Christopher Wong
999234204
MAT 128A
Programming Project 3 11/23/2016

Problem 1

Composite_trapezoidal_rule_P1.m

```
% Problem 1
% clear
format long e

% cd 'C:\Users\Christopher\Desktop\MAT 128A\Project 3';
clear E i j h T sum_vec

range = input('Specify the integration limits: ');
% Verifying the limits are valid
if (numel(range) ~= 2 || range(1) >= range(2))
  error('Your limits of integration are not valid.');
end
a = range(1);
b = range(2);

% Taking the input and converting it into a function
input_func = input('Evaluate f(x) = ', 's');
f = str2func(strcat('@(x)', input_func));

% Make sure the code does not run forever
tol = input('Error tolerance: ');
j_max = input('Safeguard: ');

% Index '1' corresponds to 0 in written vector form
% Setting up the initial trapezoidal rule step
h(1) = b-a;
T(1) = (h(1)/2)*(f(a)+f(b));
% Dividing the interval into subintervals increases precision
for j = 2:(j_max + 1)
  h(j) = (h(j-1)/2);
  for i = 1:(2^((j-1)-1))
    sum_vec(i) = f(a+(2*i-1)*h(j));
  end
  T(j) = (1/2)*(T(j-1)) + h(j)*sum(sum_vec);
  E(j) = (4/3)*abs(T(j)-T(j-1));
  % If tolerance is met, return everything
  if (E(j) <= tol)
    final_approx = T(j);
    error_return = E(j);
    iteration_index = j-1;
    nfcount = 1+2^(j-1);
    output = [final_approx, error_return, iteration_index, nfcount];
    fprintf(['Final approximation: %d\n', ...
             'Conservative error estimate: %d\n', ...
```

```
            'Final iteration index: %d\n', ...
            'Number of function evaluations: %d\n'], output);
    return
  end
end
```

Problem 2

Composite_Simpsons_rule_P2.m

```
% Problem 2
% clear
format long e

% cd 'C:\Users\Christopher\Desktop\MAT 128A\Project 3';
clear E i j h S sum_vec B A

range = input('Specify the integration limits: ');
% Verifying the limits are valid
if (numel(range) ~= 2 || range(1) >= range(2))
  error('Your limits of integration are not valid.');
end
a = range(1);
b = range(2);

% Taking the input and converting it into a function
input_func = input('Evaluate f(x) = ', 's');
f = str2func(strcat('@(x)', input_func));

% Make sure the code does not run forever
tol = input('Error tolerance: ');
j_max = input('Safeguard: ');

% Index '1' corresponds to 0 in written vector form
% Setting up the initial trapezoidal rule step
h(1) = b-a;
B(1) = 2*f((a+b)/2);
A(1) = f(a) + B(1) + f(b);
S(1) = (h(1)/6)*(A(1)+B(1));
% Dividing the interval into subintervals increases precision
for j = 2:(j_max + 1)
  h(j) = (h(j-1)/2);
  for i = 1:(2^(j-1))
    sum_vec(i) = f(a+(2*i-1)*(h(j)/2));
  end
  B(j) = 2*sum(sum_vec);
  A(j) = A(j-1) + B(j);
  S(j) = (h(j)/6)*(A(j)+B(j));
  E(j) = (16/15)*abs(S(j)-S(j-1));
  % If tolerance is met, return everything
  if (E(j) <= tol)
    final_approx = S(j);
    error_return = E(j);
    iteration_index = j-1;
```

```
    nfcount = 1+2^j;
    output = [final_approx, error_return, iteration_index, nfcount];
    fprintf(['Final approximation: %d\n', ...
             'Conservative error estimate: %d\n', ...
             'Final iteration index: %d\n', ...
             'Number of function evaluations: %d\n'], output);
    return
  end
end
```

Problem 3

NotAKnot_cubic_spline_integral_P3.m

```
% Problem 3
% clear
format long e

% cd 'C:\Users\Christopher\Desktop\MAT 128A\Project 3';

x = input('Input a vector of x values: ');
a = x(1);
b = x(numel(x));

% Taking the input and converting it into a function
input_func = input('Evaluate f(x) = ', 's');
f = str2func(strcat('@(x)', input_func));

if (numel(x) < 3)
  error('x must be of length 3 or greater');
end

% Constructing the not-a-knot cubic spline and evaluating its integral
y = f(x);
pp = spline(x, y);
I_spline = integral(@(x)ppval(pp, x), a, b);
```

Christopher Wong
999234204
MAT 128A

Problem 4

I = I(b) = erf(b) := $\frac{2}{\sqrt{\pi}} \int_0^b exp(-x^2)dx$ for b = 0.4, 0.8, 1.2, 1.6, 2.0.

Composite trapezoidal rule

b	j	nfcount	T_j	E_j	$\lvert I - T_j \rvert$
0.4	18	262145	4.283923550465203e-01	5.962637790920173e-13	1.481592626362271e-13
0.8	19	524289	7.421009647074746e-01	7.346715828286202e-13	1.859623566247137e-13
1.2	20	1048577	9.103139782295727e-01	2.837730050941900e-13	6.272760089132135e-14
1.6	19	524289	9.763483833444249e-01	8.668621376273222e-13	2.191580250610059e-13
2.0	19	524289	9.953222650188525e-01	3.976078725524227e-13	1.002531391236516e-13

Composite Simpson's rule

b	j	nfcount	S_j	E_j	$\lvert I - S_j \rvert$
0.4	8	513	4.283923550466771e-01	1.364242052659392e-13	8.604228440844963e-15
0.8	9	1025	7.421009647076676e-01	1.080024958355352e-13	7.105427357601002e-15
1.2	8	513	9.103139782296621e-01	4.121147867408581e-13	2.664535259100376e-14
1.6	9	1025	9.763483833446057e-01	6.271723880975818e-13	3.830269434956790e-14
2.0	10	2049	9.953222650189488e-01	6.608047442568932e-14	3.885780586188048e-15

Spline-based approach

Equally-spaced points

b	n	I_{spline}	$\lvert I - I_{spline} \rvert$
0.4	512	4.283923550466666e-01	1.831867990631508e-15
0.8	1024	7.421009647076594e-01	1.110223024625157e-15
1.2	512	9.103139782296437e-01	8.215650382226158e-15
1.6	1024	9.763483833446516e-01	7.549516567451065e-15
2.0	1024	9.953222650189529e-01	2.220446049250313e-16

Chebyshev points

b	n	I_{spline}	$\lvert I - I_{spline} \rvert$
0.4	512	4.283923550466592e-01	9.214851104388799e-15
0.8	1024	7.421009647076543e-01	6.217248937900877e-15
1.2	512	9.103139782299158e-01	2.803313137178520e-13
1.6	1024	9.763483833448057e-01	1.616484723854228e-13
2.0	1024	9.953222650189901e-01	3.741451592986778e-14

I notice that the composite trapezoidal rule is not as efficient as the composite Simpson's rule. The composite trapezoidal rule takes anywhere from hundreds to a few thousand times the number of function evaluations of the composite Simpson's rule and still results in a lower accuracy. On the other hand, the spline-based approach and the composite Simpson's rule are similar in accuracy and efficiency for this function. Also, the spline-based approach is more accurate using equally-spaced points compared to Chebyshev points. This is surprising because from Project 1, Chebyshev points were a huge improvement over equally-spaced points for polynomial interpolation. If I had to rank the numerical integration methods in terms of efficiency and accuracy,

(1) Composite Simpson's Rule
(2) Spline-based approach with equally-spaced points
(3) Spline-based approach with Chebyshev points
(4) Composite trapezoidal rule

$I = \pi = 2\int_{-1}^{1} \frac{1}{1+x^2} dx$

Composite trapezoidal rule

j	nfcount	T_j	E_j	$\lvert I - T_j \rvert$
21	2097153	3.141592653589641e+00	5.980401359314176e-13	1.518785097687214e-13

Composite Simpson's rule

j	nfcount	S_j	E_j	$\lvert I - S_j \rvert$
7	257	3.141592653589785e+00	6.077508866534724e-13	8.437694987151190e-15

Spline-based approach

Equally-spaced points

n	I_{spline}	$\lvert I - I_{spline} \rvert$
256	3.141592653529198e+00	6.059552859483119e-11

Chebyshev points

n	I_{spline}	$\lvert I - I_{spline} \rvert$
256	3.141592653244173e+00	3.456204211715885e-10

For this function, I notice that the composite trapezoidal rule uses about 8,000 times the number of function evaluations of the composite Simpson's rule and still results in a lower accuracy. On the other hand, the spline-based approach, while not as accurate as the composite trapezoidal rule, is still pretty accurate given that it uses 256 function evaluations compared to the 2,097,153 of the composite trapezoidal rule. Comparing the spline-based approach with the composite Simpson's rule, I find that the composite Simpson's rule is the most accurate and efficient method of the three, which is what I found in the previous function. In addition, Chebyshev points are shown to be once again inferior to equally-spaced points when using the spline-based approach. My ranking of the numerical integration methods is the same as the previous function:

(1) Composite Simpson's Rule
(2) Spline-based approach with equally-spaced points
(3) Spline-based approach with Chebyshev points
(4) Composite trapezoidal rule