Homework 5

1. Develop an algorithm which, for a given function $f(x)$, interval bounds a and b with $a < b$, and a prescribed number of subintervals n, applies the multiple application trapezoidal rule to approximate the integral $\int_a^b f(x)\,dx$.

```
import numpy as np

def func(x):
    return 3 * np.power(x, 2) + 5*x - 7

def trap_integration(a, b, n, function):
    # creates the step size
    step = (b - a) / n
    # adds the sum of the evaluation of the function at each step from a to b
    iter_sum = function(a) + function(b) + sum(list(map(lambda x: 2 * function(x),
np.arange(a + step, b, step))))
    # alters the sum by the range of the bounds and 2n
    return (b - a) * iter_sum / (2 * n)
```

2. Develop an algorithm which, for a given function $f(x)$, interval bounds a and b with $a < b$, and a prescribed number of subintervals n, approximates the integral $\int_a^b f(x)\,dx$ according to the following procedure:

   (a) If $n = 1$, it applies the trapezoidal rule.

   (b) If n is even, it applies the multiple application Simpson's 1/3 rule.

   (c) If $n \geq 3$ and n is odd, it applies the multiple application Simpson's 1/3 rule on the first $n - 3$ subintervals, and applies the Simpson's 3/8 rule on the last three subintervals.

Simpson's 1/3 Integration Formula

Formula:

$$I \approx (b-a)\,\frac{f(x_0) + 4\sum_{i=1,3,5}^{n-1} f(x_i) + 2\sum_{j=2,4,6}^{n-2} f(x_j) + f(x_n)}{3n}$$

Simson's 3/8 Integration Formula

$n-3$

$$\int_a^b f(x)\,dx = I \approx \frac{3h}{8}\left[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\right]$$

Formula:

$$= (b-a)\,\frac{f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)}{8}$$

```
import numpy as np

# the function that is being integrated
def func(x):
    return 3 * np.power(x, 2) + 5*x - 7

# the trapazoidal integration for n == 1
def trap_integration(a, b, n, function):
    # creates the step size
    step = (b - a) / n
    # adds the sum of the evaluation of the function at each step from a to b
    iter_sum = function(a) + function(b) + sum(list(map(lambda x: 2 * function(x),
np.arange(a + step, b, step))))
    # alters the sum by the range of the bounds and 2n
```

```
    return (b - a) * iter_sum / (2 * n)

# simpson's 1/3 integration rule for even amounts of subsets
def simpson_third_integration(a, b, n, function):
    if n <= 0:
        return 0.0
    step = (b - a) / n
    iter_sum = 0
    all_x = np.arange(a + step, b, step)
    '''
    In the mathematical formula, 4f(xi) is for x=1,3,5,... and 2f(xi) for x=2,4,6
    However, this means that first item in all_x is x_1 in the mathematical formula.
So in the program
    the even terms are multiplied by 4 and the odd terms are multiplied by 2
    '''
    for i in range(len(all_x)):
        if i == 0 or i % 2 == 0:  # even
            iter_sum += 4 * function(all_x[i])
        else:  # odd
            iter_sum += 2 * function(all_x[i])

    return (b - a) * (function(a) + iter_sum + function(b)) / (3*n)

# simpson's 3/8 integration rule for the last 3 sub sections
def simpson_three_eighths_integration(a, b, n, function):
    step = (b - a) / n
    return (b - a) * (function(a) + 3*function(a+step) + 3 * function(a + 2 * step) +
function(b)) / 8

# driver code for deciding which rule to use
def simpson_integration(a, b, n, function):
    # if n == 1, use trap rule
    if n == 1:
        return trap_integration(a, b, 1, function)
    # if n is even, use simpson 1/3 rule
    elif n % 2 == 0:
        return simpson_third_integration(a, b, n, function)

    # if n is odd, use simpson 1/3 for all but the last three sub-interval, which
uses simpson 3/8 rule
    elif n >= 3:
        step = (b - a) / n
        return simpson_third_integration(a, b - 3 * step, n - 3, function) \
               + simpson_three_eighths_integration(b - 3 * step, b, 3, function)

    else:
        print(f"n, {n}, is weird")
        return "Error"
```

3. Develop an algorithm which, for a given function $f(x)$, interval bounds a and b with $a < b$, and error tolerance per subinterval tol, applies adaptive quadrature to approximate the integral $\int_a^b f(x)\,dx$ (based on the pseudocode that was presented in the recorded lectures and can be found on page 642 of the textbook).

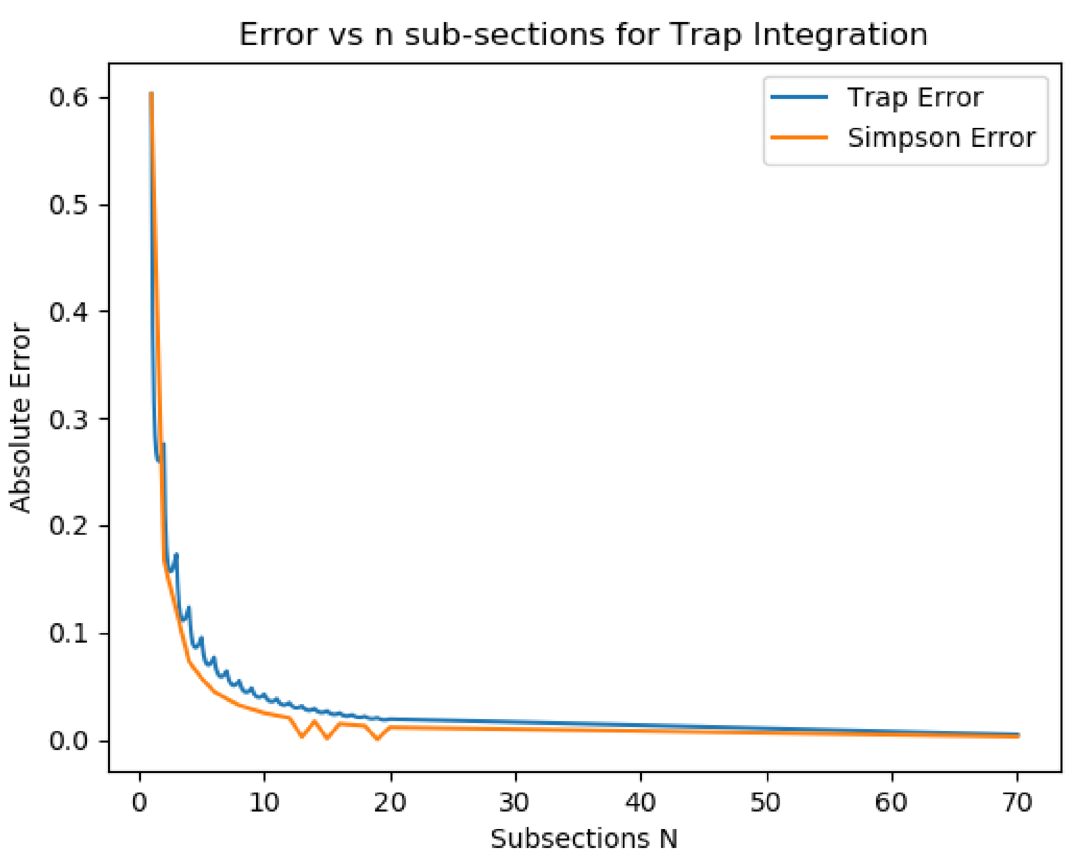
```
# Adaptive Quadrature Integration
def adap_quad_integration(a, b, function, tol):
    c = (a + b) / 2
    I_1 = ((b - a) / 6) * (function(a) + 4 * function(c) + function(b))
    I_2 = ((b - a) / 12) * (function(a) + 4 * function((a + c) / 2) + 2 *
                            function(c) + 4 * function((b + c) / 2) + function(b))

    # if the difference between I_1 and I_2 is less that user defined tolerance,
return
    if abs(I_1 - I_2) <= tol:
        return 16 * I_2 / 15 - I_1 / 15
    else:
        # else split the area is half and add the sums through recursion stack
        return adap_quad_integration(a, c, function, tol) +\
               adap_quad_integration(c, b, function, tol)
```

4. Apply the algorithms you developed in questions 1-3 above to approximate

$$\int_0^1 x^{0.1}(1.2-x)(1-e^{20(x-1)})\,dx,$$

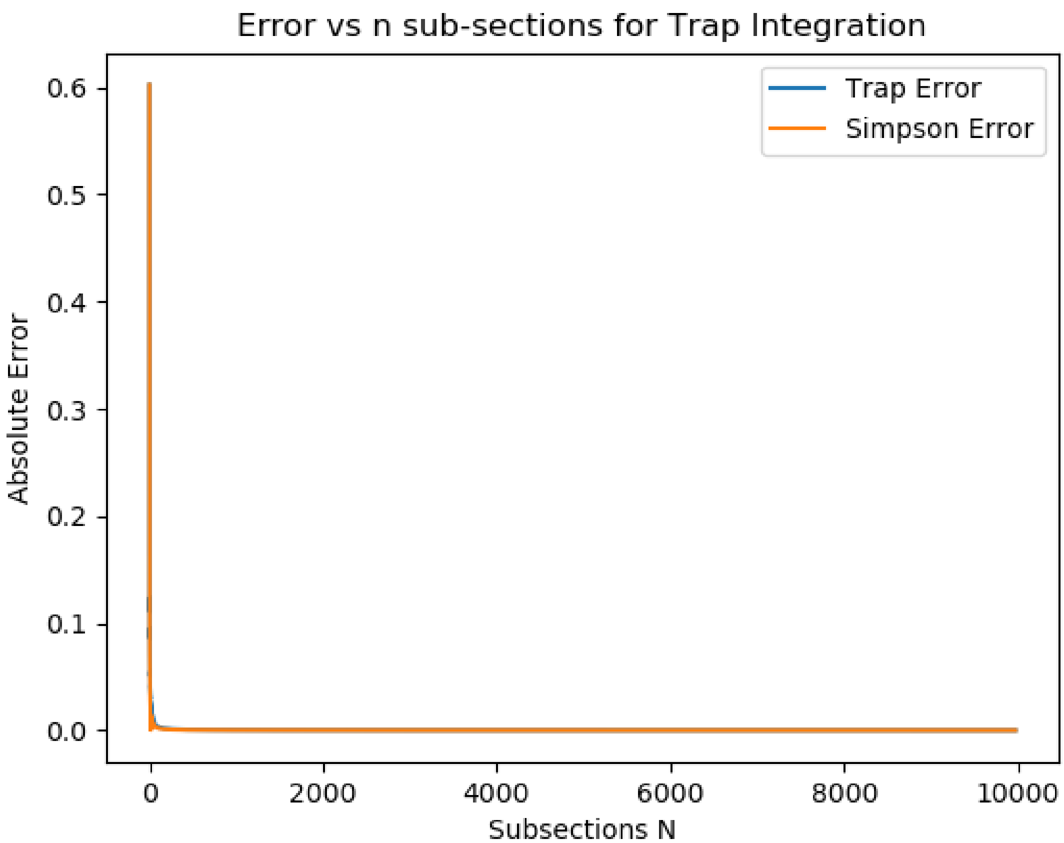
for varying values of n and tol. Note that this integral is not easy to evaluate analytically! Using the true value of 0.602298, plot ϵ_t as a function of n for the algorithms you developed in questions 1 and 2, and plot ϵ_t as a function of tol for the algorithm you developed for question 3. Use your best judgement to determine appropriate ranges of values for n and tol to be included in the plots.

This first plot shows only n up to 100. This is because this is the area of the most significant change. Some interesting things to note is that for the Simpson's Error, it is mostly exponential decay, except for n=13, n = 15, and n=19. They localized drops in error occur all at odd numbered n's, meaning that there

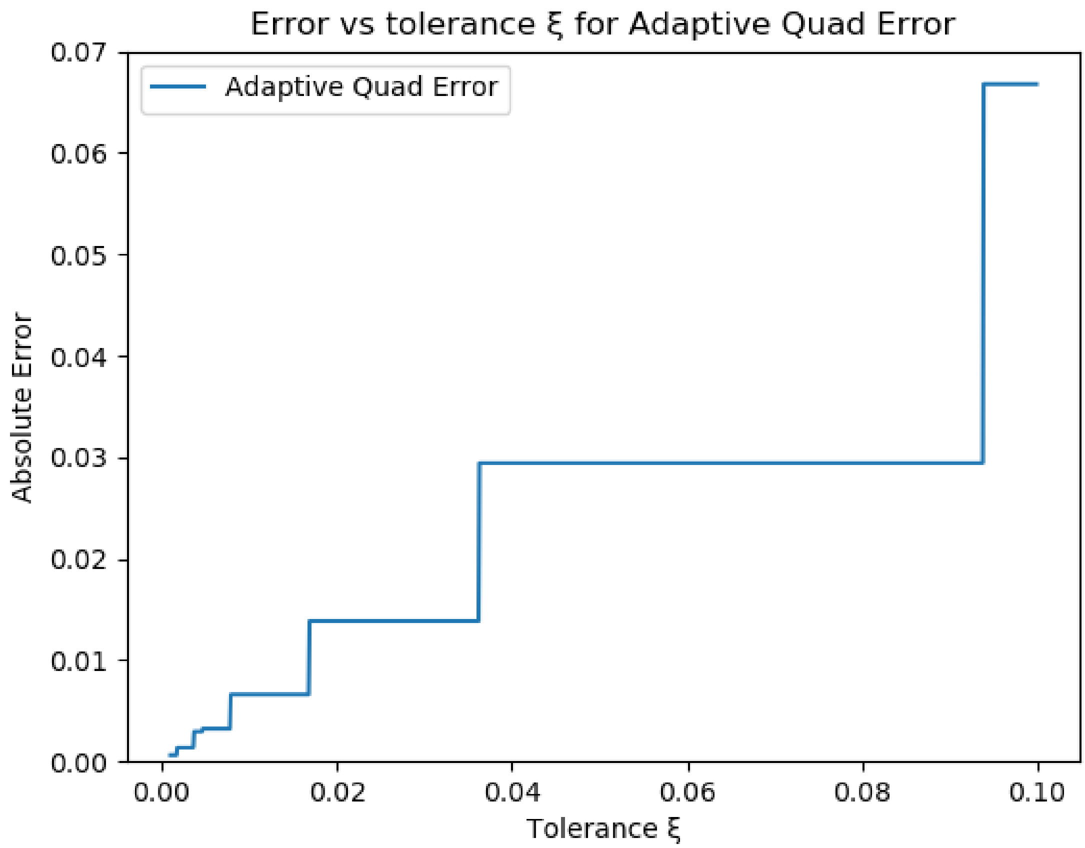


is a chance of the algorithm becoming more accurate when using Simpson's 1/3 rule and Simpson's 3/8 rule in tandem.

In this second plot, it shows Trapezoidal Error and Simpson's Error up to n = 10000. This shows the overall trend of exponential decay. However, some of the details are too small to notice from this larger scale. This shows that as $n \to \infty, \xi \to 0$. Since this is done numerically on digital systems, the dominating source of error at large N's is the numerical instability of representing increasingly small numbers in binary.



In the third plot, it shows absolute error versus the user defined tolerance ξ. This follows steps that trend down to 0 as the tolerance goes to 0.



Something to note is that changing ξ a little does not usually make much of a change. However, changing ξ from 0.1 to 0.01 makes a significant change and appears to trend downward. The trend will stop as the numerical instability described above begins to play a bigger role in the error.