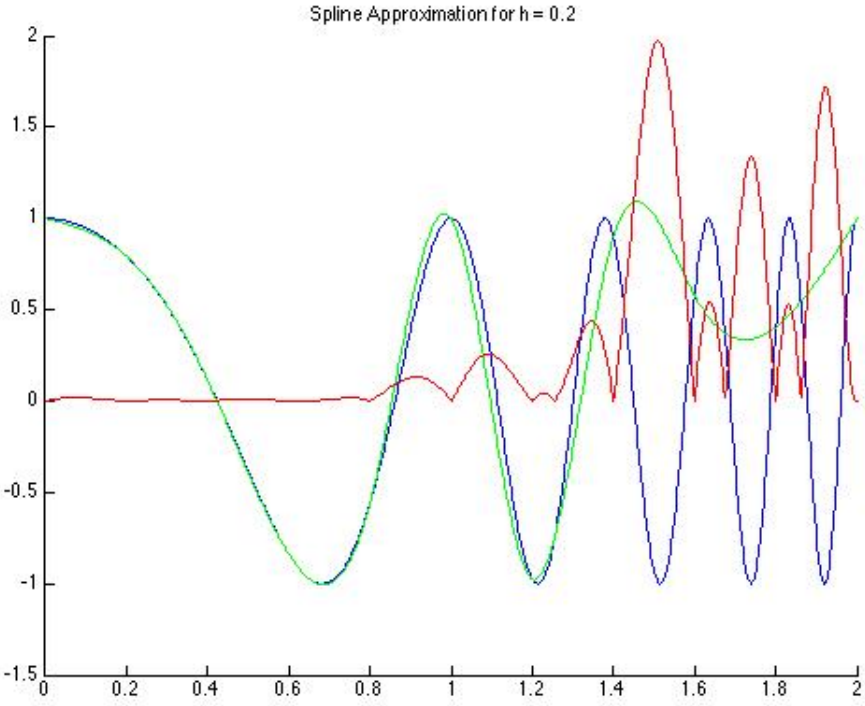
Below are five graphs comparing $f(x) = \cos(\pi(x^3 + x))$ to the interpolating spline polynomials corresponding to the five different steps sizes h. In each case, the function $f(x)$ is in blue, the interpolating spline polynomial s(x) is in green, and the absolute error in red. Below each graph is the max absolute error with a corresponding interval for x.



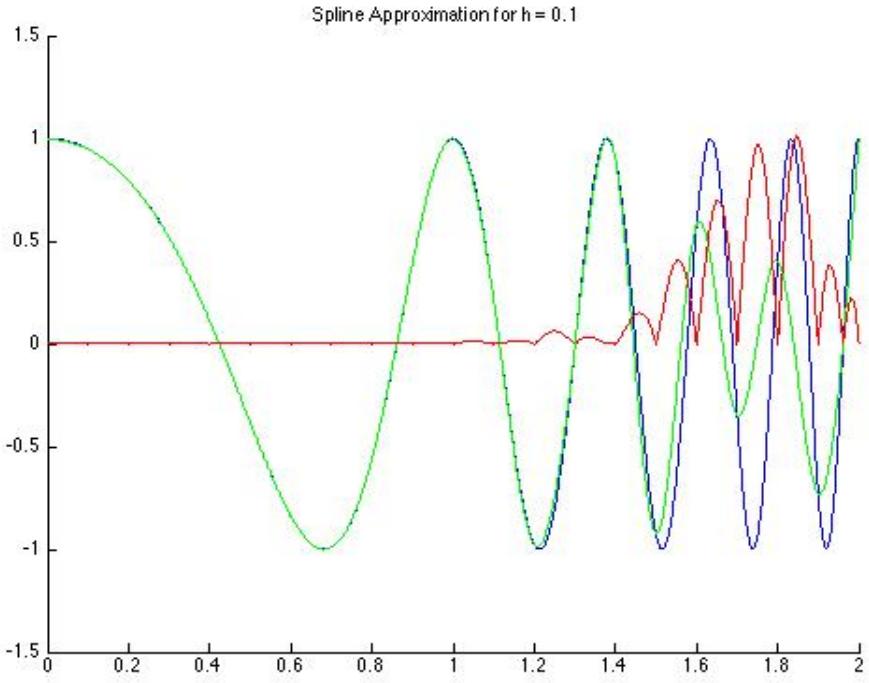
max absolute error: 2 with x in [0,2]



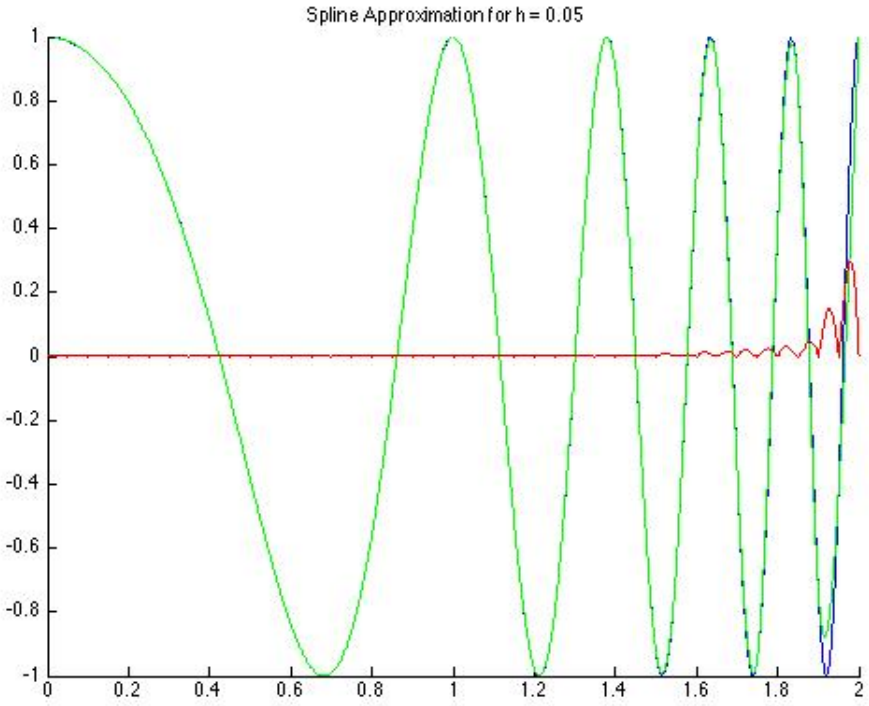
max absolute error: 1.858 with x in [1.5,2]



max absolute error: 1.974 with x in [1.4,1.6]



max absolute error: 1.010 with x in [1.8,1.9]



max absolute error: 0.2975 with x in [1.95,2]

Results and Error Analysis (Spline):

Using a uniform step size h makes for easy computation but this style of step size is not always the best for minimizing error. Notice the transition of error between step sizes. The error calms down at the start $x = 0$ but tends to increasingly isolate itself around $x = 2$. The increase of oscillations $f(x)$ yields about $x = 2$ causes steeper slopes and abrupt changes in concavity. The freedom corresponding to the first derivative of the interpolating spline polynomial's endpoints causes the polynomial trouble with maintaining a level relationship with the function but still intersecting at the associated n points. Decreasing the step size allows the behavior of the interpolating spline polynomial to be more level with the given functions since more points are used.

To form a bound for the error, observe Thm 3.13 from Burden's Numerical Analysis, $|f(x) - s(x)| = \left(\frac{5|f^{(4)}(\xi)|}{384}\right) \times max\left(x_{j+1} - x_j\right)^4$ for $\xi \in [0,2]$ and $0 \leq j \leq n-1$. The bound using the fourth derivative is $\left|f^{(4)}(\xi)\right| = 2.774758 \times 10^6$ at $x = 2$, however this explosion yields no essential information about how accurate the approximations is. Since most of the error for the most accurate case is in the interval $[1.95,2]$, that implies ξ resides there. The clamped spline polynomial has a maximum error of 3.6035 for $h = 0.1$ which is not as accurate as the natural spline polynomial's max error of 1.010 for $h = 0.1$.

Table 1: Fourth Derivative of $f(\xi)$

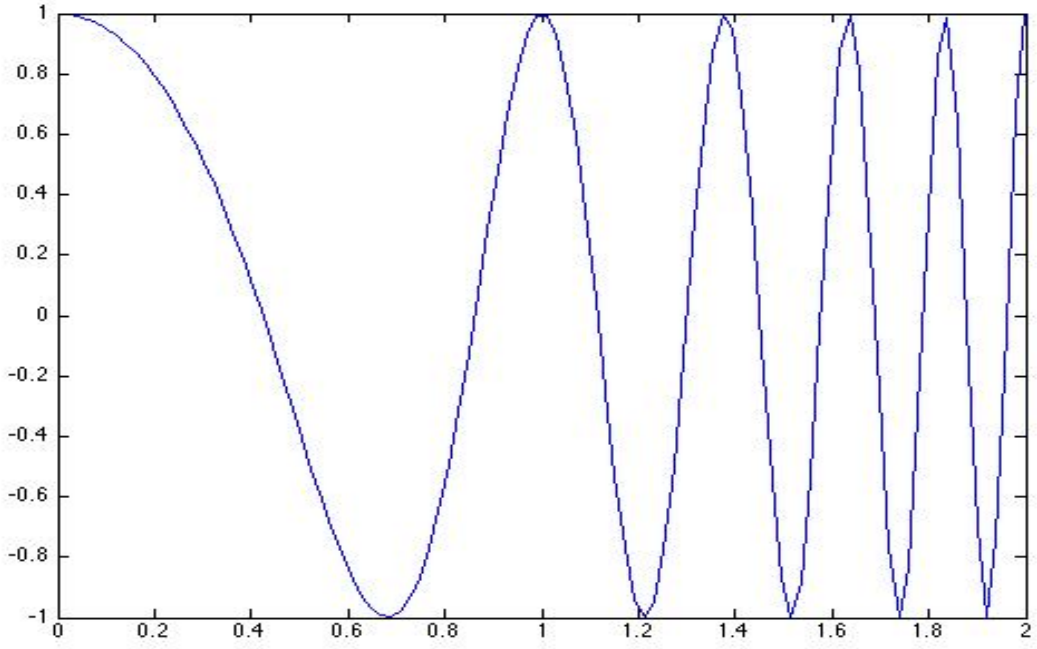
ξ	$f^{(4)}(\xi)$
1.9	-1.177944 x 10^6
1.95	-1.253286 x 10^6
1.975	1.04521 x 10^6
1.999999	2.774732 x 10^6
1.9999999	2.774755 x 10^6
1.99999999	2.774758 x 10^6
2	2.774758 x 10^6

Table 2: Absolute Error with corresponding Step Sizes

h	x Interval	Absolute Error
1	[0,2]	2
0.5	[1.5,2]	1.858
0.2	[1.4,1.6]	1.974
0.1	[1.8,1.9]	1.010
0.05	[1.95,2]	0.2975

To obtain an idea of a bound for the estimated error observe that the behavior of the interpolating spline polynomial between $y = -1$ and $y = 1$ compared to the given functions. Notice the polynomial always resides between them. This implies that the absolute error between the given function and the interpolating spline polynomial cannot exceed 2 in this interval with the given step sizes. Using $h = 1$ as the minimal case producing the most error.

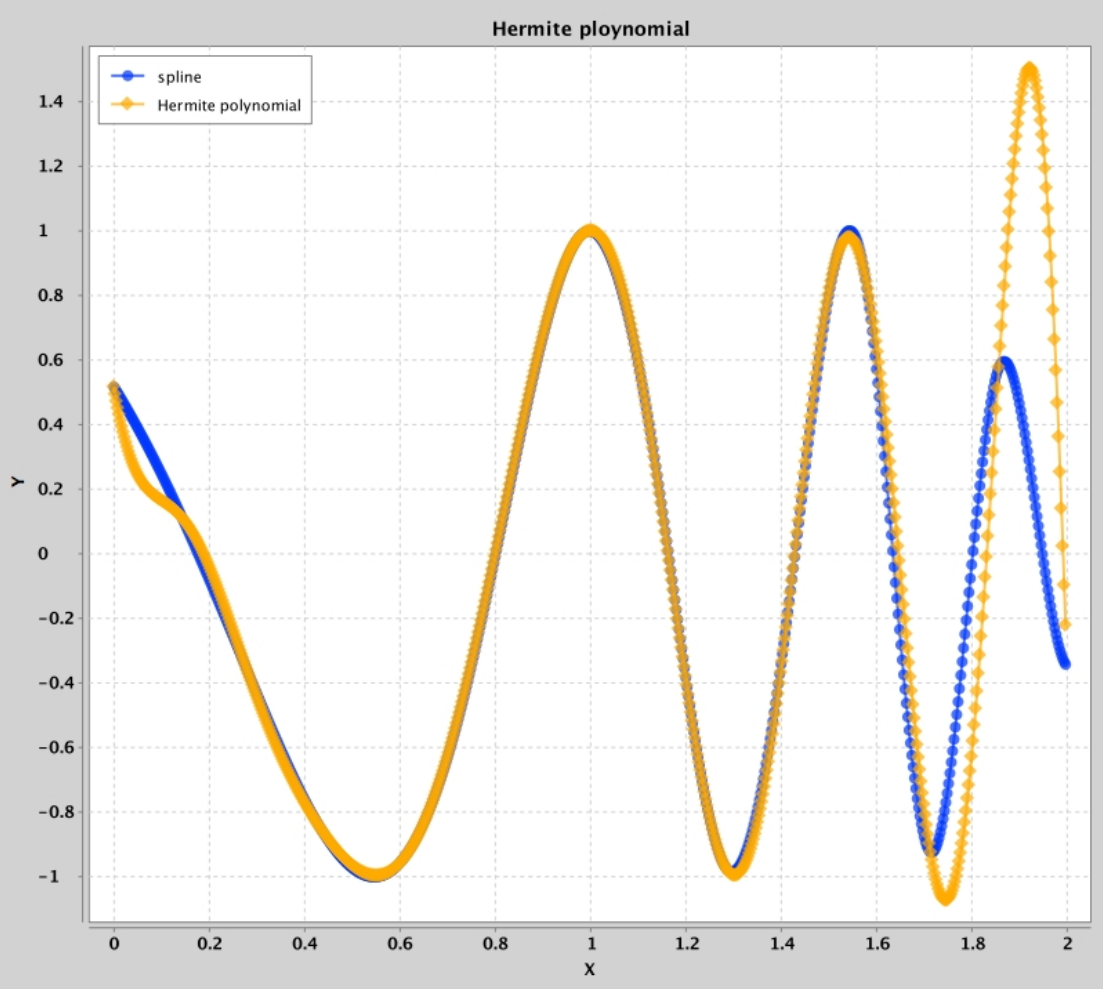
Reference the graph below for the following. Observing that most of the error occurs closer to $x = 2$, it would be appropriate to chose a non-uniform step size to obtain a better approximation. Notice that a cubic polynomial could at best cover one full oscillation due to its containing at most three distinct roots and looks very similar to $-\cos(x)$ for $x = [-\pi/2, 3\pi/2]$. By observing the graphs above closely, notice the greatest portion of the error exists in the interval [1.5,2]. This implies that the more oscillations that an interpolating spline polynomial covers the worse the approximation for $\cos(x)$ will become. To apply this concept to choosing five points, it is easier to consider h in respect to $y_i = x_i^3 + x_i$ instead of x adaptively placing most of the points in the high-erred interval $[1.5, 2]$. This is obvious because for $x > 1$, y increases considerably faster than x, causing more oscillations to occur in this interval. Choosing five points such that $\cos(\pi y_i) = \cos\left(\frac{\pi n}{2}\right)$. This implies that $y_1 = \frac{10}{4} = \frac{5}{2}$ since $y_4 = 10$. Therefore the rest of the y values should be $y_0 = 1, y_1 = \frac{5}{2}, y_2 = 5, y_3 = \frac{15}{2}, y_4 = 10$. By approximating the x_i values $x_0 = 0, x_1 = 1.11, x_2 = 1.51, x_3 = 1.79, x_4 = 2$.



Program: Interpolating Hermite Polynomial:

Using Algorithm 3.3 from Burden's Numerical Analysis an interpolating Hermite polynomial can be formed and then compared to the interpolating spline polynomial. After calculating the first derivatives of $f(x)$ at each x value, the interpolating Hermite polynomial can be constructed. The following code yields 42 coefficients for a 41 degree approximating polynomial.

Using the similar plot commands to for spline functions, the following graph compares Hermite (yellow) to spline (blue).



Results: Hermite vs. Spline:

The purpose of this programming assignment was to observe the end behavior for different types of interpolation. Above explains the error analysis for the interpolating natural spline polynomial. For $h = 0.1$ the Hermite polynomial produces a fairly good estimate as long as the endpoints are left out because it explodes to values around 100 when really close to them, within 10^{-6} distance of it. The graph does not show it is because the endpoints were removed to observe the general behavior from $(0,2)$. This occurrence is not surprising due to a Hermite polynomial because the use of first derivatives, similar to interpolating clamped spline polynomials behavior at the endpoints.

Conclusion:

In the end, each case generated a high error on the right sides of the intervals. The max error values for $h = 1$, $h = 0.5$, and $h = 0.2$ step sizes all had similar max absolute errors, but each smaller iteration of h produces an interpolating natural spline polynomial more flush with the given function $f(x)$. The smaller the step size h, the bigger n, the more points, the more control the spline has on the behavior of these oscillations because they become increasingly more frequent as x increases. This makes it difficult to develop a way to choose points, the step sizes would not be equal. A good method would be to guarantee the maximums and minimums interpolating polynomial lines up with the given function's maximums and minimums. Or cleverly distribute the points according to each oscillation, as explained on page 9 of this report.

Analysis

```
function [ b,c,d ] = fspline( n,x,a )
% fspline To construct the cubic spline interpolant S for the function f,
%       defined at the numbers x(0) < x(1) < ··· < x(n),
%       satisfying S''(x(0)) = S''(x(n)) = 0
% This is a direct implimentation of Algorithm 3.4 from Burden's % Numerical Analysis
%
%  INPUT n;
%            x(1), ..., x(n);
%            a(0) = f(x(0)), a(1) = f(x(1)), ..., a(n) = f(x(n))
%
%  OUTPUT a(j), b(j), c(j), d(j) for j = 0, 1, ..., n-1
%
%  Note: S(x) = S(j)(x)
%  S(j)(x) = a(j) + b(j)(x - x(j)) + c(j)(x - x(j))^2 + d(j)(x - x(j))^3
%  for x(j) < x < x(j+1)
%
%
%  Step 1: For i = 0, 1, ..., n-1
%            set h(i) = x(i+1) - x(i)
%
%  Step 2: For i = 1, 2, ..., n-1
%            set A(i) = 3/h(i)(a(i+1) - a(i)) - 3/h(i-1)(a(i) - a(i-1))
%
%  Step 3: Set l(0) = 1;
%            (Steps 3, 4, 5, and part of Step 6
%            solve a tridiagonal linear system
%            using a method described in Algorithm 6.7.)
%         Set u(0) = 0;
%         Set z(0) = 0;
%
%  Step 4: For i = 1, 2, ..., n-1
%            set l(i) = 2(x(i+1) - x(i-1)) - h(i-1) u(i-1);
%                u(i) = h(i)/l(i);
%                z(i) = (A(i) - h(i-1) z(i-1))/l(i);
%
%  Step 5: Set l(n) = 1;
```

```
%             z(n) = 0;
%             c(n) = 0;
%
%   Step 6: For j = n-1, n-2, ..., 0
%             set c(j) = z(j) - u(j) c(j+1);
%                 b(j) = (a(j+1) - a(j))/h(j) - h(j)(c(j+1) + 2c(j))/3;
%                 d(j) = (c(j+1) - c(j))/(3h(j));
%   Step 7: OUTPUT (a(j),b(j),c(j),d(j) for j = 0, 1, ..., n - 1);
%         STOP

hi = zeros(1,n);

for i = 1:n-1
    hi(i) = x(i+1) - x(i);
end

A = zeros(1,n-1);

for i = 2:n-1
    A(i) = (3/hi(i)) * (a(i+1) - a(i)) - (3/hi(i-1)) * (a(i) - a(i-1));
end

l = zeros(1,n-1);
l(1) = 1;

u = zeros(1,n-1);
u(1) = 0;

z = zeros(1,n-1);
z(1) = 0;
z(n) = 0;

for i = 2:n-1
    l(i) = 2*(x(i+1) - x(i-1)) - hi(i-1) * u(i-1);
    u(i) = hi(i)/l(i);
    z(i) = (A(i) - hi(i-1) * z(i-1))/l(i);
end
```

```
b = zeros(1,n-1);
c = zeros(1,n);
d = zeros(1,n-1);

c(n) = 0;

for j = n-1:-1:1
    c(j) = z(j) - u(j) * c(j+1);
    b(j) = (a(j+1) - a(j))/hi(j) - hi(j) * (c(j+1) + 2*c(j))/3;
    d(j) = (c(j+1) - c(j))/(3*hi(j));
end

c = c(1:n-1);
end
```

%% Script

```
% change h for corresponding step sizes h = (1, 0.5, 0.2, 0.1, 0.05)
h = 1;  % this is the only line that changes in the script!

n = (2/h) + 1;

x = zeros(1,n);
x(1) = 0;
x(n) = 2;

for j = 2:n-1
    x(j) = x(j-1) + h;
end

% Consider the function
% f = cos(pi * (x.^3 + x));
% on the interval 0 ≤ x ≤ 2

a = zeros(1,n);
for j = 1:n
    a(j) = cos(pi*(x(j).^3 + x(j)));
end
```

```
% natural cubic spline function
[ b,c,d ] = fspline( n,x,a );

for i = 1:n-1
    % spline polynomials on respective interval
    X(i,:) = linspace(x(i),x(i+1));
end

figure
hold on

title(['Spline Approximation for h = ', num2str(h)])

for i = 1:n-1
    % plotting function (BLUE)
    plot(X(i,:), cos(pi*(X(i,:).^3 + X(i,:))), 'b');

    % plotting spline approximation (GREEN)
    plot(X(i,:), a(i) + b(i).*(X(i,:) - x(i)) + c(i).*(X(i,:) - x(i)).^2 + d(i).*(X(i,:) -
x(i)).^3, 'g');

    % plotting error (RED)
    plot(X(i,:), abs(cos(pi*(X(i,:).^3 + X(i,:))) - (a(i) + b(i).*(X(i,:) - x(i)) + c(i).*(X(i,:)
- x(i)).^2 + d(i).*(X(i,:) - x(i)).^3)), 'r');
end


function [ Q ] = fhermite( x,a,d )
% fhermite Summary of this function goes here
%   INPUT numbers x(0),x(1), ..., x(n);
%       values  f(x(0)), ...,f(x(n))
%               f'(x(0)), ..., f'(x(n))
%
%    OUTPUT numbers Q(0,0), Q(1,1), ..., Q(2n+1,2n+1)
%           where H(x) = Q(0,0) + Q(1,1)(x − x(0)) + Q(2,2)(x − x(0))^2
%                   + Q(3,3)*(x − x(0))^2 * (x − x(1))
%                   + Q(4,4)*(x−x(0))^2 * (x−x(1))2 + ···
```

```
%                   + Q(2n+1,2n+1)*(x − x(0))^2 * (x − x(1))^2 * ···
%                   * (x − x(n−1))^2 * (x − x(n))
%
%   Step 1: For i = 0, 1, ..., n
%               do Steps 2 and 3
%
%   Step 2: Set z(2i) = x(i);
%               z(2i+1) = x(i);
%               Q(2i,0) = f(x(i));
%               Q(2i+1,0) = f(xi);
%               Q(2i+1,1) = f'(x(i)).
%
%   Step 3: If i ~= 0
%           Set Q(2i,1) = (Q(2i,0) − Q(2i−1,0))/(z(2i) − z(2i−1))
%
%   Step 4: For i = 2, 3, ..., 2n+1
%               for j = 2, 3, ..., i
%                   set Q(i,j) = (Q(i,j−1) − Q(i−1,j−1))/(z(i) - z)
%
%   Step 5: OUTPUT (Q(0,0), Q(1,1), ..., Q(2n+1,2n+1);
%               STOP

h = 0.1;
n = 2/h+1;

Q = zeros((2*n),(2*n));

for i = 1:n-1
  x(i+1) = x(i) + h;
end

for i = 1:n
  a(i) = cos(pi*(x(i).^3 + x(i)));
end

% first derivatives of f(x)
for i = 1:n
  d(i) = -1*pi*(3*x(i).^2 + 1)*sin(pi*(x(i).^3 + x(i)));
```

```
end

end
```

This next block of code is the script for running fhermite.m.

```
%%
% Construct the Hermite polynomial
% Interpolating the data:
% (x(j),f(x(j)),f'(x(j)) for j = 0, ..., n
%  n = (2/h)+1 for h = 0.1

h = 0.1;
n = 2/h+1;

x = zeros(n,1); % x values
a = zeros(n,1); % f(x) values
d = zeros(n,1); % derivatives at f(x) values
[ Q ] = fhermite( x,a,d );

z = zeros((2*n),1);
% Hermite coefficients
for i = 1:n
    z(2*i-1) = x(i);
    z(2*i) = x(i);
    Q((2*i-1),1) = a(i);
    Q((2*i),1) = a(i);
    Q((2*i),2) = d(i);

    if i ~= 1
        Q((2*i-1),2) = (Q((2*i-1),1)-Q((2*i-2),1)) / (z(2*i-1) - z(2*i-2));
    end
end

for i = 2:(2*n-1)
    for j = 2:i
       Q(i+1,j+1) = (Q(i+1,j) - Q(i,j)) / (z(i+1) - z(i+1-j));
    end
end
```