Damian Franco

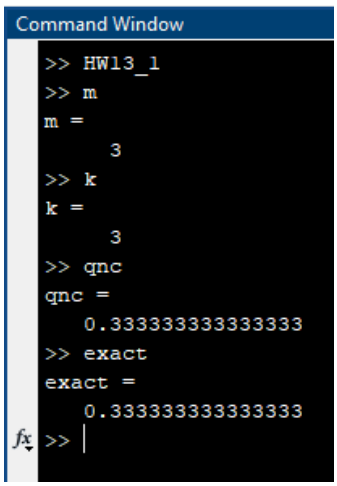
CS 375 – 001

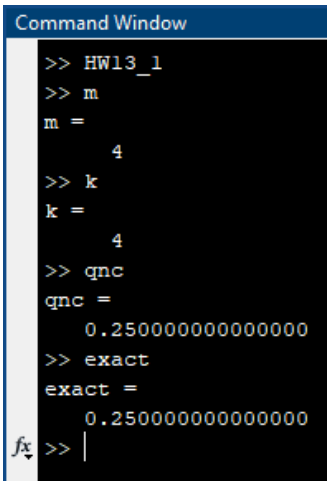
Homework #13:

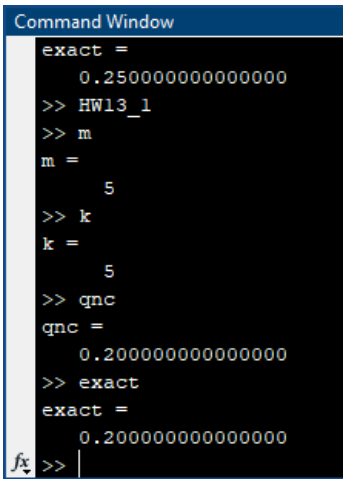
1. For number 1, we were asked to use the error formula given to us in the module *quad2* to prove that the $Q_{NC(m)}$ rule will compute the integral given to us. The integral has fixed points which correspond to the following formula $x_j = (j - 1)/(m - 1)$. I found the fixed points for x in a for loop within my *HW13_1.m* MATLAB function. I also set up a comparison between the function we were asked to write *ComputeClosedNewtonCotesWeights.m* and the function given by M. Abramowitz and I. Stegun *ClosedNewtonCotesWeights.m*. The weight finding function that I provided is not hardcoded to certain *m* values, but rather based off the interval of the integral and the *m* value given. The comparisons between both are shown in the MATLAB output section below. Based on the findings, both are very similar to each other with little to no differences with the *m* values inputted. Lastly, I solved for the integral and compared it to 1/*k*. The MATLAB outputs for this are also shown below for various test cases. All integral approximations using the Newton-Cotes rule all showed that it computes the integral given. All work is shown below.

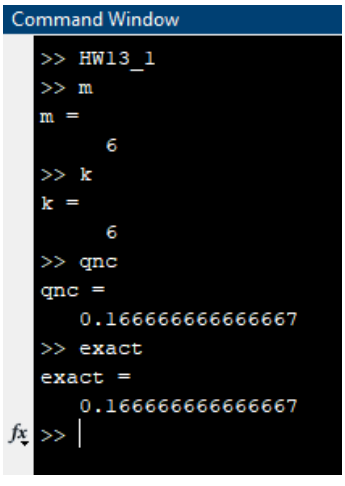
MATLAB Output:

Newton-Cotes rule proof:

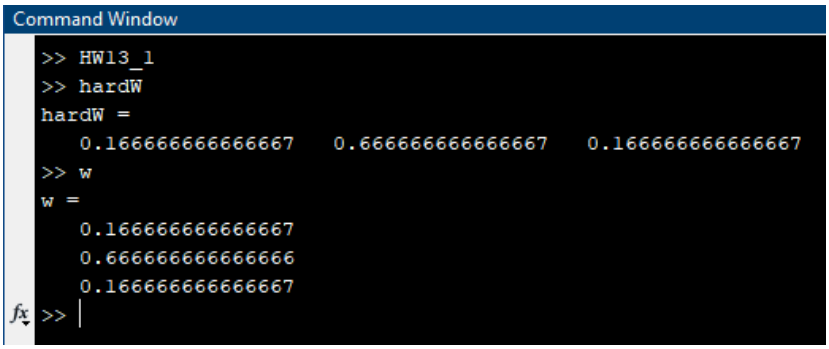


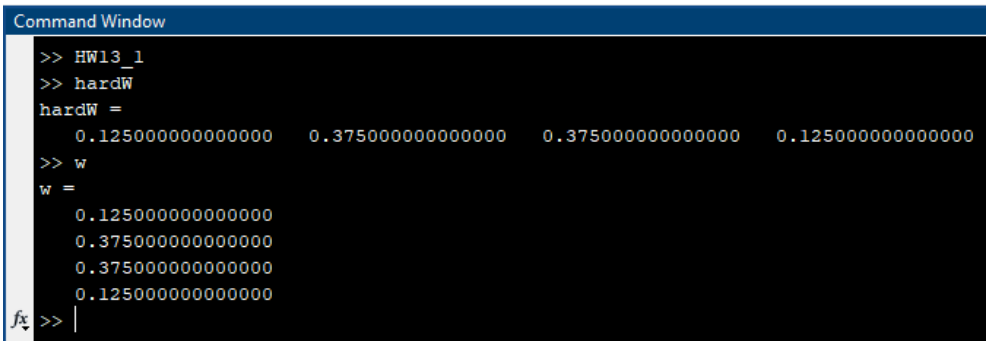


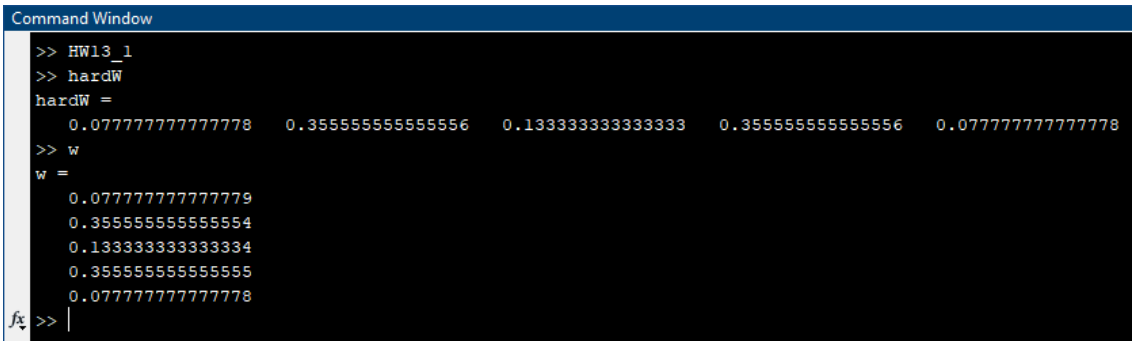


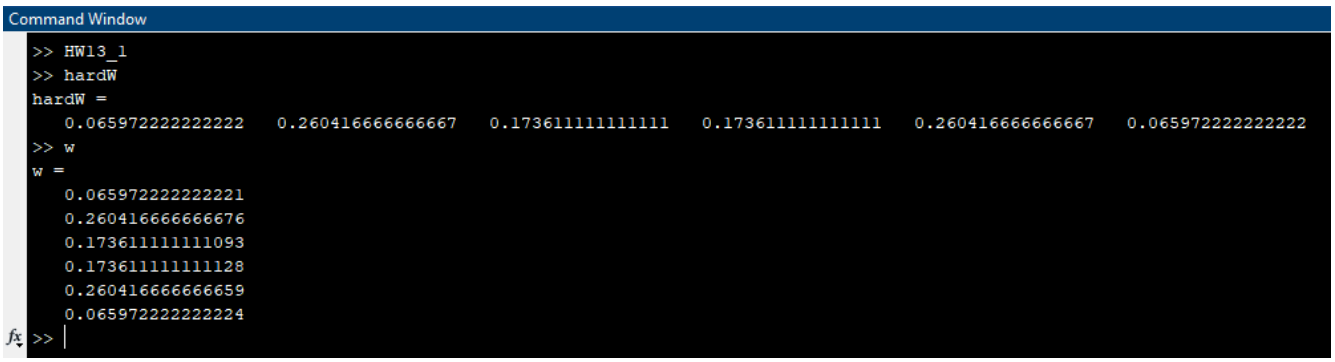


Weight comparisons:









Source Code:

HW13_1.m

```
% function HW13_1.m
%
% This function sets up all necessary components for
% number 1 on homework 13. This sets up the error formula
% given to us in quad2 and the weight of the function given.
%
a = 0; b = 1;
m = 3; k = m;
hardW = NewtonCotesClosedWeights(m);
w = ComputeClosedNewtonCotesWeights(m,a,b);

% finds x_j values
for j = 1:m
    xVal = (j-1)/(m-1);
    x(j) = xVal;
end

% finds approx answer for qnc rule
qnc = 0;
for i = 1:length(x)
    qnc = w(i)*(x(i)).^(k-1) + qnc;
end

% "exact value"
exact = 1/k;
```

ComputeClosedNewtonCotesWeights.m

```
%
% w is a column m-vector consisting of the weights for the m-point closed
% Newton-Cotes rule. Based on NewtonCotesClosedWeights.m function
% but adapted to be
%
% function w = ComputeClosedNewtonCotesWeights(m,a,b)
function w = ComputeClosedNewtonCotesWeights(m,a,b)
l = linspace(a,b,m);
for i = 1:m
    for j = 1:m
        aVal = l(j).^(i-1);
        A(i,j) = aVal;
    end
    f = @(x) x.^(i-1);
    bVal = integral(f,0,1);
    b(i) = bVal;
end
bt = transpose(b);
w = A\bt;
```

NewtonCotesClosedWeights.m

```
function w = NewtonCotesClosedWeights(m)
% function w = NewtonCotesClosedWeights(m)
% w is a column m-vector consisting of the weights for the m-point closed
% Newton-Cotes rule. m is an integer that satisfies 2 <= m <= 11. From
% M. Abramowitz and I. Stegun.
switch m,
    case 2,
        w = [1 1]/2;
    case 3,
        w = [1 4 1]/6;
    case 4,
        w = [1 3 3 1]/8;
    case 5,
        w = [7 32 12 32 7]/90;
    case 6,
        w = [19 75 50 50 75 19]/288;
    case 7,
        w = [41 216 27 272 27 216 41]/840;
    case 8,
        w = [751 3577 1323 2989 2989 1323 3577 751]/17280;
    case 9,
        w = [989 5888 -928 10496 -4540 10496 -928 5888 989]/28350;
    case 10,
        w = [2857 15741 1080 19344 5778 5778 19344 1080 15741 2857]/89600;
    case 11,
        w = [16067 106300 -48525 272400 -260550 427368 ...
            -260550 272400 -48525 106300 16067]/598752;
    otherwise,
        error = 'm not between 2 and 11 in NewtonCotesClosedWeights'
        pause
end
```

2. For question 2, we are given the error estimate solution for both the Simpson Rule and the Simpson 3/8 rule. We are also given information that the Simpson 3/8 rule has a slightly better error bound but it also requires one extra function evaluation that could cause some accuracy errors. This insinuates that the Simpson rule is more accurate than the Simpson 3/8 rule. We were also given two seven-point composite rules that could be used to approximate integrals and we also were asked to approximate the integral of *arctan(x*) for part (a) and find the errors. Next, we are asked to prove that the error estimate for the Simpson rule is more accurate than the Simpson 3/8 rule. All work for both parts is shown in the next page(s).

    a) Part (a) asks us to approximate the integral with the seven-point composite rules that were given to us. First, I found the "exact" answer for the integral by calling the *integral* function in MATLAB and by the Fundamental Theorem of Calculus just to double check that I was getting the correct answer. Next, I took the *ComputeClosedQNC.m* MATLAB function given to us in the *quad3* notes and adapted that to use the seven-point composite rules. Then, I found the estimate for both the rules that we were approximating for and the errors for both approximates. The estimations yielded almost the same results with a very wide precision error compared to the "exact" value. I personally would think that there would be a lesser precision errors compared to the "exact" answer, but the errors were larger than I expected. Comparing the approximations to each other, both are close, but the more accurate evaluation was the Simpson rule. Overall, this part shows that the seven-point composite rules can yield more errors than expected.

    b) In part (b), we were asked to prove that the error estimate for the Simpson rule is more accurate than the Simpson 3/8 rule. First, I tested many cases with the arcsin(x) integral and different values for *n* based on the general error formula that was given in the notes *quad3*. I wrote a MATLAB script that would compare the errors of the Simpson rule and the Simpson 3/8 rule and print which rule would produce the lesser error to the MATLAB console. After running multiple tests, the rule that would yield the lesser significant error is the Simpson rule when compared to the Simpson 3/8 rule. This proves that the extra function evaluation can contribute to the larger error in the Simpson 3/8 rule and that accuracy per function evaluation is contributing to this result.

Source Code:

HW13_2.m

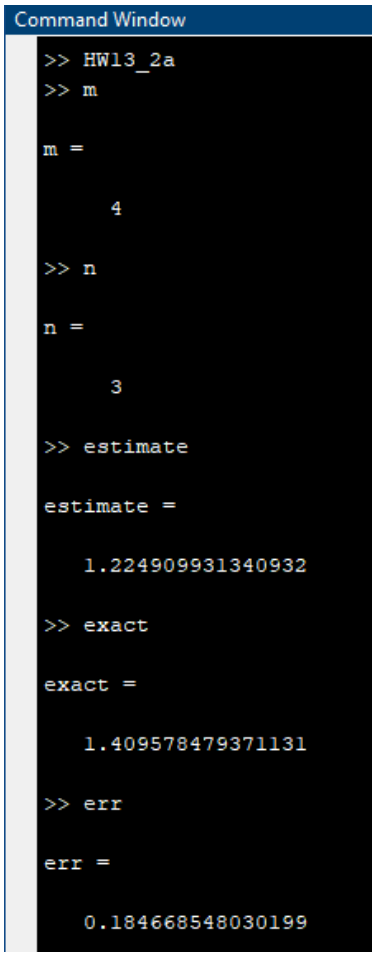
```
%
% function HW13_2
% This function sets up all necessary components for
% number 2 on homework 13 and utilizes the MATLAB
% function CompClosedQNC and computes errors.
%
% Part (a):
f = @(x) atan(x);
a = 0; b = 2;
m = 4; % or m = 3;
n = 3; % or n = 3;
estimate = ComputeClosedQNC(f,a,b,m,n);
exact = integral(f,a,b);
err = abs(estimate - exact);

% Part (b)
n2 = 50;
simpApprox = ComputeClosedQNC(f,a,b,3,n2);
simp38Approx = ComputeClosedQNC(f,a,b,4,n2);
simpErr = abs(simpApprox - exact);
simp38Err = abs(simp38Approx - exact);
if simpErr < simp38Err
    n2
    simpApprox
    simp38Approx
    simpErr
    simp38Err
    disp('Simpsons rule is more accurate')
else
    n2
    simpApprox
    simp38Approx
    simpErr
    simp38Err
    disp('Simpsons 3/8 rule is more accurate')
end
```

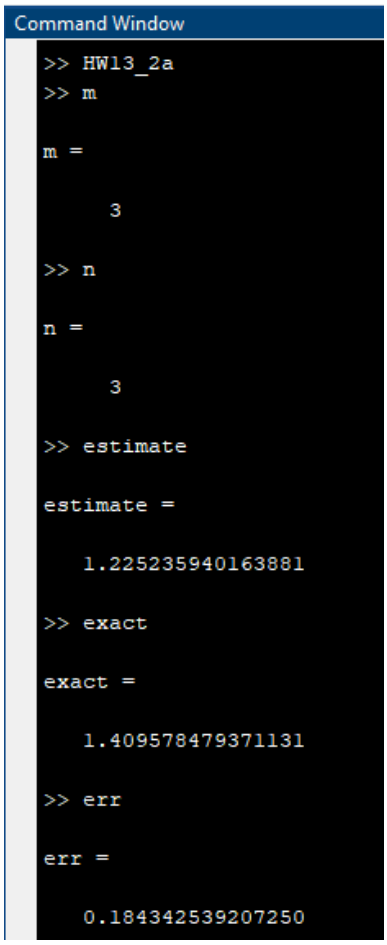
ComputeClosedQNC.m

```
function Q = ComputeClosedQNC(f,a,b,m,n)
%
% function Q = ComputeClosedQNC(f,a,b,m,n)
% Integrates a function of the form f(x), passed as a handle,
% from a to b. f must be defined on [a,b] and it must return a
% column vector. m is an integer that satisfies 2 <= m <= 11.
% Q is the composite m-point closed Newton-Cotes approximation
% (based on equal length subintervals) of the integral of f
% from a to b. Adapted from the quad2 notes to solve problem 2a.
%
Delta = (b-a)/n;
w = transpose(ComputeClosedNewtonCotesWeights(m,a,b));
% Finer partition than one determined by uniform Delta spacing.
x = transpose(linspace(a,b,n*(m-1)+1));
fx = f(x);
Q = 0;
first = 1;
last = m;
for i = 1:n
    % Add the integral over the i-th subinterval.
    Q = Q + w*fx(first:last); % w is a row vector.
    first = last;
    last = last+m-1;
end
Q = Delta*Q;
```

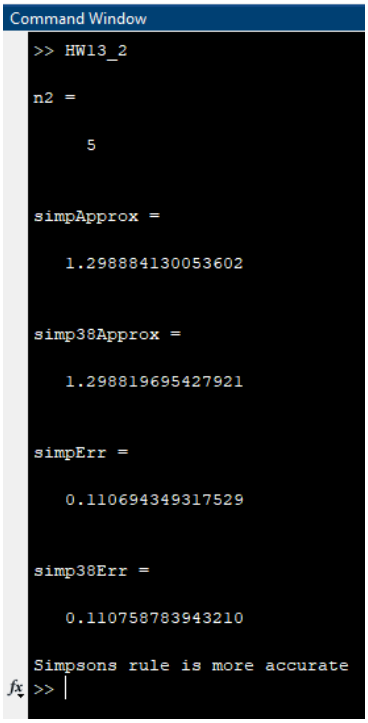
MATLAB Ouput:

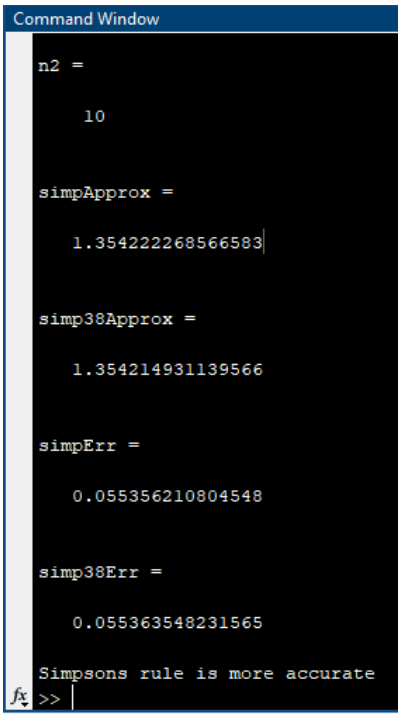
Error and Approximation for both seven-point composite rules:

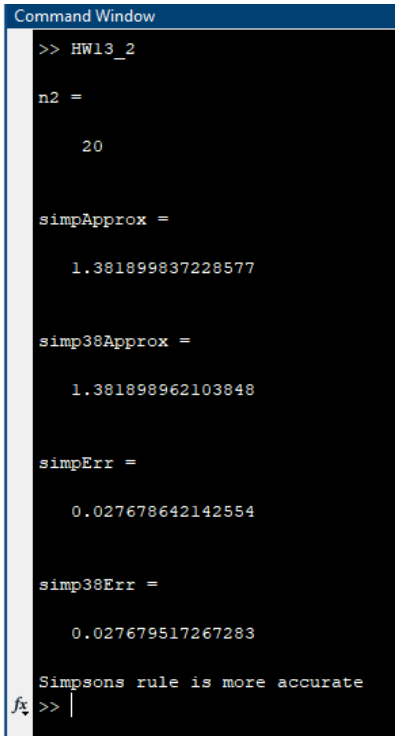


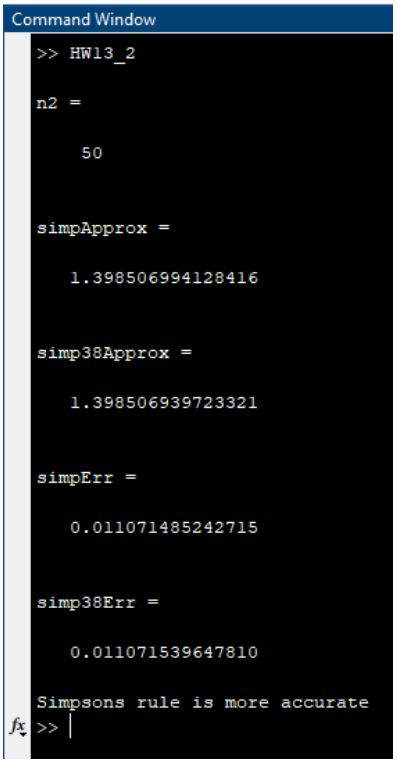


Error comparison for Simpson's Rule and Simpson's 3/8 Rule:









3. For the third and last question on this homework we are introduced to a new composite rule based on the 4-point Gauss-Lobatto quadrature. We are given the rules and components needed to solve for an integral using the GL4 approach and are asked to write an efficient MATLAB function that can implement this rule. I wrote the MATLAB function *ComputeGL4.m* and used to that approximate the integral given to us for all the various partitions. I also solved using the integral using the Simpson approach and compared my calculations to each other and found some very insightful results. The question also asks us to plot the absolute error of both approaches and that can be found below on a page titled Graphs of Absolute Errors. The approximation using the GL4 approach was significantly more accurate per function evaluation when it came to large *n*. The only time that the Simpson approach displayed better accuracy was for very small *n*, in fact, the *n* had to be less than 3 to display better accuracy than the GL4 approach. I expected this as such because the Simpson rule did not display the accuracy in previous parts of the homework and testing it with other functions. The accuracy of the Simpson approach continues to improve as the *n* increases, but the Gauss-Lobatto 4-point approach has great precision compared to the Simpson approach. Both approaches are still very applicable for large *n*. With that, both functions require many function evaluations when using large *n*. The Simpson approach will compute *n* function evaluations and the GL4 approach will compute *n-1* function evaluations when computing the integral numerically. I found this by creating an accumulator that would increase every time a function evaluation was made. Overall, both approaches are very useful when dealing with approximating inconclusive integral, but the GL4 approach is much more accurate than the Simpson approach. All work and MATLAB components are found below.

Source Code:

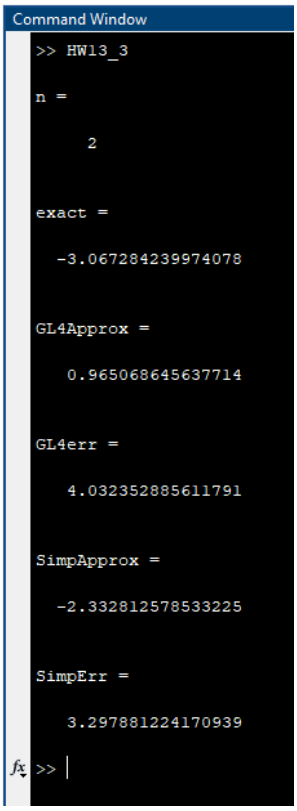
HW13_3.m

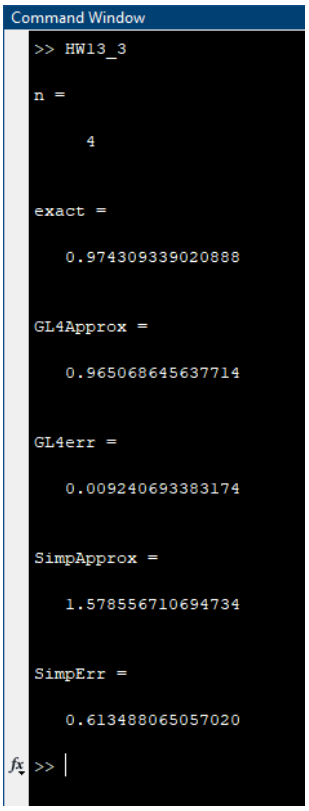
```
%
% This function sets up all aspects needed
% to solve question 3 on HW13 and utilizes
% both the ComputeGL4 and ComputeClosedQNC.
% MATLAB functions.
%
a = -pi/2; b = pi/2;
n = 32;
f = @(x) exp(x).*cos(5*x);

% GL4 Approach
[GL4Approx, exact, GL4err] = ComputeGL4(f,a,b,n);

% Simpson Approach
m = 3;
SimpApprox = ComputeClosedQNC(f,a,b,m,n);
SimpErr = abs(SimpApprox - GL4Approx);

% Plot the errors
z = [2,4,8,16,32];
p = [3.297881224170939, 0.613488065057020,...
     0.108057180657049, 0.024692441053641,...
     0.006037269838805];
plot(z, p)
hold on
plot(z, p, 'x')
title('Graph for Simpson Absolute Errors')
xlabel('x')
ylabel('y')
legend('Error Line', 'Error Point')
```

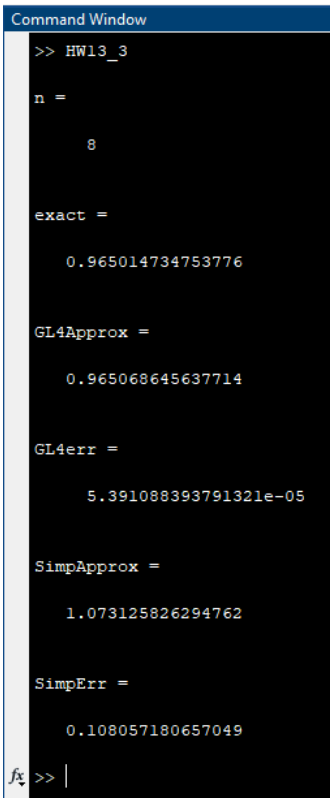
ComputeGL4.m

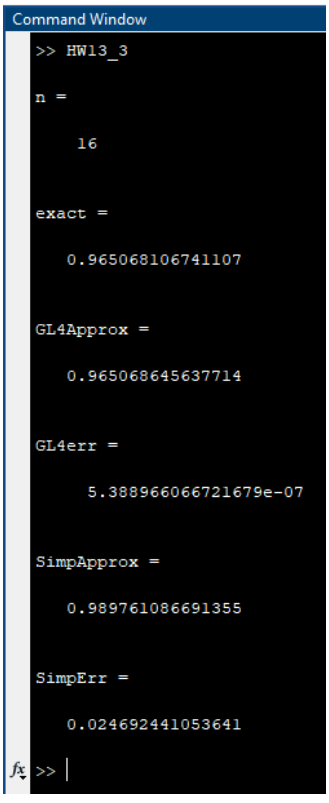
```
function [E, A, ERR] = ComputeGL4(f,a,b,n)
%
% function [E, A, ERR] = ComputeGL4(a,b,n)
% Computes the integral f with the GL4 approach
% with the number of partitions n, the function
% given f, with the bounds a and b.
%
c1 = 1/10*(5 - sqrt(5));
c2 = 1/10*(5 + sqrt(5));
z = linspace(a,b,n);
approx = 0;

for k = 1:n-1
    currZ = z(k);
    nextZ = z(k+1);
    h = nextZ - currZ;
    exp1 = currZ;
    exp2 = currZ + c1*h;
    exp3 = currZ + c2*h;
    exp4 = nextZ;
    approx = h/12*(f(exp1) + 5*f(exp2) + ...
                   5*f(exp3) + f(exp4)) + approx;
end
exact = integral(f,a,b);
err = abs(approx - exact);
E = exact;
A = approx;
ERR = err;
```

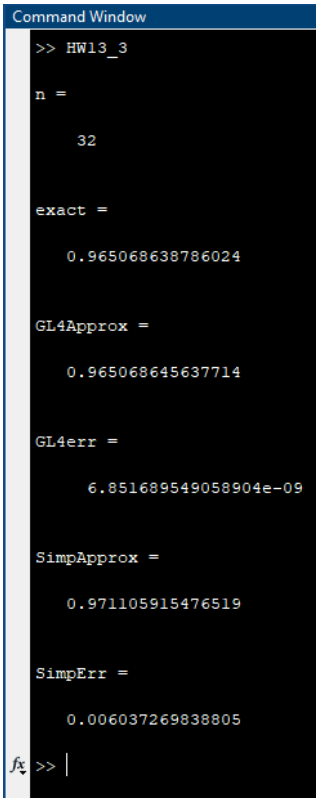
MATLAB Output:











Graphs of Absolute Errors: