Project 2: Splines Interpolation

Hieu Duy Bui

October 5, 2022

1 Introduction

We do not often get exact functions in the real world; however, we get data points. This project focuses on extracting meaning from such data points with interpolation, namely splines. We use splines to connect between data points. Each spline has its own equation, and we can also dictate the order of the splines with different methods. In this project, we go through linear, quadratic, and cubic splines.

2 Methods

2.1 Linear Interpolation

The main idea of linear interpolation is using linear function to connect 2 data points. We know the linear two-point-form looks like $y - y_i = m * (x - x_i)$, where m is the slope: $m = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$. We can rearrange the equation to solve for y.

$$y = y_i + m * (x - x_i)$$

$$1 \leq i \leq n - 1$$

n is the number of data points we wish to go through. Since each spline connects 2 points, we will have n number of points and n-1 number of splines.

2.2 Quadratic Spline

Similar to linear interpolation, the two consecutive splines will have to go through the same point. The end point of the previous spline will have to be the start point of the next spline. However, unlike linear interpolation, the first derivative of the i^{th} spline need to match the $(i+1)^{th}$ spline. Graphically,

the same first derivative will smooth out the transition at interior points. To start, we will use the first 3 data points to approximate k_1. We need 3 data points because the number of splines is n-1 data points; therefore, we need 3 data points to obtain 2 splines. With first 2 initial splines, we can solve for k_1 using Cramer's Rule. The first two splines equations:

$$y_2 = y_1 + k_1 * (x_2 - x_1) + m_1 * (x_2 - x_1)^2$$

$$y_3 = y_1 + k_1 * (x_3 - x_1) + m_1 * (x_3 - x_1)^2$$

We transform it into matrices form:

$$\begin{bmatrix} (x_2 - x_1) & (x_2 - x_1)^2 \\ (x_3 - x_1) & (x_3 - x_1)^2 \end{bmatrix} \begin{bmatrix} k_1 \\ m_1 \end{bmatrix} = \begin{bmatrix} (y_2 - y_1) \\ (y_3 - y_1) \end{bmatrix}$$

Cramer's Rule requires a modified matrix, in which we replace the column we want to solve for with the resultant column and call it A_i:

$$\begin{bmatrix} (y_2 - y_1) & (x_2 - x_1)^2 \\ (y_3 - y_1) & (x_3 - x_1)^2 \end{bmatrix}$$

Lastly, we compute the ratio between the determinant of A_i and the original matrix to get k_1.

$$k_1 = \frac{\begin{vmatrix} (y_2 - y_1) & (x_2 - x_1)^2 \\ (y_3 - y_1) & (x_3 - x_1)^2 \end{vmatrix}}{\begin{vmatrix} (x_2 - x_1) & (x_2 - x_1)^2 \\ (x_3 - x_1) & (x_3 - x_1)^2 \end{vmatrix}}$$

Since the first matching condition requires two adjacent splines to go through the same interior point, we have a quadratic equation that connects the previous

data point to the next data point:

$$y_{i+1} = y_i + k_i * (x_{i+1} - x_i) + m_i * (x_{i+1} - x_i)^2$$

We can use this equation to solve for the second derivative, which is m, with algebra. With the same idea of matching condition, we will write an equation that express the matching of first derivatives of two adjacent splines using power rule:

$$k_{i+1} = k_i + 2 * m_i * (x_{i+1} - x_i)$$

Since k and m are variables of splines, we have n-1 values of k and m because we have n-1 splines. Knowing this, we will solve the above two equations from $1 \leq i \leq n-1$ for k and m. After solving for all the k and m, we can solve for all the splines with the standard quadratic function format:

$$y = y_i + k_i * (x - x_i) + m_i * (x - x_i)^2$$
$$1 \leq i \leq n-1$$

Since we solve for the splines from the first 3 points and go from 1 to n-1, we have a left bias quadratic spline interpolation. We could also do a right bias from the last 3 data points and go from n to 2 with all the same equations and procedures. Ultimately, we can average the derivatives (k and m) to get the average result.

2.3 Clamped Cubic Spline

Similar to second order splines, we need to obtain first, second, and third derivatives. After we have the derivatives, we will plug the into the standard cubic polynomial equation for the splines. With the clamped cubic spline, we enforce the matching interior points, first and second derivatives. Firstly, we will solve

for the first derivative k. The first and last k can be solved with Cramer's Rule. The original set up for them are

$$\begin{bmatrix} (x_2 - x_1) & (x_2 - x_1)^2 \\ (x_3 - x_1) & (x_3 - x_1)^2 \end{bmatrix} \begin{bmatrix} k_1 \\ m_1 \end{bmatrix} = \begin{bmatrix} (y_2 - y_1) \\ (y_3 - y_1) \end{bmatrix}$$

$$\begin{bmatrix} (x_{n-1} - x_n) & (x_{n-1} - x_n)^2 \\ (x_{n-1} - x_n) & (x_{n-1} - x_n)^2 \end{bmatrix} \begin{bmatrix} k_n \\ m_n \end{bmatrix} = \begin{bmatrix} (y_{n-1} - y_1) \\ (y_{n-2} - y_1) \end{bmatrix}$$

After we have k_1 and k_n, we can solve for the rest by solving a tridiagonal matrix with coefficients of $\frac{1}{6}$, $\frac{4}{6}$, $\frac{1}{6}$. The resultant array is $\frac{y_{i+1} - y_{i-1}}{2*(x_{i+1} - x_i)}$, for $2 \leq i \leq n-1$. Next, we solve for m and d with the following equations:

$$m_i = \frac{3 * (y_{i+1} - y_i)}{(x_{i+1} - x_i)^2} - \frac{k_{i+1} + 2 * k_i}{x_{i+1} - x_i}$$

$$d_i = \frac{k_{i+1} + k_i}{(x_{i+1} - x_i)^2} - \frac{2 * (y_{i+1} - y_i)}{(x_{i+1} - x_i)^3}$$

$$1 \leq i \leq n-1$$

Lastly, we substitute all the derivatives into the cubic polynomial between two points for the splines.

$$y = y_i + k_i * (x - x_i) + m_i * (x - x_i)^2 + d_i * (x - x_i)^3$$

$$1 \leq i \leq n-1$$

2.4 Natural Cubic Spline

With natural cubic spline, we don't need to match the first derivative. We set the second derivative of the first and last spline to 0 for initial conditions. After which, we can solve for the rest of the second derivatives using tridiagonal

system like we did for k in clamped cubic spline. The diagonals' coefficients remain the same. The resultant array changes to $\frac{y_{i+1}-2y_i+y_{i-1}}{x_{i+1}-x_i}$, for $2 \leq i \leq n$. After which, we substitute and solve for the cubic polynomial just as we did in clamped cubic spline.

3 Programming

3.1 Linear Interpolation

- Read data points
- Create stretched grid
- Solve for required variables
- Set up 10 points per subdivision for each spline
- Solve for splines
- Plot splines incrementally

3.2 Quadratic Spline

- Read data points
- Create stretched grid
- Write sub-routine for Cramer's Rule
- Set up and solve initial conditions
- Solve for unknowns
- Set up 10 points per subdivision
- Solve for splines
- Plot splines incrementally

3.3 Clamped Cubic Spline

- Read data points
- Create uniform grid
- Set up Matrices for initial and ending conditions (k1 and kn)
- Solve for k1 and kn with Cramer's sub-routine
- Set up tridiagonal matrix system to solve for the rest of k(s)
- Solve tridiagonal matrix system with THOMAS3
- Set up 10 points per subdivision
- solve for splines
- Plot splines incrementally

3.4 Natural Cubic Spline

- Read data points
- Create uniform grid
- Set initial and ending conditions (m1 and mn)
- Set up tridiagonal matrix system to solve for the rest of m(s)
- Solve tridiagonal matrix system with THOMAS3
- Solve for all other unknown coefficients with given equations
- Set up 10 points per subdivision
- solve for splines
- Plot splines incrementally

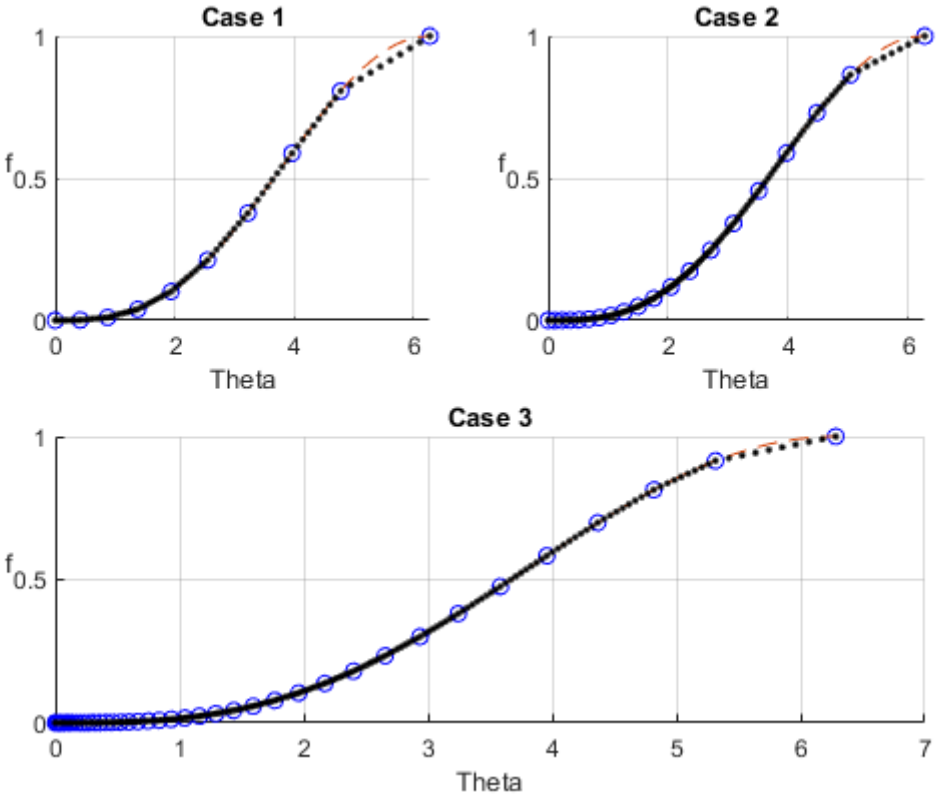
4 Code

4.1 Linear Spline

```
clear, clc;
close all;

%constant variables
n_1 = 10;
del_1 = .4186;
n_2 = 20;
del_2 = .1106;
n_3 = 40;
del_3 = .0146;
k = 1.1;

%exact original fncs
xexact = linspace(0,2*pi(),100);
yexact = sin(xexact./4).^3;

[a1,a2,a3,a4] = linsplin(del_1,k,n_1); %case 1
[b1,b2,b3,b4] = linsplin(del_2,k,n_2); %case 2
[c1,c2,c3,c4] = linsplin(del_3,k,n_3); %case 3

%integrate
area1 = splineArea(a1,n_1);
area2 = splineArea(b1,n_2);
area3 = splineArea(c1,n_3);

%differentiate
diff1 = deriv(a1);
diff2 = deriv(b1);
diff3 = deriv(c1);

%plotting all 3 cases
figure(1);
subplot(2,2,1);
hold on; grid on;
linsplinplot(xexact,yexact,a1,a2,a3,a4,'Case 1');
subplot(2,2,2);
hold on; grid on;
linsplinplot(xexact,yexact,b1,b2,b3,b4,'Case 2');
subplot(2,2,[3 4]);
hold on; grid on;
linsplinplot(xexact,yexact,c1,c2,c3,c4,'Case 3');

%plotting function
function linsplinplot(xexact,yexact,a,b,c,n,str)
    %plotting
    plot(b, c, 'bo', xexact, yexact, '--')
    for i = 1:n-1
        plot(a{i}(1,:),a{i}(2,:),'.k')
    end
    title(str);
    xlabel('Theta');
    ylabel('f','Rotation',0);
```

```
end


%linear spline function.
%The function returns splines, each with 10 discrete points per spline.
%Return data points and number of points
function [a,b,c,d] = linsplin(del, k, n)
    %generating domain points
    theta = ones(1,n);
    theta(1) = 0;
    theta(n) = 2*pi();

    for i = 1:n-2
        theta(i+1) = theta(i) + del*(k^(i-1));
    end

    %generating range points (change orginal eqn here)
    f = sin(theta./4).^3;

    %setting up 10 points for each spline
    %splines are stored as cell array, which contains 10 points between
    %data points
    s = cell(n-1,1);
    for j = 2:n
        m = (f(j)-f(j-1))/(theta(j)-theta(j-1));
        x = linspace(theta(j-1),theta(j),10);
        b = f(j)-m*theta(j);
        y = m.*x + b;
        s{j-1} = [x;y];
    end
    a = s;
    b = theta;
    c = f;
    d = n;
end

%integrating function
function area = splineArea(splines,n)
    area = 0;
    for j=1:n-1
        for i = 1:9
            littleArea = splines{j}(2,i)*(splines{j}(1,i+1)-splines{j}(1,i));
        end
        area = area + littleArea;
    end
end

%differentiating function
function diff = deriv(splines)
    diff = ones(3,1);
    for i=1:3
        diff(i) = (splines{i}(2,i+1)-splines{i}(2,i))/(splines{i}(1,i+1)-
splines{i}(1,i));
    end
```

```
end
```



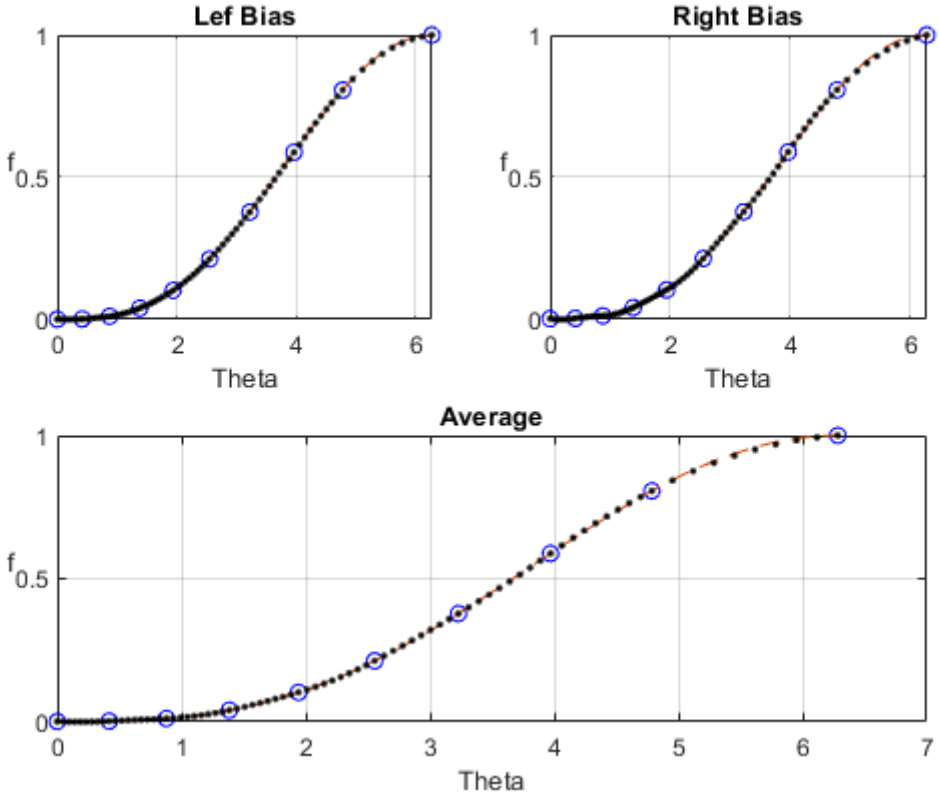
Published with MATLAB® R2021b

4.2 Quadratic Spline

```
clear, clc;
close all;

%case 1 data points and grid condition
n = 10;
del = .4186;
k = 1.1;

%generating grid
x = ones(1,n);
x(1) = 0;
x(n) = 2*pi();
for i = 1:n-2
    x(i+1) = x(i) + del*(k^(i-1));
end

%generating grid range points (change orginal eqn here)
y = sin(x./4).^3;

%exact original fncs
xexact = linspace(0,2*pi(),100);
yexact = sin(xexact./4).^3;

%running methods with inputs from case 1 to get left bias and right bias
%splines
sl = LQuadSpline(x,y,n);
sr = RQuadSpline(x,y,n);

% creating average quadratic splines by averaging x and y values of left
% and right bias splines
sa = cell(n-1,1);
for i = 1:n-1
    sa{i} = (sl{i}+sr{i})./2;
end

%Plotting left/right bias splines and the average
figure(1);
subplot(2,2,1)
QuadPlot(x,y,xexact,yexact,sl,n,'Lef Bias');
subplot(2,2,2)
QuadPlot(x,y,xexact,yexact,sr,n, 'Right Bias');
subplot(2,2,[3,4])
QuadPlot(x,y,xexact,yexact,sa,n, 'Average');

%Solving for left bias quadratic splines
function splines = LQuadSpline(x,y,n)
    %Setting up matrix system to solve for k1
    A = [(x(2)-x(1)),((x(2)-x(1))^2);(x(3)-x(1)),((x(3)-x(1))^2)];
    b = [y(2);y(3)];
    k = ones(n-1,1);
    m = ones(n-1,1);
    k(1) = cramer(A,b,1);
```

```
    %solving for m and the rest of k from left to right
    for i = 1:n-1
        m(i) = ((y(i+1)-y(i))-k(i)*(x(i+1)-x(i)))/((x(i+1)-x(i))^2);
        k(i+1) = k(i) + 2*m(i)*(x(i+1)-x(i));
    end

    %setting up 10 points for each spline and solve for splines
    splines = cell(n-1,1);
    for i = 1:n-1
        xspline = linspace(x(i),x(i+1),10);
        yspline = y(i)+k(i)*(xspline-x(i))+m(i)*(xspline-x(i)).^2;
        splines{i} = [xspline;yspline];
    end
end

function splines = RQuadSpline(x,y,n)
    %Setting up matrix system to solve for kn
    A = [(x(n-1)-x(n)),((x(n-1)-x(n))^2);(x(n-2)-x(n)),((x(n-2)-x(n))^2)];
    b = [(y(n-1)-y(n));(y(n-2)-y(n))];
    k = ones(n-1,1);
    m = ones(n-1,1);
    k(n) = cramer(A,b,1);

    %Solving for the rest of m and k from right to left
    for i = n:-1:2
        m(i) = ((y(i-1)-y(i))-k(i)*(x(i-1)-x(i)))/((x(i-1)-x(i))^2);
        k(i-1) = k(i) + 2*m(i)*(x(i-1)-x(i));
    end

    %set up 10 points per subdivision and solve for right bias splines
    splines = cell(n-1,1);
    for i = 2:n
        xspline = linspace(x(i-1),x(i),10);
        yspline = y(i)+k(i)*(xspline-x(i))+m(i)*(xspline-x(i)).^2;
        splines{i-1} = [xspline;yspline];
    end
end

% plot x-y data points
% plot exact function
% plot splines
function QuadPlot(x,y,xexact,yexact,s,n,str)
    plot(x, y, 'bo',xexact,yexact, '--')
    hold on; grid on;
    for i = 1:n-1
        plot(s{i}(1,:),s{i}(2,:),'.k')
    end
    title(str);
    xlabel('Theta');
    ylabel('f','Rotation',0);
end

%take matrix a,b, and column values then use cramer's rule
```

```
function cram = cramer(a,b,col)
    deterA = (a(1,1).*a(2,2))-(a(1,2).*a(2,1));
    ai = a;
    ai(:,col) = b;
    deterAI = (ai(1,1).*ai(2,2))-(ai(1,2).*ai(2,1));
    cram = deterAI/deterA;
end
```



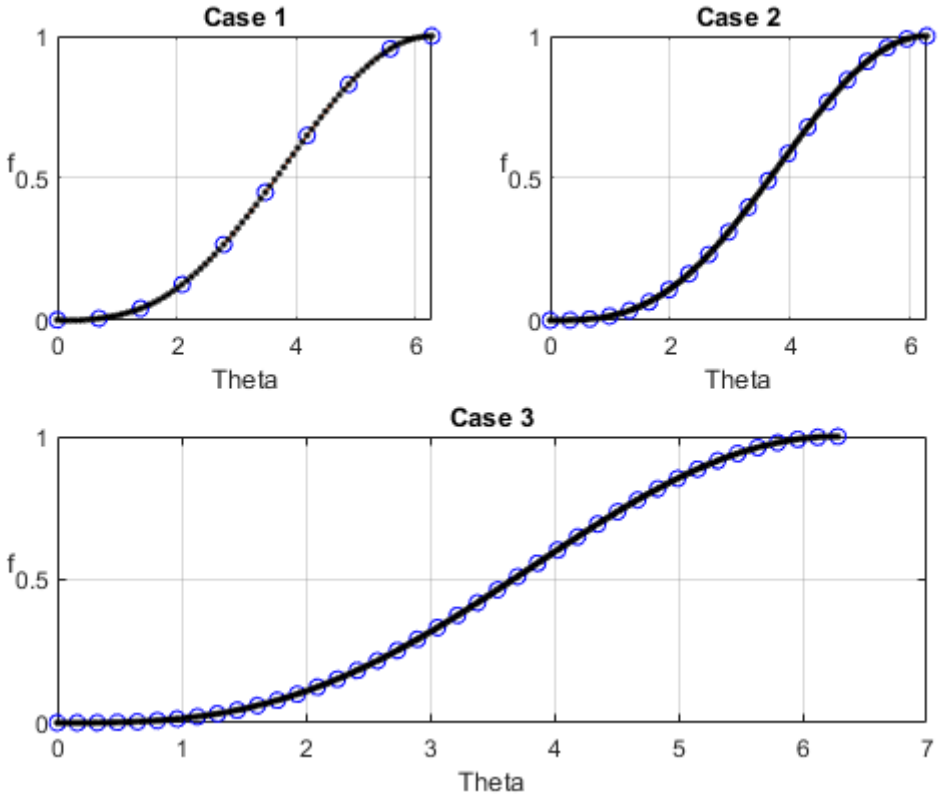
Published with MATLAB® R2021b

4.3 Clamped Cubic Spline

```
clear, clc;
close all;

% make data points for different cases
n1 = 10;
n2 = 20;
n3 = 40;

% exact case
xexact = linspace(0,2*pi(),100);
yexact = sin(xexact./4).^3;

%getting splines/x/y values
[s1,x1,y1] = clampedSpline(n1);
[s2,x2,y2] = clampedSpline(n2);
[s3,x3,y3] = clampedSpline(n3);

%Plot all 3 cases against each other
figure;
subplot(2,2,1)
cubicplot(x1,y1,xexact,yexact,s1,n1, 'Case 1')
subplot(2,2,2)
cubicplot(x2,y2,xexact,yexact,s2,n2, 'Case 2')
subplot(2,2,[3 4])
cubicplot(x3,y3,xexact,yexact,s3,n3, 'Case 3')

%Funtion that take in number of data points and return splines with domain
%and range
function [splines,x,y] = clampedSpline(n)
    %making uniform grid based on given size n
    x = linspace(0,2*pi,n);
    y = sin(x./4).^3;

    % Set up matrices for cramer's
    % Use cramer sub-routine to solve for k1 and kn
    Al = [(x(2)-x(1)),((x(2)-x(1))^2);(x(3)-x(1)),((x(3)-x(1))^2)];
    bl = [y(2);y(3)];
    Ar = [(x(n-1)-x(n)),((x(n-1)-x(n))^2);(x(n-2)-x(n)),((x(n-2)-x(n))^2)];
    br = [(y(n-1)-y(n));(y(n-2)-y(n))];
    k1 = cramer(Al,bl,1);
    kn = cramer(Ar,br,1);

    % making tridiagonal for Thomas3 and solve for k2 to kn-1
    a = (1/6)*ones(1,n);
    b = (4/6)*ones(1,n);
    c = (1/6)*ones(1,n);
    e = ones(1,n);
    for i = 2:n-1
        e(i) = (y(i+1)-y(i-1))/(2*(x(i+1)-x(i)));
    end
    e(1) = k1;
    e(n) = kn;
```

```
    k = THOMAS3(a,b,c,e,n);
    k(1) = k1;
    k(n) = kn;

    %solving for m and d
    for i = 1:n-1
        m(i) = (3*(y(i+1)-y(i))/(x(i+1)-x(i))^2)-((k(i+1)+2*k(i))/(x(i+1)-
x(i)));
        d(i) = ((k(i+1)+k(i))/(x(i+1)-x(i))^2)-((2*(y(i+1)-y(i)))/((x(i+1)-
x(i))^3));
    end

    %set up 10 points per subdivision
    %solve for splines
    splines = cell(n-1,1);
    for i = 1:n-1
        xspline = linspace(x(i),x(i+1),10);
        yspline = y(i)+k(i)*(xspline-x(i))+m(i)*(xspline-
x(i)).^2+d(i)*((xspline-x(i)).^3);
        splines{i} = [xspline;yspline];
    end
end

%Plot function
%plot the original/exact function and data points
%plot splines
function cubicplot(x,y,xexact,yexact,splines,n,str)
    figure(1);
    plot(x, y, 'bo',xexact,yexact,'--')
    hold on; grid on;
    for i = 1:n-1
        plot(splines{i}(1,:),splines{i}(2,:),'.k')
    end
    title(str);
    xlabel('Theta');
    ylabel('f','Rotation',0);
end

%take matrix a,b, and column values then use cramer's rule
function cram = cramer(a,b,col)
    deterA = (a(1,1).*a(2,2))-(a(1,2).*a(2,1));
    ai = a;
    ai(:,col) = b;
    deterAI = (ai(1,1).*ai(2,2))-(ai(1,2).*ai(2,1));
    cram = deterAI/deterA;
end

%Sub-routine to solve for tridiagonal matrix system
function x = THOMAS3(a,b,c,d,n)

    %initial condition
    bbar(1) = b(1);
    cbar(1) = c(1);
    dbar(1) = d(1);
```

```
    %making upper triangle
    for i = 2:n
        multiplier = a(i)./bbar(i-1);
        abar(i) = a(i) - bbar(i-1).*multiplier;
        bbar(i) = b(i) - cbar(i-1).*multiplier;
        cbar(i) = c(i);
        dbar(i) = d(i) - dbar(i-1).*multiplier;
    end

    %initialize x of size n
    x = ones(1,n);

    %initialize end condition
    x(n) = dbar(n)/bbar(n);

    % Upward substitution AKA zip it up
    for i = n-1:-1:1
        x(i) = (dbar(i)-(cbar(i)*x(i+1)))/bbar(i);
    end
end
```

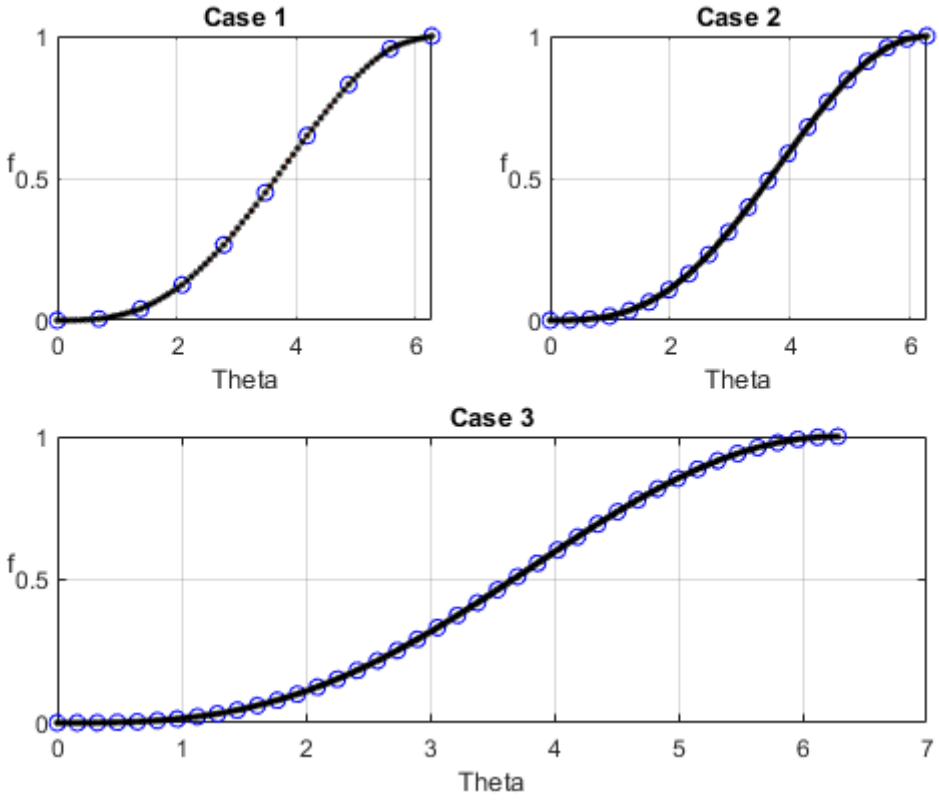


Published with MATLAB® R2021b

4.4 Natural Cubic Spline

```
clear, clc;
close all;

% make data points
n1 = 10;
n2 = 20;
n3 = 40;

% exact case
xexact = linspace(0,2*pi(),100);
yexact = sin(xexact./4).^3;

%getting splines/x/y values
[s1,x1,y1] = naturalCubic(n1);
[s2,x2,y2] = naturalCubic(n2);
[s3,x3,y3] = naturalCubic(n3);

%Plot all 3 cases against each other
figure
subplot(2,2,1)
naturalCubicPlot(x1,y1,xexact,yexact,s1,n1, 'Case 1')
subplot(2,2,2)
naturalCubicPlot(x2,y2,xexact,yexact,s2,n2, 'Case 2')
subplot(2,2,[3 4])
naturalCubicPlot(x3,y3,xexact,yexact,s3,n3, 'Case 3')

%Funtion that take in number of data points and return splines with domain
%and range
function [splines,x,y] = naturalCubic(n)
    %making uniform grid based on given size n
    x = linspace(0,2*pi,n);
    y = sin(x./4).^3;

    % making tridiagonal for Thomas3 and solve for m1 to mn
    a = (1/6)*ones(1,n);
    b = (4/6)*ones(1,n);
    c = (1/6)*ones(1,n);
    e = ones(1,n);
    for i = 2:n-1
        e(i) = (y(i+1)-2*y(i)+y(i-1))/(x(i+1)-x(i));
    end
    e(1) = 0;
    e(n) = 0;
    m = THOMAS3(a,b,c,e,n);
    m(1) = 0; %first and last m set to 0
    m(n) = 0;

    %set up 10 points per subdivision
    %solve for splines
    splines = cell(n-1,1);
    for i = 1:n-1
        xspline = linspace(x(i),x(i+1),10);
```

```
        deltax = x(i+1)-x(i);
        a0 = m(i)/(6*deltax);
        a1 = m(i+1)/(6*deltax);
        a2 = y(i)/deltax-(m(i)*deltax)/6;
        a3 = y(i+1)/deltax-(m(i+1)*deltax)/6;
        yspline = a0.*(x(i+1)-xspline).^3 + a1.*(xspline-x(i)).^3 + a2.*(x(i
+1)-xspline) + a3.*(xspline-x(i));
        splines{i} = [xspline;yspline];
    end
end

%Plot function
%plot the original/exact function and data points
%plot splines
function naturalCubicPlot(x,y,xexact,yexact,splines,n,str)
    figure(1);
    plot(x, y, 'bo', xexact, yexact, '--')
    hold on; grid on;
    for i = 1:n-1
        plot(splines{i}(1,:),splines{i}(2,:),'.k')
    end
    title(str);
    xlabel('Theta');
    ylabel('f','Rotation',0);
end

%Sub-routine to solve for tridiagonal matrix system
function x = THOMAS3(a,b,c,d,n)

    %initial condition
    bbar(1) = b(1);
    cbar(1) = c(1);
    dbar(1) = d(1);

    %making upper triangle
    for i = 2:n
        multiplier = a(i)./bbar(i-1);
        abar(i) = a(i) - bbar(i-1).*multiplier;
        bbar(i) = b(i) - cbar(i-1).*multiplier;
        cbar(i) = c(i);
        dbar(i) = d(i) - dbar(i-1).*multiplier;
    end

    %initialize x of size n
    x = ones(1,n);

    %initialize end condition
    x(n) = dbar(n)/bbar(n);

    % Upward substitution AKA zip it up
    for i = n-1:-1:1
        x(i) = (dbar(i)-(cbar(i)*x(i+1)))/bbar(i);
    end
end
```



Published with MATLAB® R2021b