

11/25

12

Problem 1:

- a.) *Write a composite trapezoidal rule routine to compute an approximation to the integral of a function, f , over an interval $[a, b]$. Your routine should take as inputs: the integrand f , the endpoints a and b , and the number the number of subintervals n .*

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define f as inline function
% a is left endpoint
% b is right endpoint
% n is the number of subintervals to be used
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function[int] = trapezoid(f, a, b, n)
h = (b-a)/n;           %Set Interval lengths based on interval
                        %length and number of subintervals
x=a;                   %Initiate beginning of 1st interval

mid = 0;               %Initialize middle term (summation term) of Composite Trap
                        %Rule
for i = 1:(n-1)        %calculate summation term in the Rule
    x=x+h;
    mid = mid + f(x);
end
                        %Calculate the integral approximation with composite Trap
int = (h/2)*(f(a) + 2*mid + f(b));
    
```

13
13
18
44

- b.) *Write a composite Simpson's rule integration routine that takes the same inputs as your composite trapezoidal rule routine.*

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define f as inline function
% a is left endpoint
% b is right endpoint
% n is the number of subintervals to be used
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function[int] = simpsons(f, a, b, n)
h = (b-a)/n;           %Calculate interval length
x_0 = a;               %initialize first x to be the left endpoint
g_0 = f(a);           %calculate function value at left end point

for i = 1:n            %Set grid spacing
    x(i) = x_0 + i*h;
end
g = f(x);              %calculate function value at the grid points

m1 = 0;                %Initialize the two summation terms in the formula
m2 = 0;
for i = 1:(n/2)-1     %calculate summation term (even terms)
    m1 = m1 + g(2*i);
end
for i = 1:(n/2)       %calculate summation term (odd terms)
    m2 = m2 + g(2*i-1);
end
                        %Calculate entire estimating integral with Simpson's composite Rule
int = (h/3)*(g_0 + 2*m1 + 4*m2 + g(n));
    
```

if n=1
value b
Simpson's rule

c.) Write a composite 3-point Gaussian quadrature routine that takes the same inputs as your composite trapezoidal rule routine.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define f as inline function
% a is left endpoint
% b is right endpoint
% n is the number of subintervals to be used
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function[int] = gaussian(f, a, b, n);

h = (b-a)/n;           %set grid spacing
int = 0;               %initialize integral to zero
for i = 1:n
    l = a + (i-1)*h;   %calculate left endpoint of the i_th subinterval
    r = a + (i)*h;     %calculate right endpoint of the i_th subinterval
    %Transform the points to be evaluated from the [-1, 1] used
    %in gaussian quadrature
    u1 = -((r-l)/2)*(sqrt(3/5)) + (r+l)/2 ;
    u2 = (r+l)/2;
    u3 = ((r-l)/2)*(sqrt(3/5)) + (r+l)/2 ;
    %compute the integral with proper weights for i_th subinterval
    comp_int = ((r-l)/2)*((5/9)*f(u1) + (8/9)*f(u2) + (5/9)*f(u3));

    %combine integral of subinterval with previously claculated integrals
    int = int + comp_int;
end
    
```

d.) Apply each of these composite quadratures to approximate

$$\int_0^1 e^x dx$$

Make a table of the results for $n = 2, 4, 8, 16, 32$, and a table of the errors.

n\error in using:	Composite Trapezoidal	Composite Simpson's	Composite Gaussian
2	0.035649264005780	0.000579323417547295	0.0000000132050086421032
4	0.008940076098471	0.000037013462701463	0.0000000002076450122956
8	0.002236763705256	0.000002326240851502	0.0000000000032500668823
16	0.000559300120949	0.000000145592846224	0.00000000000000515143483
32	0.000139831857282	0.000000009102726128	0.00000000000000008881784

How is the order of accuracy demonstrated in the table of errors? Comment on your results.

We know that the composite Trapezoidal Rule Quadrature has order of accuracy $O(h^2)$, this can be seen in the errors because if for example n goes from 4 to 8 (i.e. the length of the intervals decreases by half), the error decreases by a factor of 4 (i.e. is multiplies by $(1/2)^2$).

Ex. $0.00894/4 = 0.002235$

The Composite Simpson's rule follows the same trend except that since it has order of accuracy $O(h^4)$, the error gets multiplied by a factor of $(1/2)^4$ (or $1/16$) instead.

Ex. $0.00003701/16 = 0.00000231$

The 3-point Composite Gaussian however shows a much greater increase in accuracy since it is order $O(h^6)$. The error gets multiplied by a factor of $(1/2)^6$ (or $1/64$)

Ex. $0.0000000132/64 = 0.000000002$

13
Problem 2:

Consider the Integral

$$\int_0^1 x^{-\frac{2}{3}} e^{-x^2} dx$$

The integrand is singular, although it is integrable. The value of this integral is approximately 2.6647910962558108.

- a.) Use MATLAB's built-in quadrature, *quad*, to compute this integral, as written, with error tolerances of 10^{-8} , 10^{-10} , 10^{-12} . What values of the approximate integral does this give? What are the actual errors? How many function evaluations are required?

Tolerance	Approximate Integral	Actual Error	Function Evaluations
10^{-8}	2.664790417536985	0.000000678718826030433	642
10^{-10}	2.664790395912379	0.000000700343431692119	1542
10^{-12}	2.664790392203251	0.000000704052560251967	3830

As you can see, the error tolerance we asked for was NOT followed by the program, this is because the integrand is singular at the left endpoint. ~~OK.~~

- b.) Make the change of variables $x = u^\alpha$ in this integral. What values of α give a transformed integral that is appropriate for numerical quadrature?

Using $x = u^\alpha$ as our transformation we simply do a change of variables on our integral in the following fashion:

Since $x = u^\alpha$, then $dx = \alpha u^{\alpha-1} du$.

Our endpoints will be transformed by $\sqrt[\alpha]{x} = u$, but since every root of one is one and every root of zero is zero, our limits remain unchanged.

$$\int_0^1 (u^\alpha)^{-\frac{2}{3}} e^{-(u^\alpha)^2} (\alpha u^{\alpha-1}) du$$

If we combine our common terms we get the following:

$$\alpha \int_0^1 u^{\frac{1}{3}\alpha-1} e^{-u^{2\alpha}} du$$

The purpose of doing the change of variables is to get rid of the singularity in our integrand. Therefore α must meet the following characteristic:

$$u^{\frac{1}{3}\alpha-1} \geq 1 = u^0, \text{ therefore } \frac{1}{3}\alpha - 1 \geq 0 \text{ this implies that } \alpha \geq 3$$

Having this specific α insures that our integrand is non-singular on the interval of integration. So, to simplify our expression as much as possible we choose $\alpha = 3$.

So, our integrand simplifies to:

$$3 \int_0^1 e^{-u^6} du$$

c.) Repeat part (a) with your transformed integral, and comment on the results.

After inputting the transformed integral into quad, we get the following results:

Tolerance	Approximate Integral	Actual Error	Function Evaluations
10^{-8}	2.664791094735980	0.00000000151983092777641	69
10^{-10}	2.664791096288973	0.00000000003316236174555	153
10^{-12}	2.664791096255712	0.00000000000009903189380	321

The error tolerance we specified is now followed accurately by the program and the number of evaluations is at a much more tolerable rate. If you notice in part (a) of this problem, the number of evaluations for the tolerance of 10^{-8} was 642 and it did not even match the tolerance we asked for. However, transforming the integrand gave us a much better approximation with many less function evaluations.

Problem 3:

a.) Briefly describe each of the six blocks of code labeled 1 through 6.

1: new_count = count + 9; ~~2: 1:~~

This piece of the code calculates how many function calls are made during this round of integration (3 for [a, b], 3 for [a, c] and 3 for [c, b]) totaling to 9.

```
2:   sab = S(f,a,b); %% 2
```

This calls the function which calculates the Simpson's rule approximation of f over the entire interval which was passed to this function (this can be the entire interval (in the 1st pass through the function), or subintervals of the entire interval which have too much error).

```
3:   c = 0.5*(a+b); %% 3
sac = S(f,a,c); %%
scb = S(f,c,b); %%
```

This calculates the midpoint of the interval that was passed into the function. This is calculated to calculate the Simpson's estimate for half of interval.

```
4:   factor = 15; %% 4
```

This sets the factor of multiplication for the error estimation derived from the error formula. This can be used because we are comparing two methods of quadrature which have the same order of accuracy.

```
5:   if( abs(sac+scb-sab) < factor*ep ) %% 5
      q = sac + scb; %%
```

This checks if the estimated integral over the interval $[a, c]$ and $[c, b]$ is sufficiently better than the approximation over $[a, b]$. If it is, then q (the adaptive Simpson's quadrature) is returned.

```
6:   else
      [q1,new_count]=adapt(f,a,c,ep/2,new_count); %% 6
      [q2,new_count]=adapt(f,c,b,ep/2,new_count); %%
      q =q1+q2; %%
```

If the condition of 5 is not met, then the function is recursively called until condition 5 is met.

b.) This code is written for clarity, not for efficiency. Explain why this program is not an efficient implementation of adaptive Simpson's rule.

This code is not efficient because it recalculates function values in the sense that in block 2 of the code ($sab = S(f, a, b);$ %% 2), it calculated the function values for the endpoints and midpoint; while in block 3 ($sac = S(f, a, c);$ %% $scb = S(f, c, b);$ %%) it recalculates not only the endpoints once again, but it calculates the midpoint TWICE. This is very inefficient, especially when we start calculating over larger intervals.

c.) Change this code to make an adaptive 3-point Gaussian quadrature. Is this an efficient (not necessarily optimal) implementation?

```
% adaptive quadrature using 3-point Gaussian Quadrature
% Header the same as given

function [q,new_count] = adaptGauss(f,a,b,ep,count);
% if count was not passed in, initialize it to zero
if( nargin < 5 )
```

```

count = 0;
end

new_count = count + 9; %% 1
sab = gauss(f,a,b); %% 2
c = 0.5*(a+b); %% 3
sac = gauss(f,a,c); %%
scb = gauss(f,c,b); %%
factor = (2^6) - 1; %% 4

if( abs(sac+scb-sab) < factor*ep ) %% 5
    q = sac + scb; %%
else
    [q1,new_count]=adapt(f,a,c,ep/2,new_count); %% 6
    [q2,new_count]=adapt(f,c,b,ep/2,new_count); %%
    q =q1+q2; %%
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 3-point Gaussian Quadrature
% input: f - integrand, scalar function that takes one input.
% a,b - integration is over [a,b]
%
% output: q - simpson's rule approximation to integral of f over [a,b]
%
function q=gauss(f,a,b);
    u1 = -((b-a)/2)*(sqrt(3/5)) + (b+a)/2 ; %get points to be evaluated
    u2 = (b+a)/2;
    u3 = ((b-a)/2)*(sqrt(3/5)) + (b+a)/2 ;

q = ((b-a)/2)*((5/9)*feval(f, u1) + (8/9)*feval(f, u2) + (5/9)*feval(f, u3));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

This application is actually very efficient, maybe not optimal application of 3-point Gaussian Quadrature. Unfortunately because of the nature of Gaussian Quadrature, we cannot reuse points as we can in the Simpson's or Trapezoidal Rule; the points will not be as rigid and evenly spaced because of the odd spacing within each subinterval.

- d.) Use the given adaptive Simpson's rule routine, your adaptive Gaussian quadrature routine, and MATLAB's build-in adaptive Simpson's quadrature, *quad*, to evaluate the integral below with an error tolerance of 10^{-10} .

$$\int_0^1 \frac{1}{x+0.1} dx$$

(Handwritten version of the integral above)

Quadrature Method	Estimated Integral	Error	Function Evaluations
Given Adaptive Simpson's	2.397895272833749	0.00000000003537881099191509	3951
Adaptive Gaussian	2.397895272774391	0.00000000002397948506427383	459
Built-in Adaptive Simpson's	2.397895272800162	0.00000000001791899961745003	301

Comment on your results.

The number of evaluations used in the method given (Adaptive Simpson's) and the one created (Adaptive Gaussian) are both inadequate because the former uses more than ten times the number of evaluations of the Built-in method of the same class (Adaptive Simpson's). Although adaptive Gaussian proved to be much better than adaptive Simpson's in this case, we cannot be certain that it will be much better when we apply the method to larger intervals and more complex integrands.

e.) Analytically derive a bound for the number of points needed to guarantee that the error is below 10^{-10} for the composite Trapezoidal rule and for the composite Simpson's rule each using equally spaced points for the integral above.

The theoretical error for the **composite Trapezoidal Rule** is:

$$\left| -\frac{(b-a)}{12} f''(\gamma) h^2 \right|$$

Where $\gamma \in (a, b)$ and $h = \frac{b-a}{n}$.

So if we want to guarantee that this value is less than 10^{-10} , we need to satisfy the following inequality:

$$\frac{(b-a)}{12} [\max_{x \in [a, b]} f''(x)] h^2 \leq \frac{1}{10^{10}}$$

But since our bound of integration is $(0, 1)$, then $b-a = 1$ and $h = 1/n^2$. So the inequality simplifies to:

$$\frac{1}{12} [\max_{x \in [a, b]} f''(x)] \frac{1}{n^2} \leq \frac{1}{10^{10}}$$

But since $f''(x) = 2(x + 0.1)^{-3}$, this is a decreasing monotonic function on the interval. This implies that the maximum happens at the left endpoint of the interval (and its value is 2000). Therefore:

$$\frac{2000}{12 n^2} \leq \frac{1}{10^{10}} \rightarrow n \geq 1290995$$

Therefore to guarantee an error tolerance of 10^{-10} for the Composite Trapezoidal Rule, the number of subintervals used must be greater than the number as stated above.

Similarly, the theoretical bound for the Composite Simpson's rule is:

$$\frac{(b-a)}{180} [\max_{x \in [a,b]} f^{(4)}(x)] h^4 \leq \frac{1}{10^{10}}$$

Where $\gamma \in (a, b)$ and $h = \frac{b-a}{n}$.

Similarly as above, the fourth derivative of our function is monotonic decreasing and is $f^{(4)}(x) = 24(x + 0.1)^{-5}$, the maximum occurs at the left endpoint and equals 2,400,000. So our inequality is:

$$\frac{2400000}{180 n^4} \leq \frac{1}{10^{10}} \rightarrow n \geq 3398$$

So to guarantee an error tolerance of 10^{-10} for the Composite Simpson's Rule, the number of subintervals used must be greater than the number as stated above.

- f.) Experiment with your composite Simpson's rule code to estimate (within 1000 points) the actual number of points needed to get an error below 10^{-10} . Comment on your results taking into account the analytic bound on the number of points and the number of function evaluations used by the (efficient) adaptive routines.*

We used our composite Simpson's rule for these numbers of points and attained the following errors:

n	Estimated Integral	Error
900	2.397895273306240	0.000000000507869746257938
1200	2.397895272959084	0.000000000160713664598688
1500	2.397895272864202	0.000000000065831784468173

As you can see, the value where n is 1500 is where the Error goes below 10^{-10} . Comparing this to the Adaptive Simpson's and Gaussian, it fails in efficiency. Except that it beats the given Adaptive Simpson's quadrature in efficiency which, as stated in part (b) is not built for efficiency. However, this result truly shows how the error bounds prove inadequate when it comes to choosing the number of points; the bound gave us an n more than twice the value that we actually used.