## ABSTRACT

**Numerical Interpolation and Data fitting are fundamental aspects in scientific computing in order to turn the data points into meaningful functions that can later be used to deduce important patterns and observe trends. Interpolation schemes are used in order to approximately fit the discrete set of points in a polynomial which can then be differentiated or integrated in accordance to the constraints and conditions required. One of the objectives of using interpolation methods is to not only fit these discrete points but to also to see that the errors in the approximately fitted data is minimum. This is ensured by Least -Square approximation methods. In this project, we are using piecewise cubic polynomial interpolation approach. The interpolation polynomial using this method is smooth, second-order differentiable and converges to a decreased actual value on increasing the number of data-points, unlike the polynomial interpolation where the errors and oscillations of the interpolant increases. We are also testing the numerical accuracy of the written code by analyzing the Runge function tested for a set of 5, 9, 17 and 33 equidistant points. To depict the reduction of error with increasing the number of equi-spaced points, we are plotting the logarithm of L2 error against the logarithm of length of subinterval. The program is then extended to a problem of parametric cubic spline which has repeated abscissa and ordinates which is expected to generate a circle of unit radius 9 points.**

## I. DESCRIPTION OF METHODOLOGY

Here, we want to use the spline which should have a global smoothness property with **Not-A-Knot** condition. Considering (n+1) data-points which form (n) sub-intervals over the given range of data-sets, each sub-interval has an estimation of a cubic polynomial passing through it which will be given by:

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \{x_i \leq x \leq x_{i+1}\}$$

From this equation, we have '4n' constants which will require '4n' set of equations. The interpolant passes through each and every given data-points thus generating 2 equations per interval, hence '2n' equations in total. Since the Cubic splines are 2nd order differentiable and that gives '2(n-1)' equations more. The remaining two equations are derived from the global smoothness property of Not-A-Knot condition.

### *A. Formulation of Cubic Spline*

The following steps are followed to approximate a cubic spline.
We take the first and second derivative of Equation 8, we have;

$$s''_i(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2$$

Let 'h' denote the length of each subinterval given by the formula;

$$h_i = x_{i+1} - x_i \text{ , } i = 0, 1, ....n\text{-}1.$$

The first set of conditions for each interval is:

$$s_i(x) = f(x_i) \rightarrow a(x_i) = f(x_i)$$

$$s_i(x_{i+1}) = f(x_{i+1}) \rightarrow a_i + b_i h_i + c_i h^2_i + d_i h^3_i$$

$$b_i + c_i h_i + d_i h_i = f[x_i, x_{i+1}], \; i = 0,1..n\text{-}2$$

where, $f[x_i, x_{i+1}]$ is the Newton's divided difference.
The first and the second derivative conditions give,

$$s'_i(x_{i+1}) = s'_{i+1}(x_{i+1}) \rightarrow b_i + 2c_i h_i + 3d_i h^2_i = b_{i+1}, \; i = 0,1...n\text{-}2$$

$$s''_i(x_{i+1}) = s''_{i+1}(x_{i+1}) \rightarrow c_i + 6d_i h_i = c_{i+1}, \; i = 0,1...n\text{-}2$$

Hence, the coefficients $b_i$ and $d_i$ can be written as:

$$b_i = f[x_i, x_{i+1}] - h_i(c_{i+1} + 2c_i)/3, \; i = 0,1, ...n - 1$$

$$d_i = \{c_{i+1} - c_i\}/3h_i$$

and rearranging the above gives;

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_i = 3(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]), \; i = 1, 2, \ldots, n\text{-}1$$

## *B. Not-A-Knot Condition*

Constructing a Not-A-Knot conditioned cubic spline will close this system of equations. However, we need two more equations that we will get from the global smoothness criteria. The not-a-knot condition states that the third derivative at the first two and the last two intervals are equal, which says that

$$s'''_0(x_1) = s'''_1(x_1), \text{ and}$$
$$s'''_{n-2}(x_{n-1}) = s'''_{n-1}(x_{n-1}).$$

These two conditions lead to the following two extra equations

$$d_0 = d_1, \text{ and } d_{n-1} = d_{n-2}$$

On solving for $c_o$ & $c_n$ we obtain the following:

$$c_0 = \{(h_1 + h_0)c_1 - h_0 c_2\}/h_1,$$
$$c_n = \{(h_{n-1} + h_{n-2})c_{n-1} - h_{n-1}c_{n-2}\}/h_{n-2}$$

$$\{(h_1 + h_0)(h_0 + 2h_1)\}.\ c_1/h_1 + \{(h_1 + h_0)(h_1 \square h_0)\}.\ c_1/h_1 = r_1;$$

$$\{(h_{n-2} + h_{n-1})(h_{n-2} - h_{n-1})\}.\ c_{n-2}/h_{n-2}$$
$$+ \{(2h_{n-2} + h_{n-1})(h_{n-1} + h_{n-2})\}.\ c_{n-1}/h_{n-2} = r_{n-1}$$

Above equations are handy to solve for the tridiagonal matrix and ultimately reach to the values of coefficients $c_i$ .

## *C. Interpolation Error*

We should expect some errors to persist in the interpolated polynomial which is given by;

$$e = \int_{Ii} \{P_i(x) - f(x)\}dx$$

In order to evaluate this integral, we use Gauss Quadrature 4-point integration scheme. Since, the estimated polynomial is cubic, we can use 4 points to exactly determine the integral. The Gauss point formula and the Gauss points and weights are as listed in the table below.

$$\int_a^b f(x)dx = \sum_{i=1}^{N} w_i\, f(\xi_i)$$

| **Gauss Points** $\{\xi_i\}$ | **weights** $\{w_i\}$ |
|---|---|
| -0.8611363116 | 0.3478548451 |
| -0.3399810436 | 0.6521451549 |
| 0.3399810436 | 0.6521451549 |
| 0.8611363116 | 0.3478548451 |

Also, we have computed the L2 norm of the error and plotted the log-log plot of the error and length of sub-interval. Subsequently, using least squares a linear fit is created to approximate the order of accuracy from the slope of the line. In the above equation, m is the order of accuracy that we are supposed to obtain as '4'.

$$||e2|| = \sqrt{\sum_{i=1}^{no.of\,subint} \int (Pi(x) - f(x))^2 dx))}$$

$$\log(\,||e||_2\,) = c + m\ \log(h)$$

The spline construction and the error computation has been done in MATLAB, the programs for which are attached next. The next section shows how to implement MATLAB algorithm to approximate a cubic-spline.

## MATLAB Program to implement cubic spline interpolation using ordered discrete observation knots which has 'not-a-knot' boundary conditions.

```
clc;clear all

points = 100;
number = points + 1; %Number of data points (xi,yi) for i = 0,1,..,number-1

%% Taking examples of function f(x)= y = x^2 OR  y = 1./(1 + 25(x)^2);
x = 0:0.1:10; %Equal-spaced knots / Points
y1 = (x).^2; %Sample function for Question - 1
% y1 = 1./(1+ 25*(x1).^2); %Runge-phenomenon function for Question - 2

figure(1)
clf
grid on
plot(x,y1);title('Actual analytical function plot of (x) vs f(x)');

% Spline Interpolation method to evaluate same function f(x) = y = x^2
n = length(x);
x_range = [0 10];
xi = equal_sub_int(x_range,number);% .m file to get equal subintervals
xi = xi(:);

h = h_func(xi,number); %. m function file to calculate hi = x(i+1)-x(i)
A = CoefficientMatrix(h);%.m function file to get coefficient matrix in 'c'
r = zeros((number-2),1); % RHS of the eqn : A*c = r
% Calculating RHS Vector;
for i = 1:(number-2)
 r(i,:)= 3*((funct_of_x([xi(i+1),xi(i+2)]))-(funct_of_x([xi(i),xi(i+1)])));
end

X = For_Inverse(A,r);%. m function file to obtain solution of matrix system.
C = X(:); % values of coefficients 'c' except first and last

C_FirstRow = ((h(2) + h(1))*C(1) - h(1)*C(2))/h(2);

C_LastRow = ((h(length(h)-1) + h(length(h)-2))*C(length(h)-2)...
  - h(length(h)-1)*C(length(h)-2))/h(length(h)-2);
% Stacking the results
CE= [C_FirstRow;C;C_LastRow]; % coefficient 'c' of cubic polynomial

AE = []; % vector AE is the coeffiecient 'a' of the cubic polynomial
for i = 1:number-1
  AE(i) = y1(i);
end
AE = AE(:);

DE = zeros((number-1),1);
BE = zeros((number-1),1);
for i = 1:(number-1)
    DE(i) = (CE(i+1) - CE(i))/(3*h(i));
    BE(i) = funct_of_x([xi(i),xi(i+1)]) - (h(i)/(3))*(2*CE(i) + CE(i+1));
end

BE = BE(:); %vector BE is coeffiecient 'b' of the cubic polynomial
DE = DE(:); % vector DE is the coeffiecients 'd' of the cubic polynomial
CE = CE(1:length(CE)-1);
```

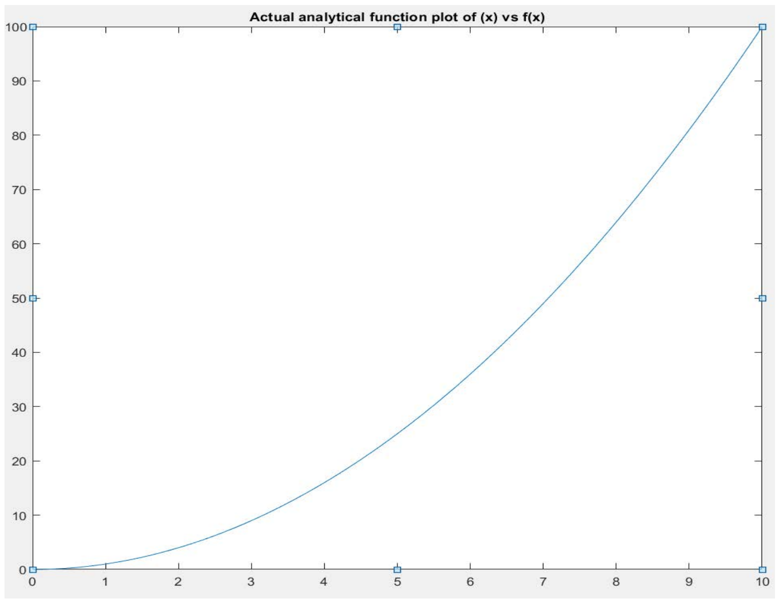
```
V = [];  %V is the complete cubic polynomial equation for x(i)=<x<=x(i+1)
for i = 1:n
  for j = 1:number-1
    if ((x(i) >= xi(j)) && (x(i) < xi(j+1)))
   %using relation: Si(x)=ai(x - xi) + bi(x-xi)^2 + ci(x-xi)^3 + di(x-xi)^3
      S = AE(j) + BE(j)*(x(i) - xi(j)) + CE(j)*(x(i) - xi(j))^2 ...
        + DE(j)*(x(i) - xi(j))^3;
      V = [V; S];
    end
  end
end
% Stacking last value of y1 manually since it is not computed in the loops.
V = [V;y1(length(x))];

figure(2)
grid on;hold on;plot(x,V,'r');
title('Cubic Spline Interpolation function plot of (x) vs f(x)');
```

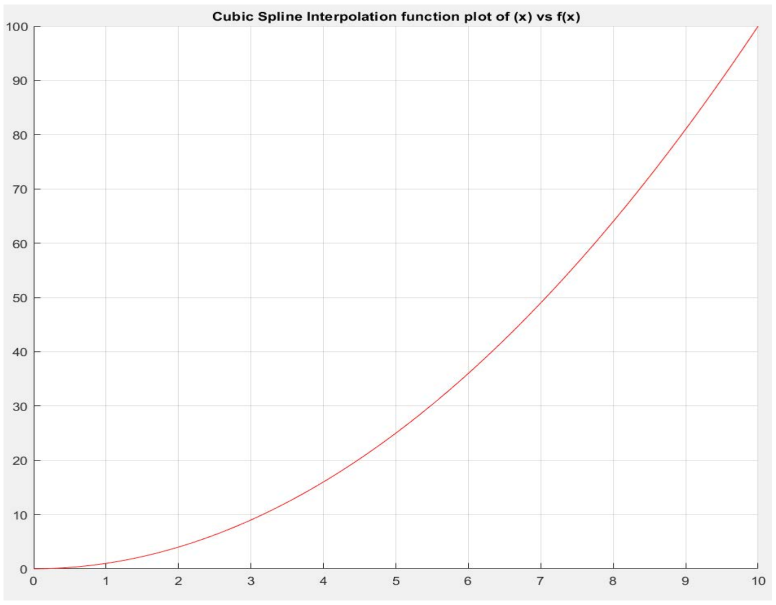
## Results:

**The plots obtained are displayed on the right. On comparison, the plot traced through Cubic Spline Interpolation program is similar to the one obtained from Analytical function plot.**

**Since we are employing 'not-a-knot' boundary condition, the Interpolated and Analytical plots will converge for any number of observation knots selected.**

**For the sake of study, in this question the equation studied is $f(x) = x^2$ .**

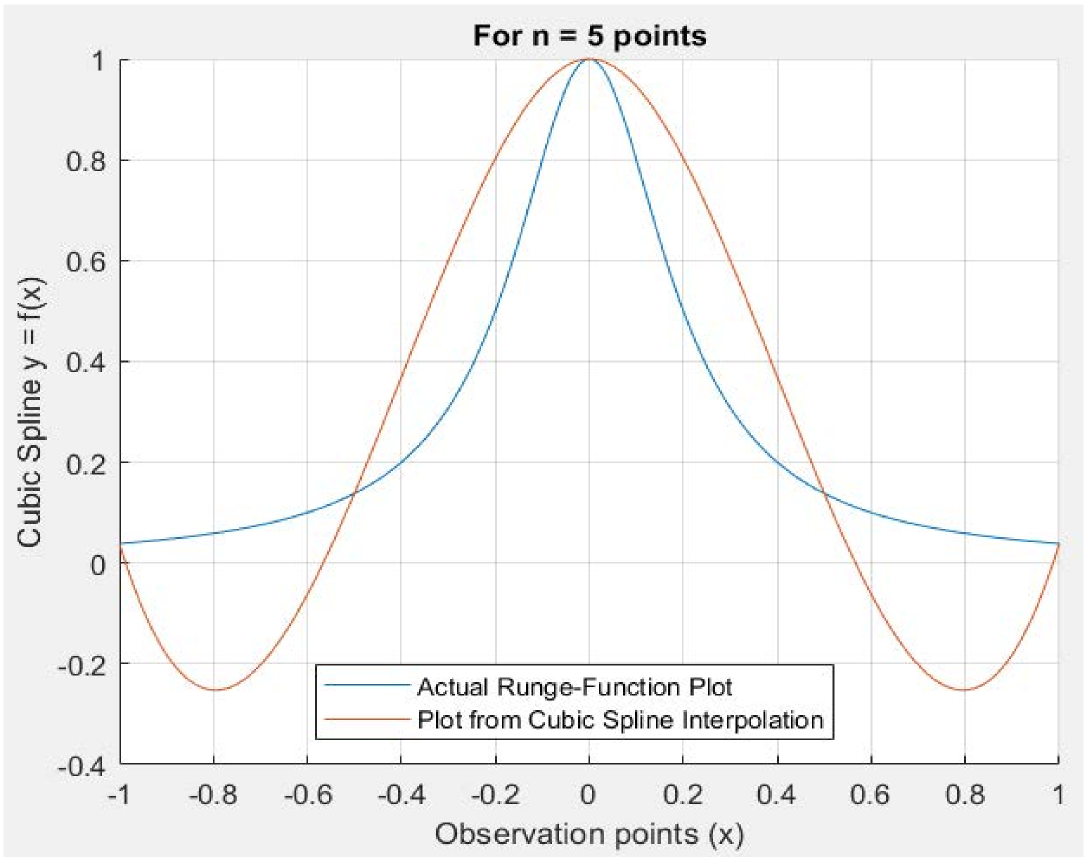


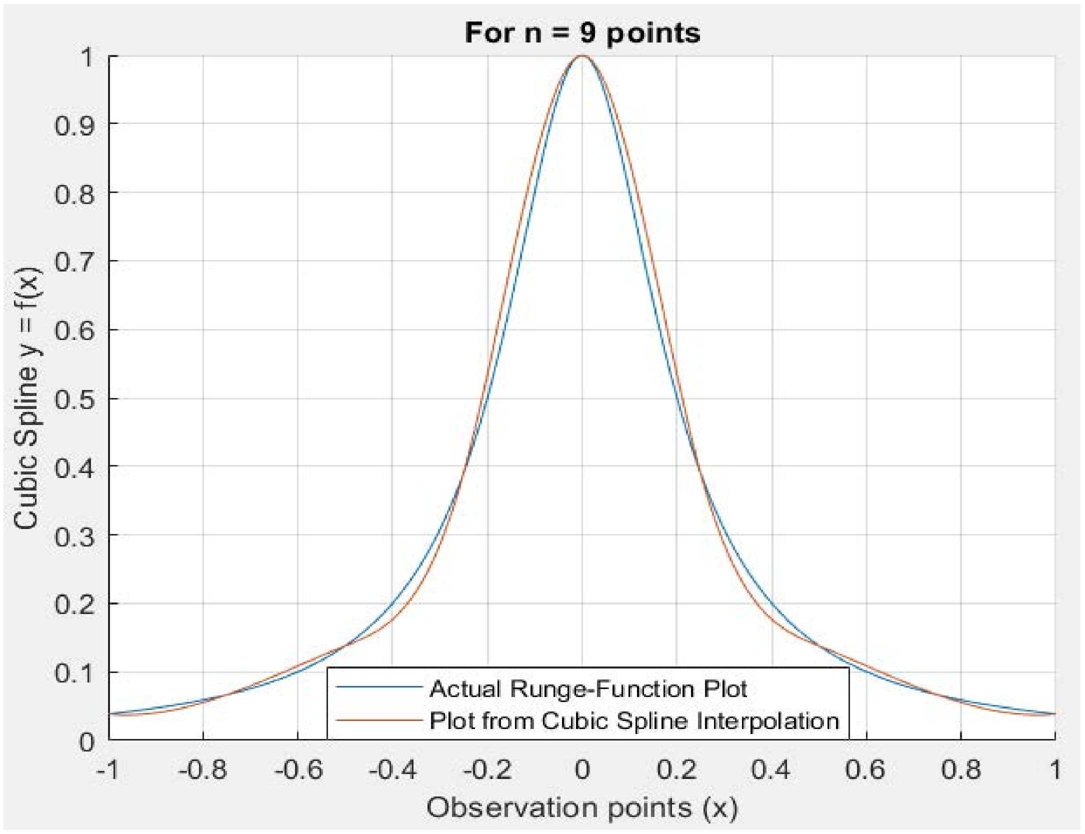


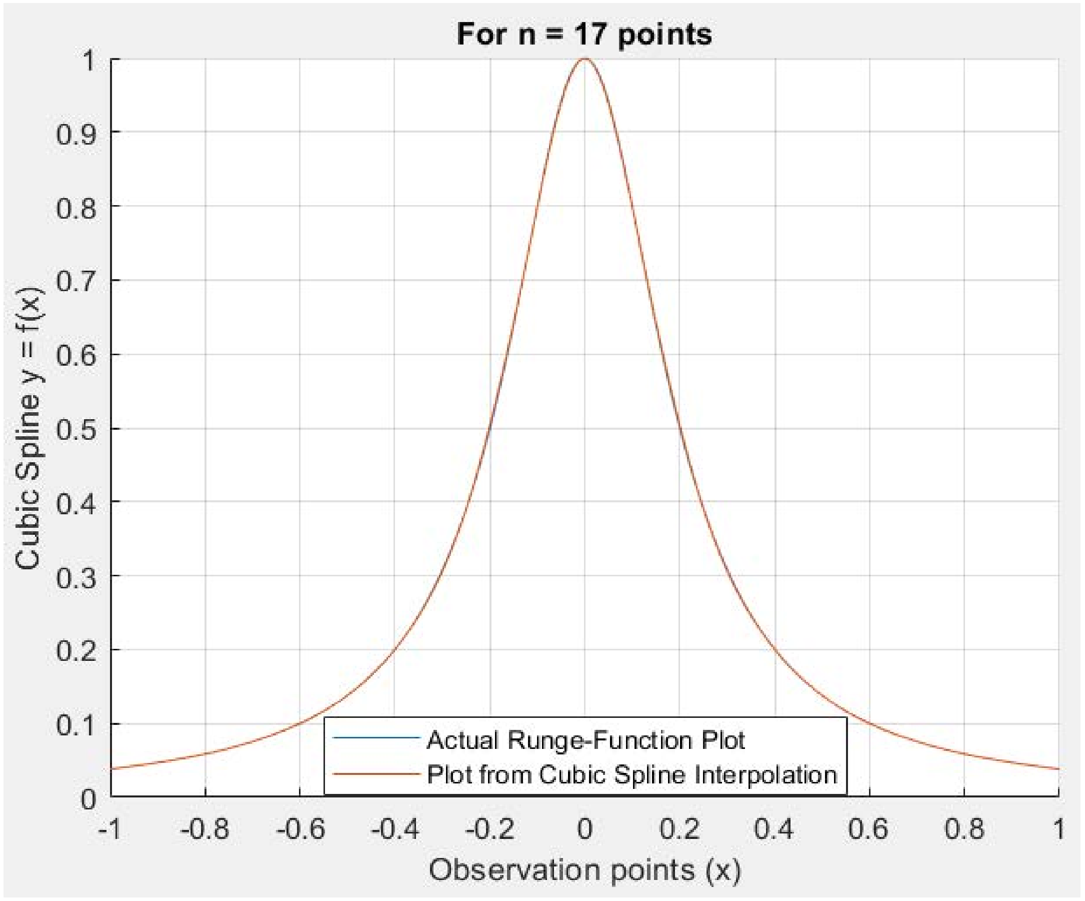
**MATLAB Program to observe Runge-Phenomenon, plotting the L2 error between Interpolating Cubic spline Polynomial and Actual Analytical Runge function, and also achieving the line of best-fit that represents the 4 points.**

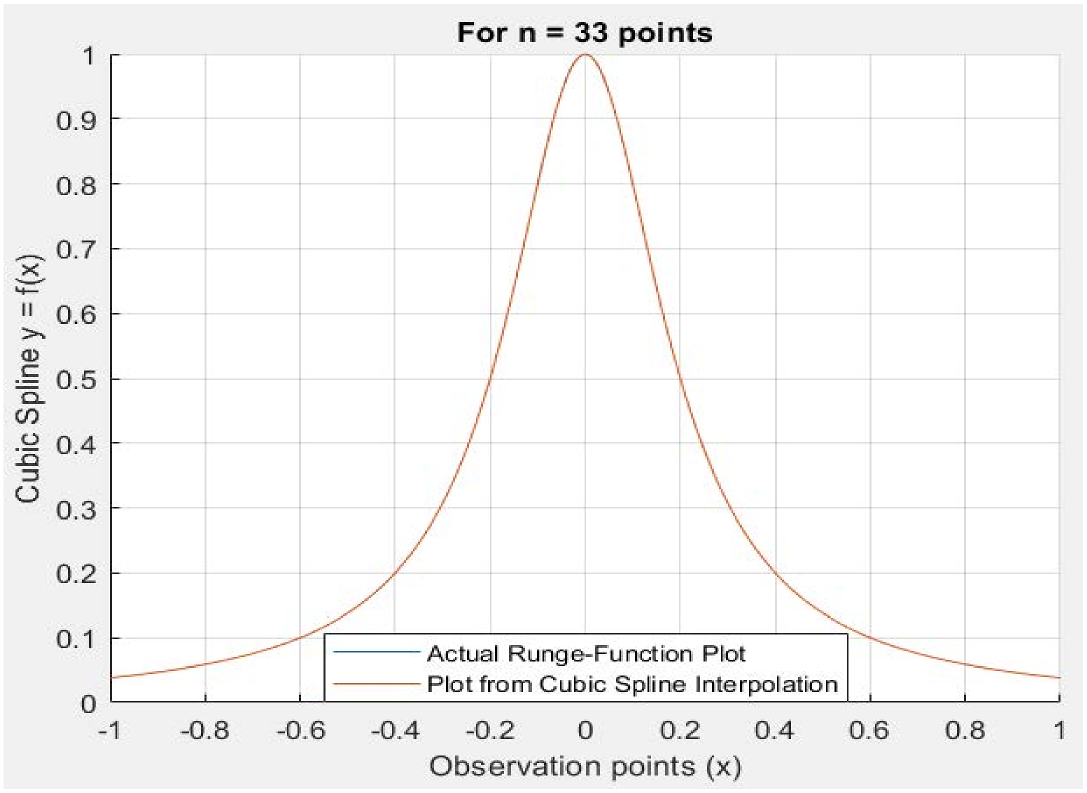
```
clc;clear all

number = [5 9 17 33];
x1 = -1:0.01:1;
y1 = myfun(x1);

%% Implementing Cubic Spline Interpolation
n = length(x1);
h = zeros(4,1);
e2 = zeros(4,1); % error
for U=1:length(number)
xrange = [-1 1];

x2 = equal_sub_int(xrange,number(U));
x2 = x2(:);

%% Solving the system of co-efficient matrix
H = h_func(x2,number(U));
A = CoefficientMatrix(H);
r = zeros((number(U)-2),1);
for i = 1:(number(U)-2)
     % RHS of the coefficient matrix system
    r(i,:) =  3*((funct_of_x([x2(i+1), x2(i+2)])) - (funct_of_x([x2(i), x2(i+1)])));
end

X = For_Inverse(A,r); %.m file to compute inverse and solve matrix system.
C = X(:);

C_FirstRow = ((H(2) + H(1))*C(1) - H(1)*C(2))/H(2);

C_LastRow = ((H(length(H)-1) + H(length(H)-2))*C(length(C)) - H(length(H)-1)*C(length(C)-1))/H ↙
(length(H)-2);
CE = [C_FirstRow;C;C_LastRow]; % Stacking results

AE = myfun(x2);
AE = AE(:);
AE = AE(1:length(AE)-1);

DE = zeros((number(U)-1),1);
BE = zeros((number(U)-1),1);
for i = 1:(number(U)-1)
    DE(i) = (CE(i+1) - CE(i))/(3*H(i));
    BE(i) = funct_of_x([x2(i),x2(i+1)]) - (H(i)/(3))*(2*CE(i) + CE(i+1));
end

BE = BE(:);
DE = DE(:);
CE = CE(1:length(CE)-1);

V = [];
for i = 1:n
  for j = 1:number(U)-1
     if ((x1(i) >= x2(j)) && (x1(i) < x2(j+1)))
        S = AE(j) + BE(j)*(x1(i) - x2(j)) + CE(j)*(x1(i) - x2(j))^2 + DE(j)*(x1(i) - x2(j))^3;
        V = [V; S];
     end
  end
end
V = [V; y1(length(y1))];

figure(U)
grid on;
hold on;
plot(x1,y1);
plot(x1,V);
legend('Actual Runge-Function Plot','Plot from Cubic Spline Interpolation')
xlabel('Observation points (x)');ylabel('Cubic Spline y = f(x)');hold off;

%% Gauss Quadrature Integration for 4 points to obtain L2_error.
SE = CubicSplPolynml(AE,BE,CE,DE,x2);
% For 4 number of points, there are 4 values of weights 'w' & nodes 'Kzi'
n_kzi = 4; % number of 'kzi' - nodes

% Weights 'w' & Nodes 'Kzi' for 4 points are;
w = [0.3478548 0.6521452 0.6521452 0.3478548]';
Kzi = [-0.86113631 -0.33998104 0.33998104 0.86113631]';

Xx = zeros(n_kzi,1);
F = zeros(n_kzi,1); F1 = zeros(n_kzi,1); F2 = zeros(n_kzi,1);
Vv = zeros(n_kzi,1);
product = zeros(n_kzi,1);
summ = 0;
for i = 1:number(U)-1

    for j = 1:n_kzi
    var1=(x2(i+1)-x2(i))/2;
    var2=(x2(i+1)+x2(i))/2;
    f1=(var1*(1/sqrt(3))+var2);
    f2=(var1*(-1/sqrt(3))+var2);
    error(j) = var1*((((1/(1+(25*(f1^2))))-(AE(i)+(BE(i)*(f1-x2(i)))+(CE(i)*((f1-x2(i)).^2))+(DE ↙
(i)*((f1-x2(i)).^3))))^2)+(1*((1/(1+(25*(f2^2))))-(AE(i)+(BE(i)*(f2-x2(i)))+(CE(i)*((f2-x2(i)) ↙
^2))+(DE(i)*((f2-x2(i)).^3))))^2));
    summ = summ + error(j);
    end
 end
summ;

e2(U) = log(sqrt(summ)); % L2 error formula

h(U) = mean(H,'all');
end

log_of_error = (e2);
log_subinterval = log(h);

%% Linear Least Squares Fitting procedure

Avg_sub_int = mean(log_subinterval,'all');
Avg_log_error = mean(log_of_error,'all');

% Calculation of slope 'm' of the line of best fit.
l1 = ((length(log_subinterval)).*sum((log_subinterval).*(log_of_error)));
```

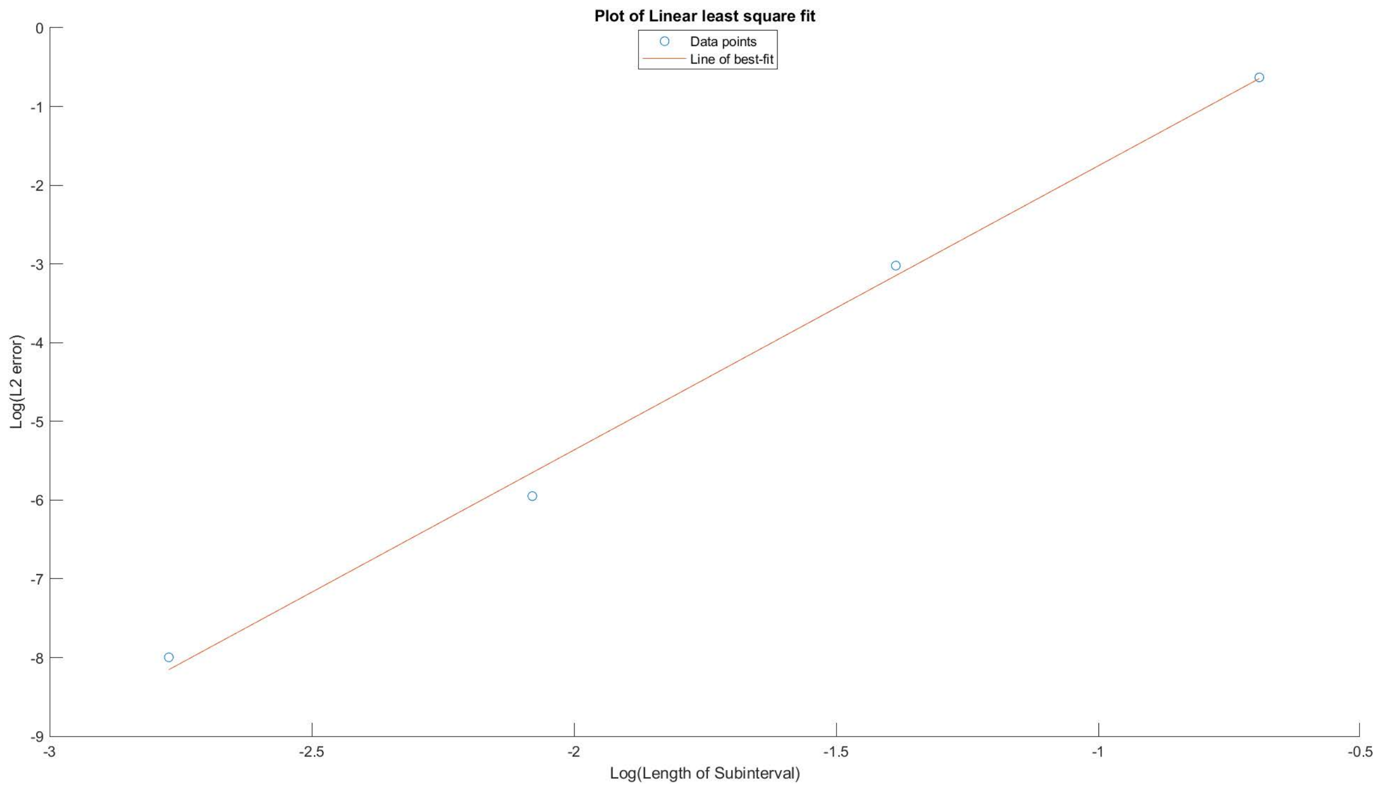
```
l2 = ((sum(log_subinterval)).*(sum(log_of_error)));
l3 = ((length(log_subinterval))*(sum((log_subinterval).^2)));
l4 = ((sum(log_subinterval)).^2);

m = (l1 - l2)/(l3 - l4)  %% Slope of the line
% % now we compute the intercept value
intercept = Avg_log_error - (m*Avg_sub_int);

LogH_plot = linspace(log_subinterval(1),log_subinterval(length(log_subinterval)),100);
LogH_plot = LogH_plot(:);

LogError_plot = [];
Eqn_Line = 0;
for i = 1:length(LogH_plot)
  Eqn_Line = m*LogH_plot(i) + intercept  % y = mx + C
  LogError_plot = [LogError_plot; Eqn_Line];
end
% Plot of log||e2||-vs-log||subinterval_size||, scatter plot of those 4 pts
figure(5)
hold on;
scatter(log_subinterval,log_of_error,'+');
plot(LogH_plot,LogError_plot)
xlabel('Log(Length of Subinterval)');ylabel('Log(L2 error)')
title('Plot of L2 error')
legend('Data points','Line of best-fit')
```




For n = 9 points
Cubic Spline y = f(x)
Observation points (x)
Actual Runge-Function Plot
Plot from Cubic Spline Interpolation
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1


For n = 17 points
Cubic Spline y = f(x)
Observation points (x)
Actual Runge-Function Plot
Plot from Cubic Spline Interpolation
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1





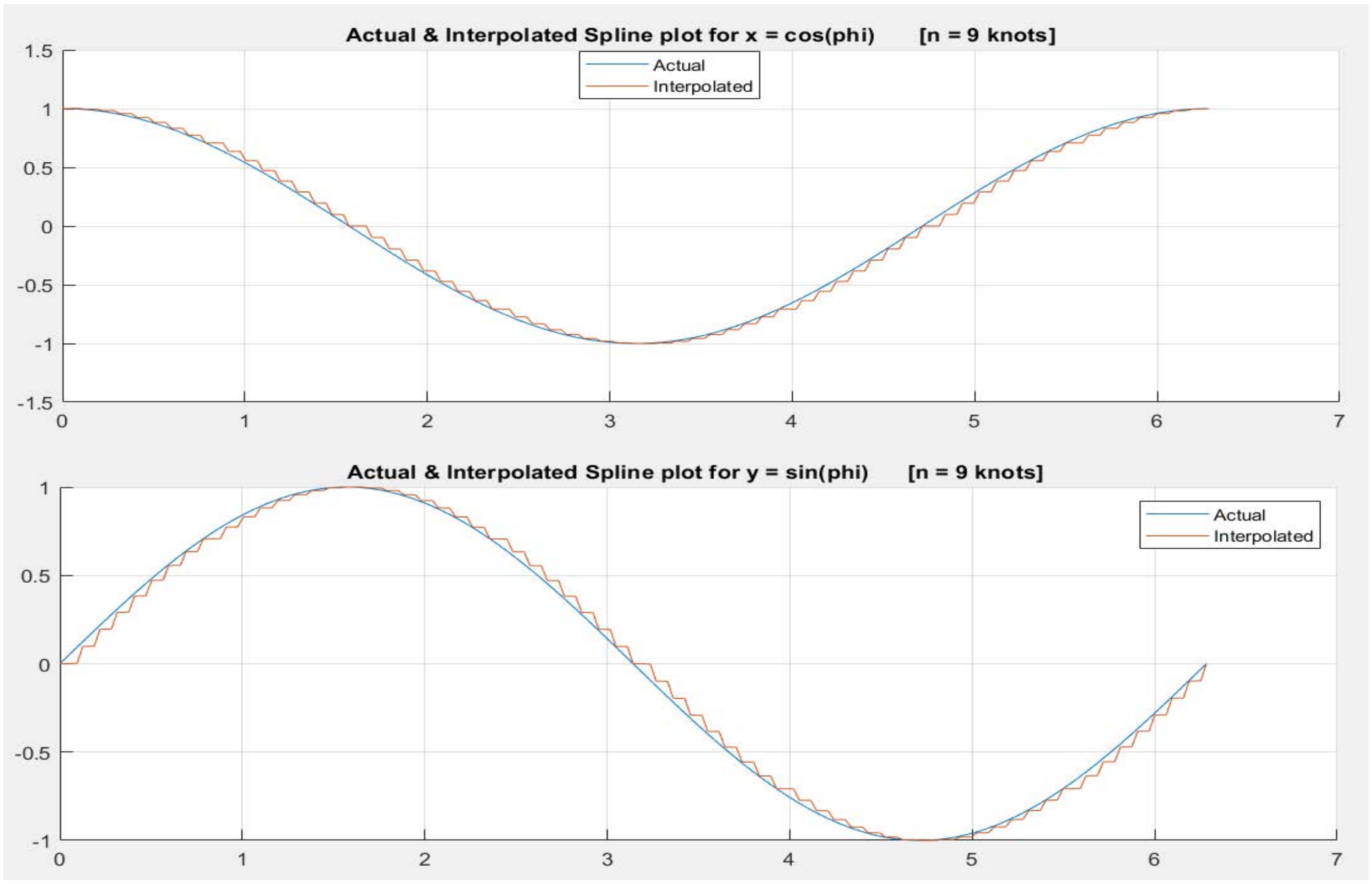
From observation of the L2 error plot versus the log of subintervals, we can estimate the slope of a line that exist if a linear fit is estimated to represent all these data point. From the order of accuracy, the value of slope is expected to be 4. However, due to ill-conditioned linear system of equations and errors in estimation of Gauss-Quadrature 4-point Integration, the slope of the best-fit line turns out to be 3.61 which can be considered to be close to 4.

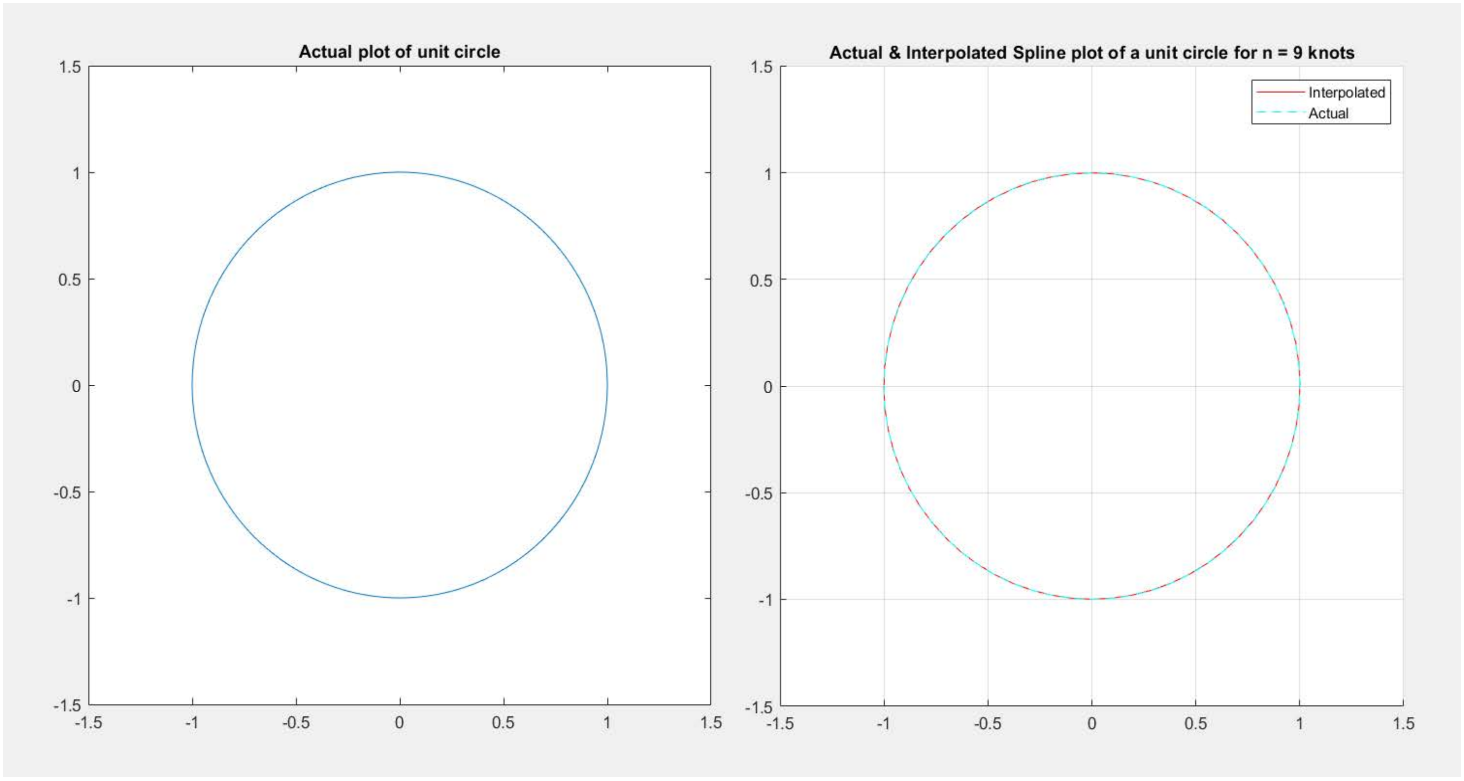
Hence, it can be said that the line of best-fit fairly approximate these 4 set of data-points.

## MATLAB Program to construct cubic spline for a set of data points which repeat on the abscissa and ordinates

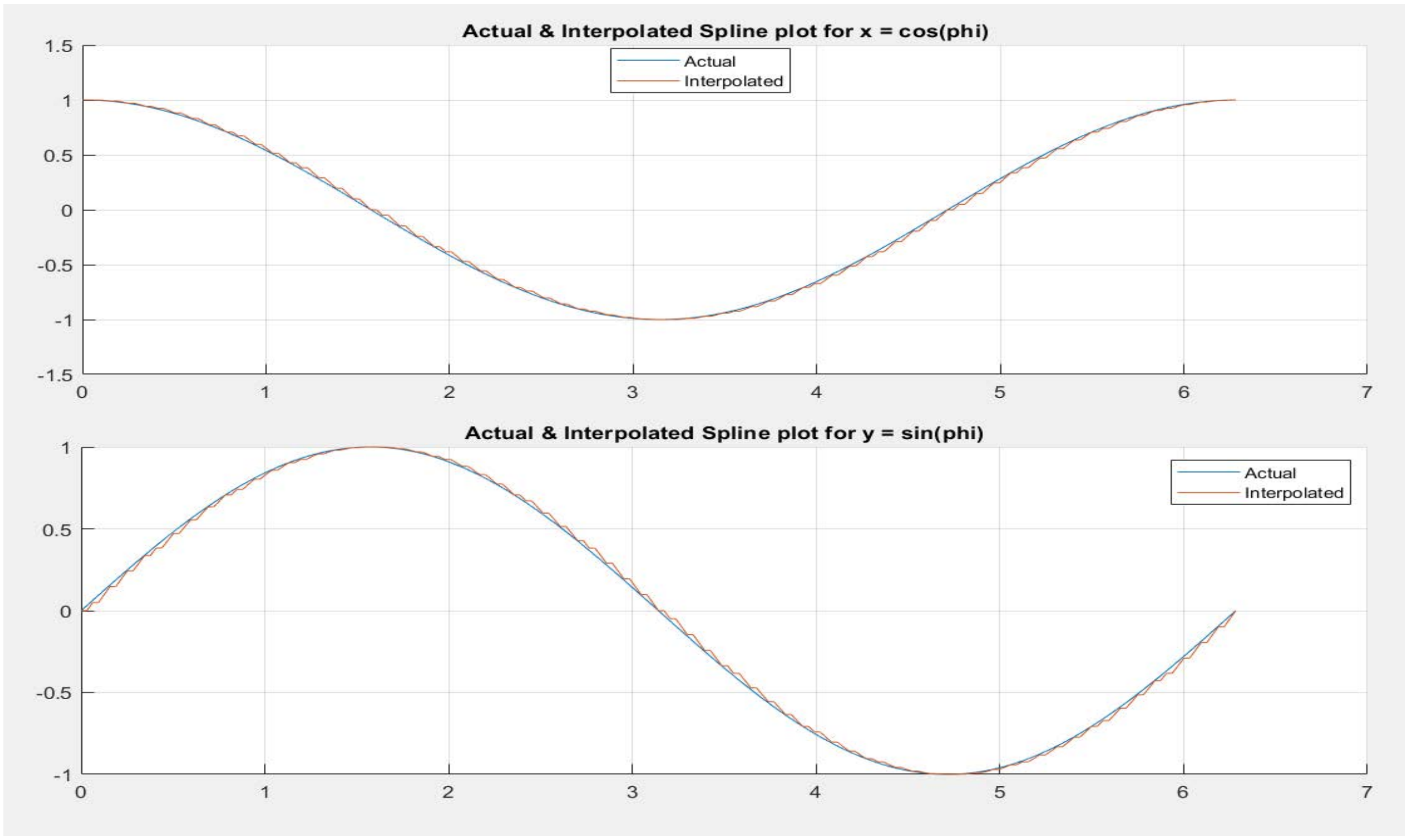
```
clc;clear all
clf
points = 16;
number = points + 1;

phi = [0:pi/100:2*pi];
x1 = cos(phi);
y1 = sin(phi);

V1 = []; V2 = [];
V1last = 0; V2last = 0; % because the f(phi) at start & end points are '0'.
n = length(phi);

phi_b = 0;
phi_a = pi/4;
phi_range = zeros(8,2); phi_i = zeros(8,number);
for j = 1:8

  phi_range(j,:) = [phi_b phi_a];
  phi_i(j,:) = equal_sub_int(phi_range(j,:),number);

  phi_b = phi_b + pi/4;
  phi_a = phi_a + pi/4;

  phi_i = phi_i(j,:);
  H = h_func(phi_i,number);
  A = CoefficientMatrix(H);
  r = zeros((number-2),1);
  for i = 1:(points-1)
    % 'r' is the RHS vector to coefficient matrix system,
    % 'funct_of_x' = .m file to compute f(x), like the Runge-function equation
    r(i,:) =  3*((funct_of_x_Try([phi_i(i+1), phi_i(i+2)])) - (funct_of_x_Try([phi_i(i), phi_i ↙
(i+1)])));

  end

  X = For_Inverse(A,r);
  C = X(:);

  C_FirstRow = ((H(2) + H(1))*C(1) - H(1)*C(2))/H(2);

  C_LastRow = ((H(length(H)-1) + H(length(H)-2))*C(length(C)) - H(length(H)-1)*C(length(C)-1))/H ↙
(length(H)-2);
  CE = [C_FirstRow;C;C_LastRow]; % Stacking results

  AE1 = cos(phi_i); % representing xi = cos(phi(i))
  AE1 = AE1(:);
  AE1 = AE1(1:length(AE1)-1);

  AE2 = sin(phi_i); % representing yi = sin(phi(i))
  AE2 = AE2(:);
  AE2 = AE2(1:length(AE2)-1);

  DE = zeros((number-1),1);
  BE = zeros((number-1),1);
  for i = 1:(number-1)
  DE(i) = (CE(i+1) - CE(i))/(3*H(i));
  BE(i) = funct_of_x_Try([phi_i(i),phi_i(i+1)]) - (H(i)/(3))*(2*CE(i) + CE(i+1));
 end
 BE = BE(:);
 DE = DE(:);
 CE = CE(1:length(CE)-1);

 for i = 1:n
  for j = 1:number-1
   %V is the complete cubic polynomial equation for x(i)=<x<=x(i+1)
   if ((phi_i(j) <= phi(i)) && (phi(i) < phi_i(j+1)))
    % Spline 'S1' is for spline in 'x = cos(phi)';
    % Spline 'S2' is for spline in 'y = sin(phi)';
    S1 = AE1(j) + BE(j)*(phi(i) - phi_i(j)) + CE(j)*(phi(i) - phi_i(j))^2 + DE(j)*(phi(i) - ↙
phi_i(j))^3;
    S2 = AE2(j) + BE(j)*(phi(i) - phi_i(j)) + CE(j)*(phi(i) - phi_i(j))^2 + DE(j)*(phi(i) - ↙
phi_i(j))^3;
    V1 = [V1; S1];
    V2 = [V2; S2];
   end
  end
 end
end
V1 = [V1;x1(length(x1))]; % Spline Polynomial for x = cos(phi - phi_i)
V2 = [V2;y1(length(y1))]; % Spline Polynomial for y = sin(phi - phi_i)

figure(1)
subplot(2,1,1)
grid on;hold on;% Plotting splines in x = cos(phi)
plot(phi,x1);
plot(phi,V1);title('Actual & Interpolated Spline plot for x = cos(phi)')
hold off;legend('Actual','Interpolated')

subplot(2,1,2)
grid on;
hold on;  % Plotting splines in y = sin(phi)
plot(phi,y1);
plot(phi,V2);title('Actual & Interpolated Spline plot for y = sin(phi)')
hold off;legend('Actual','Interpolated')

figure(2)
subplot(1,2,1)
plot(x1,y1);title('Actual plot of unit circle');
axis([-1.5 1.5 -1.5 1.5])
daspect([1 1 1])

subplot(1,2,2)
grid on;
hold on;
plot(V1,V2,'r',x1,y1,'--c') % Plotting unit circle
hold off;title('Actual & Interpolated plot of Spline of unit circle')
axis([-1.5 1.5 -1.5 1.5]);legend('Interpolated','Actual')
daspect([1 1 1])
```

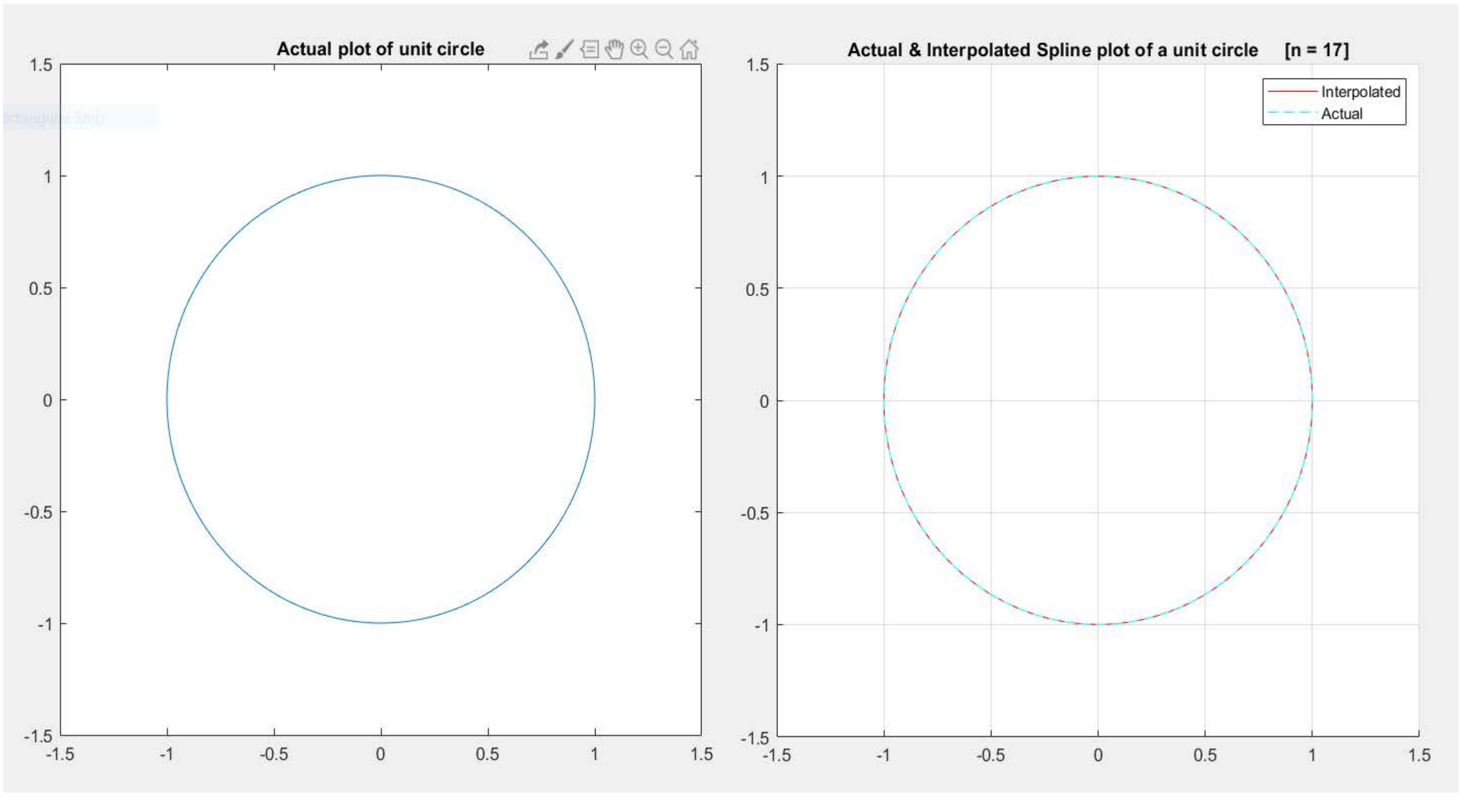
## Results obtained for n = 9 observation points



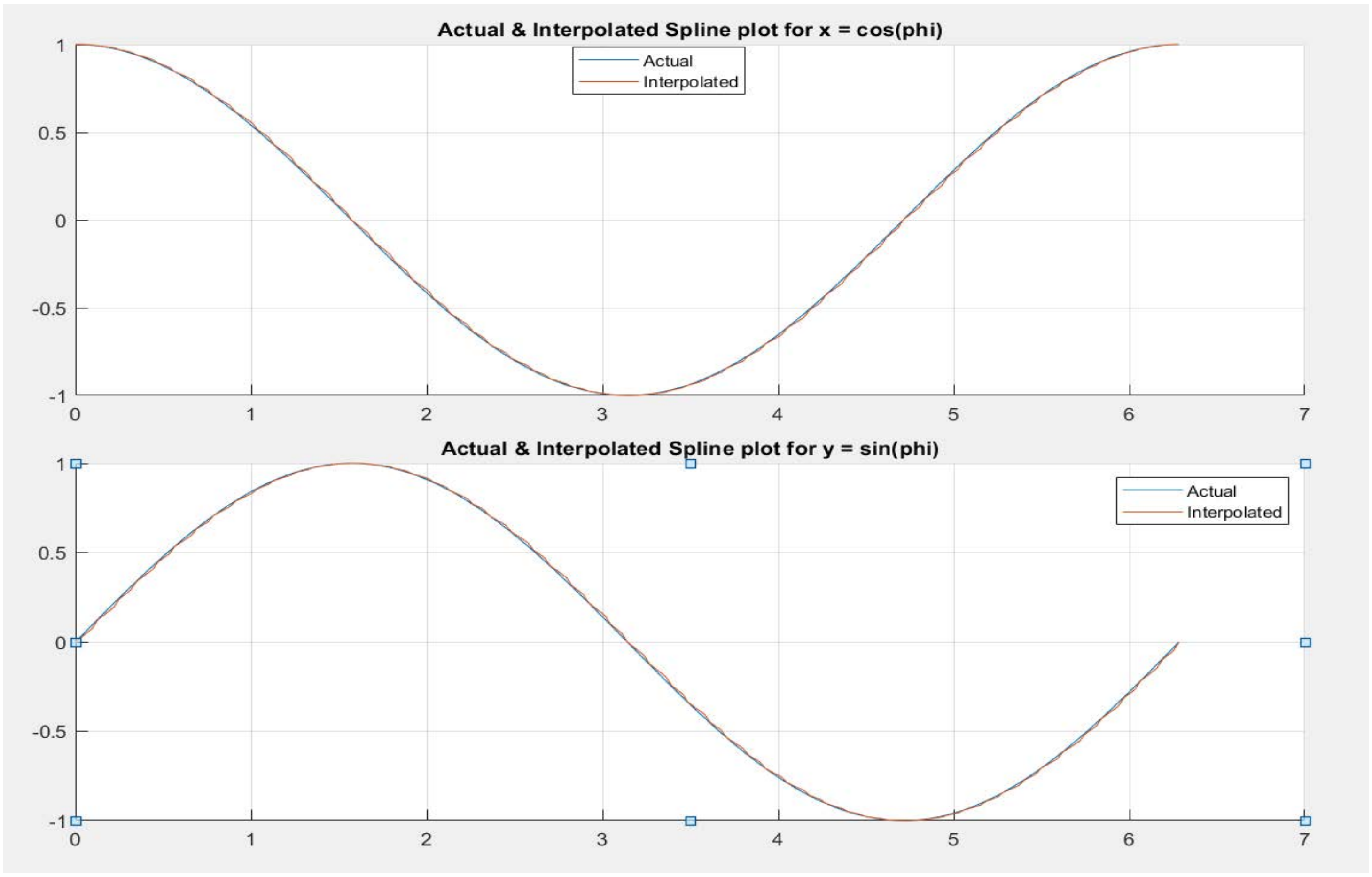


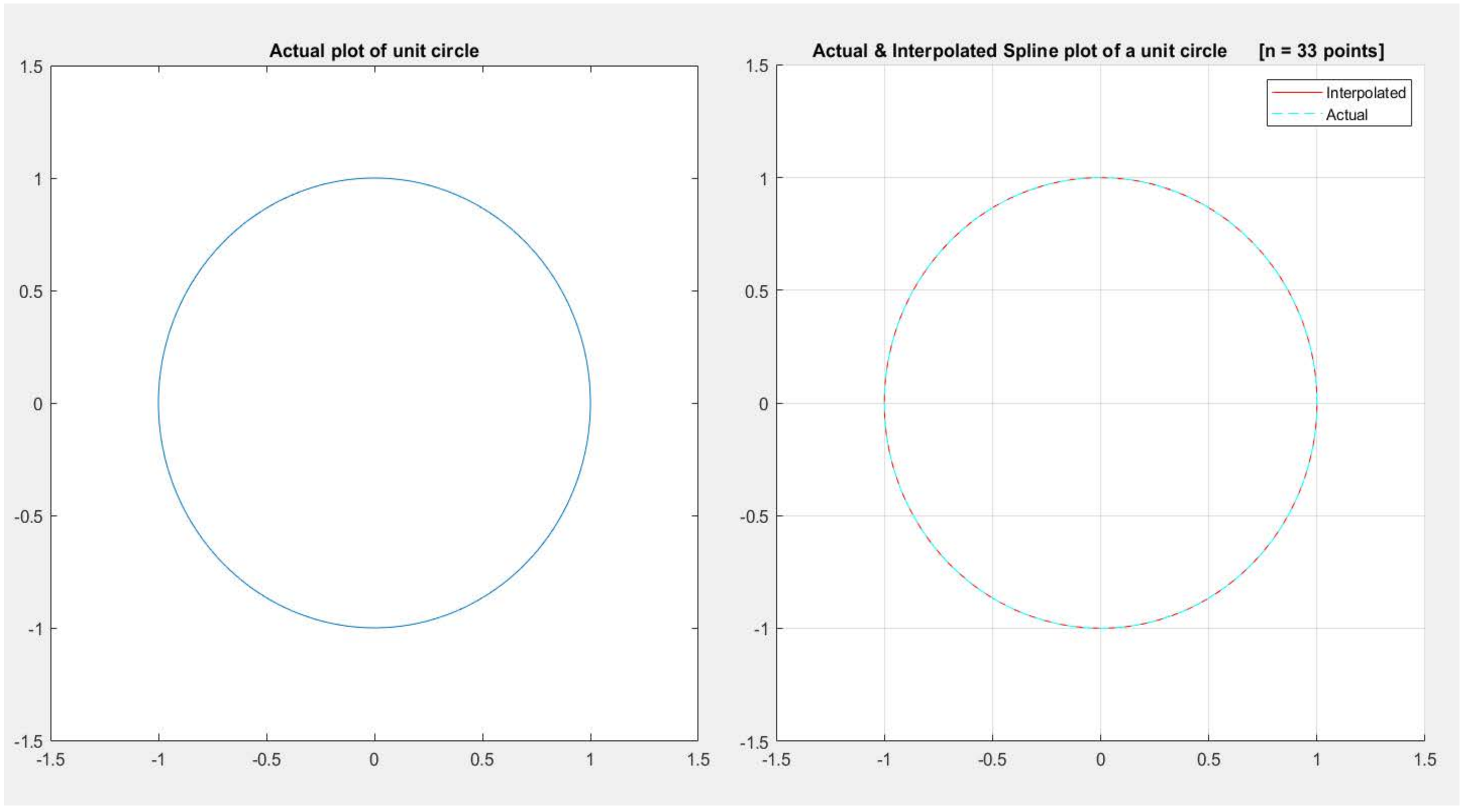
## Results obtained for n = 17 observation points





## Results obtained for n = 33 observation points





## References

1. Numerical Methods for Engineers and Scientists  Using MATLAB.
   *Ramin S. Esfandiari*
2. Numerical Methods for Engineers and Scientists An Introduction with Applications using MATLAB.
   *Amos Gilat, Vish Subramaniam*
3. Applied Numerical Methods with MATLAB® for Engineers and Scientists
   Steven C. Chapra, Raymond P. Canale
4. Web Sources :-

    1. Article describing how to compute intercept and slope in the method of linear least square fit:-
       https://www.technologynetworks.com/informatics/articles/calculating-a-least-squares-regression-line-equation-example-explanation-310265
    2. Article explaining the method of cubic spline interpolation and several of its boundary conditions:-
       http://www.maths.lth.se/na/courses/FMN081/FMN081-06/lecture11.pdf
    3. https://www.cae.tntech.edu › 2004-11-02_handouts.pdf - Process of obtaining the C coefficient matrix and solving the system of matrix.

# Appendix

**All the function files (.m files) used in the programming of Cubic spline interpolation questions are detailed out below;**

## 1.) "funct_of_x" computes the divided difference.

```
function f = funct_of_x(x)

%% Test equations to test for Question - 1.
equation = @(x) (x).^2; % Sample equation to test the program
%% Runge - phenomenon equations to test for Question - 2.
% equation = @(x) 1./(1+ 25*(x).^2); % Runge-phenomenon for Question - 2
%% Obtaining the divided difference.
x = sort(x);
length = numel(x);
xa = [];
xb = [];
for i = 2:length
    xa = [xa, x(i)];          % Big value
    xb = [xb, x(i-1)];        % Small value
end
    % The equation below is the equation of 'yi' which is f(x). It needs to
    % be changed as per the question being studied.
    f1 = equation(xa);
    f2 = equation(xb);
    f = (f1 - f2)/(x(length) - x(1)); % divided difference formula
end
```

## 2.) "myfun" comprises of the Runge-Phenomenon equation.

```
function y = myfun(x)

x = x(:)';
y = 1./(1 + 25*(x.^2));
end
```

## 3.) "For_Inverse" Gauss-Seidel Method to solve linear system of equations.

```
% clc;clear all %% Gauss-Seidel Method to solve linear system of equations
function x = For_Inverse(A,R)

x=zeros(length(A),1);n = size(x,1);
for iteration_count = 1:20
x_old = x;
  for i = 1:n
    sum1 = 0;sum2 = 0;
    for j = 1:i-1
      if j~=i
      sum1 = sum1 + A(i,j)*x(j);
      end
    end
       for j = i+1:n
         sum2 = sum2 + A(i,j)*x_old(j);
       end
       x(i) = -(sum1 + sum2 - R(i))/A(i,i);
  end
end
end
```

## 4.) "h_func" computes x(i+1) - x(i)

```
function h = h_func(x,number)

h = [];
for i = 1:(number-1)
  hi = x(i+1) - x(i);
  h = [h, hi];
end
end
```

## 5.) "equal_sub_int" provides the equally spaced subintervals or step-size.

```
function X = equal_sub_int(xrange,number)

X = xrange(1);
 for i = 1:(number-1)
   xtemp = xrange(1) + ((xrange(2) - xrange(1))/(number-1))*i;
   X = [X,xtemp];
 end
end
```

## 6.) "CubicSplPolynml" formulates the cubic polynomial for spline.

```
function S = CubicSplPolynml(A,B,C,D,x)

xsort = sort(x);
xsort(:);
A = A(:); B = B(:); C = C(:); D = D(:);
syms x;
len = length(A);
for i = 1:(len)
  S(i) = A(i) + B(i)*(x - xsort(i)) + C(i)*(x - xsort(i)).^2 + D(i)*(x - xsort(i)).^3;
end
S = S(:);
end
```

## 7.) "CoefficientMatrix" develops Coefficient matrix [A] in [A][c] = [r]

```
function A = CoefficientMatrix(h)

h = h(:)';
sz = length(h)-1;

%% 1st & 2nd element of 1st row
A = zeros(sz);
A(1,1) = (h(2) + h(1))*(h(1) + 2*h(2))/h(2);
A(1,2) = ((h(2) + h(1))*(h(2) - h(1)))/h(2);
%% Second Last & Last element of last row
A(sz,(sz-1)) = ((h(sz) + h(sz+1))*(h(sz) - h(sz+1)))/h(sz);
A(sz,sz) = ((2*h(sz) + h(sz+1))*(h(sz) + h(sz+1)))/h(sz);

%% middle elements on the tri-diagonal
middle = zeros((sz-2),3);
for i =1:(sz-2)
    middle(i,:) = [h(i+1), 2*(h(i+1)+h(i+2)), h(i+2)];
    A(i+1,i:i+2) = middle(i,:);

end
end
```

## 8.) "funct_of_x_Try" used in last problem for plotting unit circle spline

```
function f = funct_of_x_Try(phi)
phi = sort(phi);
% phi = 0:pi/4:(2*pi);
lent = numel(phi);
phi_a = []; phi_b = [];
for i = 2:lent
  x(i) = cos(phi(i));
  phi_a = [phi_a, phi(i)];          % Big value
  phi_b = [phi_b, phi(i-1)];        % Small value
end
  %y(i) = sin(x(i));
  f1 = sin(phi_a);
  f2 = sin(phi_b);
  f = (f1 - f2)/(2*pi);
end
```