Adaptive Quadrature

Kenneth Potts

December 9, 2016

Abstract

In regular quadrature, an integral approximation is found using a method of numerical integration such as Simpson's Rule, or Simpson's composite rule. These methods use equidistant subintervals. The function is divided into subdivisions of equal length, and the function is evaluated at the points of division, or nodes. These evaluated nodes are then used in a weighted algorithm for approximation. Using this method of quadrature, incrementally smaller subintervals must be used for higher precision approximations. This task may be time and resource intensive when computed. Certain subintervals of the approximation may be sufficiently precise while others contain the majority of the error of the full approximation. Instead of equally shortening all of the subintervals, it may be more efficient to place more nodes in regions where the error is relatively large while maintaining fewer nodes in regions where the error is already sufficiently small. This method is called adaptive quadrature. It generally uses a standard quadrature method on a region of a function to obtain an integral approximation I_1 which is then compared to an approximation I_2, produced by the corresponding composite quadrature method. Whenever the two approximations are sufficiently close the approximation is kept, otherwise the region is divided into two sections and the process is repeated. It is a recursively defined algorithm. This produces a sufficiently accurate approximation that uses more densely spaced nodes wherever necessary.

1 Introduction

There are many different methods for approximating an integral. Some of the most common are Riemann sums, the trapezoidal rule, Simpson's rule, and Simpson's $\frac{3}{8}$ rule. Each rule consist of an integral approximation and an error term which provides an upper bound or an estimate for the error of the integral approximation. There are multiple reasons why one would desire a numerical method to approximate an integral. Firstly, some functions do not have antiderivatives.

Theorem 1 *[5] Let f be a function which is continuous on an interval $[a, b]$. Let F be the antiderivative of f, then*

$$\int_a^b f(x)dx = F(b) - F(a) \tag{1}$$

By theorem 1, the fundamental theorem of calculus, if one can obtain an antiderivative, then one can solve a definite integral. If no antiderivative exists, then one cannot solve for the integral in this way. In this case, a numerical method is required to get an approximation for the integral. Secondly, a numerical integration method may be used to program a computer to produce integral approximations. A numerical algorithm is in general required when programming a computer to solve an integral because a computer uses rounded floating point numbers for numerical computations. In either scenario it is important to consider the computational speed in which a sufficient level of precision is reached. In general numerical integration algorithms are performed iteratively with increasing levels of precision until a desired level is reached.

2 Standard Numerical Integration

2.1 Example: The Trapezoidal Rule

To begin, it is necessary to understand standard numerical integration. These methods generally evaluate the function at evenly spaced points called nodes, and then assign weights to the resulting values. The space between nodes will be denoted as a subinterval. The composite trapezoidal rule will be use as an example. The rule is defined as the following: [1]

$$\int_a^b f(x)dx = \frac{h}{2}[f(a) + 2\sum_{i=1}^{n-1} f(a + i \cdot h) + f(b)] + E_t(x) \tag{2}$$

where n is the number of equal-length subintervals of size $h = \frac{b-a}{n}$ and $E_t(x)$ is the error of the approximation. The error term is then defined as:

$$E_t(x) = \frac{(b-a)h^2}{12} f''(\xi_t) \tag{3}$$

holding equality for some $\xi_t \in (a, b)$ [1]. The error $E_t(x)$ is a function of h. To perform this approximation, it is necessary to first choose a value for n, the number of subintervals. The value of h depends on n, as n increases h decreases, and as a result, the error also decreases as h decreases. In fact, the error term decreases by a factor of h^2, this is denoted as $O(h^2)$ in big-oh notation [2]. The error for the approximation with n subintervals can then be expressed as $e_n \approx Mh^2$ where M is some constant when ξ is assumed to be constant [2].

$$e_n \approx Mh^2 \tag{4}$$

$$e_{2n} \approx M(\frac{h}{2})^2 \tag{5}$$

$$\implies \frac{e_n}{e_{2n}} \approx \frac{Mh^2}{M(\frac{h}{2})^2} = 4 \tag{6}$$

$$\implies e_n \approx 4e_{2n} \tag{7}$$

This means that doubling the number of subintervals used in a trapezoidal rule approximation will yield a new approximation with $\frac{1}{4}$ the error of the previous approximation. To increase the precision of an approximation while maintaining uniformly sized intervals between nodes, it is necessary to add more nodes and therefore uniformly decrease the size of h between all nodes. That is, when more nodes are added in the approximation, the size h between nodes decreases uniformly across the interval $[a, b]$, which produces a more accurate approximation.

2.2 Other Standard Methods of Numerical Integration

Another numerical method of integration is Simpson's rule:

$$\int_a^b f(x)dx = \frac{h}{3}[f(a)+2\sum_{i=0}^{\frac{n}{2}} f(a+(2i-2)\cdot h)+4\sum_{i=0}^{\frac{n}{2}} f(a+(2i-1)\cdot h)+f(b)]+E_s(x) \tag{8}$$

where n is an even number of equal-length subintervals of size $h = \frac{b-a}{n}$ between adjacent nodes, and $E_s(x)$ is the error of the approximation. The error term is then defined as:

$$E_s(x) = \frac{h^4}{180} f^{[4]}(\xi_s) \tag{9}$$

holding equality for some $\xi_s \in (a, b)$ [6]. This method is $O(h^4)$. Then following the same logic as equations 4-7 produces the error relationship:

$$e_n \approx 16e_{2n} \tag{10}$$

3 Adaptive Quadrature

3.1 Theory

Standard methods of integration can be used to obtain accurate integral approximations. However, when considering their computational effort, these methods seam inefficient. Many functions do not exhibit uniform characteristics. They have sections which change dramatically while having other sections which have a smaller, more gentle change. Standard approaches split these functions up into even subintervals to obtain sufficient accuracy. Depending on the nature of each division of the function, an integral approximation may be more accurate in some sections than others. If the error of an approximation is broken up and analyzed for each section between nodes, the approximations for the sections which contain mild rates of change will achieve the proper accuracy with larger subintervals than would a section which contains a more variant rate of change. An adaptive quadrature algorithm is one that addresses the varying concentrations of error in an approximation, and uses subinterval sizes that are inversely proportional to the error of a specific region. In other words the algorithm more densely distributes nodes in regions which would otherwise produce greater error, which overall reduces the total number of nodes required.

3.2 The Recursive Algorithm

The primary way to implement an adaptive quadrature algorithm is to define it recursively. This means it is necessary to write an algorithm that will repeat itself with a different input before the original run of the algorithm has completed. In other words, the algorithm will call itself from within with different inputs. Without a recursive definition, it would be necessary to record and access a large amount of data from many calculations. This is quite cumbersome and can be bypassed when using a recursive algorithm. Adaptive quadrature can be done with many basic quadrature methods such as the trapezoidal rule, Simpson's rule, or Simpson's $\frac{3}{8}$ rule. Each of these methods has a composite version wherein approximations from multiple subintervals are added together to produce a more accurate approximation. A basic adaptive quadrature method can then be described in a generalized form using any quadrature method and its corresponding composite version, e.g., the trapezoidal rule and the trapezoidal composite rule. As with any numerical method, certain assumptions must be true before the algorithm can be used. An outline of the general adaptive quadrature method is as follows [2]:

Assumptions: $f''(x)$ is continuous on the interval $[a, b]$

Inputs: A function f, the interval $[a, b]$ over which to integrate, and a set tolerance level tol

Step 1: Set $m = \frac{a+b}{2}$, then find integral estimates I_1 and I_2

Step 2: If $|I_2 - I_1| < C \cdot tol$ then return I_2

Step 3: Do steps 1-5 with inputs f; $[a, \frac{a+b}{2}]$; and $\frac{tol}{2}$, and set A equal to the result

Step 4: Do steps 1-5 with inputs f; $[\frac{a+b}{2}, b]$; and $\frac{tol}{2}$, and set B equal to the result

Step 5: Return $A + B$

Output: Approximate value of the integral $\int_a^b f(x)dx$

The approximation I_1 is the integral approximation with the fewest nodes (e.g. two nodes with the trapezoidal rule, three nodes with Simpson's rule). The approximation I_2 is the composite integral approximation with the fewest nodes (e.g. three nodes with the composite trapezoidal rule, five nodes with the composite Simpson's rule). Whenever the error for the approximation I_2 is not below the desired tolerance level within a given subsection, the subsection is split into two halves, and the algorithm is recursively called for both halves with half the error tolerance. In step 2, C is a constant ratio that depends on the error of the method being used. When this factor C is used to approximate error, it is called a linear error estimator and can be used when comparing errors produced by methods with the same order of convergence[4]. To find C start with the following:

$$\int_a^b f(x)dx = I_2 + e_2 = I_1 + e_1 \tag{11}$$

I_2 and e_2 are the resulting approximation and error when twice the number of nodes are used from I_1 and e_1. The relationship between e_2 and e_1 depends on the order of the method being used. The relationship for I_1 and e_1 for the composite trapezoidal rule is given in equation 7. The relationship can then be used to make a substitution in equation 11:

$$I_2 - I_1 = e_1 - e_2 \tag{12}$$

$$I_2 - I_1 \approx 4e_2 - e_2 \tag{13}$$

$$I_2 - I_1 \approx 3e_2 \tag{14}$$

[2] The substitution shows that the difference between approximations I_2 and I_1 is approximately $3 \cdot e_2$. The adaptive quadrature algorithm stops when $|I_2 - I_1| < C \cdot tol$, and for the trapezoidal rule $C = 3$. Using the substitution from equation 10, it follows that $C = 15$ for the composite Simpson's rule. An upper bound for the total error can be obtained by then summing the upper bound for the error for each subinterval, where $|I_2 - I_1|$ is the upper bound for a given subinterval when $|I_2 - I_1| < C \cdot tol$ is achieved. Particularly, the global error is the sum of all the local errors [4]. See appendix A for a simple Python implementation of the recursive algorithm.

4 Coding and Practical Implementation

MATLAB is somewhat of a standard computing environment for numerical methods. MATLAB has many built in methods for integration including one called `quad`. The `quad` function standard in MATLAB implements the extrapolated Simpson's rule in an adaptive recursive algorithm [7]. Recently the fields of numerical analysis, computational science, and engineering have been working extensively with Python as well. This is due to its modularity, dynamic data types, and interactive environment among other advantages [3]. Python employs its SciPy library for many scientific numerical computations including integration. The library also uses a function called `quad` which uses a recursive adaptive quadrature method utilizing Simpson's method. Adaptive quadrature is widely used as a standard method for integration primarily because of its superior performance in computational effort while sacrificing little in accuracy. Another advantage of adaptive quadrature over standard methods of integration is that an adaptive quadrature algorithm branches into two calculations at each level. This allows for parallel computing, where computers which are optimized for parallel computations can perform multiple unrelated calculations at the same time [8]. This benefit does not reduce the computational effort, however it greatly reduces the time it takes a program to execute. This gives adaptive quadrature a powerful advantage over standard quadrature methods.

5 Efficacy Analysis

An adaptive quadrature algorithm, as the name suggests, adapts the placement of the nodes to the variation of the input function. This presents a few complications when calculating the convergence of the algorithm. Calculating the order of convergence for the composite trapezoidal rule and composite Simpson's rules was moderately straight forward, $O(h^2)$ and $O(h^4)$ respectively. The order of convergence in either case depends on h with the assumption that h is the uniform distance between all adjacent nodes. This is not the case for adaptive quadrature where h adapts to the given function, and therefore the order of convergence does not depend on h, but on the input function itself.

5.1 A Visual Demonstration

An integral approximation for the function

$$\int_{-1}^{3} x \cdot sin(2x)dx \tag{15}$$

can be obtained using the standard Simpson's method with a tolerance of 10^{-3}.

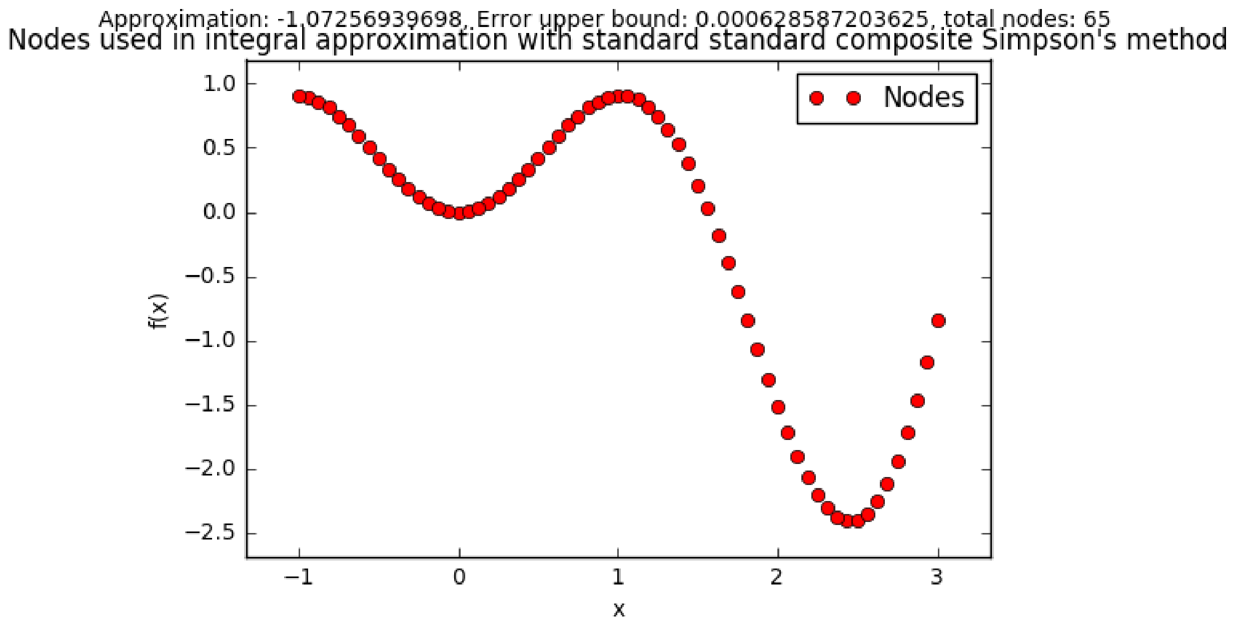

Figure 1: Approximating $\int_{-1}^{3} x \cdot sin(2x)dx$ With Standard Simpson's

This approximation used 65 nodes to approximate the integral to with an error $e \approx 0.000628587 < 10^{-3}$. Note that the nodes are all equidistant from their adjacent nodes. The integral above can then be approximated with the adaptive composite Simpson's rule with an error tolerance or 10^{-3}.

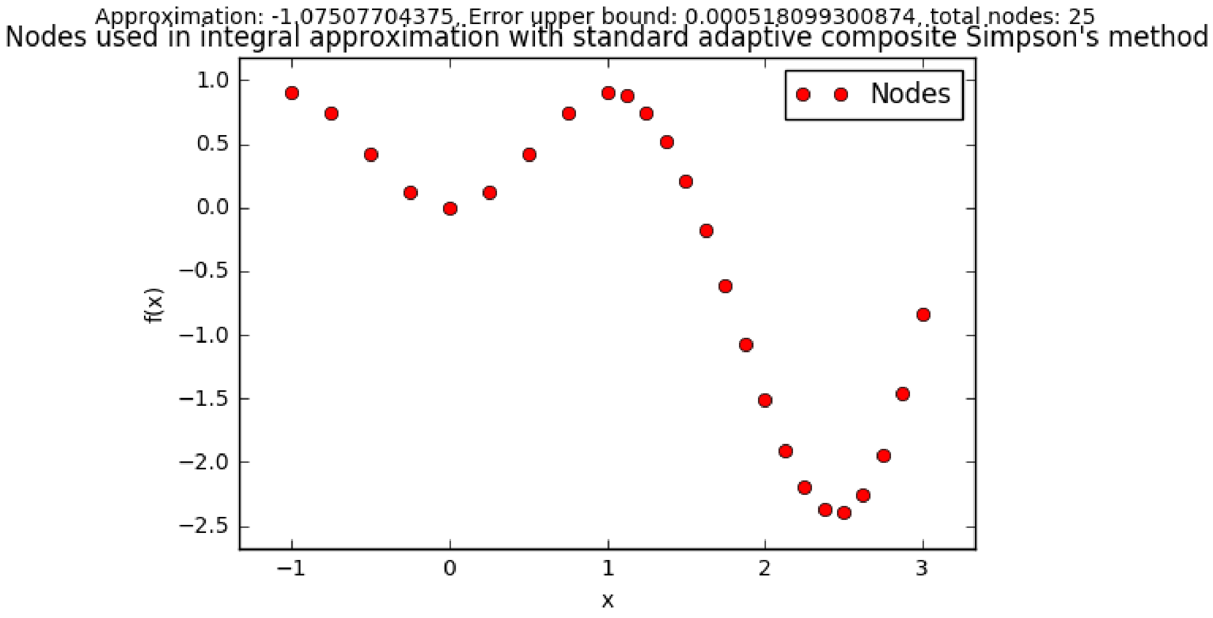

Figure 2: Approximating $\int_{-1}^{3} x \cdot sin(2x)dx$ With Adaptive Simpson's

This approximation used 25 nodes to approximate the integral to with an error $e \approx 0.000518099 < 10^{-3}$. Note that there are varying distances between adjacent nodes. The adaptive method required less than half the number of nodes as the standard method, while still delivering an approximation with an error under the desired tolerance.

5.2 Convergence Comparison

Since order of convergence of an adaptive quadrature algorithm does not depend on the value of h, an indirect way to analyze convergence is to collect data from the algorithm. The question to ask is what data would allow for such a comparison? The following chart was produced using data for logarithmically decreasing error upper bounds and logarithmically increasing number of nodes for various methods for approximating the integral:

$$\int_{-1}^{3} x \cdot sin(2x)dx \tag{16}$$

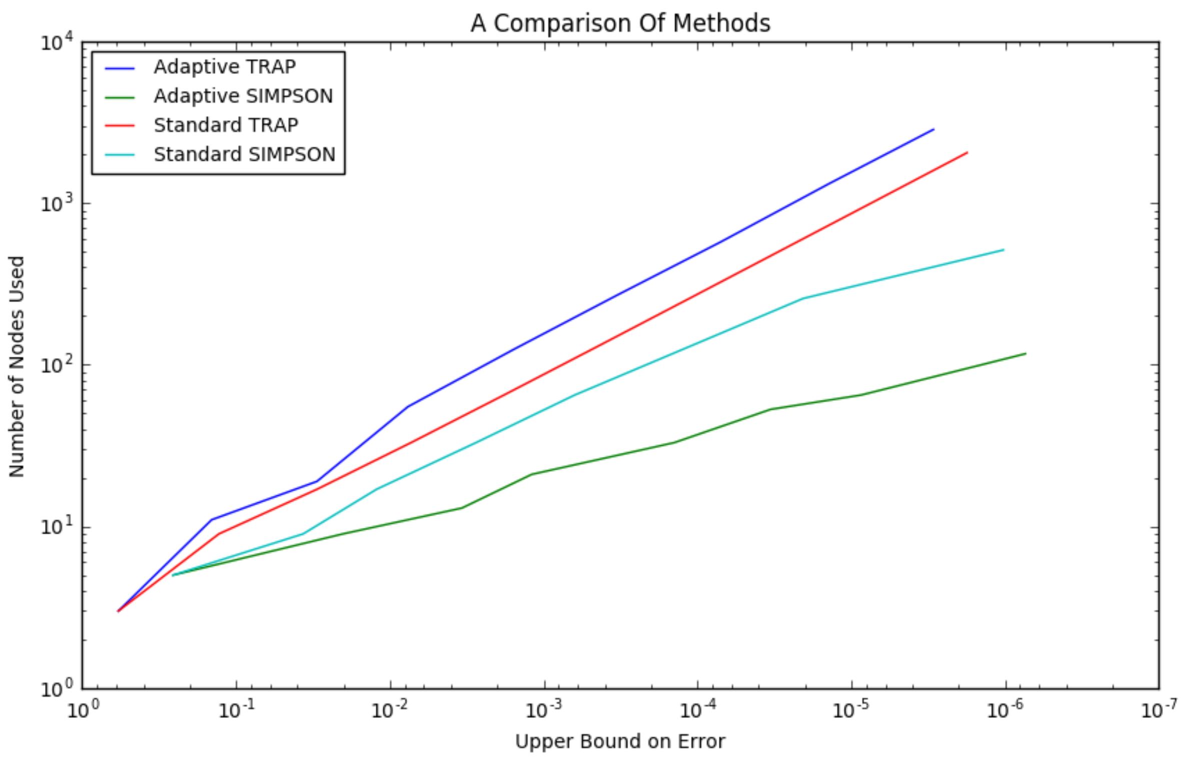

Figure 3: Approximating $\int_{-1}^{3} x \cdot sin(2x)dx$

The library containing the algorithms used to generate these data are located in appendix B. The log-log scale allows for the visualization of the convergence of the methods. Clearly, the adaptive Simpson's method seems to be converging to the true integral value using fewer nodes than the other methods. It uses fewer nodes than other methods to achieve a given error tolerance. This illustrates adaptive quadrature's superior computational advantage. The next best method appears to be the standard Simpson's method, followed by the standard and adaptive trapezoidal methods. Other than the adaptive Simpson's method leading the way, the results of this chart are not intuitive. This may be due to the manner in which the number of nodes where being calculated in the algorithm. Uncovering the cause will take further investigation and optimization. The slopes of these lines vary as the style of function changes, which is to be expected. The nature of the function itself is what dictates where and how many nodes are to be used. More plots and charts of algorithm data are located in appendix C.

6 Conclusion

Adaptive Quadrature adjusts the number of nodes used to approximate the integral of a function based on the specific variation of the function. It places more nodes in sections where function variation is large and fewer nodes in sections where the variation is smaller. The nodes used in the approximation are not evenly distributed. The overall effect is that an integral can be approximated to within a specified error tolerance using fewer nodes, and therefore fewer computations than a standard quadrature method would require. Moreover, The algorithm is best implemented using recursion which allows for potential parallelization of computations, further speeding up the time it takes a computer to perform the algorithm. These advantages have made adaptive quadrature a standard algorithm in scientific and numerical computing, with implementation in standard MATLAB, and Python SciPy quadrature functions.

References

[1] D. N. Arnold, *A concise introduction to numerical analysis*, 2001.

[2] L. Q. Brin, *Tea Time Numerical Analysis*, GNU, 2 ed., 2016.

[3] H. Fangohr, *Python for Computational Science and Engineering Python for Computational Science and Engineering Python for Computational Science and Engineering*, University of Southampton, 2015.

[4] P. Gonnet, *A review of error estimation in adaptive quadrature*, ACM Computing Surveys, 44 (2012).

[5] D. R. Guichard, *Calculus*, David R. Guichard under the Creative Commons, 2009.

[6] D. Levy, *Introduction to Numerical Analysis*, University of Maryland (CSCAMM), 2010.

[7] C. Moler, *Numerical Computing with MATLAB: Quadrature*, SIAM, 2004.

[8] J. R. Rice, *Adaptive quadrature: Convergence of parallel and sequential algorithms.*, Bulletin of the American Mathematical Society, 80 (1974).

Appendices

Appendix A

```
# Simple Python Implementation
# using the composite trapezoidal rule
import numpy as np

def simpleAdaptQuad(func,a,b,tol):
    m = (a+b)/2      #midpoint
    h_1 = b-a           #interval size for aprox I_1
    h_2 = (b-a)/2       #interval size for aprox I_2

    f_a = func(a) #f(a)
    f_b = func(b) #f(b)
    f_m = func(m) #f(m)

    I_1 = (h_1/2)*(f_a+f_b) # approximation I_1
    I_2 = (h_2/2)*(f_a+(2*f_m)+f_b) # approx I_2
    max_error = np.abs(I_2-I_1)/3 #upper bound on error for trapezoidal

    if max_error < (tol):
        return I_2, max_error
    else:  #recursive call on left and right halves
        I_left, left_error = simpleAdaptQuad(func, a, (a+b)/2, tol/2)
        I_right, right_error = simpleAdaptQuad(func, (a+b)/2, b, tol/2)

    return I_left + I_right, left_error + right_error
    # returns: sum of left and right aprox
    # sum of left and right error
```

The code can be executed and the approximation obtained with the following command:

```
# To approximate f(x)=sin(x^2)x on the interval [-2,5], with a tolerance of 0.00001
approx, error = simpleAdaptQuad(lambda x: np.sin(x**2)*x, -2, 5, 1e-5)
```

```
print("Approximation = %s, Error = %s" % (approx, error))
```

This yields the output:

```
Approximation = -0.822423114722, Error = 5.41493601912e-06
```

Appendix B

The following code was used to produce all charts and plots.

```
"""
Contains functions that approximate integrals by performing adaptive quadrature
"""

from matplotlib import pyplot as plt
from matplotlib import style
import numpy as np
from pandas import DataFrame, Series
import pandas as pd


def quad(func, a, b, tol=1e-5, method='TRAP', adaptive=True, double_nodes=True, visual_mode=False):
    """
    Approximate the integral of function func, on the interval [a,b]
    return the approximation and an upper bound on the error.

    Calls the ad_quad function which uses the an adaptive quadrature method.

    Args:
        func: the function to be integrated (callable)
        a (float): the lower bound of the interval of integration
        b (float): the upper bound of the interval of integration
        tol (float, optional): desired error tolerance, defaults to 1e-5
        method (str, optional): quadrature method to be used, defaults to 'TRAP'
        adaptive (bool, optional): True opts for adaptive quadrature,
                                   false uses standard, defaults to True
        double_nodes (bool, optional): True doubles the amount of nodes at each step,
                                       only used with standard quadrature, defaults to True
        visual_mode (bool, optional): True will produce an interactive plot of nodes,
                                      defaults to False

    Returns:
        Integral Approximation (float): The Approximation of the integral
        Error upper bound (float): An upper bound on the error of the approximation
    """

    def _ad_quad(function, a_1, b_1, tol_1, method_1, total_lines_1, val_dict_1={}):
        """
            Approximate the integral of function func, on the interval [a,b]
            return the approximation and an upper bound on the error.

            Called from quad. This function uses the an adaptive quadrature method.
            This function is called recursively. _ad_quad is defined from
            within quad because it should be only be called from within a_quad
            but must remain separate to perform recursive calls.

            Args:
                function: the function to be integrated (callable)
                a_1 (float): the lower bound of the interval of integration
                b_1 (float): the upper bound of the interval of integration
                tol_1 (float): desired error tolerance, defaults to 1e-6
                method_1 (str): quadrature method to be used, defaults to 'TRAP'
                total_lines_1 (int): the total number of lines run
                val_dict_1 (dict): should be an empty dictionary when first called


            Returns:
                Integral Approximation (float): The Approximation of the integral
                Error upper bound (float): An upper bound on the error of the approximation
                Nodes dictionary (dict): A dictionary of the points used to
                    approximate the integral
                Total lines run (int): The total number of lines run by the function
        """

        line_sum = total_lines_1

        # interval validity
        if a_1 > b_1:
            a_1, b_1 = b_1, a_1

        # common points for all methods:

        mid = (a_1 + b_1) / 2  # midpoint

        val_dict = val_dict_1

        if a_1 in val_dict:
            fa = val_dict[a_1]
        else:
            fa = function(a_1)
            val_dict[a_1] = fa
        if b_1 in val_dict:
            fb = val_dict[b_1]
        else:
            fb = function(b_1)
            val_dict[b_1] = fb
        if mid in val_dict:
            fm = val_dict[mid]
        else:
            fm = function(mid)
            val_dict[mid] = fm

        # Choose Method and calculate

        if method_1 == 'SIMPSON':  # Simpson's rule
            h1 = (b_1 - a_1) / 2  # interval size for approx i_one
            h2 = (b_1 - a_1) / 4  # interval size for approx i_two
            quarter_1 = a_1 + h2
            quarter_3 = mid + h2

            if quarter_1 in val_dict:
                fq_1 = val_dict[quarter_1]
            else:
                fq_1 = function(quarter_1)
                val_dict[quarter_1] = fq_1
            if quarter_3 in val_dict:
                fq_3 = val_dict[quarter_3]
            else:
                fq_3 = function(quarter_3)
```

```
            val_dict[quarter_3] = fq_3

        i_one = (h1 / 3) * (fa + (4 * fm) + fb)
        i_two = (h2 / 3) * (fa + (4 * fq_1) + (2 * fm) + (4 * fq_3) + fb)
        max_error = np.abs(i_two - i_one) / 15  # calculates upper bound on
                                                # error for Simpson's composite

        if max_error < tol_1:
            return i_two, max_error, val_dict, line_sum
        else:  # recursive call on left and right halves
            i_left, left_error, left_val_dict, left_lines = _ad_quad(function,
                                                                     a_1,
                                                                     (a_1 + b_1) / 2,
                                                                     tol_1 / 2,
                                                                     method_1,
                                                                     line_sum,
                                                                     val_dict)
            i_right, right_error, right_val_dict, right_lines = _ad_quad(function,
                                                                         (a_1 + b_1) / 2,
                                                                         b_1, tol_1 / 2,
                                                                         method_1,
                                                                         line_sum,
                                                                         val_dict)

        z_dict = left_val_dict
        z_dict.update(right_val_dict)

        return i_left + i_right, left_error + right_error, z_dict, left_lines + right_lines

        # merge dict: {**left_val_dict, **right_val_dict} can be used
        #     in certain newer versions of python
        # returns: sum of left and right approximations
        # sum of left and right error
        # merged left and right val_dicts
        # and the sum of the lines run

    elif method_1 == 'SIMPSON_38':  # Simpson's 3/8 rule
        # http://mathfaculty.fullerton.edu/mathews/n2003/Simpson38RuleMod.html

        h1 = (b_1 - a_1) / 3  # interval size for approx i_one
        h2 = (b_1 - a_1) / 6  # interval size for approx i_two
        x_1 = a_1 + h2
        x_2 = x_1 + h2
        x_3 = x_2 + h2
        x_4 = x_3 + h2
        x_5 = x_4 + h2

        if x_1 in val_dict:
            fx_1 = val_dict[x_1]
        else:
            fx_1 = function(x_1)
            val_dict[x_1] = fx_1
        if x_2 in val_dict:
            fx_2 = val_dict[x_2]
        else:
            fx_2 = function(x_2)
            val_dict[x_2] = fx_2
        if x_3 in val_dict:
            fx_3 = val_dict[x_3]
        else:
            fx_3 = function(x_3)
            val_dict[x_3] = fx_3
        if x_4 in val_dict:
            fx_4 = val_dict[x_4]
        else:
            fx_4 = function(x_4)
            val_dict[x_4] = fx_4
        if x_5 in val_dict:
            fx_5 = val_dict[x_5]
        else:
            fx_5 = function(x_5)
            val_dict[x_5] = fx_5

        i_one = (3*h1 / 8) * (fa + (3 * fx_2) + (3 * fx_4) + fb)
        i_two = (3*h2 / 8) * (fa + (3 * fx_1) + (3 * fx_2) + (3 * fx_3)
                             + (3 * fx_4) + (3 * fx_5) + fb)
        max_error = np.abs(i_two - i_one) / 15  # calculates upper bound on error
                                                # for Simpson's 3/8 composite

        if max_error < tol_1:
            return i_two, max_error, val_dict, line_sum
        else:  # recursive call on left and right halves
            i_left, left_error, left_val_dict, left_lines = _ad_quad(function,
                                                                     a_1,
                                                                     (a_1 + b_1) / 2,
                                                                     tol_1 / 2,
                                                                     method_1,
                                                                     line_sum,
                                                                     val_dict)
            i_right, right_error, right_val_dict, right_lines = _ad_quad(function,
                                                                         (a_1 + b_1) / 2,
                                                                         b_1, tol_1 / 2,
                                                                         method_1,
                                                                         line_sum,
                                                                         val_dict)

        z_dict = left_val_dict
        z_dict.update(right_val_dict)

        return i_left + i_right, left_error + right_error, z_dict, left_lines + right_lines

        # merge dict: {**left_val_dict, **right_val_dict} can be used in certain
        #     newer versions of python
        # returns: sum of left and right approximations
        # sum of left and right error
        # merged left and right val_dicts
        # and the sum of the lines run

    else:  # other condition performs TRAPEZOIDAL RULE
        h1 = b_1 - a_1  # interval size for approx i_one
        h2 = (b_1 - a_1) / 2  # interval size for approx i_two

        i_one = (h1 / 2) * (fa + fb)
        i_two = (h2 / 2) * (fa + (2 * fm) + fb)
        max_error = np.abs(i_two - i_one) / 3  # calculates upper bound on error for trap

        if max_error < tol_1:
            return i_two, max_error, val_dict, line_sum
        else:  # recursive call on left and right halves
            i_left, left_error, left_val_dict, left_lines = _ad_quad(function,
                                                                    a_1,
                                                                    (a_1 + b_1) / 2,
                                                                    tol_1 / 2,
                                                                    method_1,
                                                                    line_sum,
                                                                    val_dict)
            i_right, right_error, right_val_dict, right_lines = _ad_quad(function,
                                                                       (a_1 + b_1) / 2,
                                                                       b_1, tol_1 / 2,
                                                                       method_1,
                                                                       line_sum,
                                                                       val_dict)

        z_dict = left_val_dict
        z_dict.update(right_val_dict)

        return i_left + i_right, left_error + right_error, z_dict, left_lines + right_lines

        # merge dict: {**left_val_dict, **right_val_dict} can be used in certain
        #   newer versions of python
        # returns: sum of left and right approximations
        # sum of left and right error
        # merged left and right val_dicts
        # and the sum of the lines run

def _std_quad(function, a_1, b_1, tol_1, method_1, total_lines_1, double_nodes_1=True):
    """
        Approximate the integral of function func, on the interval [a,b]
        return the approximation and an upper bound on the error.

        Called from quad. This function uses the standard quadrature method.
        _st_quad is defined from within a_quad because it should be only be
        called from within quad but must remain separate for multiple calls.

        Args:
            function: the function to be integrated
            a_1 (float): the lower bound of the interval of integration
            b_1 (float): the upper bound of the interval of integration
            tol_1 (float): desired error tolerance, defaults to 1e-6
            method_1 (str): quadrature method to be used, defaults to 'TRAP'
            total_lines_1 (int): the total number of lines run
            double_nodes_1 (bool, optional): True doubles the amount of nodes at each step,
                defaults to True



        Returns:
            Integral Approximation (float): The Approximation of the integral
            Error upper bound (float): An upper bound on the error of the approximation
            Nodes dictionary (dict): A dictionary of the points used to
                approximate the integral
            Total lines run (int): The total number of lines run by the function
    """

    line_sum = total_lines_1

    # interval validity
    if a_1 > b_1:
        a_1, b_1 = b_1, a_1

    if method_1 == 'SIMPSON':
```

```
    # calculates based on the composite Simpson's rule
    n = 4
    h2 = (b_1 - a_1) / n
    h1 = h2 * 2
    x_0, x_1, x_2, x_3, x_4 = a_1, a_1 + h2, a_1 + h1, a_1 + 3*h2, b_1
    y_0, y_1, y_2, y_3, y_4 = function(x_0), function(x_1), function(x_2),\
                              function(x_3), function(x_4)

    val_dict = {x_0: y_0, x_1: y_1, x_2: y_2, x_3: y_3, x_4: y_4}

    sum_1 = (y_0 + 4 * y_2 + y_4)
    sum_2 = (y_0 + (4 * y_1) + (2 * y_2) + (4 * y_3) + y_4)

    i_one = (h1 / 3) * sum_1
    i_two = (h2 / 3) * sum_2

    while np.abs(i_two - i_one) > 15 * tol_1:
        if double_nodes:
            n *= 2  # vs += 2 which only adds 2 (minimal nodes)
        else:       # but does not calc accurately
            n += 2
        h2 = (b_1 - a_1) / n
        i_one = i_two
        sum_2 = val_dict[a_1] + val_dict[b_1]
        for i in range(1, n):
            x_val = a_1 + i*h2
            if x_val in val_dict:
                y_val = val_dict[x_val]
            else:
                y_val = function(x_val)
                val_dict[x_val] = y_val
            if i % 2 == 1:  # MADE A CHANGE HERE TO CORRECT ISSUE
                sum_2 += 4 * y_val
            else:
                sum_2 += 2 * y_val
        i_two = (h2 / 3) * sum_2
    return i_two, np.abs(i_two - i_one)/15, val_dict, line_sum

elif method_1 == "SIMPSON_38":
    # calculates based on the  Simpson's 3/8 rule

    n = 6
    h2 = (b_1 - a_1) / n
    h1 = h2 * 2
    x_0, x_1, x_2, x_3, x_4, x_5, x_6 = a_1, a_1 + h2, a_1 + 2*h2,\
                                        a_1 + 3 * h2, a_1 + 4*h2, a_1 + 5*h2, b_1
    y_0, y_1, y_2, y_3, y_4, y_5, y_6 = function(x_0), function(x_1),\
                                        function(x_2), function(x_3),\
                                        function(x_4), function(x_5), function(x_6)

    val_dict = {x_0: y_0, x_1: y_1, x_2: y_2, x_3: y_3,
                x_4: y_4, x_5: y_5, x_6: y_6}

    sum_1 = (y_0 + 3*y_2 + 3*y_4 + y_6)
    sum_2 = sum_1 + 3*y_1 + 3*y_3 + 3*y_5

    i_one = (3*h1 / 8) * sum_1  # 3/8 multiplier
    i_two = (3*h2 / 8) * sum_2

    while np.abs(i_two - i_one) > 15 * tol_1:
            if double_nodes_1:
                n *= 2   # vs += 3 which only adds 3 (minimal nodes)
            else:        # but does not calc accurately
                n += 3
            h2 = (b_1 - a_1) / n
            i_one = i_two
            sum_2 = val_dict[a_1] + val_dict[b_1]
            for i in range(1, n):
                x_val = a_1 + i * h2
                if x_val in val_dict:
                    y_val = val_dict[x_val]
                else:
                    y_val = function(x_val)
                    val_dict[x_val] = y_val
                sum_2 += 3*y_val
            i_two = (3*h2 / 8) * sum_2  # 3/8 multiplier
        return i_two, np.abs(i_two - i_one)/15, val_dict, line_sum
    else:
        # calculates based on the trapezoidal rule

        n = 2
        h2 = (b_1 - a_1) / n
        h1 = h2 * 2
        x_0, x_1, x_2 = a_1, a_1 + h2, b_1
        y_0, y_1, y_2 = function(x_0), function(x_1), function(x_2)

        val_dict = {x_0: y_0, x_1: y_1, x_2: y_2}

        sum_1 = (y_0 + y_2)
        sum_2 = sum_1 + 2 * y_1

        i_one = (h1 / 2) * sum_1
        i_two = (h2/ 2) * sum_2

        while np.abs(i_two - i_one) > 3 * tol_1:
            if double_nodes:
                n *= 2  # vs += 1 which only adds 1 (minimal nodes)
            else:       # but does not calc accurately
                n += 1
            h2 = (b_1 - a_1) / n
            i_one = i_two
            sum_2 = val_dict[a_1] + val_dict[b_1]
            for i in range(1, n):
                x_val = a_1 + i * h2
                if x_val in val_dict:
                    y_val = val_dict[x_val]
                else:
                    y_val = function(x_val)
                    val_dict[x_val] = y_val
                sum_2 += 2 * y_val
            i_two = (h2 / 2) * sum_2
        return i_two, np.abs(i_two - i_one)/3, val_dict, line_sum

if adaptive:
    packed_data = _ad_quad(func, a, b, tol, method, total_lines_1=0, val_dict_1={})
else:
    packed_data = _std_quad(func, a, b, tol, method, total_lines_1=0,
                            double_nodes_1=double_nodes)

if visual_mode:
        method_dict = {"TRAP": "composite␣trapezoidal", "SIMPSON": "composite␣Simpson's",
                       "SIMPSON_38": "composite␣Simpson's␣3/8"}  # used for formatting of the plots

        points_dict = packed_data[2]
        x_values = np.array(list(points_dict.keys()))
        y_values = np.array(list(points_dict.values()))
        y_max = np.max(y_values)
        y_min = np.min(y_values)

        plt.plot(x_values, y_values, "ro", label="Nodes")
        plt.xlabel("x")
        plt.ylabel("f(x)")
        plt.ylim(y_min - (y_max - y_min) / 12, y_max + (y_max - y_min) / 12)
        plt.xlim(a - (b - a) / 12, b + (b - a) / 12)
        plt.title("Nodes␣used␣in␣integral␣approximation␣with␣%s␣%s␣method"
                  % ("adaptive" if adaptive else "standard", method_dict[method]))

        plt.suptitle("Approximation:␣%s,␣Error␣upper␣bound:␣%s,␣total␣nodes:␣%s"
                     % (packed_data[0], packed_data[1], len(packed_data[2])))

        plt.legend()
        plt.show()

    return packed_data[:3]


def quad_comparison(func, a, b, num_tests=11, visual_mode=True):
    """
        Compares the error vs number of nodes used for 4 integration techniques.

        Compares the adaptive and non-adaptive trapezoidal and Simpson's rules
        by running tests with difference error tolerances and recording the
        number of nodes used to achieve a given error bound

        Args:
            func: the function to be integrated
            a (float): the lower bound of the interval of integration
            b (float): the upper bound of the interval of integration
            num_tests (int): The number of tests to be performed, defaults to 11
            visual_mode (bool): Activates/deactivates graph mode, defaults to True



        Returns:
            node_data (pandas DataFrame): Node data produced by tests
            Error (pandas DataFrame): Error data produced by tests
            """
    num_methods = 4
    node_methods = ['Adaptive␣TRAP', 'Adaptive␣SIMPSON', 'Standard␣TRAP', 'Standard␣SIMPSON']
    error_methods = ['Adaptive␣TRAP', 'Adaptive␣SIMPSON', 'Standard␣TRAP', 'Standard␣SIMPSON']
    errors = np.array([10.0 / (5 ** i) for i in range(num_tests)])
    node_data = DataFrame(np.ones([num_tests, num_methods]),
                          index=errors, columns=node_methods)
    error_data = DataFrame(np.ones([num_tests, num_methods]),
                           index=errors, columns=error_methods)

    for i in range(num_methods):
        for j in range(num_tests):
            if node_methods[i].split()[0] == 'Adaptive':
                p_data = quad(func, a, b, tol=errors[j],
                              method=node_methods[i].split()[1],
                        adaptive=True)
                nodes = len(p_data[2])
                err = p_data[1]
                node_data.iloc[j, i] = nodes
                error_data.iloc[j, i] = err
            elif node_methods[i].split()[0] == 'Standard':
                if node_methods[i].split()[1] == 'SIMPSON':
                    p_data = quad(func, a, b, tol=errors[j],
                                  method=node_methods[i].split()[1],
                                  double_nodes=True,
                                  adaptive=False)
                else:
                    p_data = quad(func, a, b, tol=errors[j],
                                  method=node_methods[i].split()[1],
                                  adaptive=False)
                nodes = len(p_data[2])
                err = p_data[1]
                node_data.iloc[j, i] = nodes
                error_data.iloc[j, i] = err


            # print(data.iloc[j, i * 2])

    if visual_mode:
        fig = plt.figure(figsize=(10, 6))
        plt.plot(error_data, node_data)
        plt.xscale('log')
        plt.yscale('log')
        plt.gca().invert_xaxis()
        plt.legend(node_data.columns, loc=2, prop={'size': 10})
        plt.xlabel('Upper Bound on Error')
        plt.ylabel('Number of Nodes Used')
        plt.title('A Comparison Of Methods')

        plt.show()
    # print(node_data)

    return node_data, error_data

if __name__ == "__main__":

    # print(len(quad(lambda x: np.sin(2 * x) * x, -2, 13, tol=1e-5,
    #                method="SIMPSON", adaptive=True, visual_mode=False)[2]))
    # print(len(quad(lambda x: np.sin(2 * x) * x, -2, 13, tol=1e-8,
    #                method="SIMPSON", adaptive=True, visual_mode=False)[2]))
    # quad_comparison(lambda x: np.sin(2*x)*x, -1, 3, num_tests=11)
    # quad_comparison(lambda x: np.exp(np.power(x, 2)), -3, 1, num_tests=6,
    #                 visual_mode=True)
    # quad(lambda x: np.exp(np.power(x, 2)), -3, 1, method='SIMPSON', visual_mode=True)
    quad(lambda x: x * np.sin(2 * np.log(x)), 1, 18, tol=1e-4, method='TRAP', adaptive=False,
         double_nodes=True, visual_mode=True),
```

Appendix C

The following plots are supplemental material.

The following plots show data for approximations of the function

$$\int_1^{18} sin(2 \cdot ln(x))dx \tag{17}$$

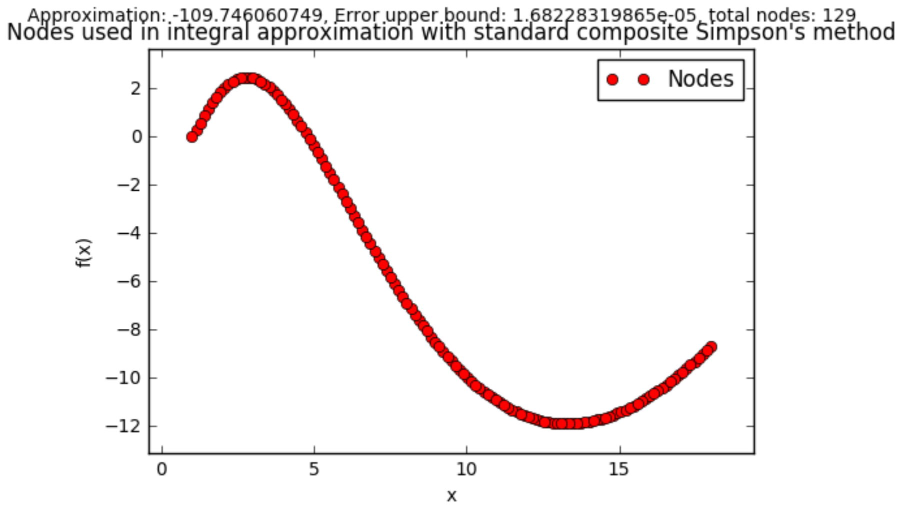

Figure 4: Approximating $\int_1^{18} sin(2 \cdot ln(x))dx$

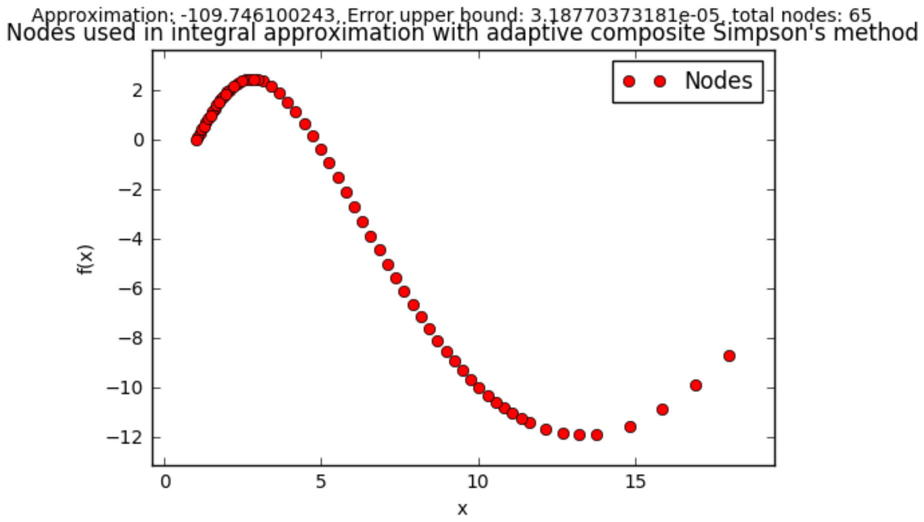

Figure 5: Approximating $\int_1^{18} sin(2 \cdot ln(x))dx$

The following plots show data for approximations of the function

$$\int_{-1}^{\frac{5}{4}} (-19 \cdot sin(x^9) + 2 \cdot cos(x^2) + 5)dx \tag{18}$$

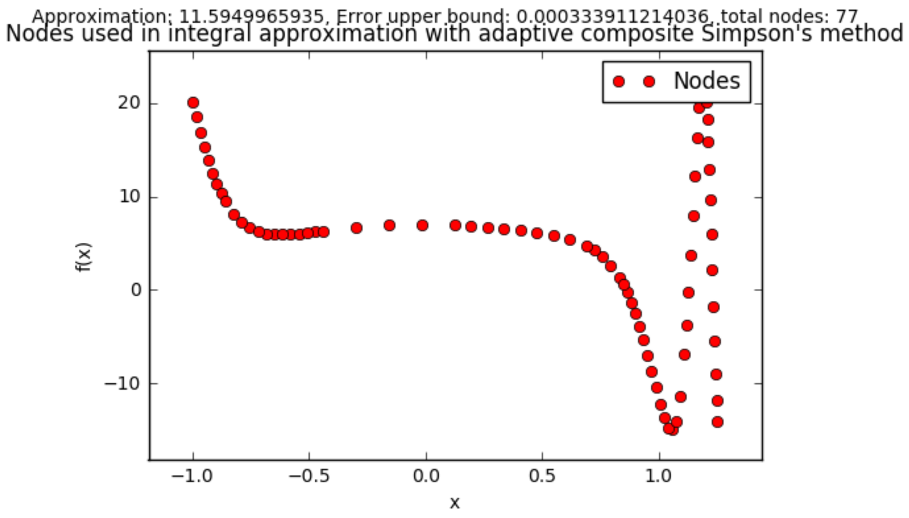

Figure 6: Approximating $\int_{-1}^{\frac{5}{4}}(-19 \cdot sin(x^9) + 2 \cdot cos(x^2) + 5)dx$

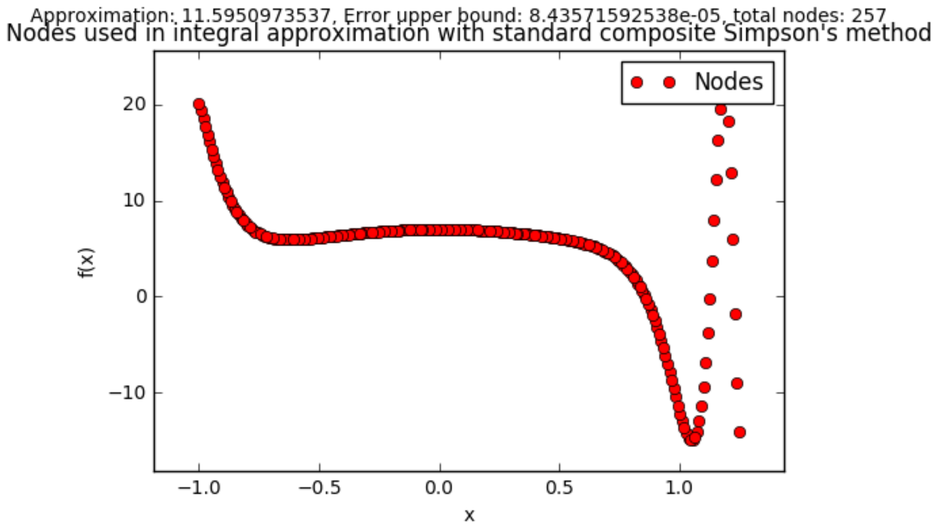

Figure 7: Approximating $\int_{-1}^{\frac{5}{4}}(-19 \cdot sin(x^9) + 2 \cdot cos(x^2) + 5)dx$

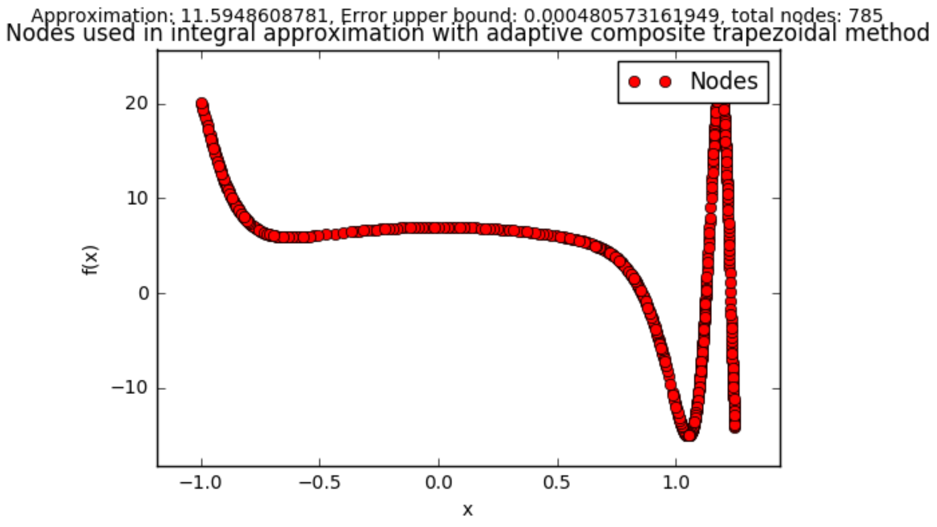

Figure 8: Approximating $\int_{-1}^{\frac{5}{4}}(-19 \cdot sin(x^9) + 2 \cdot cos(x^2) + 5)dx$

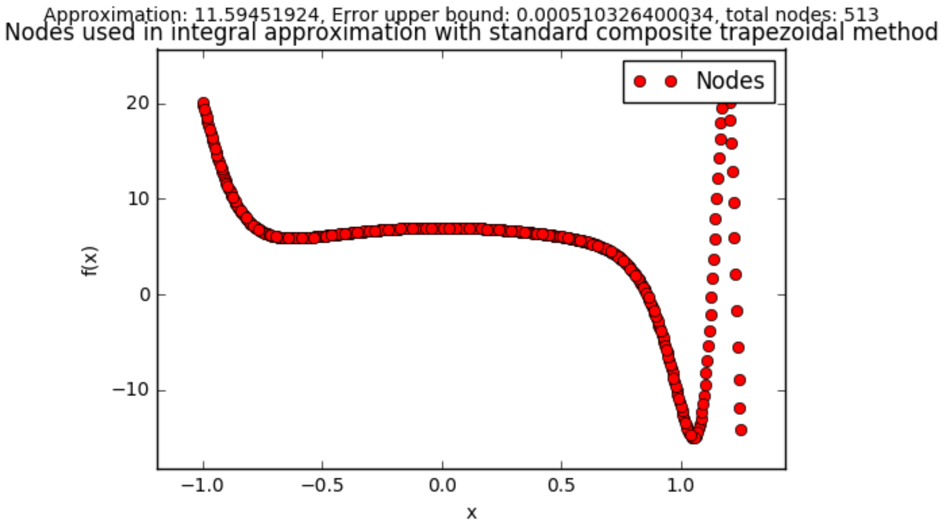

Figure 9: Approximating $\int_{-1}^{\frac{5}{4}}(-19 \cdot sin(x^9) + 2 \cdot cos(x^2) + 5)dx$

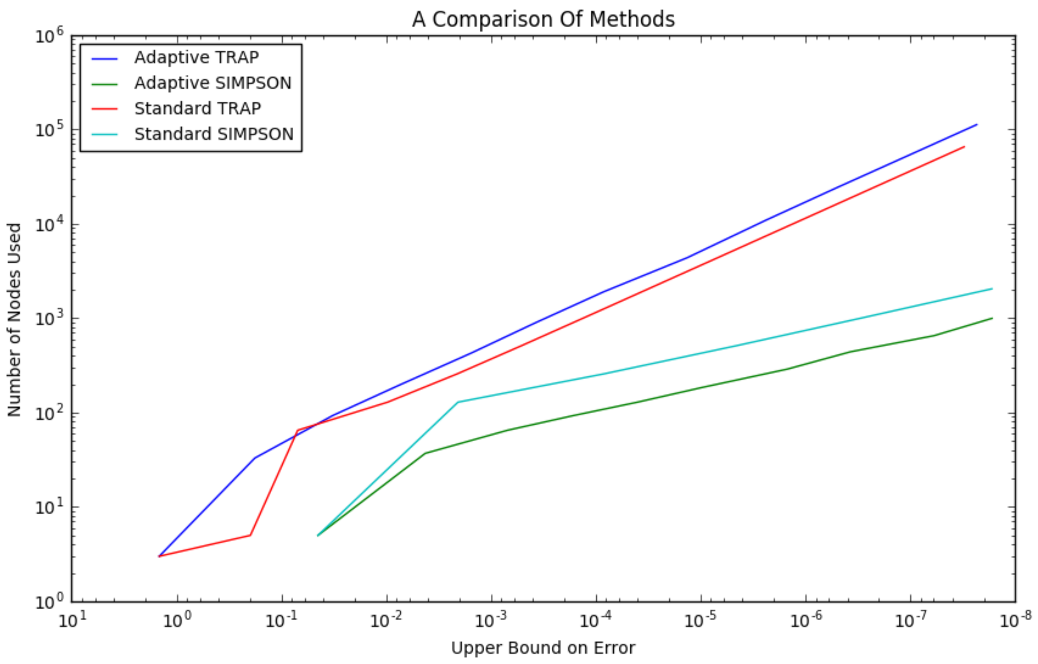

Figure 10: Approximating $\int_{-1}^{\frac{5}{4}}(-19 \cdot sin(x^9) + 2 \cdot cos(x^2) + 5)dx$