Tristan Millman

Project 1

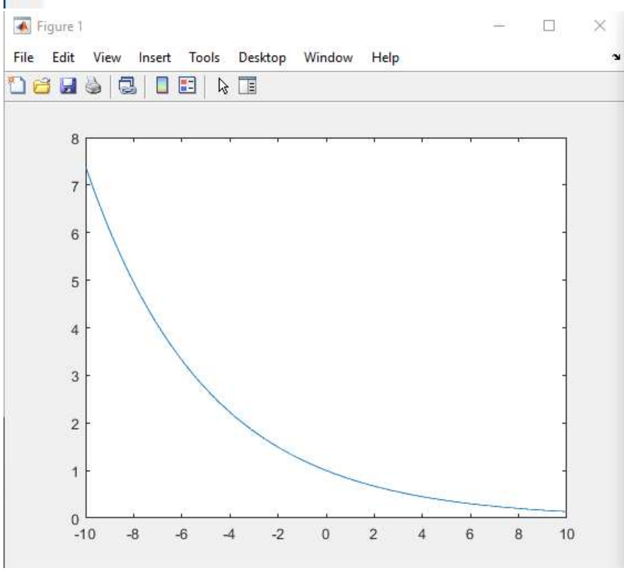
UFID: 4396-7960

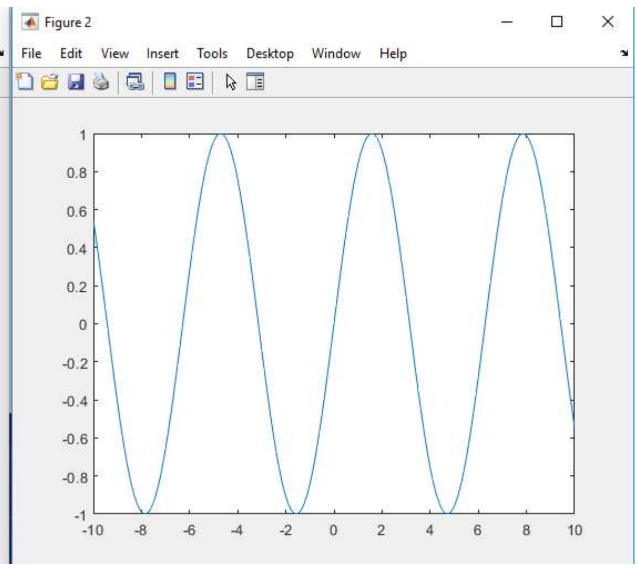
1. [Visual Inspection]

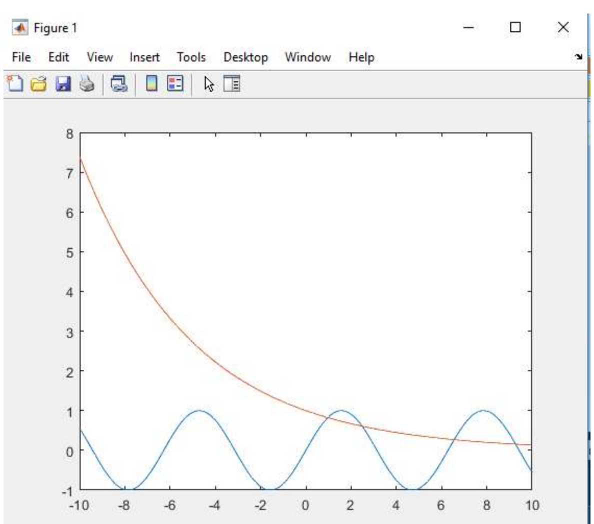
Below are the commands used to plot and an image of the two functions plotted. Based on this plot with x ranging from -10 to 10 I would guess that the points are most likely to be equal at x values at 1, 3, 6.5, and 9 along with many points afterwards.

Command Window

```
>> x=-10:0.1:10;
>> y = exp(-x/5);
>> plot(x,y)
>> y = sin(x);
>> plot(x,y)
>> g = exp(-x/5);
>> plot (x,y,x,g)
>>
```







2. [Bisection Method]

Code Used:

Editor - D:\matlab\bin\bisectionMethod.m

bisectionMethod.m

```
function bisectionMethod(f,a,b)

format long
c = f(a); d = f(b);
if c*d > 0.0
 error('There is no zero between a and b')
end
y = 100;
i = 0;
while abs(y) > 10^(-7)
 i = i+1;
 x = (a + b)/2;
 y = f(x);
    if y == 0.0
        break
    end
    if c*y < 0
        b=x;
    else
        a=x;
    end
end

disp('Number of steps:')
disp(i)
disp('Solution:')
disp(x)
```

Outputs & Command Line:

```
Command Window
>> f = @(x) exp(-x/5)-sin(x);
>> bisectionMethod(f, 0, 2)
Number of steps:
    21

Solution:
   0.968319892883301

>> bisectionMethod(f, 2, 4)
Number of steps:
    20

Solution:
   2.488080978393555

>> bisectionMethod(f, 6, 8)
Number of steps:
    23

Solution:
   6.556052446365356

>> bisectionMethod(f, 8, 10)
Number of steps:
    21

Solution:
   9.267439842224121

fx >> 
```

3. [Newton's Method]

Code Used:

```
function newtonsMethod(f,g,a)
format long
y = 100;
i = 0;
x0 = a;
while abs(y) > 10^(-7)
 i = i+1;
 x1 = x0 - f(x0)/g(x0);
 y = f(x1);
 x0 = x1;
end
disp('Number of steps:')
disp(i)
disp('Solution:')
disp(x0)
```

Outputs & Command Line:

```
Command Window
>> f = @(x) exp(-x/5)-sin(x);
>> g = @(x) -exp(-x/5)/5 - cos(x);
>> newtonsMethod(f,g,1)
Number of steps:
     3

Solution:
   0.968319799485087

>> newtonsMethod(f,g,3)
Number of steps:
     4

Solution:
   2.488081010196820

>> newtonsMethod(f,g,6.5)
Number of steps:
     2

Solution:
   6.556052464510788

>> newtonsMethod(f,g,9)
Number of steps:
     3

Solution:
   9.267439925784480

>>
```

Newton's Method was much quicker than the Bisection Method.

The steps for each point in Newton's Method were 3, 4, 2, and 3, respectively. The steps for each point in Bisection Method were 21, 20, 23, and 21 respectively. While both processed very quickly in this case, Newton's method would work much better for larger scale problems.

4. [Newton's Method Part 2]

a.

Command Window

```
>> f =  @(x)  (x-3)^(4)*sin(x);
>> g = @(x)  4*(x-3)^(3)*sin(x)+cos(x)*(x-3)^4;
>> newtonsMethod(f,g,2)
Number of steps:
    15

Solution:
   2.977784480543898

>>
```

b.

Code

Editor - D:\matlab\bin\altNewt.m

altNewt.m

```
function altNewt(f,g,a,m)
%f is the function
%g is the derivative
%a is your guess
%m is the multiplicity of the root you are trying to find
format long
y = 100;
i = 0;
x0 = a;
while abs(y) > 10^(-7)
 i = i+1;
 x1 = x0 - m * (f(x0)/g(x0));
 y = f(x1);
 x0 = x1;
end
disp('Number of steps:')
disp(i)
disp('Solution:')
disp(x0)
```

Results:

```
Command Window
>> f =  @(x)  (x-3)^(4)*sin(x);
>> g = @(x)  4*(x-3)^(3)*sin(x)+cos(x)*(x-3)^4;
>> altNewt(f,g,2,4)
Number of steps:
     2

Solution:
   2.990414230286607
```

In this case Newton's method took 15 steps whereas altered Newton's method only took 2 and returned a better result. This indicates that Altered Newton's Method works better for solving this particular root. (Note that I plugged in the exact multiplicity to do this, which is why it only took two steps).

5. [Lagrange Interpolation] (I used Maxima to get Lagrange Polynomials, and Matlab to plot them)

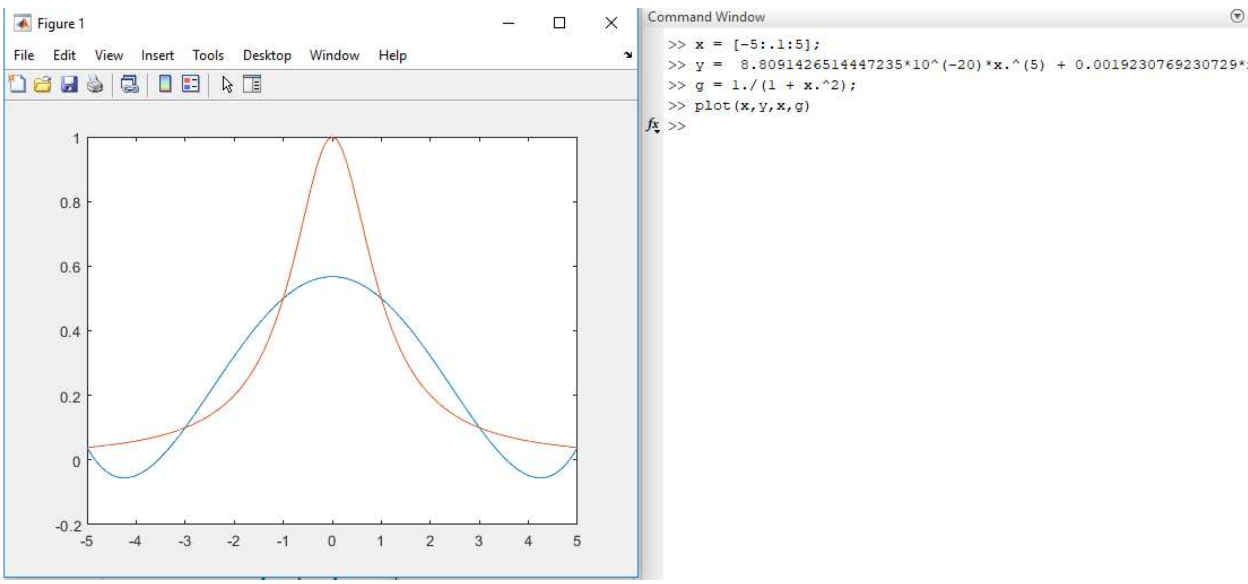
Using 6 points

[[-5, 0.03846153846],[-3, 0.1],[-1,0.5],[1,0.5],[3, 0.1],[5, 0.03846153846]]

The Lagrange polynomial is simplified to :

$Y = 8.8091426514447235\times10^{-20}x^5 + 0.0019230769230729x^4 + 5.8275866771095863\times10^{-19}x^3 - 0.069230769230729x^2 + 1.8810907692623502\times10^{-17}x + 0.56730769230766$

Plotting the function as shown below shows its not a very good approximation



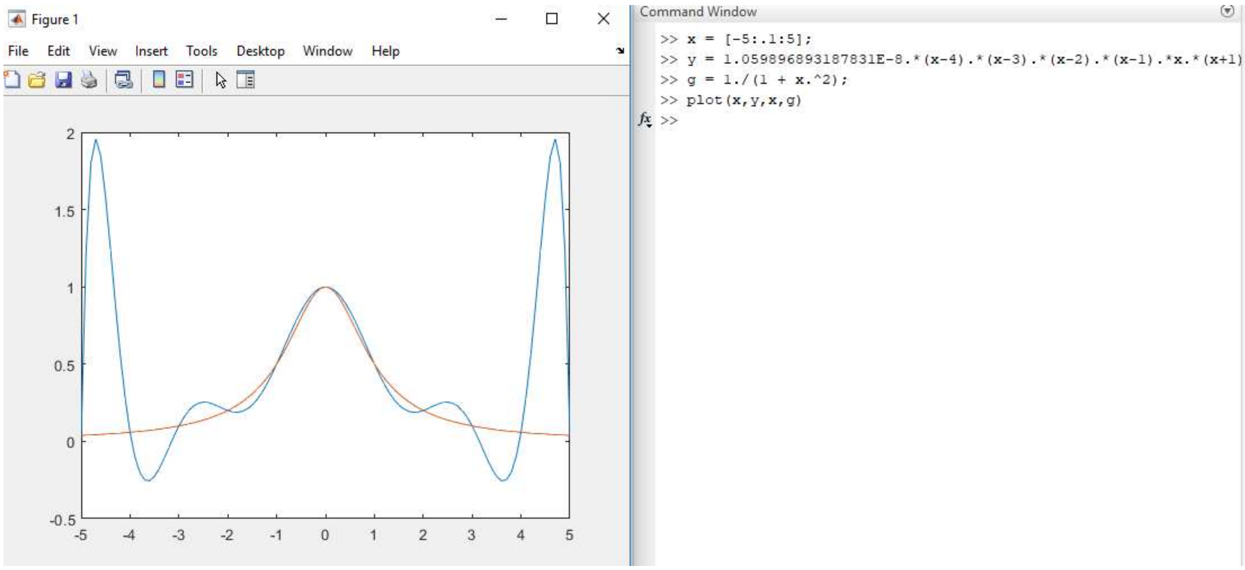
Using 11 points

[[-5,0.03846153846], [-4,0.05882352941], [-3,.1], [-2,.2], [-1,.5], [0,1], [1,.5], [2,.2], [3,.1], [4,0.05882352941], [5,0.03846153846]]

The Lagrange Polynomial is:

y = 1.059896893187831E-8*(x-4)*(x-3)*(x-2)*(x-1)*x*(x+1)*(x+2)*(x+3)*(x+4)*(x+5)-1.621018777832892E-7*(x-5)*(x-3)*(x-2)*(x-1)*x*(x+1)*(x+2)*(x+3)*(x+4)*(x+5)+1.240079365079365E-6*(x-5)*(x-4)*(x-2)*(x-1)*x*(x+1)*(x+2)*(x+3)*(x+4)*(x+5)-6.613756613756615E-6*(x-5)*(x-4)*(x-3)*(x-1)*x*(x+1)*(x+2)*(x+3)*(x+4)*(x+5)+2.893518518518519E-5*(x-5)*(x-4)*(x-3)*(x-2) * x*(x+1)*(x+2)*(x+3)*(x+4)*(x+5)-((x-5)*(x-4)*(x-3)*(x-2)*(x-1) * (x+1)*(x+2)*(x+3)*(x+4)*(x+5))./14400+2.893518518518519E-5*(x-5)*(x-4)*(x-3)*(x-2)*(x-1)* x*(x+2)*(x+3)*(x+4)*(x+5)-6.613756613756615E-6*(x-5)*(x-4)*(x-3)*(x-2)*(x-1) *x*(x+1)*(x+3)*(x+4)*(x+5)+1.240079365079365E-6*(x-5)*(x-4)*(x-3)*(x-2)*(x-1) *x*(x+1)*(x+2)*(x+4)*(x+5)-1.621018777832892E-7*(x-5)*(x-4)*(x-3)*(x-2)*(x-1) *x*(x+1)*(x+2)*(x+3)*(x+5)-1.059896893187831E-8*((-x)-4)*(x-5)*(x-4)*(x-3)*(x-2)*(x-1) *x*(x+1)*(x+2)*(x+3)

Plotting the function shows the approximation got better



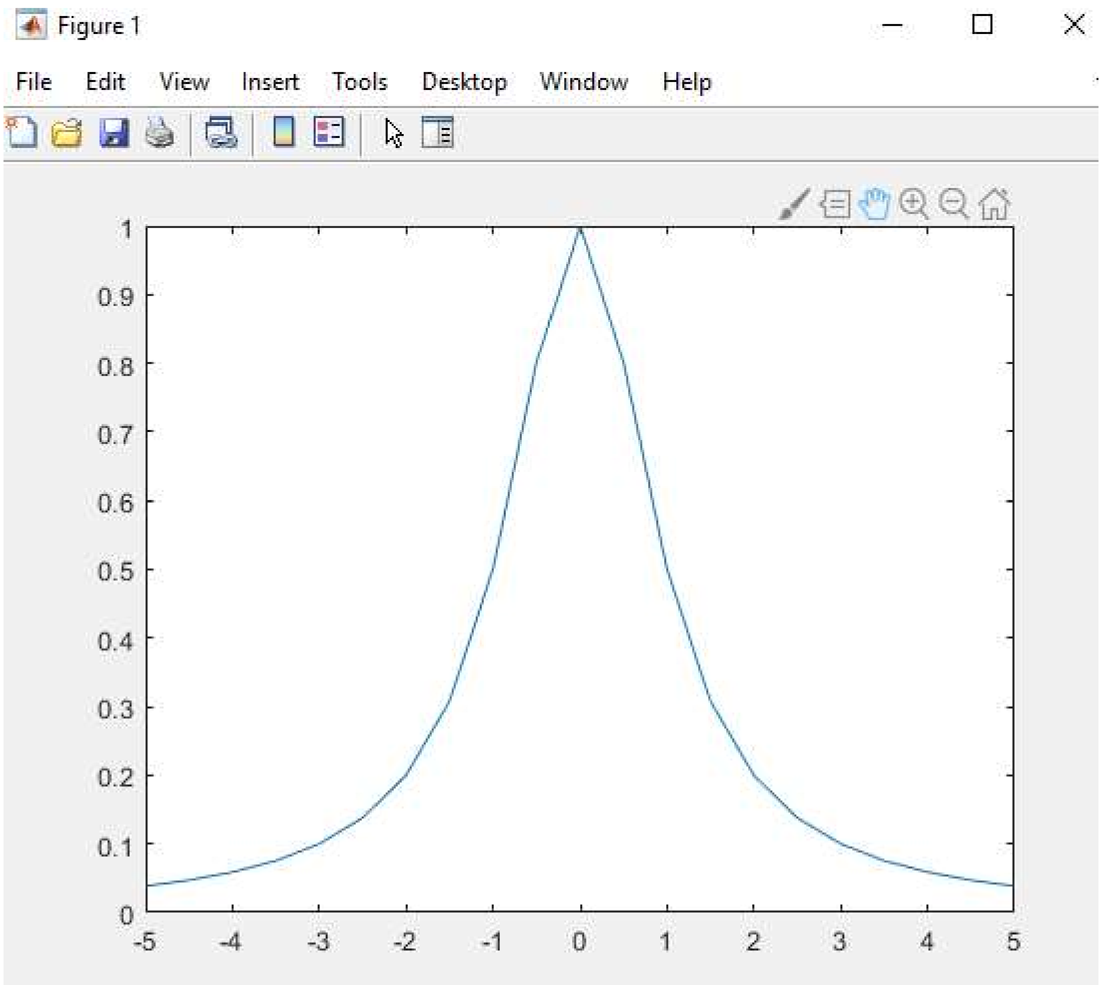
Using 21 points

[[-5,0.03846153846], [-4.5, 0.04705882352], [-4,0.05882352941], [-3.5,0.07547169811], [-3,.1], [-2.5,0.13793103448], [-2,.2], [-1.5,0.30769230769], [-1,.5], [-0.5,0.8], [0,1], [0.5,0.8], [1,.5],[1.5,0.30769230769], [2,.2], [2.5,0.13793103448], [3,.1], [3.5,0.07547169811], [4,0.05882352941], [4.5, 0.04705882352], [5,0.03846153846]]

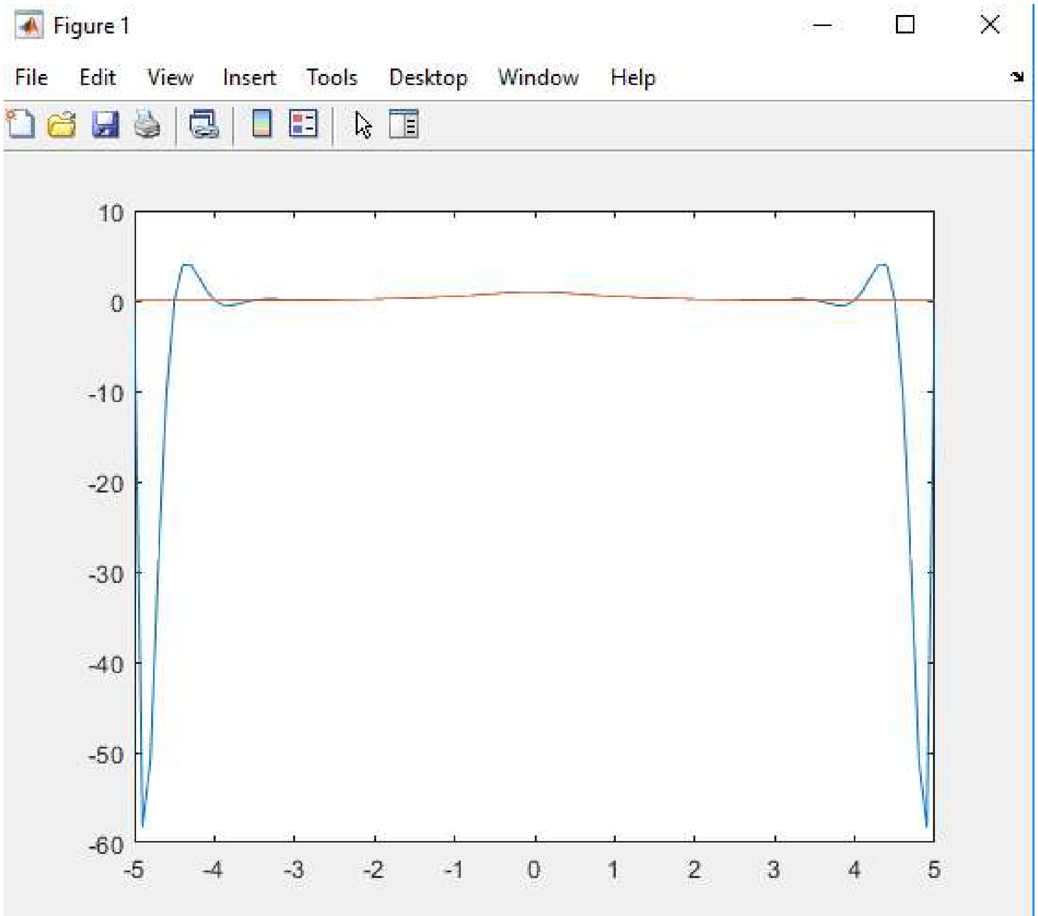
The Lagrange Polynomial is:

Y = 1.657684773849914E-14*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-4.056452151830759E-13*(x-5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+4.81703693111792E-12*(x-5)*(x-4.5)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-3.708209562021781E-11*(x-5)*(x-4.5)*(x-4)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+2.088185509702264E-10*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-9.21681880126014E-10*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+3.341096815523622E-9*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-1.028029789384174E-8*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+2.714641162612944E-8*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-5.791234480240947E-8*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+7.962947410331301E-8*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-5.791234480240949E-8*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+2.714641162612944E-8*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-1.028029789384174E-8*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+3.341096815523623E-9*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-9.216818801260139E-10*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+3)*(x+3.5)*(x+4)*(x+4.5)*(x+5)+2.088185509702265E-10*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3.5)*(x+4)*(x+4.5)*(x+5)-3.708209562021784E-11*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+4)*(x+4.5)*(x+5)+4.81703693111792E-12*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4.5)*(x+5)-4.056452151830756E-13*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+5)+1.657684773849914E-14*(x-5)*(x-4.5)*(x-4)*(x-3.5)*(x-3)*(x-2.5)*(x-2)*(x-1.5)*(x-1)*(x-0.5)*x*(x+0.5)*(x+1)*(x+1.5)*(x+2)*(x+2.5)*(x+3)*(x+3.5)*(x+4)*(x+4.5)

Plotted with x= [-5:.5:5] the approximation looks very close to the original function



However, Plotted with x= [-5:.1:5] it looks pretty good up until passing 4 and -4 while still being a better approximation overall than orders of n=5 or n= 10



Overall, the approximation got better as the order increased.

6. [Piecewise Linear Interpolation]

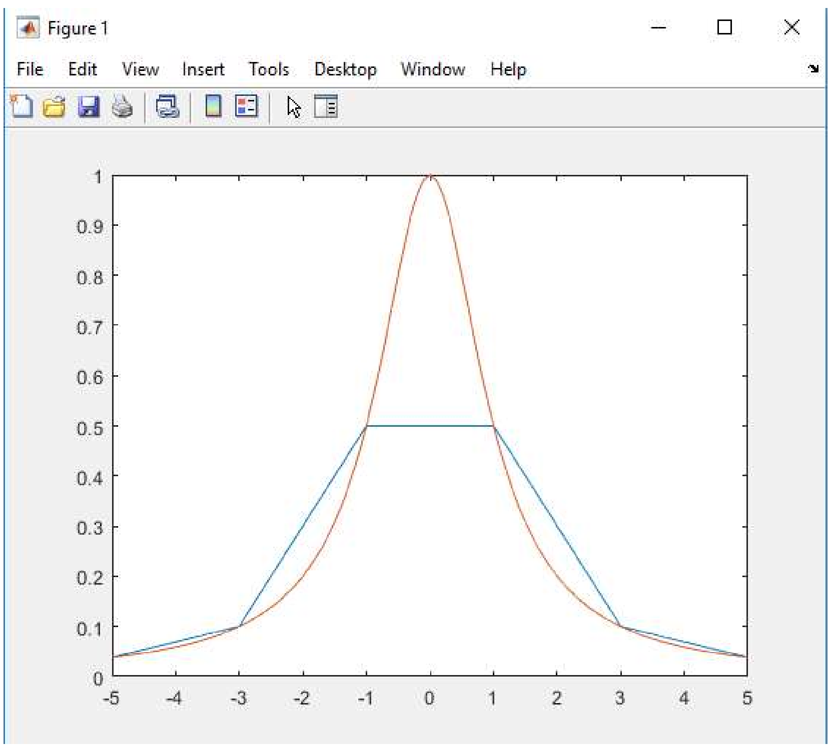
Code:

Editor - D:\matlab\bin\piecelin.m

piecelin.m

```
function v = piecelin(x,y,u)
delta = diff(y)./diff(x);
n = length(x);
k = ones(size(u));
for j = 2:n-1
    k(x(j) <= u) = j;
end

% Evaluate interpolant
s = u - x(k);
v = y(k) + s.*delta(k);
```

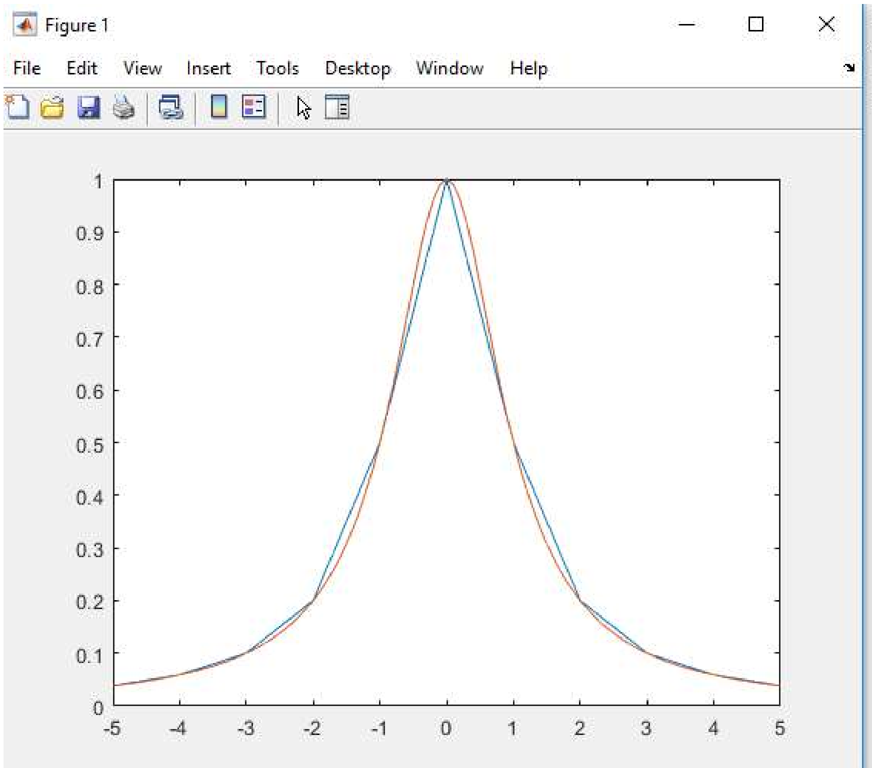
N = 6 points



Command Window

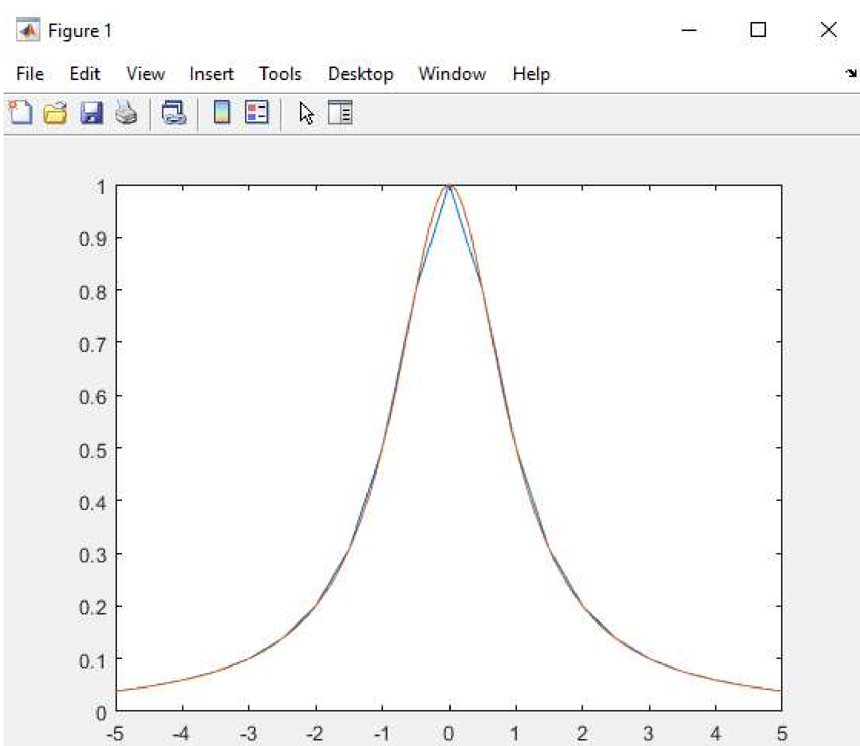
```
>> x = [-5:2:5];
>> f = @(x) 1./(1+x.^2);
>> u = [-5:.1:5];
>> y = f(x);
>> g = 1./(1 + u.^2);
>> plot(u,piecelin(x,y,u),u,g)
>>
```

N = 11 points




```
>> x = [-5:1:5];
>> y = f(x);
>> plot(u,piecelin(x,y,u),u,g)
>>
```

N = 21 points




```
>> x = [-5:.5:5];
>> y = f(x);
>> plot(u,piecelin(x,y,u),u,g)
>>
```

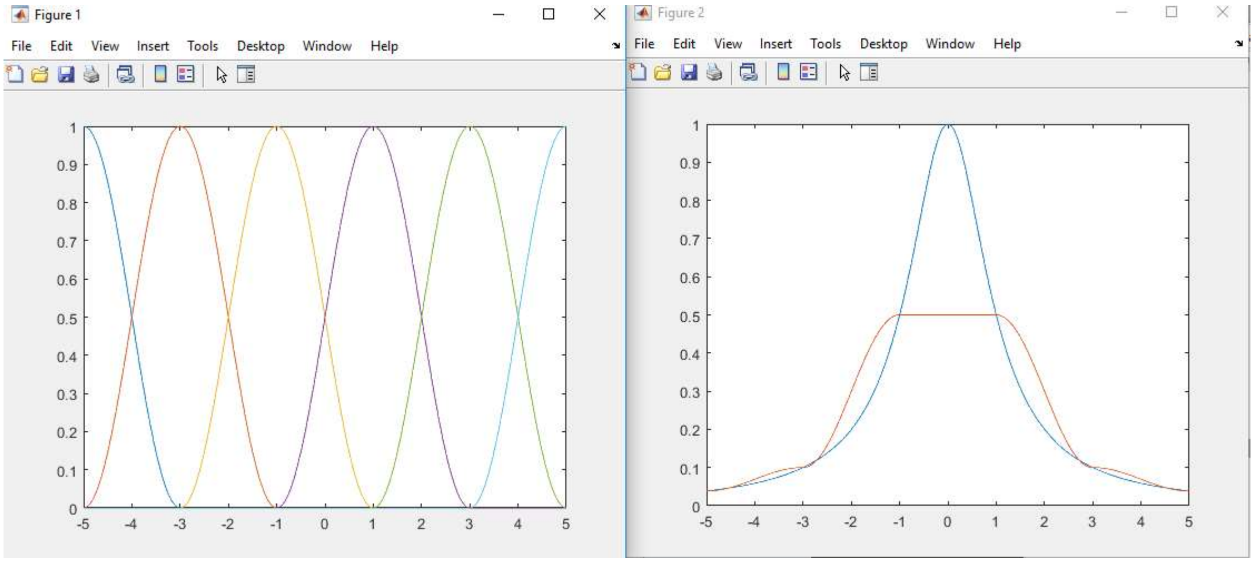
The approximation got better with more points being used.

7. [Raised Cosine Interpolation]

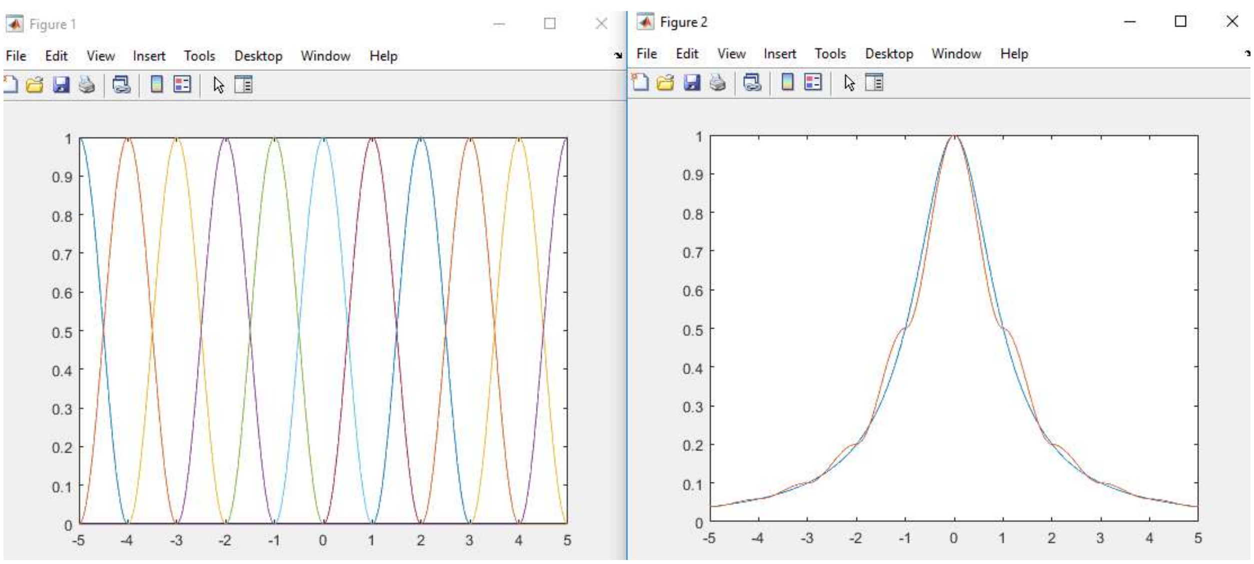
Code:

```
function raisedCos(n)
x = [-5:.01:5];
fx = 1./(1.+x.^2);
length = size(x,2);
mat = zeros(length,n);
len = (length -1)/(n-1);
t = linspace(-pi, pi, len*2+1);
rc = 1/2*(1+cos(t));
mat(1:(len+1),1) = rc(len + 1:size(t,2))';
for i = 1:(n-2)
    i_start = (i-1)*len+1;
    i_end = (i-1)*len+2*len+1;
    mat(i_start:i_end,i+1) = rc';
end

mat(((n-2)*len+1):((n-1)*len+1),n) = rc(1:len+1)';
figure(1)
plot(x', mat);
xx = linspace(-5,5,n);
xxx = (1./(1.+xx.^2))';

fit = mat*xxx;
figure(2)
plot(x,fx,x,fit);
```

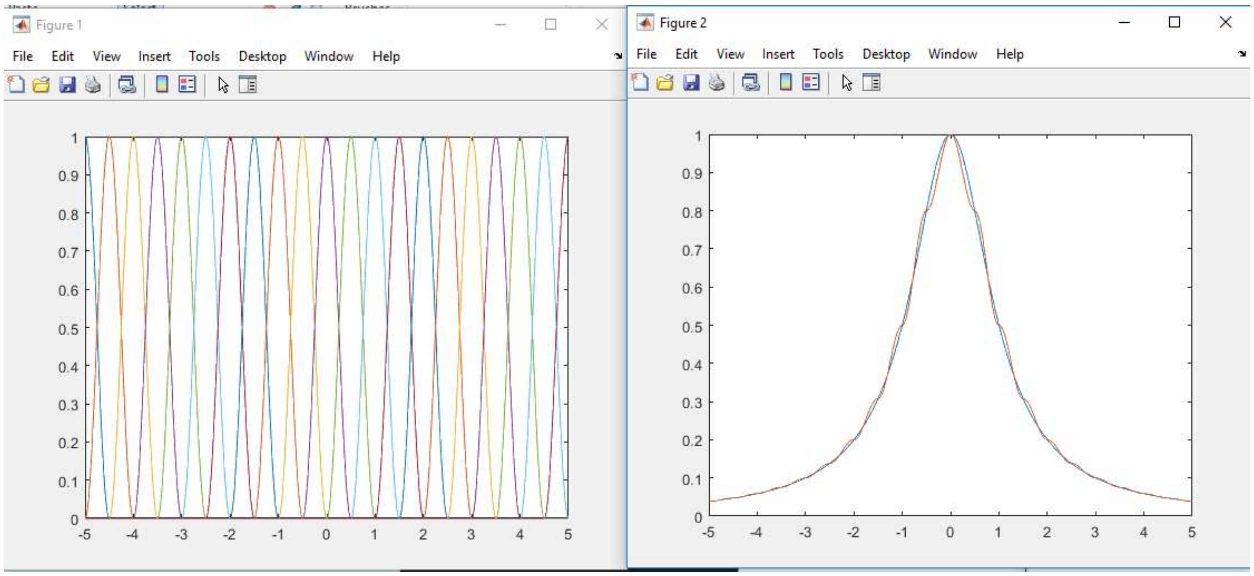
6 point



11 Point



21 Point



The Approximation gets better with more points and seems to be better than or equal to the rest of the approximations.

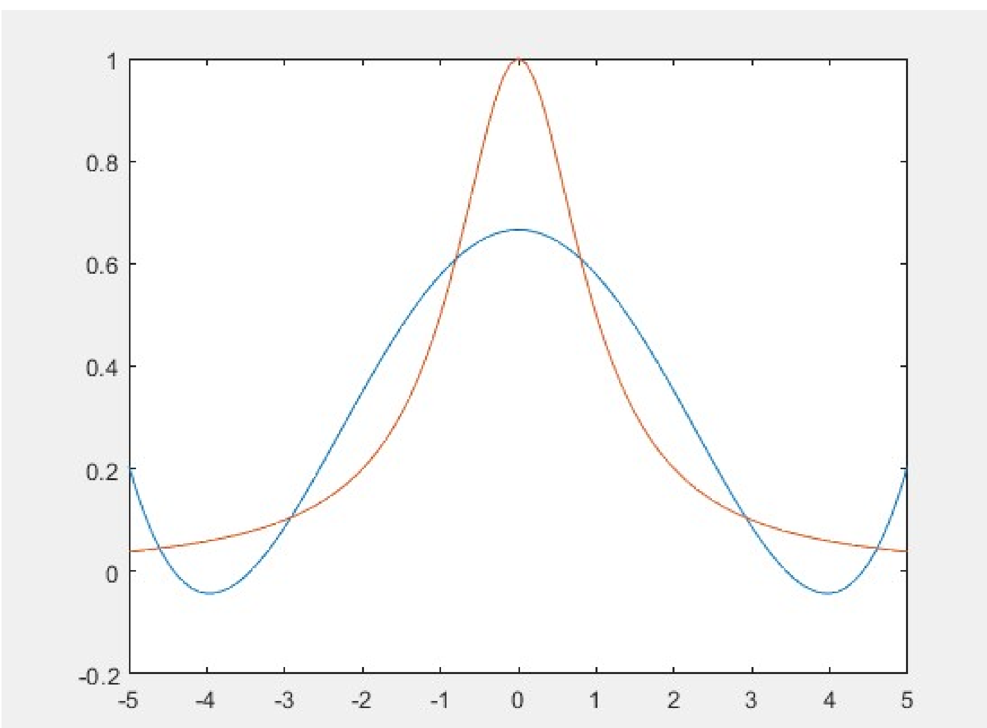
8. [Least Squares Approximation]

Code

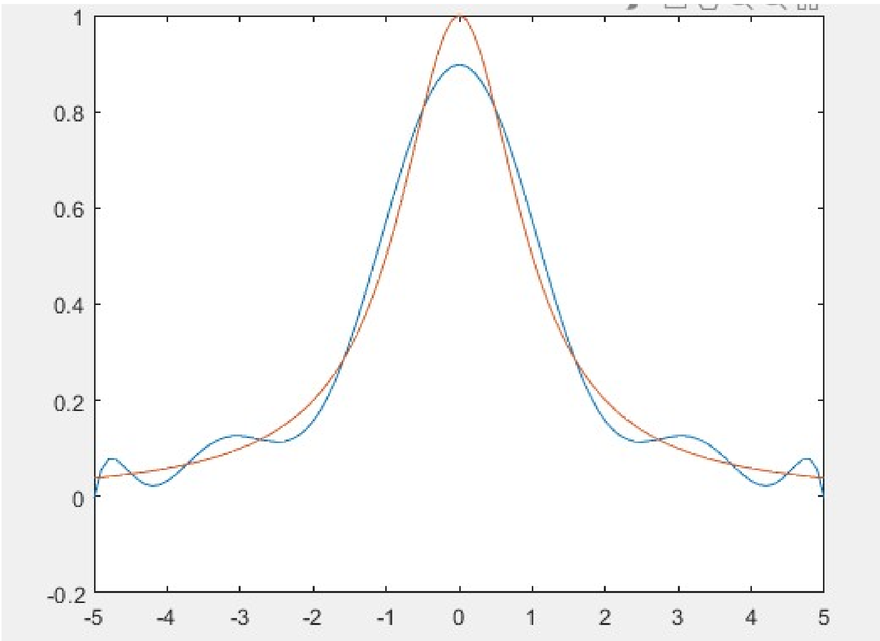
leastSquares.m

```
function leastSquares(n)
x = -5:0.1:5;
f = 1./(1+x.^2);
a = zeros(n);
b = zeros(n,1);
for j=1:n
    b(j) = sum(f.*(x.^(j-1)));
end
for j=1:n
    for c = 1:n
        a(j,c) = sum((x.^(j-1)).*(x.^(n-c)));
    end
end

ans = a\b;
fit = polyval(ans,x);
plot(x,fit,x,f);
end
```

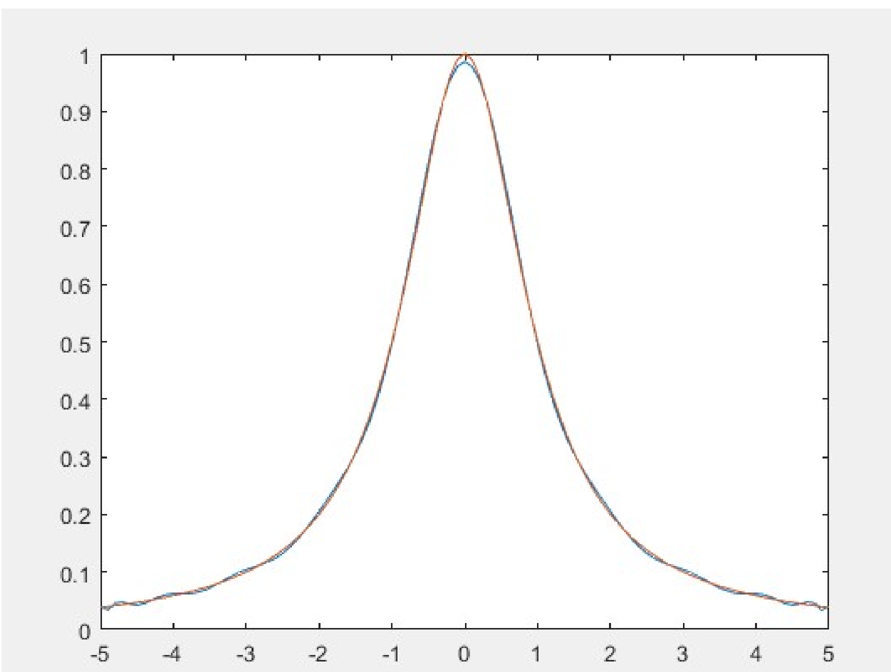
For 6 points



For 11 Points



For 21 Points



This one seems better than the approximations in 4, 5, and 6. It also gets better with higher order polynomials.