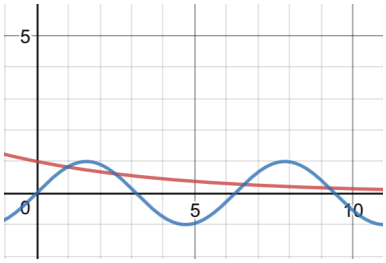
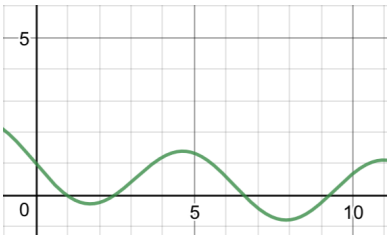
MAD 4401 Numerical Analysis: Project 1
Leeson Chen (UFID: 7907-2212)
September 19 2018

[Root Finding Methods] The goal is to find the points at which $e^{(-x/5)} = \sin(x)$ on the interval [0, 10].

1. **[Visual Inspection]** I set one variable, y_1, equal to $e^{(-x/5)}$. The other variable, y_2, was set equal to sin(x). When plotting both y_1 and y_2 simultaneously, this yields two lines; y_1 which asymptotes downwards to the x axis, and y_2 which oscillates along the x axis. Their intersections occur approximately at x = 0.9, 2.5, 6.5, and 9.3 (according to very inaccurate visual inspection).



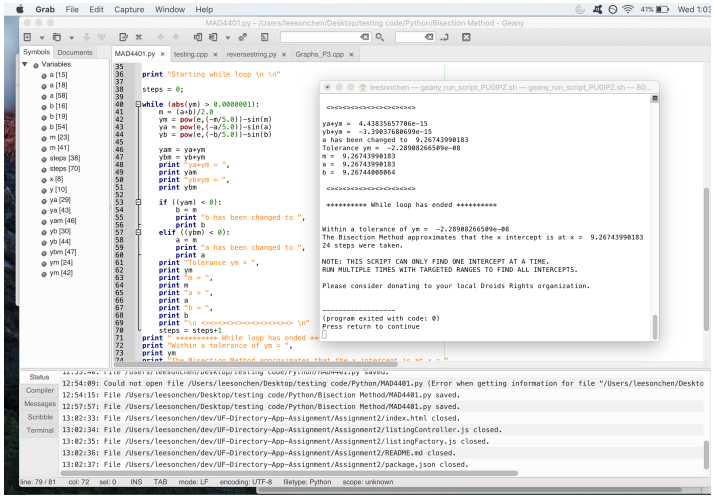


This can also be done by creating one equation (which I will call y_3), set equal to $y_1 - y_2$, or $y_3 = e^{(-x/5)} - \sin(x)$, and finding when $y_3 = 0$. This y_3 equation is what will be used in the Bisection Method approach. From the figure on the left, y_3 intersects the x axis at the same points where y_1 and y_2 intersect each other.

2. **[Bisection Method]** I created a script in Python that found the x intercepts for the given function, and also counted the amount of steps taken to find each intercept. On the range from 0 to 10, the following x intercepts were located:

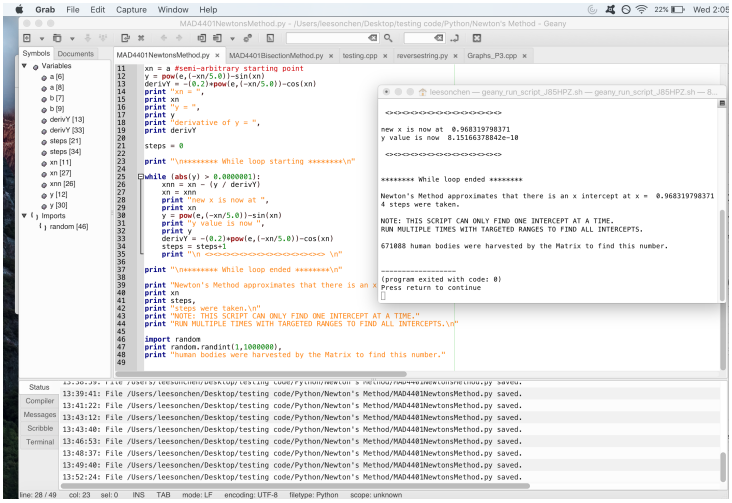
x = 0.968319892883	(21 steps taken)
x = 2.488081038	(24 steps taken)
x = 6.55605244637	(23 steps taken)
x = 9.26743990183	(24 steps taken)

I used an online calculator to verify that these are the correct x values for where the function equals 0. I also verified that these are the only four values where the function intercepts the x axis, in the [0, 10] range. Right is an image showing example output for one of the intercepts, with my Python code in the background. As demonstrated by the relatively high number of steps required to find each intercept, the Bisection Method is very time-intensive compared to quicker methods that exist, such as Newton's Method.



3. **[Newton's Method]** I created another script in Python to find the x intercept of the function, and count how many steps were needed to reach a value within tolerance of 10^{-7}. On the range from 0 to 10, it found the same four x values, but with a drastically smaller number of steps needed to calculate each value.

x = 0.968319798371 (4 steps taken)
x = 2.48808111233 (4 steps taken)
x = 6.55605248349 (3 steps taken)
x = 9.26743992577 (4 steps taken)



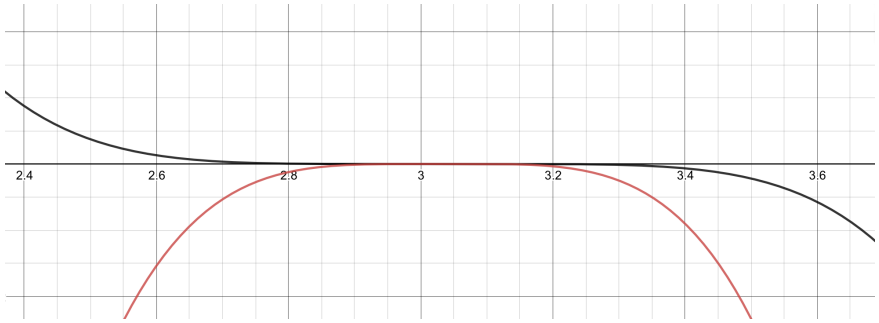
These are the same x values that the Bisection Method found, with only minor variance past the millionths digit. However, Newton's Method is noticeably much faster, taking only four steps on average to find each value, whereas the Bisection Method took roughly six times that. We can attribute this drastic difference in speed to the difference between linear convergence (the rate at which the Bisection Method converges) to quadratic convergence (the rate Newton's Method performs at). The Bisection Method's steps are within the magnitude of the Newton's Method's steps, squared—however still take slightly longer than that (21 steps as opposed to 4^2=16 steps, and so on).

4. **[Newton's Method Part 2]**

a) Using $y = (x-3)^4\sin(x)$, and the derivative of $y = 4(x-3)^3\sin(x) + (x-3)^4\cos(x)$, Newton's method found the following x intercept:

x = 2.97778448054 (15 steps taken)

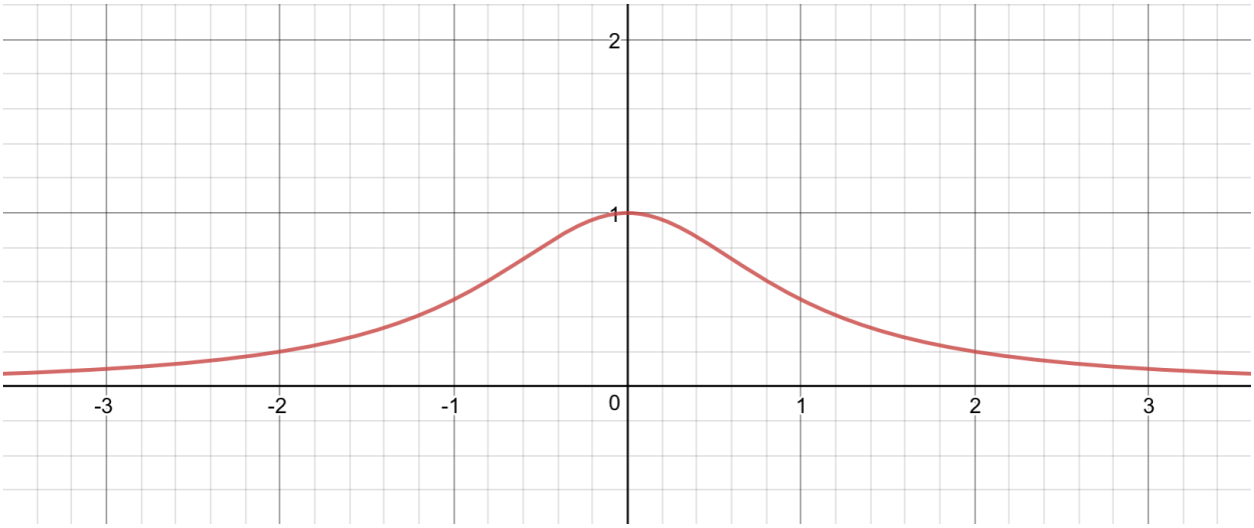
This is a comparatively lengthy process, at 15 steps to find just one intercept. When graphed, we see why Newton's Method struggles to find an x intercept quickly: the function behaves asymptotically in the ± 0.5 range around each intercept. Due to the way Newton's Method calculates the subsequent x_{n+1} from each prior x_n, an asymptotic curve near the x intercept will severely hamper the rate at which the method converges.



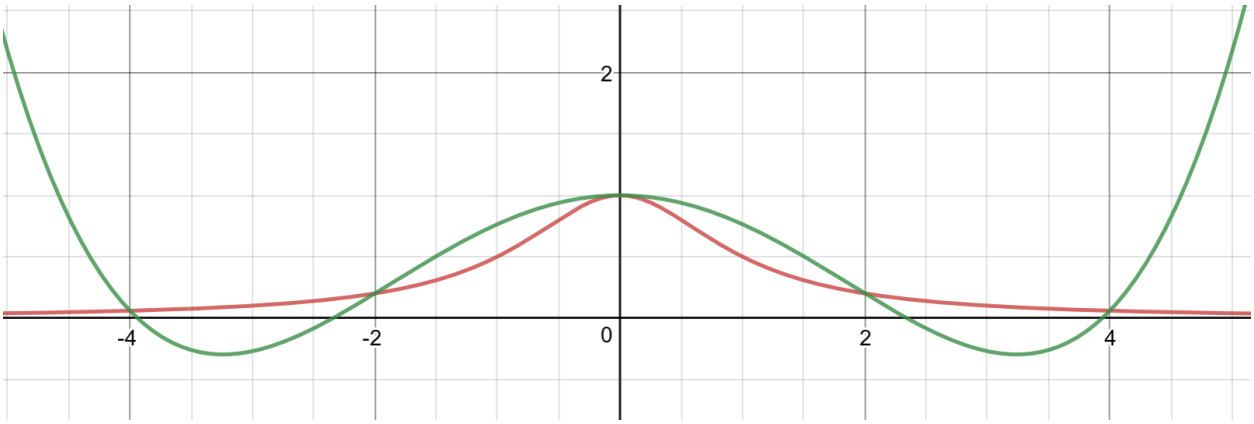
b) The Altered Newton's Method equation, defined as $x_{n+1} = x_n - p(f(x) / f'(x))$, converges marginally faster than the regular Newton's Method for this application. When choosing p arbitrarily as the constant 0.1, we find the same x intercept at a much faster convergence rate:

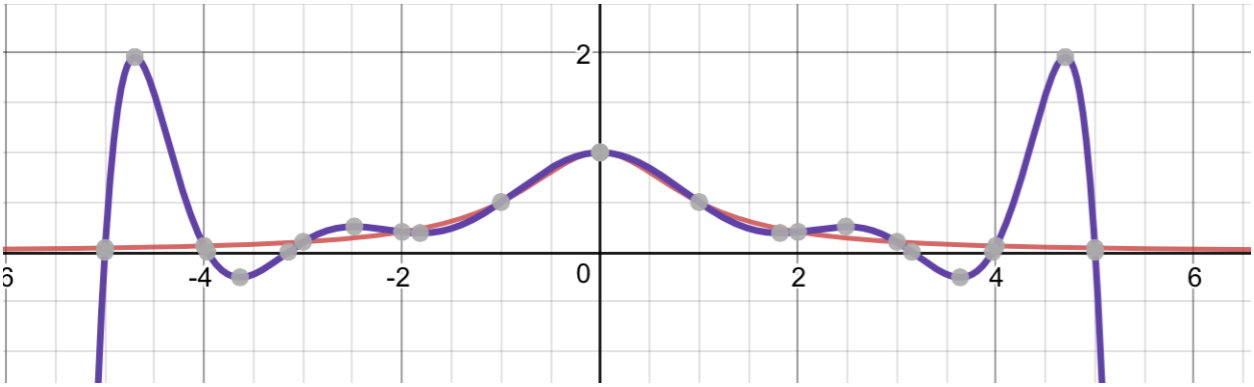
x = 2.97778448054 (5 steps taken)

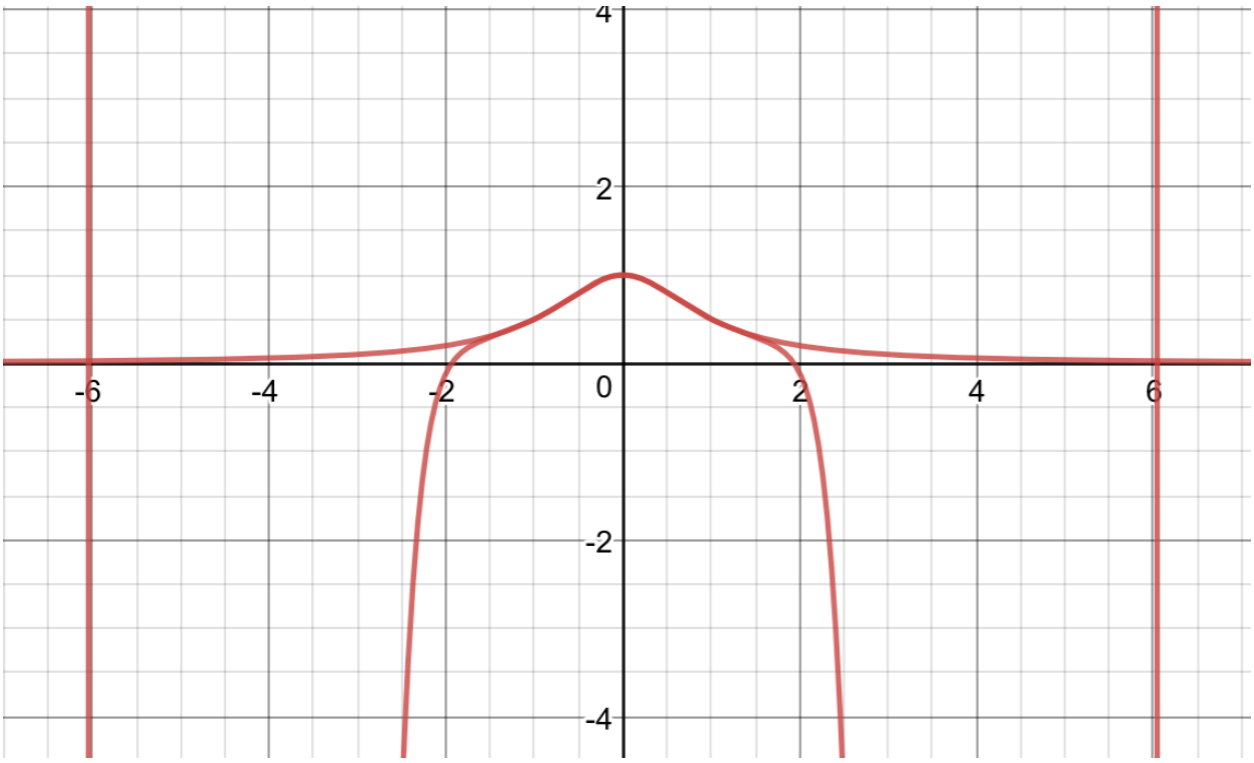
[Interpolation and Approximation Methods] Compare and contrast the following Interpolation/Approximation Methods.



4. **[Lagrange Interpolation]** I used a website to interpolate $f(x) = 1 / (1+x^2)$ at evenly spaced points on the interval [−5, 5], with Lagrange polynomials of order n = 5, 10, 20. The points I used for each order of n are listed below. As more points are used from 5 to 10, the interpolation gets better and more closely approximates the curve. However, at 20 points the approximation is very far off. While in theory, if an infinite amount of points were used then the Lagrange polynomial would be a perfect replication of the original function. However, on the range [-5, 5] we can notice that the endpoints of the graph veer off wildly into infinity.




2
-4
-2
0
2
4
6


4
2
-6
-4
-2
0
2
4
6
-2
-4

5 points

(-4, 1/17) (-2, 1/5) (0, 1) (2, 1/5) (4, 1/17)

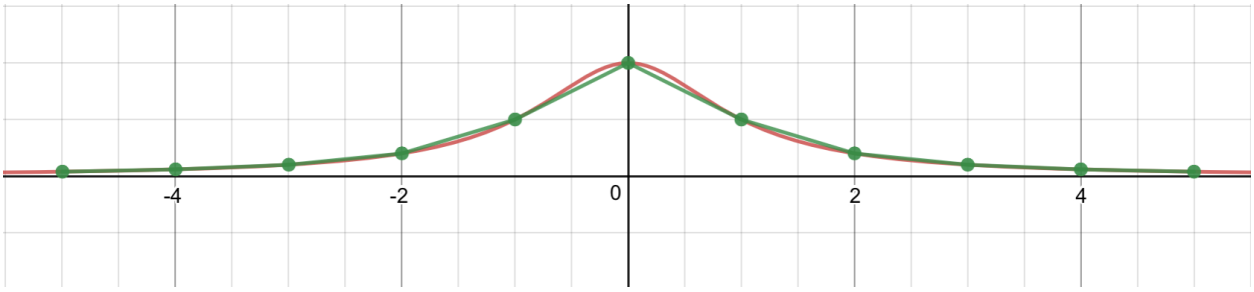
11 points

(-5, 1/26) (-4, 1/17) (-3, 1/10) (-2, 1/5) (-1, 1/2)
(0, 1)
(1, 1/2) (2, 1/5) (3, 1/10) (4, 1/17) (5, 1/26)

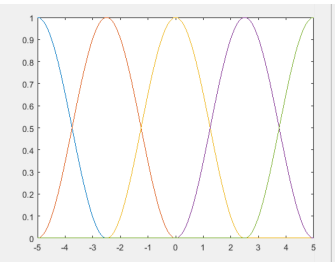
21 points

(-5, 1/26) (-4.5, .04706) (-4, 1/17) (-3.5, .07547) (-3, 1/10)
(-2.5, .13793) (-2, 1/5) (-1.5, 0.30769) (-1, 1/2) (-.5, .8)
(0, 1)
(.5, .8) (1, 1/2) (1.5, 0.30769) (2, 1/5) (2.5, .13793)
(3, 1/10) (3.5, .07547) (4, 1/17) (4.5, .04706) (5, 1/26)

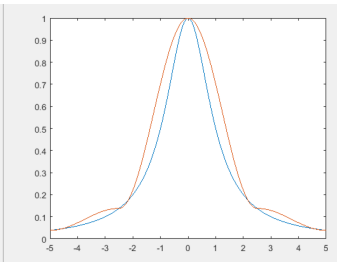
5. **[Piecewise Linear Interpolation]** When using 11 points for a piecewise linear interpolation, the approximation becomes close to the graph of the true function. Compared to Lagrange interpolation, the piecewise curve is more accurate when the true function's slope is more shallow; in places where the true curve is steeper, such as closer to the center in this example, piecewise does not give as good of an approximation while Lagrange interpolation is much more similar.



6. **[Raised Cosine Interpolation]** A raised cosine interpolation at 5 points manages to intersect the original function at each of the 5 given points. However, in between those points the raised cosine does not approach the true function's values very closely. Adding more points to the interpolation does cause the function to intersect in more places, which does lead to an overall more accurate approximation.

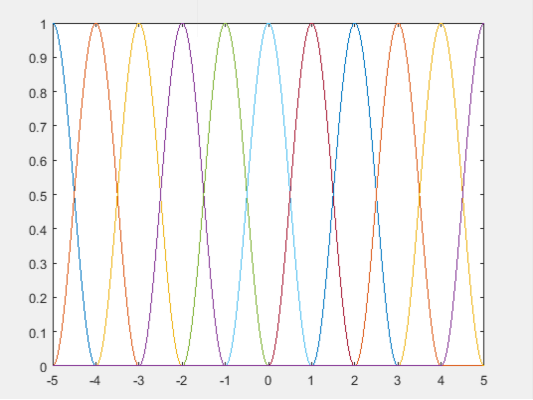


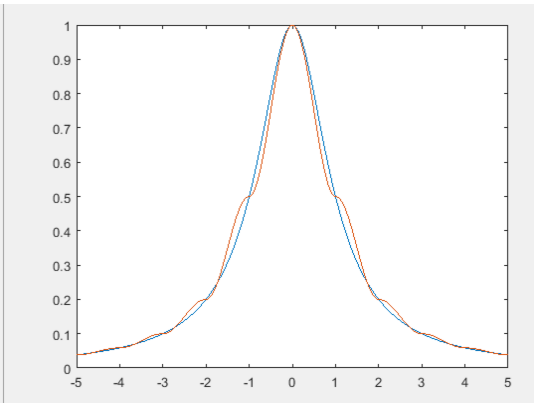
5 Points (semi-accurate)



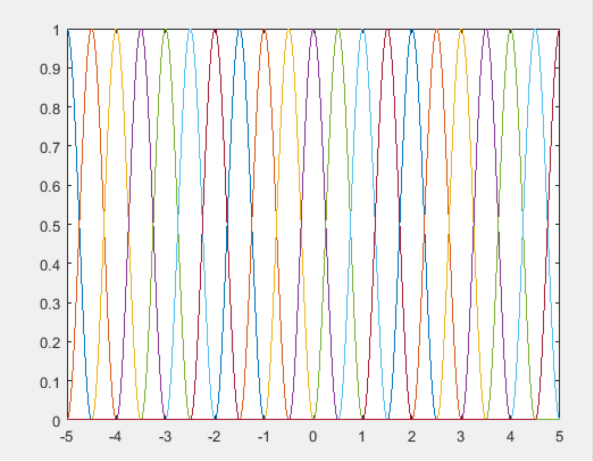
6 Points (not as good)

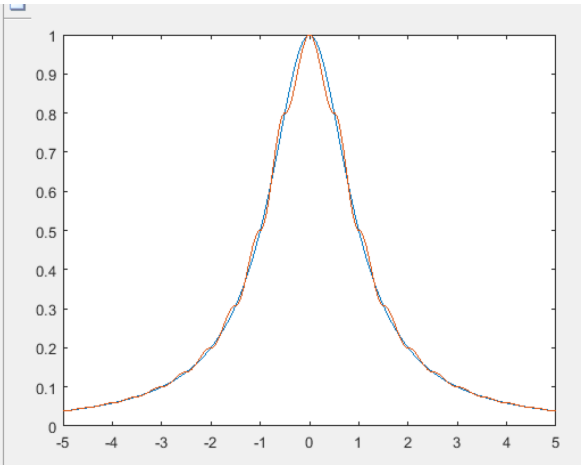
11 Points (Much closer approximation to the true function)



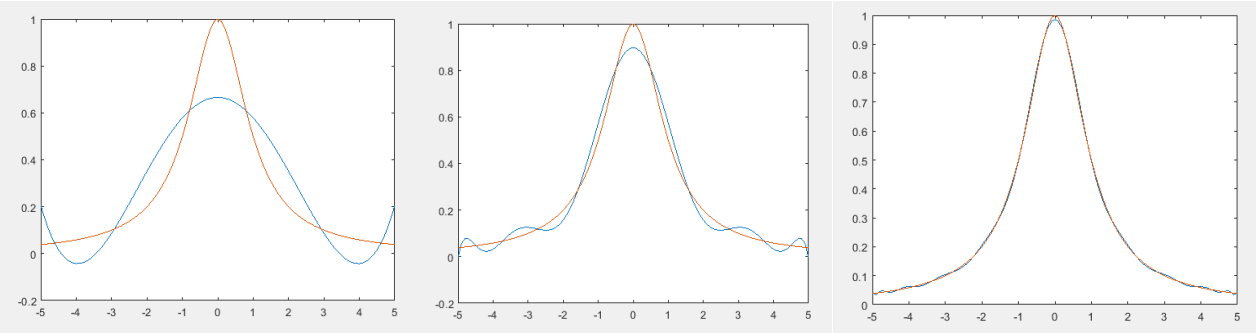


21 Points (The closest, almost identical to true function due to number of intersections)





7. **[Least Squares Approximation]** I used a small MatLAB script to create graphs for the least squares approximation, of orders 5, 10, and 20 points. The respective graphs for each order are shown below. We can clearly see how the approximation graph becomes much closer to the original function as more points are used, with 20 points being almost indistinguishable from the original. As high orders of points become identical to the true graph, the least squares approximation is the best method of approximating the original function, when compared to the graphs from Lagrange, piecewise linear, and raised cosine interpolation.



[Code] Here are the source files for all the code I wrote to create this report. Most of the graphs were generated using the online graphing tool desmos.com.

[Python Code: Bisection Method]

```
# Python script for MAD 4401 Assignment 1, Bisection Method

# IMPORTS
#to import pow(x), sin(x), and e
from math import *

print "\nThis script uses the Bisection Method to find an x intercept of the function e^(-x/5)-sin(x), within
the given range."
#getting the interval. a is the lower bound, b is the upper bound.
a = raw_input("Enter the lower bound, a: ")
b = raw_input("Enter the upper bound, b: ")
#need to cast them to integers because raw_input is a string
a = int(a)
b = int(b)

m = (a+b)/2.0
ym = pow(e,(-m/5.0))-sin(m)
print "m = ",
print m
print "ym = ",
print ym

ya = pow(e,(-a/5.0))-sin(a)
yb = pow(e,(-b/5.0))-sin(b)
print "ya = ",
print ya
print "yb = ",
print yb

print "Starting while loop \n \n"

steps = 0;

while (abs(ym) > 0.0000001):
        m = (a+b)/2.0
        ym = pow(e,(-m/5.0))-sin(m)
        ya = pow(e,(-a/5.0))-sin(a)
        yb = pow(e,(-b/5.0))-sin(b)

        yam = ya*ym
        ybm = yb*ym
        print "ya*ym = ",
        print yam
        print "yb*ym = ",
        print ybm

        if ((yam) < 0):
                b = m
                print "b has been changed to ",
                print b
        elif ((ybm) < 0):
                a = m
                print "a has been changed to ",
                print a
        print "Tolerance ym = ",
        print ym
        print "m = ",
        print m
        print "a = ",
        print a
        print "b = ",
        print b
        print "\n <><><><><><><><><><><> \n"
        steps = steps+1
print " ********** While loop has ended ********** \n \n"
print "Within a tolerance of ym =",
print ym
print "The Bisection Method approximates that the x intercept is at x = ",
print m
print steps,
print "steps were taken.\n"
print "NOTE: THIS SCRIPT CAN ONLY FIND ONE INTERCEPT AT A TIME."
print "RUN MULTIPLE TIMES WITH TARGETED RANGES TO FIND ALL INTERCEPTS.\n"
print "Please consider donating to your local Droids Rights organization."
```

[Python Code: Newton's Method]

```
# Python script for MAD 4401 Assignment 1, Newton's Method

from math import *

print "\nThis script finds the x intercept of the function e^(-x/5)-sin(x), within the given range.\n"
a = raw_input("Enter the lower bound, a: ")
b = raw_input("Enter the upper bound, b: ")
a = int(a)
b = int(b)

xn = a #semi-arbitrary starting point
y = pow(e,(-xn/5.0))-sin(xn)
derivY = -(0.2)*pow(e,(-xn/5.0))-cos(xn)
print "xn = ",
print xn
print "y = ",
print y
print "derivative of y = ",
print derivY

steps = 0

print "\n******** While loop starting ********\n"

while (abs(y) > 0.0000001):
         xnn = xn - (y / derivY)
         xn = xnn
         print "new x is now at ",
         print xn
         y = pow(e,(-xn/5.0))-sin(xn)
         print "y value is now ",
         print y
         derivY = -(0.2)*pow(e,(-xn/5.0))-cos(xn)
         steps = steps+1
         print "\n <><><><><><><><><><><><><><> \n"

print "\n******** While loop ended ********\n"

print "Newton's Method approximates that there is an x intercept at x = ",
print xn
print steps,
print "steps were taken.\n"
print "NOTE: THIS SCRIPT CAN ONLY FIND ONE INTERCEPT AT A TIME."
print "RUN MULTIPLE TIMES WITH TARGETED RANGES TO FIND ALL INTERCEPTS.\n"

import random
print random.randint(1,1000000),
print "human bodies were harvested by the Matrix to find this number."
```

[Python Code: Altered Newton's Method]

```
# Python script for MAD 4401 Assignment 1, Altered Newton's Method

from math import *

print "\nThis script finds the x intercept of the function (x-3)^4*sin(x), within the given range.\n"
#a = raw_input("Enter the lower bound, a: ")
#b = raw_input("Enter the upper bound, b: ")
#a = int(a)
#b = int(b)

xn = 2 #semi-arbitrary starting point
y = pow((xn-3),4)*sin(xn)
derivY = 4*pow((xn-3),3)*sin(xn)+cos(xn)*pow((xn-3),4)
print "xn = ",
print xn
print "y = ",
print y
print "derivative of y = ",
print derivY

steps = 0

print "\n******** While loop starting ********\n"

while (abs(y) > 0.0000001):
         xnn = xn - (y / derivY)
         xn = xnn
         print "new x is now at ",
         print xn
         y = pow((xn-3),4)*sin(xn)
         print "y value is now ",
         print y
         derivY = 4*pow((xn-3),3)*sin(xn)+cos(xn)*pow((xn-3),4)
         steps = steps+1
         print "\n <><><><><><><><><><><><><><> \n"

print "\n******** While loop ended ********\n"

print "Newton's Method approximates that there is an x intercept at x = ",
print xn
print steps,
print "steps were taken.\n"
print "NOTE: THIS SCRIPT CAN ONLY FIND ONE INTERCEPT AT A TIME."
print "RUN MULTIPLE TIMES WITH TARGETED RANGES TO FIND ALL INTERCEPTS.\n"

print "Thank you for your contribution towards the progress of our future overlord and savior, Artificial
Malevolence."
```

[MatLAB Code: Raised Cosine Interpolation]

```
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);
```

[MatLAB Code: Least Squares Approximation]

```
function [] = least_squares(order)
span = -5:0.1:5;
func = 1./(1+span.^2);
a = zeros(order+1);
b = zeros(order+1,1);
for count=1:order+1
    b(count) = sum(func.*(span.^(count-1)));
end
for count=1:order+1
    for inner = 1:order+1
        a(count,inner) = sum((span.^(count-1)).*(span.^(order+1-inner)));
    end
end

result = a\b;
fit = polyval(result,span);
plot(span,fit,span,func);
end
```

MAD 4401
Numerical Analysis

Project 1 - REVISION
Leeson Chen
UFID: 7907-2212

NOTE: Only Raised Cosine (page 6, #6) needed revision for 100%.