**ENSC 180 – Introduction to Engineering Analysis Tools**

**Assignment #6: Numerical Integration**

## Background

The biggest drawback of composite quadrature rules is the dilemma of the step-size that will be used. For example, if we have a monotonic function that increases at a steady rate, we may be able to use a large step-size to obtain a highly accurate estimate of its integral while minimizing computing time. However, this large step-size may fail miserably when applied to a wildly varying function. In this case, we would prefer a smaller step-size. An even greater dilemma occurs if we are faced with a function with both of these characteristics – what would we do then? We must look towards an adaptive approach where the function is recursively broken down into smaller and smaller intervals, but only where necessary. An interval will be broken down into two subintervals only if the integral done independently on the two subintervals is far more accurate. This is known as adaptive quadrature.

## Objectives

This assignment will cover the following topics:

1. Adaptive quadrature methods for numerical integration.
2. Recursive programming in MATLAB.
3. Line plotting tools to help visualize results.
4. Passing functions as arguments using function handles.
5. Simple computational complexity analysis using tic and toc functions.

You are required to integrate an unknown function across a finite range using adaptive quadrature. You will gain insight on how the adaptive algorithm works by creating visualizations of its optimally-chosen intervals of integration.

**Preliminary:** The unknown function that you are given is positive within an interval [1, 9] and 0 elsewhere. Before moving on, it would be wise to have a look at it. It is a function of only one variable (i.e., of the form ) and can be sampled at any point. This unknown function is defined in “f.m”.

**Task 1:** Write a MATLAB function with the following description:

function estIntegral = integralSimpsons(f, a, b)

This function should be stored in a file named “integralSimpsons.m”. It should estimate integral of a function across a specified interval using the Simpson’s rule, where:

f – a MATLAB function handle.  
a – scalar value defining the lower limit of integration.  
b – scalar value defining the upper limit of integration.  
estIntegral – the value of the estimated integral.

**Task 2:** Write a MATLAB function with the following description:

function [estIntegral, intervals] = adaptiveSimpsons(f, a, b, tol, s)

This function will be called recursively to perform the numerical integration. It will also return the endpoints of the *M* subintervals formed throughout the process in an *M*-by-2 matrix intervals. This function should be stored in a file named “adaptiveSimpsons.m” and it should perform the following steps:

1. Find the midpoint of the current interval [a, b] and store the result in a variable c.
2. Apply Simpson’s rule, using integralSimpsons twice on the unknown function (pointed at by MATLAB function handle f): Once on the left subinterval [a, c], and again on the right subinterval [c, b].
3. In the current recursion, we must check for the stopping criteria. Let denote the estimate of our integral across an interval . If   
     
     
     
   where is tol in our implementation, stop the recursion and:
   1. calculate the sum of , , and the following correction term  
        
        
        
      Store the result in estIntegral.
   2. Store the subintervals [a, c] and [c, b] as rows a matrix intervals.
   3. Exit out of the function using the return keyword.
4. If the stopping criteria is not satisfied, continue on with the recursion as follows:
   1. Call the adaptiveSimpsons function twice: once on the left subinterval and again on the right subinterval. Use half of the current tol as the error tolerance. Store the outputs of both function calls in appropriately named variables.
   2. Calculate the sum of the estimated integral outputs for these two recursive function calls and store the result in estIntegral.
   3. Vertically concatenate the matrix outputs containing the intervals created by these two recursive function calls and store the result in intervals. You can do this either by using matrix construction syntax, or by using the MATLAB cat function.
   4. Exit out of the function using the return keyword.

**Task 3:** Write a script to make it easier to execute these functions properly, and to create the visualizations for the intervals. Give the script an intuitive name like “demoSimpsons.m” and organize it as follows:

1. It is wise to start any script with:  
   clear all  
   close all  
   to ensure that it is entirely self-contained and can run with an initially empty workspace.
2. Ensure that the unknown function defined in “f.m” is inside your working folder along with functions from Task 1 and 2. A MATLAB function handle is a data type that points to a function and allows it to be passed as an argument of another function. For example, minHandle = @(x) min(x) + max(x); would add the minimum and maximum values of an input x when called as minHandle(x). Create a function handle that can be used to call our unknown function f.
3. We wish to integrate the unknown function only in the interval [1, 9], where it is not 0. Store the upper and lower limits in appropriate variables.
4. Set the tolerance, tol = 1e-2.
5. Obtain an initial estimate of the integral using integralSimpsons.
6. Call the function adaptiveSimpsons with this initial estimate and the previously defined parameters.
7. Plot the unknown function using the plot command and a generous number of sample points.
8. Query the y-axis limits using the get command on the current axis. To see which attribute contains the y-axis limits, you can create a dummy plot,   
   e.g., plot(x, x.^2), select the axis object using the arrow, open up the “Property Editor” under “View” of the figure window, and click on “More Properties”.
9. Set hold on and iterate through the rows of the intervals matrix.
   1. Using the MATLAB line function, draw a vertical line that begins at the lower y-axis limit and terminates at the upper y-axis limit for each x-axis value where an interval endpoint occurs. Assign a variable to store the handle of this line object for the next step, i.e., h = line(…). (Hint: a line object is actually what is created whenever you use the function plot, except the plot function also makes a figure and axis to go along with it. So all we are doing is overlaying straight lines on top of our original plot).
   2. Use the set command with the line handle h to make the line dotted (':') and black ([0, 0, 0]). Find the appropriate properties by searching “lineseries properties” in the documentation.
10. Set hold off and try running everything.

**Task 4:** Once the previous tasks have been completed, the next step is to repeat them using the trapezoid rule. Make copies of the files “integralSimpsons.m”, “adaptiveSimpsons.m”, and “demoSimpsons.m”. Rename the copies appropriately: “integralTrap.m”, “adaptiveTrap.m”, and “demoTrap.m”. Make sure the function names within the files are consistent with the new filenames. There are only two meaningful changes here:

1) Inside “integralTrap.m”, you must repeat Task 1 for the trapezoid rule.

2) Inside “adaptiveTrap.m”, there is a very slight change in the stopping criteria:

and the correction term becomes:

All other changes are file/function name-related and are trivial.

## Analysis

1. Compare the results obtained from adaptive Simpson’s rule with those from the adaptive trapezoid rule. In either case, why are the intervals chosen the way they are by the algorithm? You may wish to vary the tolerances and re-run both cases to obtain presentable result.
2. Compare the estimates for various tolerances with the MATLAB integral function, which numerically evaluates the integral for a given function handle on a specified interval. To get MATLAB to show more digits in your answer, type format long in the terminal. This change may be reverted by typing format short, or simply format, in the terminal.
3. Computational complexity is usually determined by the number of multiplications and divisions. Since our function is unknown, we cannot accurately measure this. An alternative is to measure the amount of time it takes to run our implementations integrateSimpsons and integrateTrap on the unknown function. Use the MATLAB tic and toc functions to perform the necessary measurements. Perform the measurements many times and take the average to make sure they are reliable. Which method is more computationally demanding? Why? Which method is more computationally demanding in the context of the adaptive algorithm? Why?