University of Zurich[UZH]

Bachelor's thesis
Physics

Numerical integration of single variable functions using TensorFlow

---

Mohammad Alminawi

Supervisors:
Prof. Nicola Serra
Jonas Eschle

July, 2020

Abstract

Integration of single variable functions is needed in many branches of physics. Python implementations of numerical integration methods are available, but none have utilized **TensorFlow**, which allows for parallelized computing, as well as being able to operate on GPU to increase efficiency. Seven integration methods have been implemented using **TensorFlow**, including two historically untested methods. Results for tests of these methods against one another and against similar implementations from the library scipy.integrate using a range of different integrals are presented. The primary goals are: Determining whether **TensorFlow** yields a boost in performance compared to currently available options, comparing the performance when running the methods using a CPU or a GPU, identifying which method is best suited for which case and determining whether there is merit to using the historically untested methods. This project is the initial step of developing a single variable integration library based on **TensorFlow**.

Acknowledgment

Working on this project has been a true joy as it is very different from anything else that I had done throughout the last three years. For that I must thank Professor Nicola Serra and Jonas Eschle, as their holistic approach to physics is what allowed for such a unique bachelor's project to exist.

Professor Nicola Serra has been incredible to work with, his trust in my ability and the abilities of the members of his group astonished and inspired me, I am very thankful for his supervision and guidance.

I cannot begin to express my gratitude to Jonas Eschle. I could not have asked for a better supervisor, his knowledge fascinated me, his ambition inspired me and his work ethic simply stunned me. Regardless of the type of obstacle I faced or when I encountered it, he always could guide me through it and I cannot begin to picture what this project would have looked like without his supervision. At times it was difficult to keep in mind that he was my supervisor as he would get as invested into the project as I was, this resulted in the project expanding to a larger scale than either of us could have predicted initially. It has been a great honor to work with him and I look forward to any opportunities in the future where I may get the chance to do so again.

I would like to thank Abdulrahman Tabaza, MD and Omar Marouf for their help with testing the program. I would also like to thank Amr Fatafta and Dana Kleu for their help with the editing and finalization of the thesis.

Studying at the University of Zurich has been an marvelous experience, I am grateful to the incredible people that have taught me and the amazing friends that made the last three years truly special.

Lastly there are three individuals, to whom I owe a debt of gratitude for so many different things that I could not begin to name. From the bottom of my heart I would like to thank Aisha Zaidan, Dawud Alminawi and Omar Alminawi.

Contents

1 Introduction **3**

2 Computing and numerical integration **5**
2.1 The importance of numerical integration . . . 5
2.2 Newton-Cotes formulas . . . 6
2.2.1 Extended Formulas . . . 9
2.2.2 Adaptive integration methods . . . 11
2.3 Romberg Integration . . . 13
2.4 Gaussian quadrature . . . 14
2.4.1 Calculation of abscissa and weights . . . 15
2.4.2 Gauss-Legendre and Gauss-Kronrod quadrature . . . 16
2.5 Gauss-Romberg method . . . 17
2.6 Implementation in Tensorflow . . . 18

3 Tests and results **19**
3.1 Preliminary testing . . . 20
3.2 Secondary tests . . . 24
3.3 Final tests . . . 29
3.3.1 Benchmark using difficult integrals . . . 29
3.3.2 Benchmark of the methods running on CPU vs GPU . . . 33
3.3.3 Newton-Cotes: Tensorflow vs Numpy . . . 35
3.3.4 Performance test against scipy.integrate . . . 37

4 Conclusion and outlook **41**

A Appendix **44**
A.1 Code and testing devices . . . 44
A.2 Implementation of adaptive methods . . . 46
A.3 Improving error-tolerance matching . . . 48

1. Introduction

We frequently study physical quantities through their rate of change with respect to a variable, this yields differential equations that are commonly solved through integration.

A key example of the need for integration is the probabilistic nature of quantum mechanics. Rather than looking for exact answers from individual cases, quantum mechanics is understood through calculation of the expectation value and variance for a given problem, this is accomplished through integration of operators and wave functions [9].

A large number of modern physics branches are rooted in quantum mechanics to a certain extent, thus it is not surprising that integration plays a key role in those fields as well, for example, in high energy physics (HEP) we can study particle interactions through Fermi's golden rule, which requires us to evaluate the magnitude of the matrix elements of the transition matrix through integration [6].

To showcase the importance of integrals in high energy physics, let us consider a scattering experiment consisting of a particle beam and a fixed target. As the particle beam collides with the target, the particles will scatter at different angles. Integrating over the angular distribution allows us to identify and study the events, despite the individual events providing no information on their own.

We have shown that the integrals contain valuable information, yet we have not outlined any methods to obtain this information. Functions arising in high energy physics are generally difficult or impossible to integrate using analytic methods, this can be due to a plethora of different reasons, such as the integrand containing too many independent functions or containing functions, whose integrals cannot be expressed in terms of elementary functions. [2]

Since analytical solutions are not an option, we turn our attention to numerical integration methods. For multidimensional integrals the most important method is **Monte-Carlo integration**, in contrast it is at best mediocre when considering single dimensional integrals, which are the focus of this thesis.

The error for the Monte-Carlo method is given by $\epsilon \propto 1/\sqrt{N}$ where N is the total number of points, which is equal to the number of points in one dimension n raised to the number of dimensions d; $N = n^d$, for single variable integrals this yields an error of $\epsilon \propto 1/\sqrt{n}$ for the Monte-Carlo integration method, which is significantly worse than the standard methods: Mid-point rule $\epsilon \propto 1/n$, trapezoidal rule $\epsilon \propto 1/n^2$, Simpson's rule $\epsilon \propto 1/n^4$.

We are interested in three types of integration methods: **Newton-Cotes methods**, **Gaussian quadrature rules** and **Romberg methods**. These methods are commonly used nowadays as they approach the task of approximating an integral from different angles.

We aimed to explore whether implementing the methods using the Python library Tensorflow would yield any benefits. Despite machine learning being the primary purpose for the development of Tensorflow, it provides a powerful infrastructure for mathematical operations, especially for more complex functions that can be sped up through utilizing the compiling feature of TensorFlow, which constructs a computational graph to improve efficiency.[11]

While TensorFlow provides the infrastructure for numerical integration, the library does not contain any implementations that could fulfill the requirements of high energy physics experiments. Hence a primary goal of this thesis is the implementation of well known single dimensional numerical integration algorithms using TensorFlow.

Since the methods differ fundamentally from one another, it is likely that they do not benefit equally from being implemented using Tensorflow. To examine this we compare the implementations in Tensorflow with native Python and Scipy implementations.

The integral being approximated is also likely to affect the performance of the methods, thus we need to test them using a range of different integrals, in order to determine the ideal conditions for each method.

The methods, as well as the theoretical benefit of implementing them using Tensorflow are explored in chapter 2, the progress of the Tensorflow implementations, comparisons with different implementations and comparisons of the individual methods are discussed in chapter 3 and a review of the results and an overview of future plans are given in chapter 4.

2. Computing and numerical integration

Numerical integration, formerly called quadrature, has a history extending all the way back to the invention of calculus, with the term "numerical integration" first appearing in 1915 in the publication *A Course in Interpolation and Numeric Integration for the Mathematical Laboratory* by **David Gibb** [8]. We will be using "numerical integration" as an umbrella term to refer to the use of different algorithms in order to obtain numerical approximations of definite integrals.

2.1 The importance of numerical integration

Before diving into numerical integration we must first recall the first fundamental theorem of calculus.[17]

Theorem 2.1.1 (First fundamental theorem of calculus) *Let f be a continuous function on the interval $[a, b]$ and let F be its anti-derivative, then:*

$$\int_a^b f(x)dx = F(b) - F(a)$$

Applying the theorem to the appropriate intervals we can deduce the existence of the integrals of elementary functions. However, the integrals of elementary functions could not, in general, be computed analytically.

In the absence of the exact solutions obtained analytically, the approximate solutions yielded by numerical integration become more appealing. Nonetheless, the effort required to use these methods by hand could not be understated, which stifled the progress of the field during the 18th and 19th centuries.

The field experienced a resurgence due to the invention of automatic computing; running a hundred or even a thousand iterations of an algorithm became possible. Numerical methods became capable of obtaining results within a very small margin of the exact results.

Automatic computing changed the goals of numerical integration methods: The aim was no longer to find methods that are possible for humans to utilize, rather, the aim was to develop algorithms such that computers could approximate integrals very precisely, which revived interest in methods that were rather inefficient for human use such as the Newton-Cotes formulas.

2.2 Newton-Cotes formulas

Newton-Cotes formulas refer to a class of numerical integration algorithms that are based on evaluations of the integrand function at equally spaced points.

There are two main categories of Newton-Cotes formulas, closed formulas and open formulas. The terms "open" and "closed" refer to the interval on which the formula is applied, with closed formulas containing the end points of the interval in the evaluation and open formulas neglecting said points as shown in Figure 2.1.

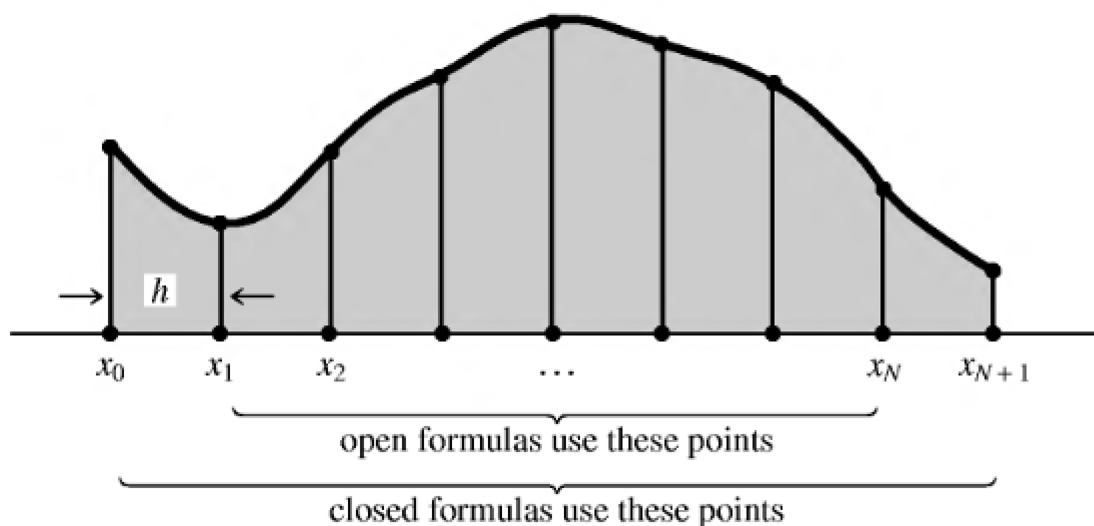


Figure 2.1: Comparison of closed and open Newton-Cotes formulas [2]

Newton-Cotes formulas are a quintessential part of the history of numerical integration, they revolutionized the field and are surely elegant, which makes their irrelevance in modern times quite disheartening. Indeed, there are only two Newton-Cotes formulas that are regularly used nowadays, those being the "extended trapezoidal formula" and the "extended mid point rule". Although far less popular, the "extended Simpson's rule" also makes appearances from time to time.

Before showcasing some Newton-Cotes formulas, an explanation for why these formulas fell out of favor is in order. The Newton-Cotes formulas function through the use of polynomial interpolation, with higher order formulas being needed for higher order integrand functions, the formulas function exceptionally well for low order polynomials [1]. However, the dependence on polynomial interpolation sometimes results in catastrophic Runge's phenomena for the closed formulas, in which the error grows exponentially with the degree of the polynomials used, leading to oscillations at the edges of the interval which prevent convergence. The open formulas are just outdone by Gaussian quadrature rules in all regards.

[1] up to degree 5

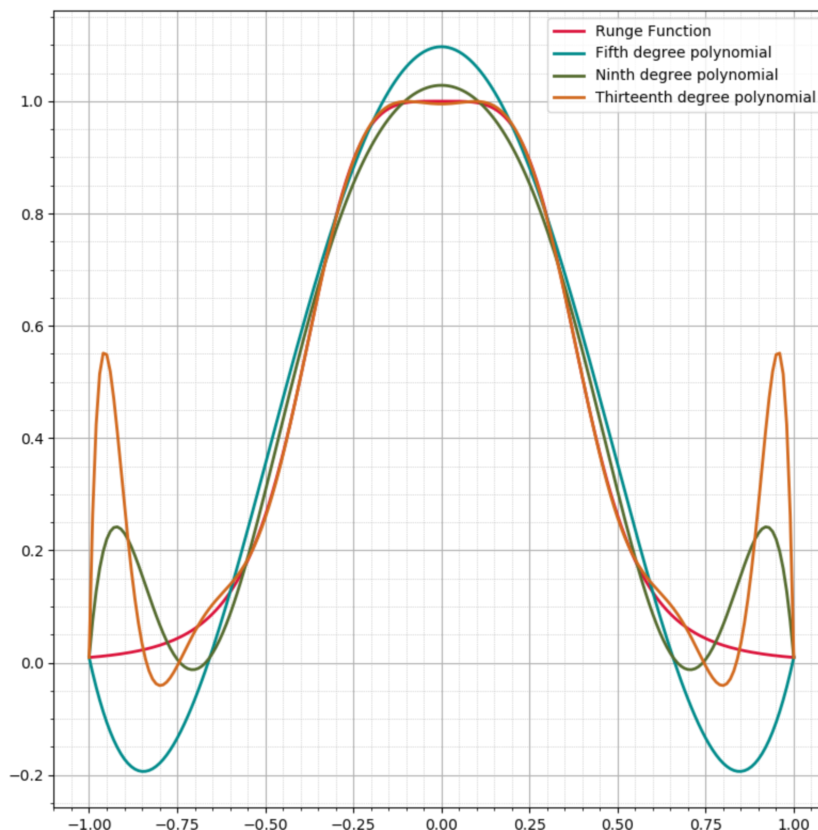


Figure 2.2: Runge's phenomenon for interpolation of $\frac{1}{1+25x^4+80x^6}$ using 5th, 9th and 13th degree polynomials

The failures of the Newton-Cotes formulas could be mitigated to a certain extent, but the effort needed to do so far outweighs the reward: The open formulas remain inferior to Gauss quadrature formulas and cannot be chained together to make extended formulas. The closed formulas can be substantially improved, but they are still outdone by other methods such as Romberg integration, which will be shown later.

There are four historically important closed Newton-Cotes formulas, using these formulas one could obtain exact results for polynomials of up to degree 5.

Trapezoidal rule

The trapezoidal rule evaluates the integrand at 2 points, approximating the integral using a linear function, therefore it is exact for degree one polynomials.

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f(x_1) + \frac{1}{2}f(x_2)\right] + O(h^3 f'') \tag{2.1}$$

The error term depends on some coefficient multiplied by the cube of the step size h and the second derivative of the integrand evaluated at some point in the domain, this point is unknown, so we can only obtain an upper bound on the error.

Simpson's rule

Simpson's rule evaluates the integrand at 3 points and due to cancellation of coefficients due to left-right symmetry, the formula is exact for polynomials up to and including degree three.

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f(x_1) + \frac{4}{3}f(x_2) + \frac{1}{3}f(x_3)\right] + O(h^5 f^{(4)}) \tag{2.2}$$

Once again the error is unknown, although it is now dependent on the fourth derivative and the fifth power of the step size, leading to a lower upper bound in most cases.

Simpson's 3/8th rule

Using four points instead of three yields no direct benefit as there is no cancellation of coefficients and the error term does not change in any significant way. The main benefit of this formula is that it allows for the derivation of Boole's rule.

$$\int_{x_1}^{x_4} f(x)dx = h\left[\frac{3}{8}f(x_1) + \frac{9}{8}f(x_2) + \frac{9}{8}f(x_3) + \frac{3}{8}f(x_4)\right] + O(h^5 f^{(4)}) \tag{2.3}$$

Boole's rule

As a result of left-right symmetry, this formula is exact for polynomials up to and including degree five.

$$\int_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f(x_1) + \frac{64}{45}f(x_2) + \frac{24}{45}f(x_3) + \frac{64}{45}f(x_4) + \frac{14}{45}f(x_5)\right] + O(h^7 f^{(6)}) \tag{2.4}$$

The error term now depends on the sixth derivative and seventh power of the step size, further decreasing the upper bound in most cases.

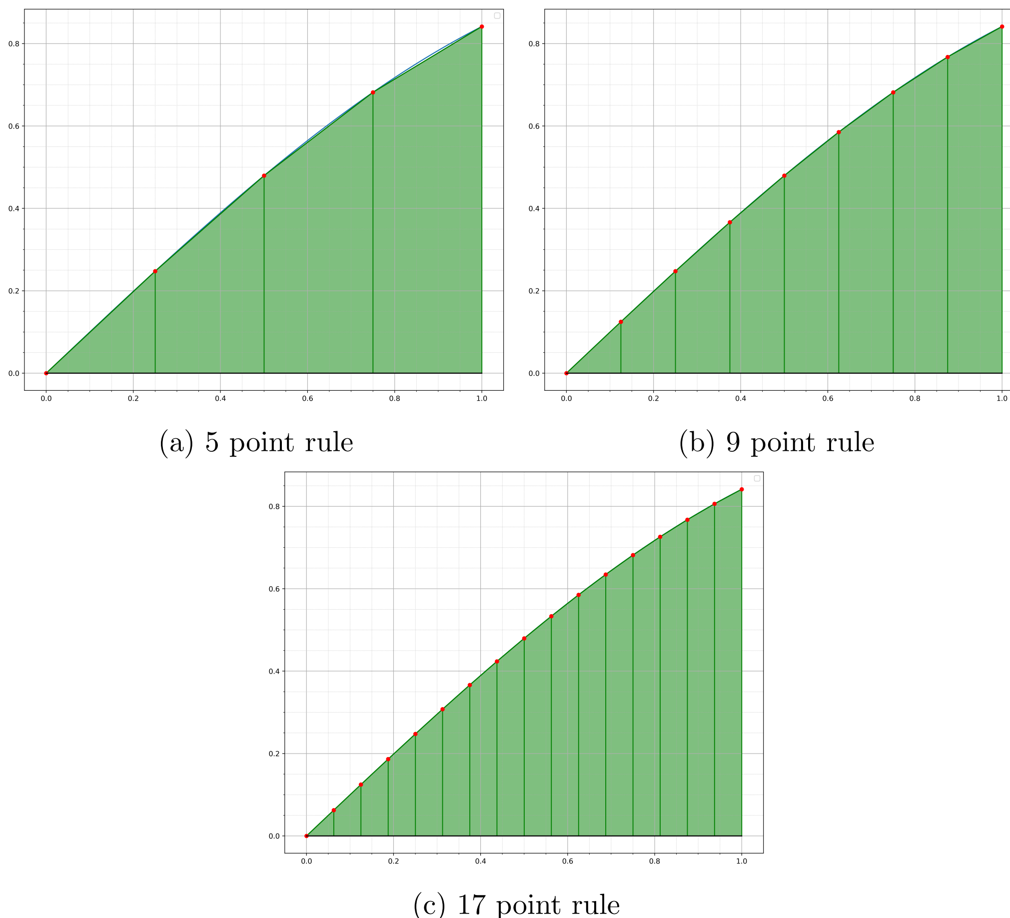
It is possible to derive higher order formulas, but this is the point at which it becomes likely to encounter Runge's phenomenon, thus it is a good point to stop and search for alternative approaches.

2.2.1 Extended Formulas

Extended formulas refer to the use of an equation such us (2.1) $N-1$ times on intervals $[x_1, x_2], [x_2, x_3], ..., [x_{N-1}, x_N]$ to obtain a composite rule of sorts such as the following.

$$\int_{x_1}^{x_N} f(x)dx = \frac{(x_N - x_1)}{N}\left[\frac{f_N + f_1}{2} + \sum_{k=1}^{N-1}\frac{f_k}{2}\right] + O\left(\frac{(x_N - x_1)^3 f''}{N^2}\right) \tag{2.5}$$

The error term being inversely proportional to the square of the number of intervals provides us with a clear method to obtain more accurate results. Naively one might assume that we could obtain arbitrarily accurate results through dividing the interval into more and more sub-intervals, this can be efficient as in Figure 2.3, but due to the dependence on the second derivative, the error is significantly larger around peaks, hence simply adding intervals uniformly would result in a substantial amount of unnecessary computing as shown in Figure 2.4.

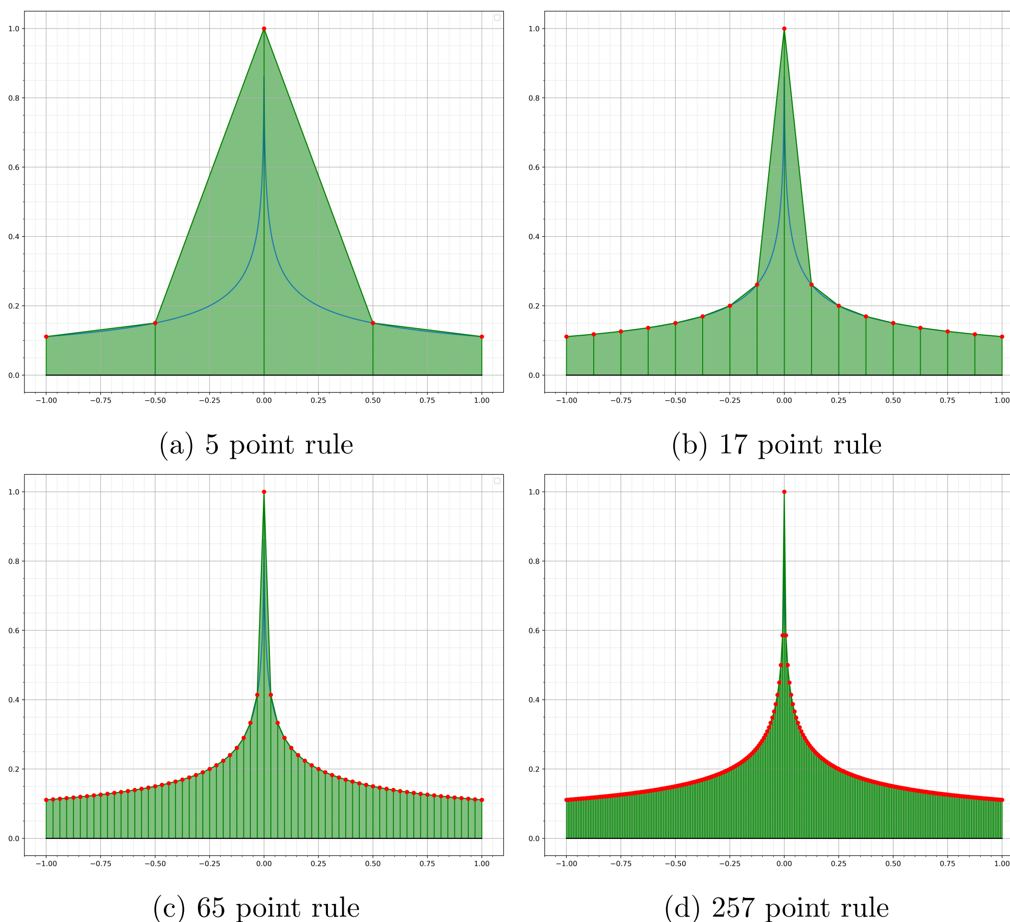


(a) 5 point rule

(b) 9 point rule

(c) 17 point rule

Figure 2.3: Simpson's rule approximations for $sin(x)$ on the interval $x \in [0, 1]$



(a) 5 point rule

(b) 17 point rule

(c) 65 point rule

(d) 257 point rule

Figure 2.4: Simpson's rule approximations for $\frac{1}{1+8\sqrt{|x|}}$ on the interval $x \in [-1, 1]$

There are a few ideas that one could implement in an attempt to improve upon the results obtained through using the extended trapezoidal rule, the simplest is to start with a higher order method in the first place, such as Simpson's rule.

$$\int_{x_1}^{x_N} f(x) = \frac{(x_N - x_1)}{N} \left[\frac{f_N + f_1}{3} + \sum_{k=1}^{(N-1)/2} \frac{4}{3} f_{2k} + \sum_{k=0}^{(N-1)/2} \frac{2}{3} f_{2k+1} \right] + O\left(\frac{(x_N - x_1)^5 f^{(4)}}{N^4} \right) \tag{2.6}$$

Using Simpson's method results in a significant performance improvement in the majority of cases when compared to the trapezoidal method, but it remains inefficient when compared to currently used integration methods.

As mentioned previously, one of the biggest obstacles one faces when using these methods is the unnecessary computing when adding sub-intervals uniformly, it is then natural to ponder what would happen if we were to add intervals in a non-uniform manner.

The idea described in the previous paragraph is known as an "adaptive" approach, instead of treating all integrands in the same manner, the method now adapts to the specific problem at hand. An appealing idea, but it is not immediately evident how one would go about implementing such a concept.

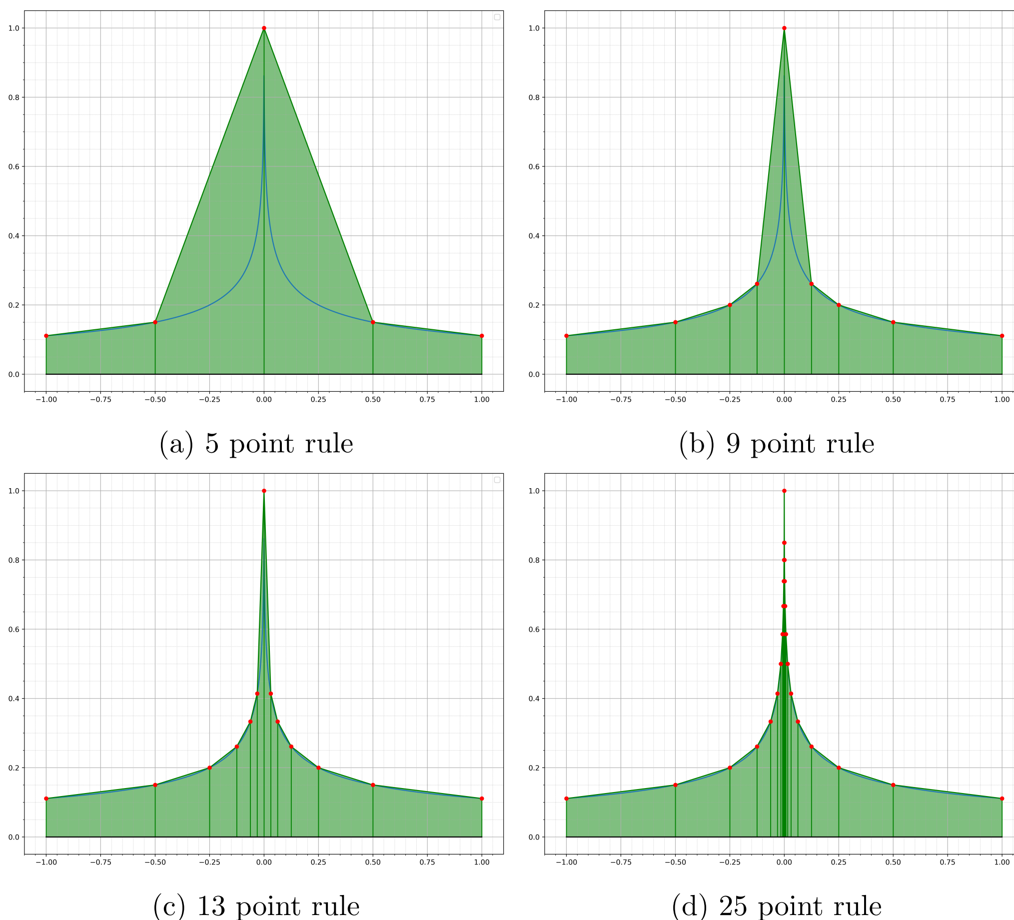
2.2.2 Adaptive integration methods

The first step to an adaptive method is establishing a criterion which would be used to determine where an interval should be split. The most common approach is to assign a tolerable absolute error and to split the interval in half if the absolute error is above the tolerance.

As mentioned previously, the absolute error from Newton-Cotes formulas is unknown, thus we use the difference between the results of consecutive iterations of the algorithm as an estimate of the error.

Once an interval is split we apply the method on each of the smaller intervals until an interval converges; its absolute error drops below the tolerance. Then the method terminates for that interval.

As explained earlier, the error term is largest near peaks, so the majority of interval splits will be occurring in that region. Utilizing this method, we can improve the precision of the result at a significantly lower computing power cost when compared to uniformly adding intervals, which is illustrated in the figures 2.4 and 2.5.



(a) 5 point rule

(b) 9 point rule

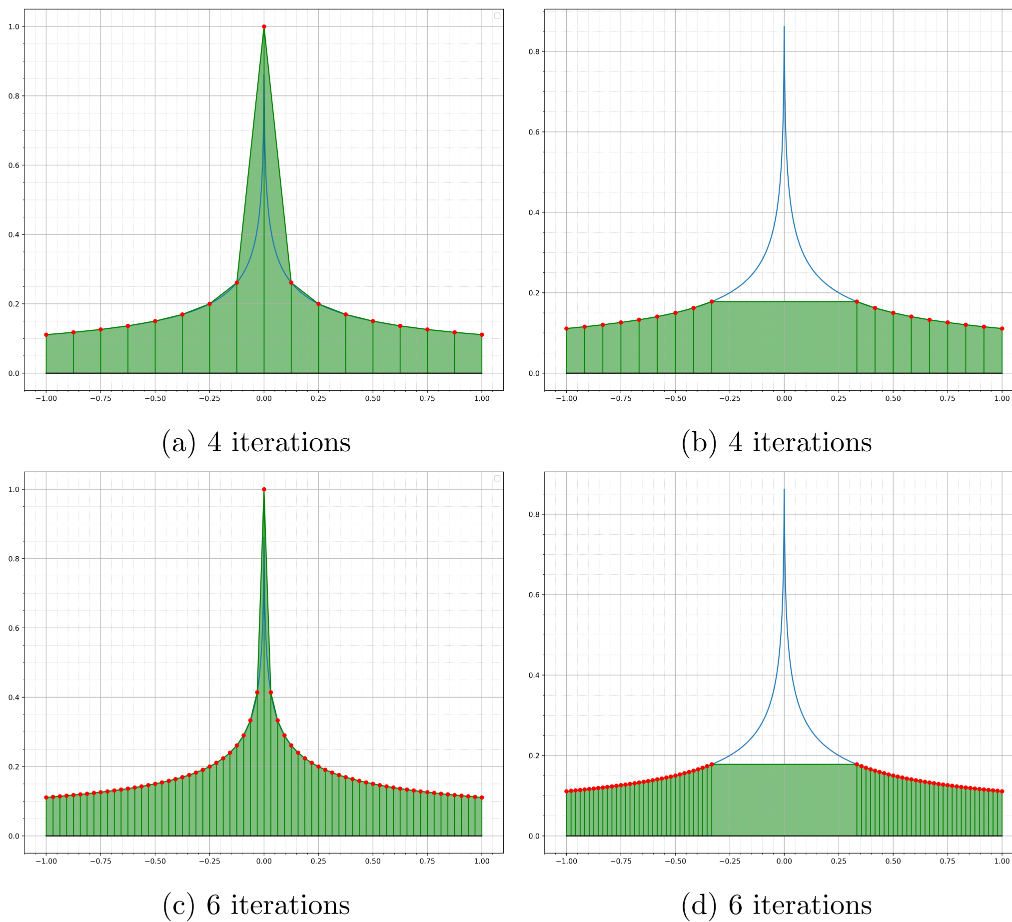
(c) 13 point rule

(d) 25 point rule

Figure 2.5: Adaptive Simpson's rule approximations for $\frac{1}{1+8\sqrt{|x|}}$ on the interval $x \in [-1, 1]$

Recursion is a powerful tool for the implementation of such methods when it is supported by the programming language being used. Alternative approaches are possible, although they typically require different logical flow and different interval splitting criteria. An example of such an implementation can be found in section A.2.

These methods can mitigate the flaws of the Newton-Cotes methods greatly, but the initial number of intervals plays a major role in determining their effectiveness.



(a) 4 iterations (b) 4 iterations

(c) 6 iterations (d) 6 iterations

Figure 2.6: Adaptive trapezoidal rule approximations for $\frac{1}{1+8\sqrt{|x|}}$ on the interval $x \in [-1, 1]$, starting with 3 points (Left) vs 4 points (Right)

2.3 Romberg Integration

A fascinating aspect of the extended trapezoidal rule is that its error term not only begins with a $1/N^2$ dependence, rather it depends on only *even* powers of N, a property that is not shared by most higher order Newton-Cotes formulas.

If we were to evaluate an integral using the trapezoidal rule once with N steps and once with $2N$ steps to obtain the results S_N and S_{2N}, then the leading error term for S_{2N} would be 1/4 the size of the leading error term for S_N, thus we could remove the leading error term using the combination.

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \tag{2.7}$$

Where we divide by 3 since all terms other than the leading error term would have been amplified by a factor of $4 - 1$.

It should be relatively simple to see that the above equation is in fact the extended Simpson's rule, with terms alternating between 2/3 and 4/3. This implies that we could replace the trapezoidal rule with Simpson's rule. We wil explore whether there is a benefit in this thesis, as this possibility has historically been ignored or avoided.

Naturally, we can generalize the previous procedure to any number of iterations that we would like, this is known as Romberg's method and it is an application of a more general idea known as *Richardson's deferred approach to the limit* or *Richardson extrapolation*.

The main premise of the Romberg's method is to use the result from k successive iterations of the trapezoidal method in order to cancel the leading error terms up to but not including $O(1/N^{2k})$. This routine is referred to as **sequence acceleration**, in which the rate of convergence is improved through use of a sequence transformation.

Romberg's method enjoys a few additional benefits when it comes to computation; when doubling the number of points used by the trapezoidal rule, half of the new points will be the ones that had already been evaluated. Through reusing these points we can significantly reduce the computation cost. Additionally, we can easily obtain an estimate for the error by comparing entries in different columns or rows, which provides us with a range of termination criteria to choose the ideal one from.

Theorem 2.3.1 (Romberg's method) *Let R be an $n \times n$ matrix, T_m be the trapezoidal rule evaluated using m points and j, k indices such that $j, k \leq n$, then Romberg's method is defined by the following equations.*

$$\begin{aligned} R_{k,0} &= T_{2k} \\ R_{k,j} &= \frac{1}{4^j - 1}(4^j R_{k,j-1} - R_{k-1,j-1}) \end{aligned} \tag{2.8}$$

2.4 Gaussian quadrature

The Newton-Cotes formulas we have seen up until this point all used equally spaced points within an interval in combination with variable weights for the evaluation of the integral, the adaptive methods allowed us the freedom to choose these intervals, but the evaluation was still performed using equidistant points. In comparison Gaussian quadrature rules aim to approximate an integral by picking optimal points to evaluate the integral at, often referred to as “abscissa” or “nodes”, and pairing them with the appropriate weights, thus doubling the degrees of freedom.

Doubling the degrees of freedom essentially allows for the derivation of quadrature formulas that are of twice the order of the Newton-Cotes formulas for the same number of function evaluations. Gaussian quadrature are open rules, thus they are not susceptible to Runge’s phenomenon

While the Gaussian quadrature rules improve upon the Newton-Cotes rules in some notable ways, they are not without fault. Since they are typically of significantly higher order, their precision is directly tied to the smoothness [2] of the integrand over the interval of interest. The derivation of Gaussian quadrature rules requires calculation of weights and abscissa, which is a very demanding task and there is a constant need for higher order rules to obtain better approximations.

There is, however, one additional feature of Gaussian quadrature formulas. We can arrange the weights and abscissa to make the methods exact for a different class of integrands, namely “polynomials times some weight function $w(x)$” instead of the usual class of polynomials, allowing for exact calculation of integrands that could be factorized into a polynomial and the weight function $w(x)$.

$$\int_a^b g(x) = \int_a^b w(x)f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j) \tag{2.9}$$

The abscissa are chosen according to the fundamental theorem of Gaussian quadrature. [18]

Theorem 2.4.1 (Fundamental theorem of Gaussian quadrature) *The abscissas of the N-point Gaussian quadrature formula are precisely the roots of the orthogonal polynomial for the same interval and weight function.*

For commonly used weight functions, the weights and abscissa have already been calculated and tabulated in various books and online sources, in such a case we can simply use the quadrature without understanding the underlying theory. Nonetheless, it is instructive for us to familiarize ourselves with a calculation process, in case we are interested in an uncommon weight function.

[2]Smoothness is a measure of the number of continuous derivatives that a function has over some interval

2.4.1 Calculation of abscissa and weights

The key to the calculation of abscissa and weights lies in the properties of orthogonal polynomials. Monic orthogonal polynomials fulfill a three term recurrence relation. [3]

$$\begin{aligned} p_n(x) &= (x - a_n)p_{n-1}(x) - b_n p_{n-2}(x), n = 2, 3, ... \\ p_0(x) &= 1 \\ p_1(x) &= x - A_1. \end{aligned} \tag{2.10}$$

The coefficients are given by the following equations, where $(,)$ represents the scalar product [3] and p_k is the kth order polynomial .

$$\begin{aligned} a_n &= \frac{(p_n, xp_n)}{(p_n, p_n)} \\ b_n &= \frac{(p_n, p_n)}{(p_{n-1}, p_{n-1})} \end{aligned} \tag{2.11}$$

The zeros of orthogonal polynomials are the eigenvalues of a particular tridiagonal matrix, the matrix is generated by the coefficients as follows.

$$T_n = \begin{bmatrix} a_1 & \sqrt{b_2} & 0 & 0 & \ddots \\ \sqrt{b_2} & a_2 & \sqrt{b_3} & 0 & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & \sqrt{b_{n-1}} & a_{n-1} \end{bmatrix} \tag{2.12}$$

The characteristic polynomial P_n satisfies the same recursion relation as the monic orthogonal polynomials p_n, in fact $P_i = p_i$ for all i.

The matrix T_n can be diagonalized $T_n = VDV^T$, where $D = diag(\lambda_1, ..., \lambda_n)$ with $V = [v_1]...[v_n]$ then the nodes x_j and weights w_j are given by the following equations.

$$x_j = \lambda_j \quad , \quad w_j = 2(v_j)_1^2 \quad , \quad j = 1, 2, ..., n \tag{2.13}$$

[3] $(a(x), b(x)) = \int_a^b a(x)b(x)dx$

2.4.2 Gauss-Legendre and Gauss-Kronrod quadrature

The Gauss-Legendre quadrature is perhaps the most commonly used Gaussian quadrature, this is due to the Legendre polynomials being orthogonal with respect to the weight function $w(x) = 1$, thus allowing this method to be applied to any choice of integral without any manipulation of the integrand.

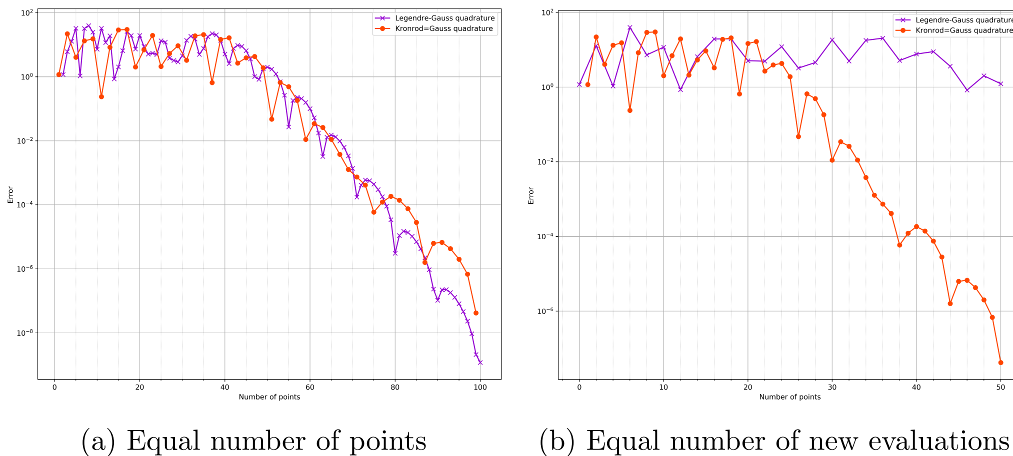
The Gauss-Legendre quadrature is exact for polynomials of degree $2N - 1$ where N is the number of evaluation points. Hereby, it is key to obtain the evaluation points to a large number of significant figures as the small number of evaluations results in each point significantly affecting the final result.

The points of different N-point Gaussian quadrature never coincide, this means that we would require $N + 1$ new points and weights for a small improvement in accuracy, this motivated the derivation of nested quadrature rules.

A nested quadrature method combines two quadrature rules, such that all the points of the first are incorporated into the second, thus allowing for a larger improvement in the order for the same increase in the number of evaluations. In the case of the Gauss-Legendre quadrature, we can combine it with the Gauss-Kronrod quadrature to obtain a nested quadrature formula.

The Gauss-Legendre and Gauss-Kronrod quadrature rules are related by a $2N + 1$ relation where N is the number of points. All the points of an N order Gauss-Legendre quadrature are incorporated into a $2N + 1$ Gauss-Kronrod quadrature, meaning that with $N + 1$ new points we can increase the order significantly[4]. A common example is a 7 point Gauss-Legendre quadrature combined with a 15 point Gauss-Kronrod quadrature.

It is worth noting that for the same number of points the Gauss-Kronrod quadrature is typically less accurate than the Gauss-Legendre quadrature. However, when we compare the number of new evaluations needed, then an improvement in accuracy is observed for the nested quadrature rule.



(a) Equal number of points (b) Equal number of new evaluations

Figure 2.7: Gauss-Legendre quadrature vs Gauss-Kronrod quadrature for $\int_{-0.5}^{-0.02} \frac{-1}{x^2} \cos(\frac{1}{x})$

[4] Increase in order from $2N - 1$ to $3N + 1$ instead of $2N + 1$

2.5 Gauss-Romberg method

Since Gaussian quadrature methods have an edge in precision over the extended Newton-Cotes formulas, one natural development is to try and implement them within the Romberg integration framework, thereby gaining an improvement over the traditional Romberg integration methods using the trapezoidal rule or Simpson's rule.

It was shown in 1972 by **J.N.Lyness** that the algorithm for Gauss-Romberg integration is actually identical in form to that of the traditional Romberg integration method.[1]

Theorem 2.5.1 (Gauss-Romberg method) *Let G be an $n \times n$ matrix and L_m the Gaussian quadrature evaluated using m points with j, k indices fulfilling $j, k \leq n$, then the Gauss-Romberg method is given by the following equations.*

$$\begin{aligned} R_{k,0} &= L_k \\ R_{k,j} &= \frac{1}{4^j - 1}(4^j G_{k,j-1} - G_{k-1,j-1}) \end{aligned} \tag{2.14}$$

This method is largely untested and is often seen as excessive, with potentially unforeseen complications arising when combining two powerful integration methods. In this thesis we will explore its efficiency when compared to other, better established methods.

2.6 Implementation in Tensorflow

Our choice to implement the methods in Tensorflow is motivated by a desire to examine whether Tensorflow's compiling feature and its ability to run on GPU could result in improvements in performance when compared to implementations of the methods in other libraries.

The compiling feature refers to Tensorflow building a computational graph of the various operations prior to execution. This feature is particularly valuable when we have repeating tasks, as the program could then identify them and carry them out before evaluation instead of going through them one by one, thereby reducing the run-time.

Tensorflow's compiling feature places a few constraints on the permitted code: Any variables must be created before creating the graph, assignment for non-variables is not supported and the data types need to remain the same throughout the graph. This can lead to some inefficiencies when running individual functions, which is why this feature is mainly intended for large functions that consist of multiple smaller parts.

Recursion is not supported when using the compiling feature of Tensorflow, the reasoning for this is rooted in the evaluation only taking place at the end. The parameters of one function cannot be used as inputs for another function until the run is completed, this results in the code being stuck at the recursion step.

Losing the ability to use recursive code seems catastrophic initially, but as proven by the Church-Turing thesis, any recursive function can be converted to an iterative function, which is supported by Tensorflow's graph framework through `tf.while_loop`.

Tensorflow can also run on GPU, which provide superior processing power and memory bandwidth when compared to CPU [10]. CPUs are better adjusted for general computing purposes, while GPUs excel at handling multiple functions at the same time. Consequently, running on GPU can result in significant performance improvements when running a large number of operations and evaluations [7], although it can also lead to decrease in efficiency when the number of evaluations is small as shown in Figure 3.13.

3. Tests and results

The testing of our implementations was carried out in three phases, the preliminary and secondary testing phases shared similar goals which can be summarized as follows:

- Verify that all the methods are operating correctly for a range of different functions.
- Identify the shortcomings of our implementations and potential ways to mitigate them.
- Examine whether the implemented adjustments addressed the shortcomings, or if further adjustments are needed.

The final testing phase commenced after satisfactory results were obtained from the prior testing phases. Since the methods were fully operational at this point we had a different set of goals:

- Benchmark the performance of each method according to agreement between absolute error and desired tolerance, then according to run-time.
- Examine the performance of our methods running on CPU compared to running on a GPU.
- Compare our implementations with pure python implementations as well as well established integration libraries such as scipy.integrate.

The preliminary and secondary testing phases were carried out only on the first CPU testing device, while the final testing phase was run on all four available testing devices to ensure that the observed patterns were not dependent on the testing device. Only results from the first CPU testing device and the GPU testing device are shown to preserve visual clarity.

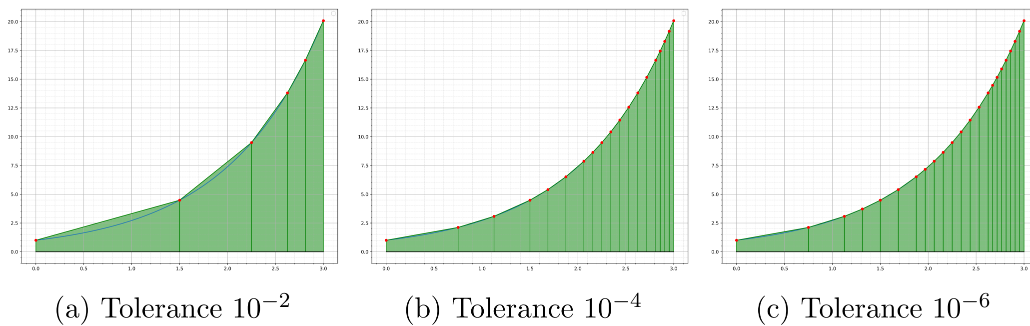
3.1 Preliminary testing

Following the initial implementation of the previously mentioned integration methods, a few tests were run to probe the methods for any deficiencies.

The following set of integrals was one of the two sets used for the tests.

1. $\int_0^1 \sin(x)$
2. $\int_0^3 \exp(x)$
3. $\int_5^9 x^3 + x^2 + 9$
4. $\int_{-2}^2 x^5 - 2x^6 \cos(x) + 7$
5. $\int_5^{10} \frac{1}{x+5} - \frac{3}{2x-1}$

These integrals were chosen due to their simplicity, they could be well approximated using the Newton-Cotes methods within a few iterations as shown in the following figures.



(a) Tolerance 10^{-2} (b) Tolerance 10^{-4} (c) Tolerance 10^{-6}

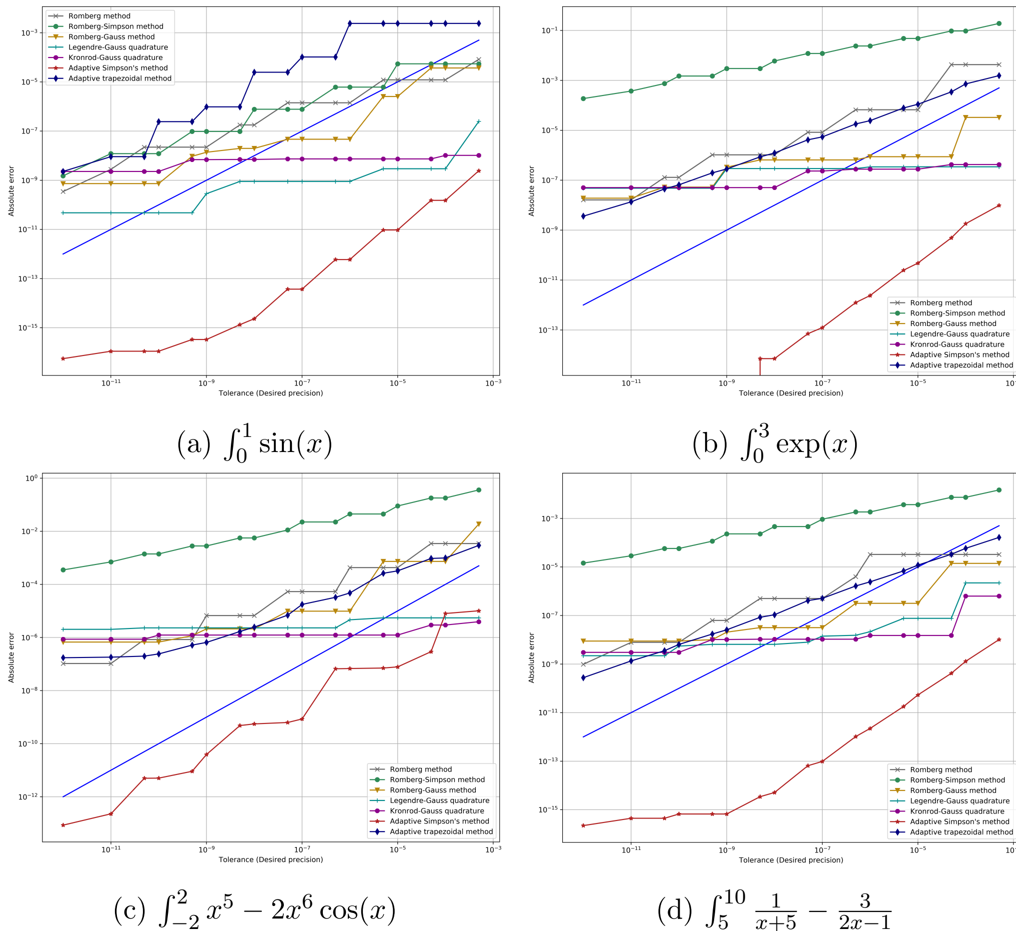
Figure 3.1: Simpson's rule approximations for $\int_0^3 \exp(x)$

By choosing simple functions we could check the reliability of our methods without having to worry about any strange behavior being caused by the nature of the integrals.

A vital criterion for determining the quality of the implementations of the methods was the comparison between the absolute error, defined as the absolute difference between the approximate result and the exact value of the integral, and the tolerance, which we define as the desired absolute error. A method would be declared to work ideally if the tolerance and absolute error are equal for any choice of tolerance.

The test consisted of comparing the numerical results obtained by the methods with the analytic result according to a range of tolerance values from 10^{-12} to $5 \cdot 10^{-4}$ as well as recording the run-time for each of the methods. The run-time was obtained through running each of the methods 30 times and averaging the results.

At the time of testing, the Newton-Cotes formulas were allowed a maximum of 15 iterations, the Romberg methods were allowed a maximum of 20 iterations and the quadrature methods were implemented up to 25 points for the Legendre-Gauss quadrature and up to 21 points for the Kronrod-Gauss quadrature.



(a) $\int_0^1 \sin(x)$ (b) $\int_0^3 \exp(x)$

(c) $\int_{-2}^2 x^5 - 2x^6 \cos(x)$ (d) $\int_5^{10} \frac{1}{x+5} - \frac{3}{2x-1}$

Figure 3.2: Error vs tolerance plots for the preliminary testing functions, linear blue line represents perfect agreement

For a tolerance of 10^{-6} or greater, the absolute error showed acceptable agreement with the tolerance, with the Romberg methods and the trapezoidal method showing better agreement than the other methods.

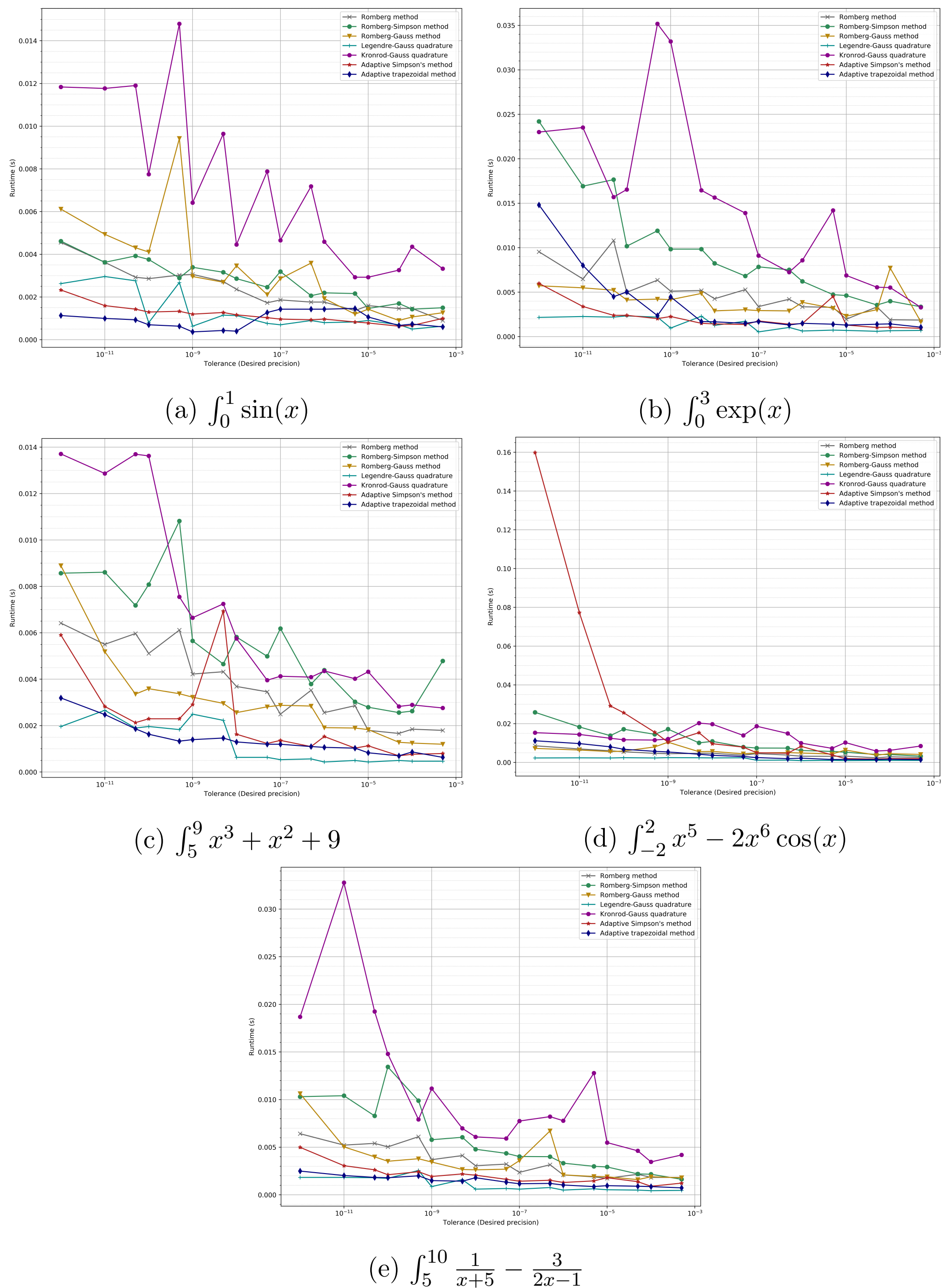
For a tolerance of 10^{-7} or less, the agreement was quite poor, this becomes more noticeable as the tolerance decreases. For the majority of the methods, this manifested through the methods obtaining results that are less accurate than desired, conversely the adaptive Simpson's method consistently obtained results that were more accurate than requested.

We attributed the disagreement to two factors:

- The methods were not allowed enough iterations to achieve greater accuracy which can be seen for the trapezoidal method by comparing Figure 3.2(a) and Figure 3.4(a) as well as for the quadrature methods through comparing Figure 3.2(d) with Figure 3.4(d).
- The termination criteria which consisted of comparing the absolute difference of the last two outputs of any method with the tolerance was too naive at greater accuracy. This is illustrated for the Romberg method through comparing Figure 3.2 with Figure 3.4 and noting that maximum number of iterations was not reached in either test.

The run-time of the individual methods was the second aspect of interest. During this phase the run-time was not used as a criterion for comparing the methods, but rather as way to examine the efficiency of the individual implementations to determine whether changes were needed.

Through observing the run-time we could see the downside of the adaptive Simpson method overachieving in terms of precision, in particular this can be seen in the plot for the fourth integral.



(a) $\int_0^1 \sin(x)$

(b) $\int_0^3 \exp(x)$

(c) $\int_5^9 x^3 + x^2 + 9$

(d) $\int_{-2}^2 x^5 - 2x^6 \cos(x)$

(e) $\int_5^{10} \frac{1}{x+5} - \frac{3}{2x-1}$

Figure 3.3: Run-time vs tolerance plots for the preliminary testing functions

The two main takeaways from the preliminary tests were that the methods needed more precise error to tolerance matching, a process for doing so is described in section A.3, and that they needed a higher limit on the number of iterations to allow for greater accuracy to be achieved.

3.2 Secondary tests

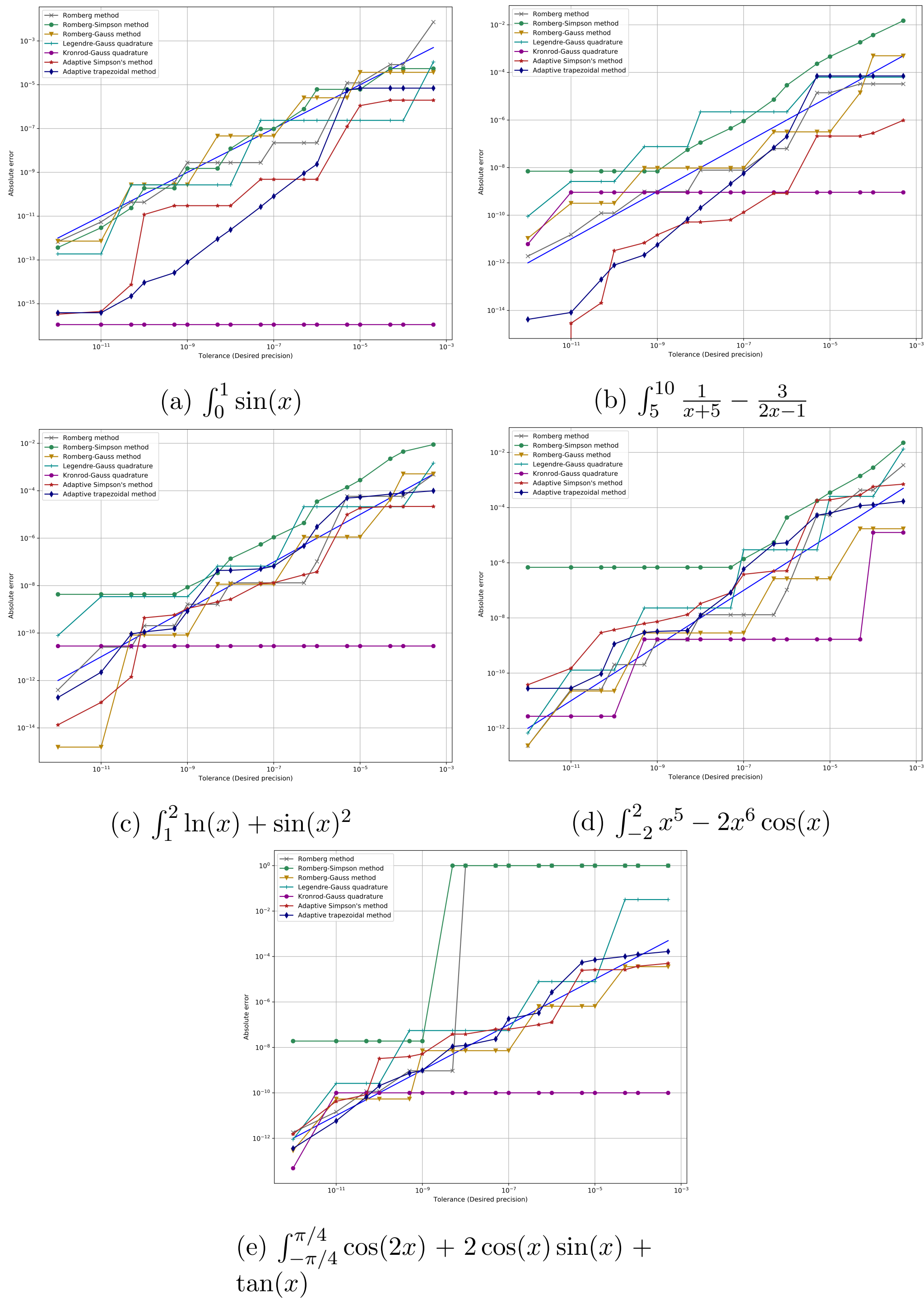
After addressing the issues mentioned in the previous section and changing the implementation for the Romberg-Simspon method, the Newton-Cotes and Romberg methods were allowed the maximum number of iterations possible for the testing device, which was 25, while the Legendre-Gauss and Kronrod-Gauss methods were implemented up to 40 and 43 points respectively.

To verify the effectiveness of the adjustments, the methods were tested once more using the following integrals:

1. $\int_0^1 \sin(x)$
2. $\int_1^2 \ln(x) + \sin(x)^2$
3. $\int_{-2}^2 x^5 - 2x^6 \cos(x) + 7$
4. $\int_5^{10} \frac{1}{x+5} - \frac{3}{2x-1}$
5. $\int_{-\pi/4}^{\pi/4} \cos(2x) + 2\cos(x)\sin(x) + \tan(x)$

A remarkable improvement can be seen in the error-tolerance matching for all the method by comparing Figure 3.2 with Figure 3.4. We also observe a clear enhancement in the accuracy of the Romberg-Simpson method, although it left a lot to be desired at higher precision.

Increasing the maximum number of iterations for the Newton-Cotes and Romberg methods as well as supporting a larger number of points for the quadrature methods allowed for lower absolute error values to be achieved on a regular basis, comparing Figure 3.2 with Figure 3.4 we see that the methods are now consistently capable of achieving an absolute error of 10^{-12} or lower.



(a) $\int_0^1 \sin(x)$

(b) $\int_5^{10} \frac{1}{x+5} - \frac{3}{2x-1}$

(c) $\int_1^2 \ln(x) + \sin(x)^2$

(d) $\int_{-2}^2 x^5 - 2x^6 \cos(x)$

(e) $\int_{-\pi/4}^{\pi/4} \cos(2x) + 2\cos(x)\sin(x) + \tan(x)$

Figure 3.4: Error vs tolerance plots for the secondary testing functions

The goal of these tests was to examine whether the methods were ready to be compared with one another, as well as with other integration methods. The data was promising but a few things stood out:

- The Kronrod-Gauss quadrature seemed rather unresponsive to the approach used to improve the matching between the error and the tolerance, this is likely due to two factors working in unison. The first being the simplicity of the functions used in the test, while the second is the fact that this quadrature only uses an odd number of points, so the change in the absolute error between iterations may be significantly larger than the requested tolerance; for example, the first integral is approximated to an error of $1 \cdot 10^{-1}$ with 5 points, and to an error of $1 \cdot 10^{-11}$ with 7 points.
- The adaptive Newton-Cotes methods showed some volatility, which resulted in some integrals being approximated slightly better than required, while others were approximated marginally worse than required. This volatility is a result of our chosen interval splitting which indicated that adjustments were needed.
- The Romberg-Simpson method was still performing significantly worse than the Romberg method, this was expected to a certain extent, but the difference was extreme enough for another change to the algorithm to be considered.
- The Romberg-Gauss method performed well, particularly in comparison to the Gauss-Legendre quadrature upon which it is based. It allowed for better error-tolerance matching in most cases, likely due to the variation between iterations being more marginal than for the Gauss-Legendre quadrature.
- In the case of the second integral we can see the quadrature methods struggling at the lowest tolerance values, which indicated that it may be beneficial to support an even larger number of points for the implementation.
- The fifth integral highlights a problem that one may face when using a Romberg implementation of any integration method. If the base method can get a reasonable approximation, then the Romberg implementation will further improve it. This can be seen by comparing the Legendre-Gauss quadrature result with the Romberg-Gauss result. On the other hand, if the base method fails, then the Romberg implementation will also fail, as seen for both the Romberg method and the Romerg Simpson methods at low accuracy.

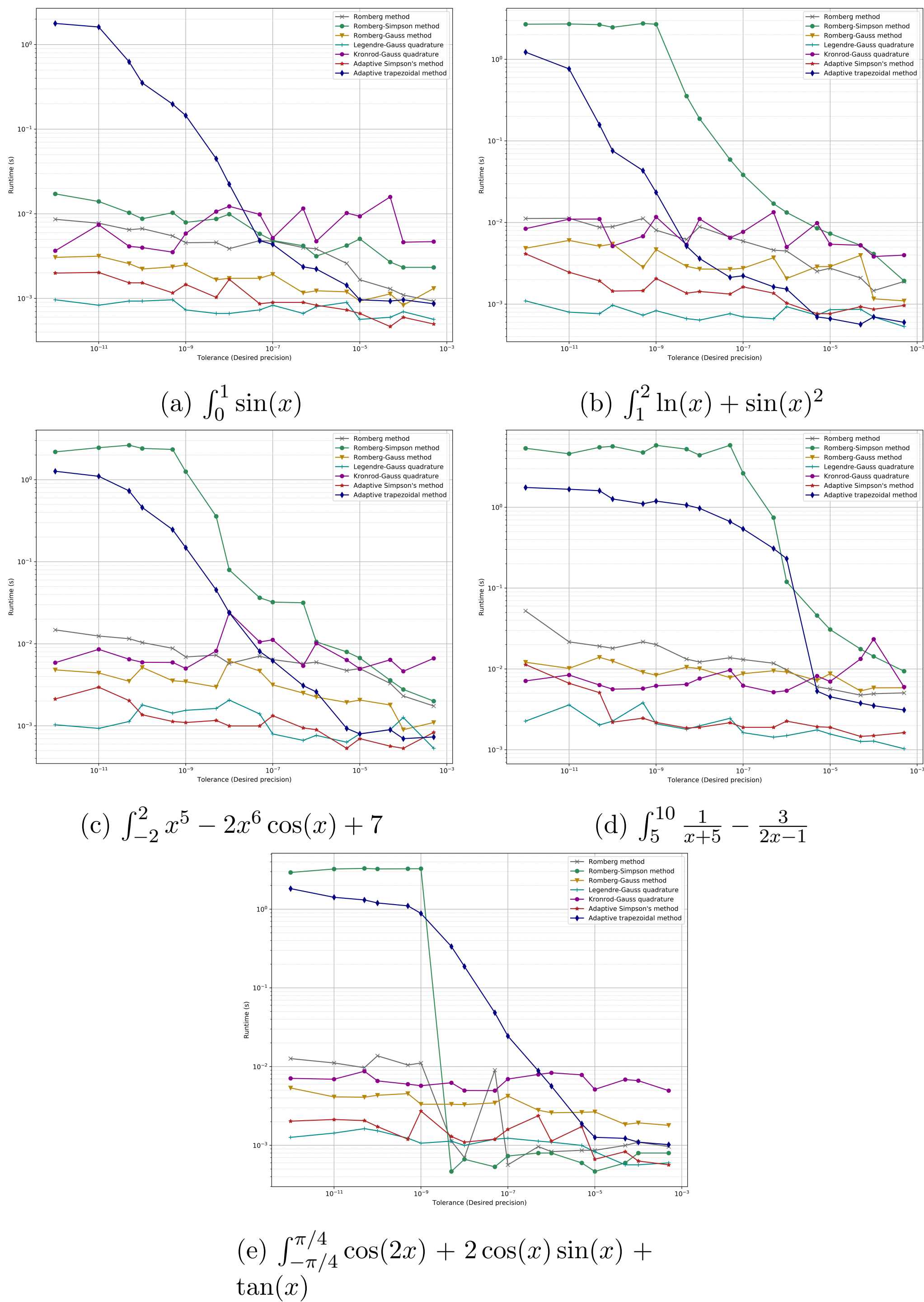
Improving the matching took a small toll on the run-time, thus a few tweaks were made to improve it. For the quadrature based methods this was done through importing and processing their nodes and weights in advance before the integration process, while for the Newton-Cotes methods, the main improvement came from increasing the number of sub-intervals at the start of the process, this allowed for quicker convergence as it was less likely to miss curves and peaks in the earlier iterations. The Romberg methods did not have much room for improvement without a complete rewrite, so they remained unchanged.

The Romberg and Romberg-Simpson methods use the largest number of evaluations for high precision results, this is also the case for simple functions, which suggests that they may be sub-optimal in comparison to quadrature methods or the adaptive Simpson's method when handling simple functions.

The run-time tests showed an interesting but unsurprising result as the adaptive trapezoidal method and the Romberg-Simpson method ran far slower at higher precision than any of the other methods. The adaptive trapezoidal method is known to converge slowly, so it was expected to require a longer time to obtain accurate results. The Romberg-Simpson suffered from a different problem, it could not achieve the desired precision in many cases, which resulted in it running for the maximum number of iterations, coupled with its inability to reuse points, unlike the Romberg method, this inflated the run-time. This is indicated by the run-time plot for the first integral, where its speed matches that of the remaining methods as it can achieve the desired precision.

Comparing the run time for the Romberg-Gauss method with the Gauss-Legendre quadrature we see that the improved error-tolerance matching comes at a comparably small cost, which hints at the Romberg-Gauss method being a viable option for numerical integration applications.

The Gauss-Kronrod quadrature regularly ran slower than the Gauss-Legendre quadrature, this is likely due to it being less accurate at the same number of points, thus requiring more iterations.



(a) $\int_0^1 \sin(x)$

(b) $\int_1^2 \ln(x) + \sin(x)^2$

(c) $\int_{-2}^2 x^5 - 2x^6 \cos(x) + 7$

(d) $\int_5^{10} \frac{1}{x+5} - \frac{3}{2x-1}$

(e) $\int_{-\pi/4}^{\pi/4} \cos(2x) + 2\cos(x)\sin(x) + \tan(x)$

Figure 3.5: Run-time for secondary testing functions (Logarithmic scale)

3.3 Final tests

After further upgrading the quadrature methods by allowing up to 100 points for the Legendre-Gauss quadrature and up to 99[1] points for the Kronrod-Gauss quadrature [14] [15] [16], we were ready to run a final round of tests, these tests aimed to accomplish the following:

- Benchmark the methods when applied to difficult integrals.
- Benchmark the methods running on a CPU in comparison to running on a GPU.
- Compare the Tensorflow adaptive Newton-Cotes methods with equivalent simple python implementations.
- Compare the implemented methods with the scipy.integrate library.
- Identify the upsides and shortcomings of the methods.

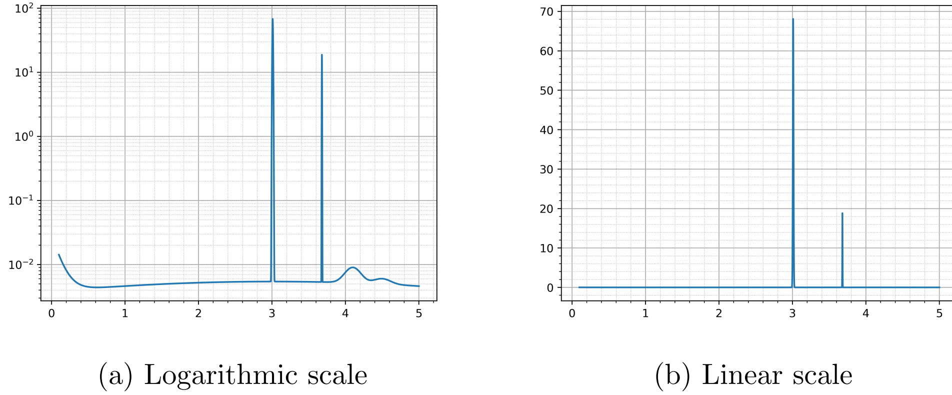
3.3.1 Benchmark using difficult integrals

The goal of this test was to identify whether the methods could accurately approximate rapidly oscillating functions or functions with sharp peaks, for that reason, the following functions were chosen.

1. $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$
2. $\int_{0}^{pi} (2\sin(x))^7$
3. $\int_{5}^{9} \cos(\exp(x))\exp(x)$
4. $\int_{-0.5}^{-0.02} \frac{-1}{x^2}\cos(\frac{1}{x})$

[1] Only defined for odd numbers of points.

Additionally an approximation of the branching fraction of the decay $B \to K^{*0}\mu^+\mu^-$ as a function of momentum transfer q^2 was used to test the methods, the distribution has the following form. [12]



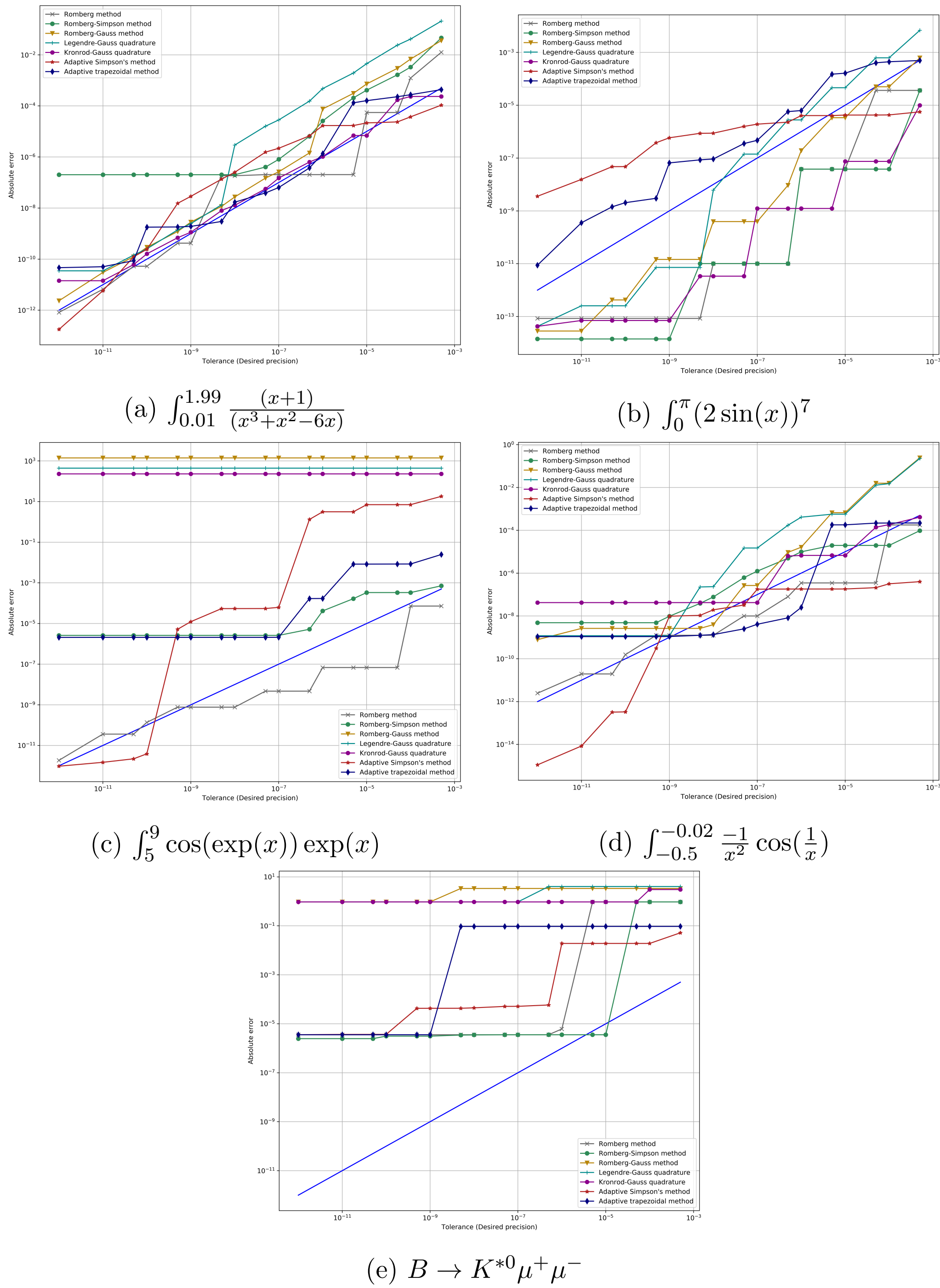
(a) Logarithmic scale (b) Linear scale

Figure 3.6: Approximate distribution of branching fraction as a function of q^2

During these tests, two limitations were frequently encountered. The available testing devices could not evaluate more than $7 \cdot 10^7$ points in a reasonable amount of time, which restricted the Newton-Cotes formulas and the Romberg methods based upon them. The second limitation relates to the quadrature methods, in a few cases the maximum available amount of points was insufficient for convergence. However, most sources for calculating the nodes and weights only go up to 100 points and the computation of more points using the method mentioned in subsection 2.4.1 requires an inordinate amount of time and computing power, thus we are restricted to no more than 100 points.

As mentioned in section 2.4, the quadrature methods are only well suited for the integration of smooth functions, thus it is unsurprising that they failed to converge for rapidly oscillating functions Figure 3.7(c) and functions with sharp peaks Figure 3.7(e).

The implemented Newton-Cotes and Romberg methods are of lower order than the quadrature methods, this results in smoothness being far less important, which allowed for reasonable approximations to be obtained.



(a) $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$

(b) $\int_0^{\pi} (2\sin(x))^7$

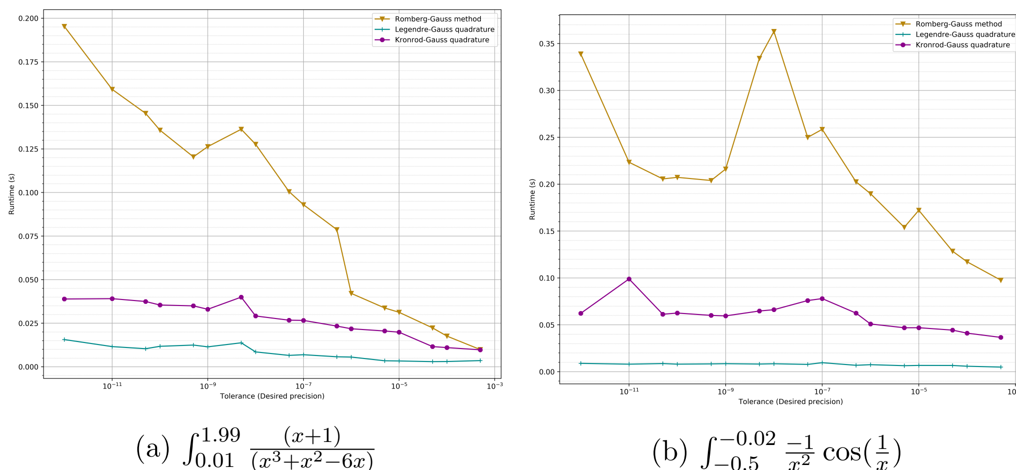
(c) $\int_5^9 \cos(\exp(x))\exp(x)$

(d) $\int_{-0.5}^{-0.02} \frac{-1}{x^2}\cos(\frac{1}{x})$

(e) $B \to K^{*0}\mu^+\mu^-$

Figure 3.7: Error vs tolerance for the final testing integrals

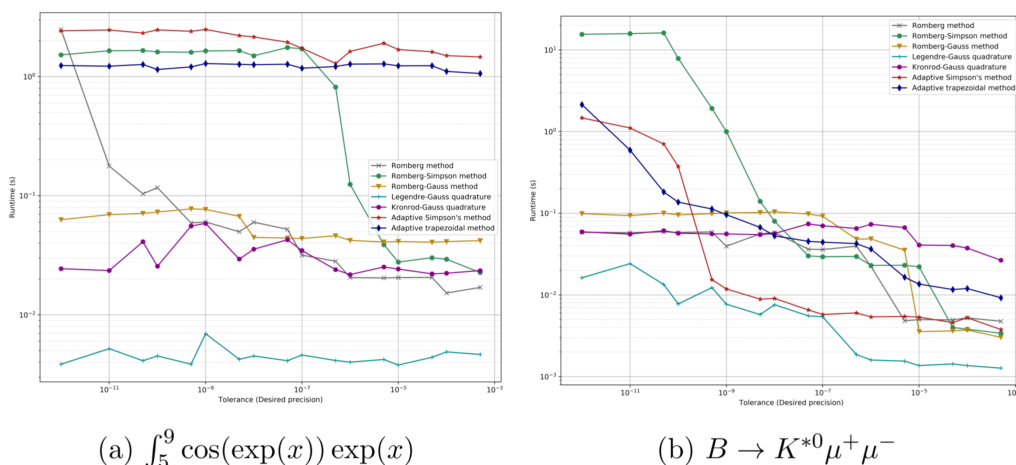
We consistently observe the Romberg-Gauss method providing better error-tolerance matching than the Legendre-Gauss method whenever the initial approximation is reasonable. This comes at the cost of a slower run-time, the improvement in accuracy relative to run-time seems to be worthwhile at lower tolerance values.



(a) $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$ (b) $\int_{-0.5}^{-0.02} \frac{-1}{x^2} \cos(\frac{1}{x})$

Figure 3.8: Run-time vs tolerance for the three quadrature methods

The third and fifth integrals were the least smooth of the set. Only the Newton-Cotes and Romberg methods converged to the correct result. Based on the run-time, we deduce that the Romberg method is ideal for most tolerance ranges, with the adaptive Simpson's method trailing, the Romberg-Simpson method and the trapezoidal method can provide reasonable results, but they pale in comparison to the previously mentioned methods.



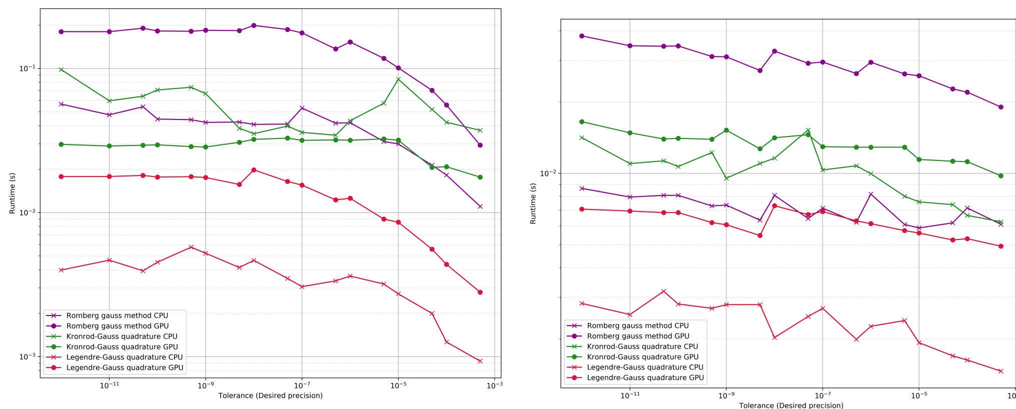
(a) $\int_5^9 \cos(\exp(x))\exp(x)$ (b) $B \to K^{*0}\mu^+\mu^-$

Figure 3.9: Run-time vs tolerance for the Newton-Cotes and Romberg methods

3.3.2 Benchmark of the methods running on CPU vs GPU

Tensorflow supports GPU usage, which can lead to improvements in performance.

The Newton-Cotes and Romberg methods were allowed the same maximum number of iterations on both devices, this meant that the error obtained would be the same regardless of the processing unit. The quadrature methods were only running up to 40 and 43 points respectively. The quadrature methods ran at similar speeds on GPU as on CPU, but run-time is not a major concern for quadrature methods as they are implemented up to 100 points only.

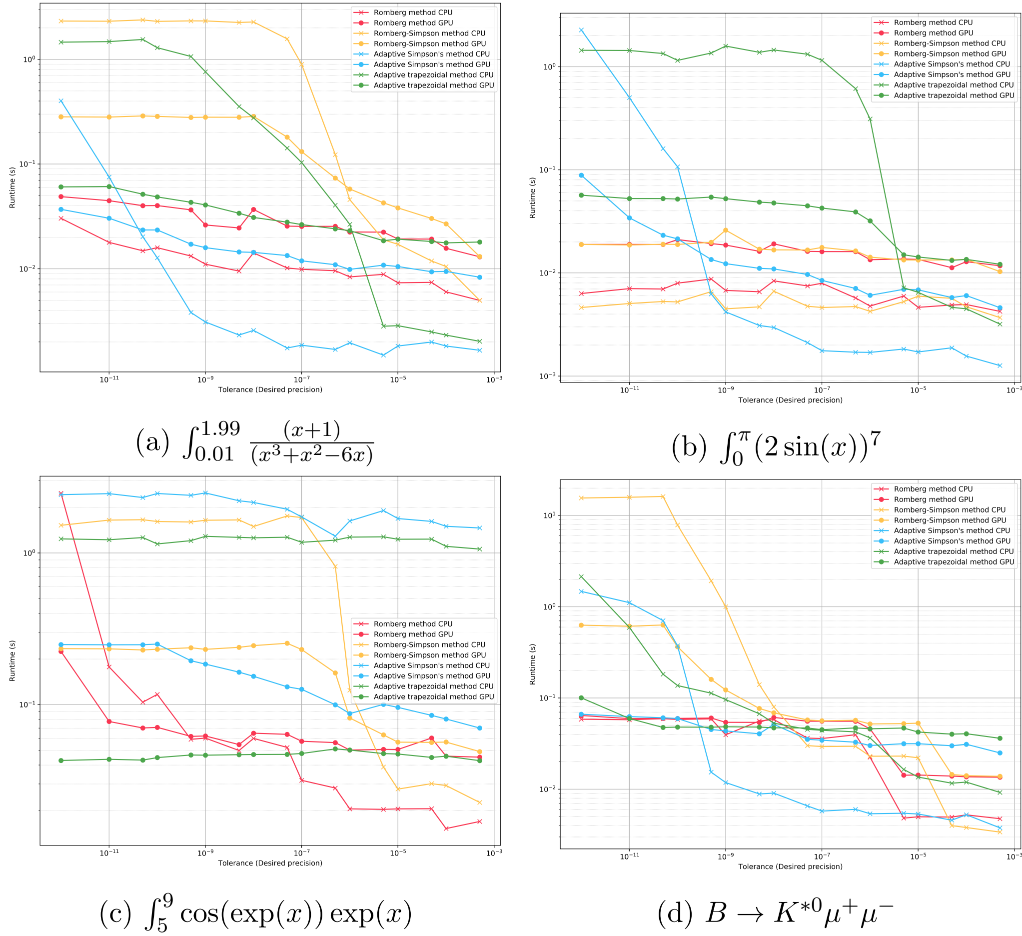


(a) $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$ (b) $\int_0^{\pi} (2\sin(x))^7$

Figure 3.10: Run-time vs tolerance for the quadrature methods running on CPU vs GPU (Logarithmic scale)

A far more interesting comparison would be the Newton-Cotes and Romberg methods as they are based on the repetition of a large number of simple steps, which are theoretically ideal processes for a GPU.

The results were surprising, running on CPU was faster initially, but for higher precisions operating on GPU yielded a noticeable improvement.



(a) $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$ (b) $\int_0^\pi (2\sin(x))^7$

(c) $\int_5^9 \cos(\exp(x))\exp(x)$ (d) $B \to K^{*0}\mu^+\mu^-$

Figure 3.11: Run-time vs tolerance for the Newton-Cotes based methods running on CPU vs GPU

The run-time experienced an improvement of a factor 10 for large numbers of evaluations. However, the program requires a large amount of memory when running on GPU, these tests were performed using a graphics card with 8 GB of GDDR6 memory, running for the maximum number of iterations required all the available memory. Running for about 20 iterations, which corresponds to using $2 \cdot 10^6$ points, required only 2GB, which should allow a tolerance of 10^-8 to be achieved.

The testing was performed using an *Nvidia RTX 2070*, which has poor double precision performance, thus it is possible to further reduce the run-time through using a graphics card that is better suited to the task at hand.

3.3.3 Newton-Cotes: Tensorflow vs Numpy

In this section we aim to examine whether there is a benefit to implementing the methods using *Tensorflow* in comparison to native python and Numpy.

The underlying methods are the same, thus any difference in error is either due to the number of evaluations or the libraries being used to evaluate the input function.

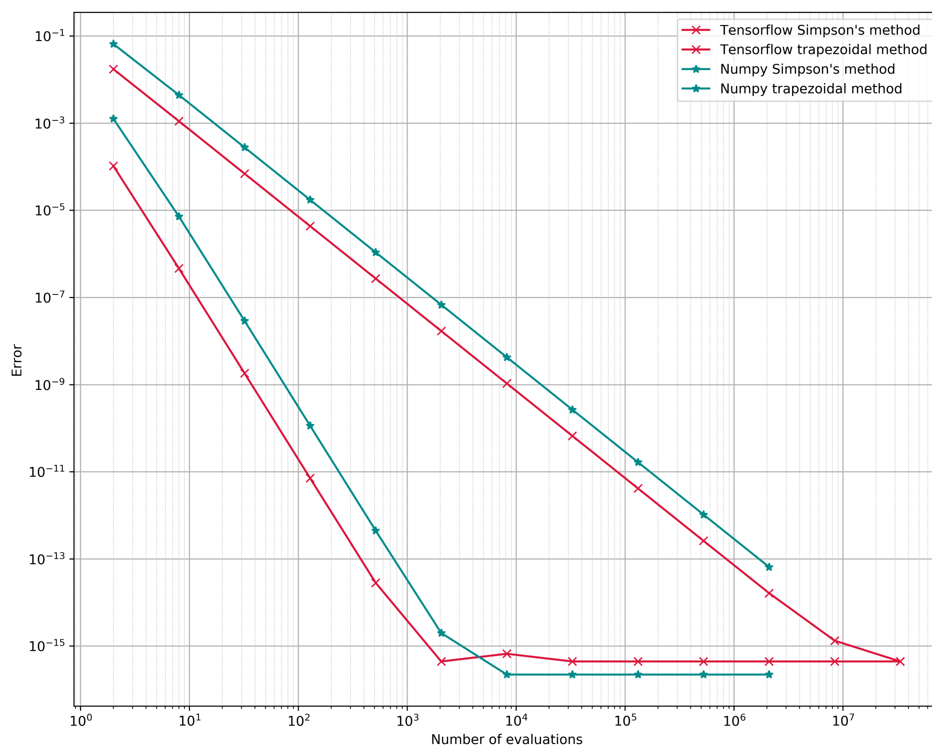


Figure 3.12: Error vs Number of evaluations for $\int_{4\pi/6}^{5\pi/6} \sin(x)\tan(x) + \frac{3}{x^2} + \ln(x^3)$

As we can see from the graph, the error plots are similar for both Tensorflow and Numpy, this is better showcased by the trapezoidal method, where it is clear that both have the same gradient. The cause for the separation between the two implementations is not clear, but it is not a major concern as the minimum relevant absolute error of 10^{-12} can be achieved by both methods using either library.

Although the Numpy implementations starts out as the fastest, this is only the case for a small number of evaluations, it falls behind in terms of run-time at about 10^3 evaluations and remains the slowest at larger numbers of evaluations. Interestingly, the CPU implementation of the methods is faster than the GPU version up to about 10^5 points, beyond that point we see a rapid increase in run-time.

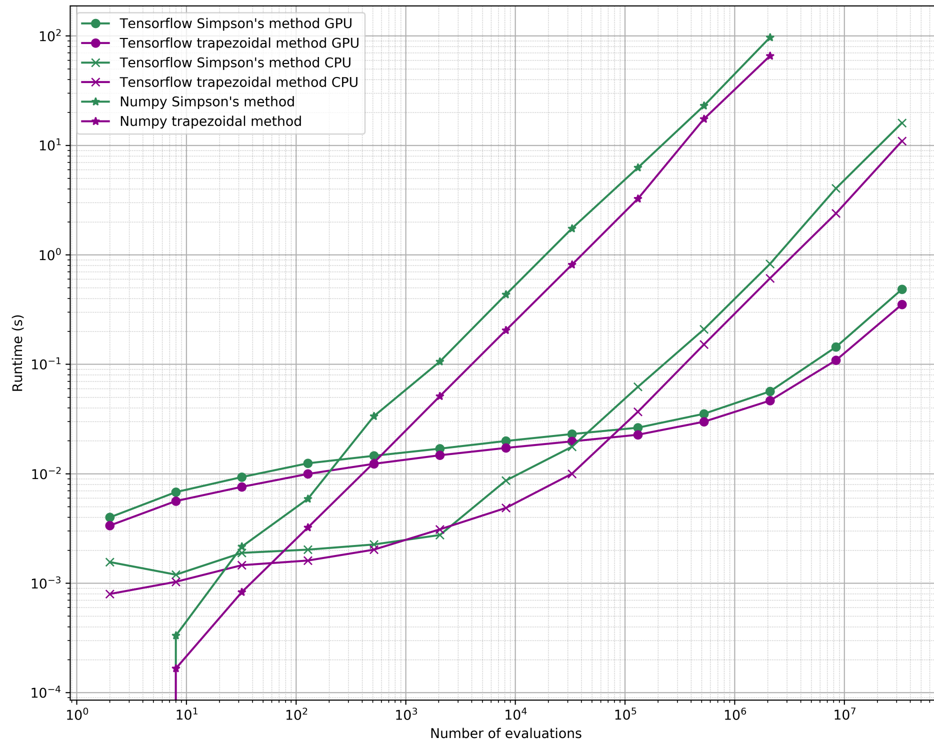


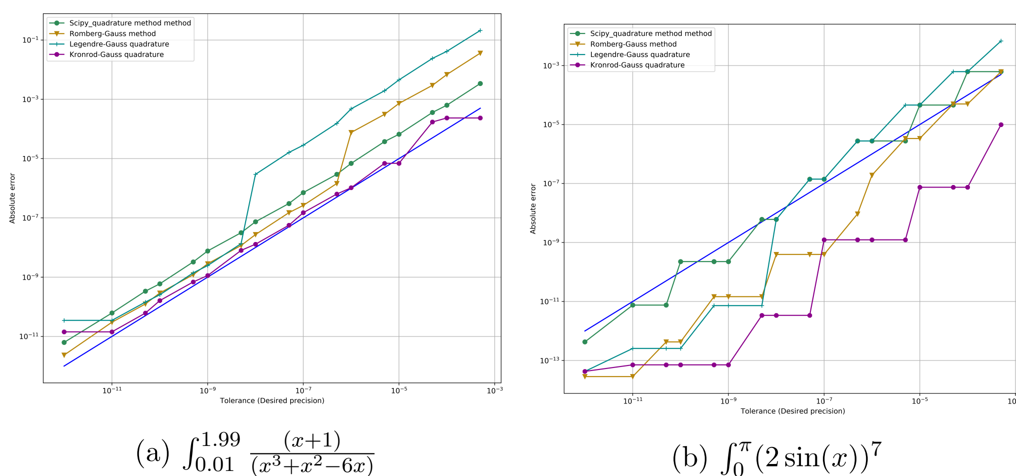
Figure 3.13: Run-time vs Number of evaluations for $\int_{4\pi/6}^{5\pi/6} \sin(x)\tan(x) + \frac{3}{x^2} + \ln(x^3)$

The improvement in run-time indicates that with proper optimization, it may be possible for a *Tensorflow* based integration library to surpass currently available python integration libraries.

3.3.4 Performance test against scipy.integrate

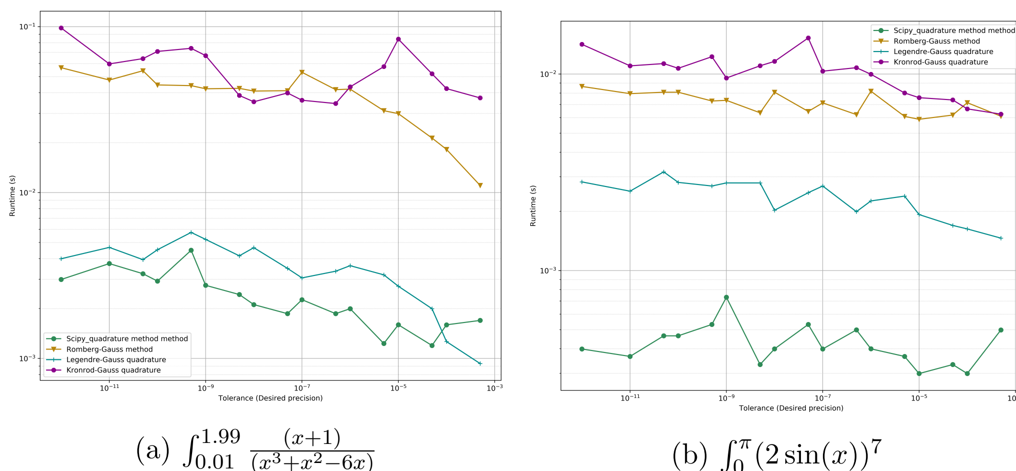
The scipy.integrate library provides three methods of integrating a single variable function up to a certain tolerance. Those being `scipy.integrate.quad` which utilizes the Fortran library QUADPACK to compute a definite integral, which will be referred to as "the default Scipy method", the second method is `scipy.integrate.quadrature` which is simply a Gaussian quadrature method and lastly `scipy.integrate.romberg`; an implementation of the Romberg method.[13]

The results from the quadrature integration were rather unsurprising, as the Scipy implementation is only accurate up to 100 points as well, hence the error results were quite similar and the run-time did not show any significant difference between the two approaches.



(a) $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$ (b) $\int_0^{\pi} (2\sin(x))^7$

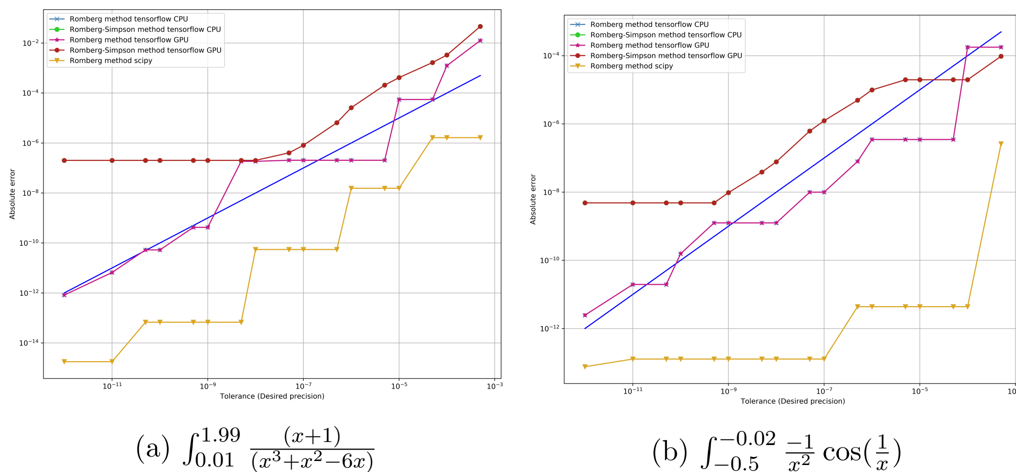
Figure 3.14: Error vs tolerance for the quadrature methods



(a) $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$ (b) $\int_0^{\pi} (2\sin(x))^7$

Figure 3.15: Run-time vs tolerance for the quadrature methods

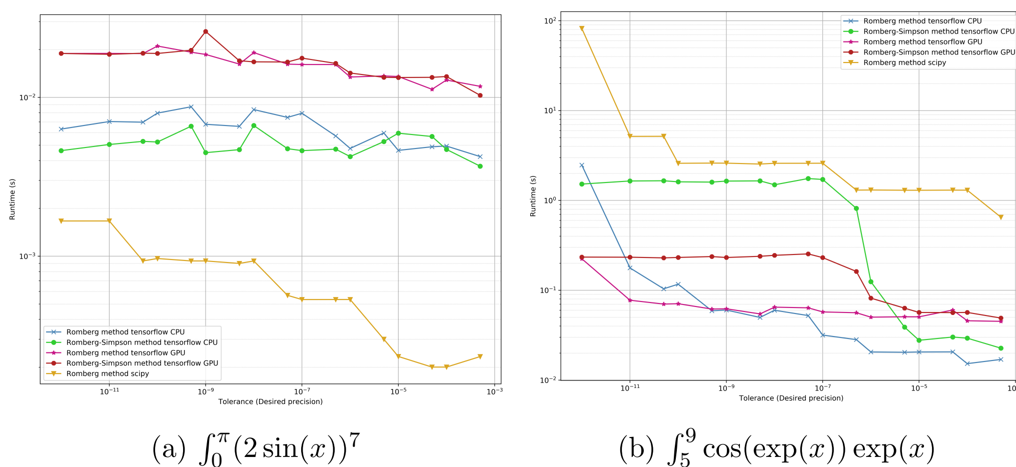
The comparison of Romberg methods yielded some compelling results, the error-tolerance matching of the Scipy implementation was worse than for the methods we implemented, but it almost exclusively obtained results that were too accurate, thus this is a minor downside at most.



(a) $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$ (b) $\int_{-0.5}^{-0.02} \frac{-1}{x^2} \cos(\frac{1}{x})$

Figure 3.16: Error vs tolerance for the Romberg methods

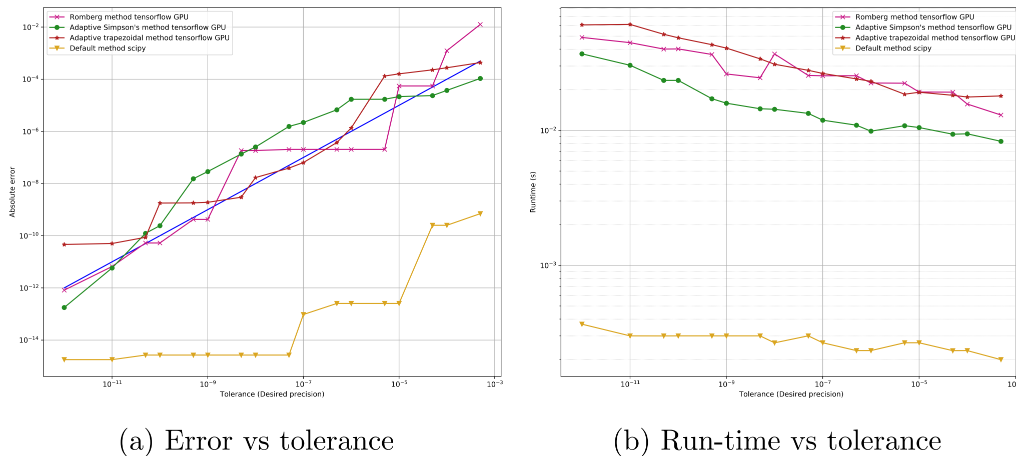
The refinement procedure seemed to be improved in the Scipy implementation as it required two or three fewer iterations to converge than our implementations, this led to an improved run-time whenever the maximum number of iterations was not needed. However, in the case of the third integral the method required all 25 iterations, which resulted in it requiring a far longer run-time than our implementations. This highlights an avenue for improving our algorithms.



(a) $\int_0^{\pi} (2\sin(x))^7$ (b) $\int_5^9 \cos(\exp(x))\exp(x)$

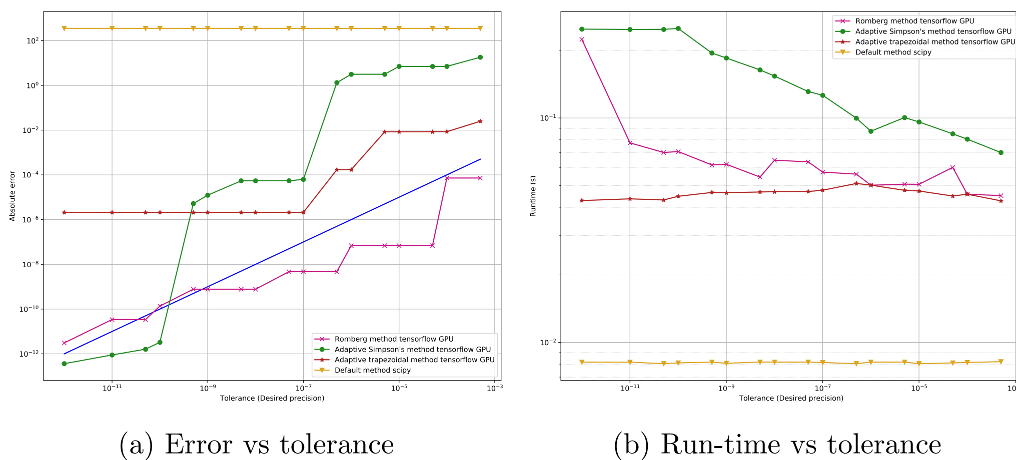
Figure 3.17: Run-time vs tolerance for the Romberg methods

The default Scipy integration method is incredibly quick, it easily matched or surpassed all of our methods whenever it could obtain a correct result within a few iterations. There are two caveats however, the first being that the method simply failed to achieve a reasonable result in some cases and the other being that it seemed far better adjusted to integrals of elementary functions as it slowed down considerably when applied to the approximate distribution shown in Figure 3.6.



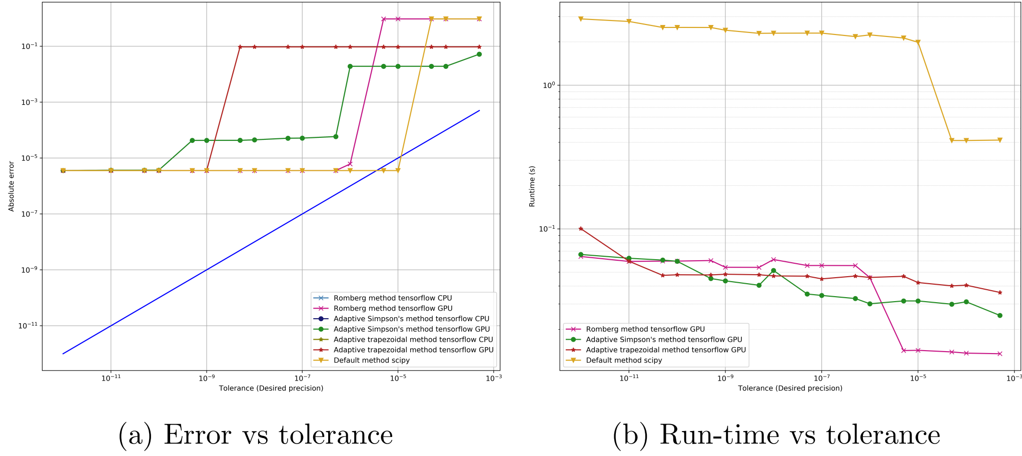
(a) Error vs tolerance (b) Run-time vs tolerance

Figure 3.18: Comparison between default Scipy method and the three fastest non-quadrature methods for $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$



(a) Error vs tolerance (b) Run-time vs tolerance

Figure 3.19: Comparison between default Scipy method and the three fastest non-quadrature methods for $\int_5^9 \cos(\exp(x)) \exp(x)$



(a) Error vs tolerance

(b) Run-time vs tolerance

Figure 3.20: Comparison between default Scipy method and the three fastest non-quadrature methods for $B \to K^{*0}\mu^+\mu^-$

The run-time for $\int_{0.01}^{1.99} \frac{(x+1)}{(x^3+x^2-6x)}$ and $\int_5^9 \cos(\exp(x))\exp(x)$ indicates that the default Scipy method utilizes a quadrature method when provided with an elementary input, which would explain its failure to obtain a reasonable result for the latter.

The poor error-tolerance matching observed for the default Scipy method is due to the library QUADPACK, the library supports a large number of points for quadrature rules, but the increase in the number of points between each rule is large, thus providing poor error-tolerance control.

When applied to a non-elementary function it seems like the QUADPACK routine **QAGS** is used, which is intended to be used when inefficient computing can be tolerated and when the problem at hand cannot be analyzed further, which explains the significant slowdown observed in Figure 3.20(b).

4. Conclusion and outlook

Based on the results presented in chapter 3 we can see that the implementations of the various integration algorithms using Tensorflow match or surpass the library *scipy.integrate* when applied to difficult integrals, which are common in HEP. These results indicate that Tensorflow may be ideal for building a powerful single variable integration library suited for HEP applications.

Methods depending on Newton-Cotes formulas experienced a large improvement in performance, particularly for large numbers of evaluations. These methods are typically constrained by the computing power available and the efficiency of the implementation, thus the improvement from Tensorflow at large numbers of evaluations, especially when running on GPU is very noticeable as shown in Figure 3.13.

It is difficult to identify one method as ideal when it comes to single variable integrals. Nonetheless we can draw the following conclusions based on the results from chapter 3:

- **Legendre-Gauss-quadrature** This method was quite successful at approximating simple functions as well as a few difficult ones, its run-time is among the lowest of any of the tested methods as shown in Figure 3.5, which makes it optimal when dealing with common integrals. When faced with pathological integrals[1] this method failed completely as shown in Figure 3.7.
- **Kronrod-Gauss-quadrature** This method is quite similar to the Legendre-Gauss-quadrature, they can achieve comparable precisions, although the Legendre-Gauss-quadrature has a slight edge when using a large number of evaluations as shown in Figure 2.7. The Kronrod-Gauss quadrature is only defined for an odd number of points, which results in worse error-tolerance matching when compared to the Legendre-Gauss-quadrature, it also ran marginally slower, making it an inferior option overall.
- **Romberg-Gauss-quadrature** Despite this method being unconventional and rather untested throughout history, it performed remarkably well in comparison to the Legendre-Gauss-quadrature upon which it is based. It was regularly able to achieve more precise results as in Figure 3.7, while only running marginally slower than the other quadrature methods as shown in Figure 3.5.

[1] Integrals possessing irregular features such as rapid oscillations or extremely sharp peaks

- **Adaptive trapezoidal method** Despite its simplicity, this method was quite potent. It was capable of achieving a sufficiently high accuracy for most applications regardless of the type of integral as shown in Figure 3.7. This method is ideal for double exponential decay functions and performs significantly better than expected for periodic functions, but it is generally outdone by similar methods.[4]

- **Adaptive Simpson's method** The development of this method was motivated by a desire to improve upon the adaptive trapezoidal method, in that regard it is undeniable that it is successful, regularly obtaining better accuracy as seen in Figure 3.7. Individual iterations of the Simpson's method are slower than those of the trapezoidal method Figure 3.13, but it requires fewer iterations to achieve a certain accuracy, resulting in a shorter run-time Figure 3.5.

- **Romberg method** This is perhaps the most well rounded of all the methods that have been tested. It is perfectly capable of achieving the desired accuracy for the majority of the tested integrals 3.7 and 3.4 while running at a reasonable speed 3.5.

- **Romberg-Simpson method** This method was intended to improve upon the Romberg method, but it falls behind both in terms of accuracy Figure 3.7 and run-time Figure 3.17. The slower run-time is explained through this method being unable to reuse points as the Simpson's rule evaluates are four points within an interval instead of two, but the worse accuracy is unexplained and could be worth exploring in detail. While this method was interesting to test, it is never ideal.

While none of the methods is always optimal , some methods outperformed the rest for specific integrals and tolerance ranges, the results are summarized in the following table.

Table 4.1: Optimal methods and processors for different types of integrals and tolerance

Type of integral	Desired tolerance	Optimal method	Optimal processor
Simple	$10^{-8} \leq$	Legendre-Gauss quadrature	CPU
Simple	$\leq 10^{-8}$	Romberg-Gauss quadrature	CPU
Difficult	$10^{-8} \leq$	Romberg-Gauss quadrature	CPU
Difficult	$\leq 10^{-8}$	Romberg method	CPU/GPU
Pathological	$10^{-8} \leq$	Romberg method	CPU
Pathological	$\leq 10^{-8}$	Romberg method	GPU

While the Newton-Cotes and Romberg methods enjoyed significant enhancements in performance due to Tensorflow's special features, the implementations are not optimized. The scipy implementation of the Romberg method could regularly best our implementation when the maximum number of iterations was not needed, which indicates that there is room for improvement.

The quadrature methods are the unlikely to see an improvement in terms of evaluation speed from Tensorflow. However, the determination of weights and abscissa is a computationally demanding task which could benefit significantly from Tensorflow, thereby allowing us to accurately determine more than 100 points. New algorithms for calculating the weights and abscissa have emerged which could aid with this process have emerged in the last decade.[5]

Supporting variable transformations that make the integrals easier for the methods to approximate is a direct path to improving efficiency. The implementation of such transformations would also allow for integration over infinite domains, which is not currently supported by the Tensorflow versions of the methods, thus it is a priority for future development.

An algorithm similar to `scipy.integrate.quad` which analyzes the integral to identify and apply the ideal method would be required to build a complete integration library, this is likely to be the end goal of the project once the methods are optimized.

A. Appendix

A.1 Code and testing devices

The code for each of the shown methods can be found at `https://github.com/M-AlMinawi/Integration-of-single-variable-functions-using-TensorFlow` along with the plots and *.csv* files containing the data used for the production of said plots for the following 10 integrals.

- $\int_{0.01}^{1.99} \frac{x+1}{x^3+x^2-6x}$
- $\int_0^{\pi} (2\sin(x))^7$
- $\int_5^9 \cos(\exp(x))\exp(x)$
- $\int_{-1}^{1} \frac{1}{(1+2500x^2)}$
- $\int_{-0.5}^{-0.02} -\frac{1}{x^2}\cos(\frac{1}{x})$
- $\int_{-1}^{1} x^2\cosh(x)\exp(\exp(x^2))\sinh(x)$
- $\int_0^{35} x^6\exp(-x)$
- $\int_{-0.04}^{0.04} 250\exp(-15000x^2)$
- $\int_1^7 \exp(\cos(\exp(x))) - x\exp(\cos(\exp(x)) + x)\sin(\exp(x))$
- $\int_{0.1}^{5} B \to K^{*0}\mu^+\mu^-$

Testing was performed using 4 devices, one of which used a graphics card that is supported by Tensorflow, thus it was used exclusively for GPU testing.

Table A.1: Testing devices

	CPU Device 1	CPU Device 2	CPU Device 3	GPU Device
CPU	Intel Core i7-7500U	Intel Core i7-4770	Intel Core i7-4770	AMD Ryzen Threadripper 1920x
GPU	AMD R5 M430	Nvidia GTX 770	AMD R9 390	Nvidia RTX 2070
RAM	8GB DDR3	8GB DDR3	16GB DDR3	16GB DDR4
Operating system	Windows 10	Windows 10	Windows 10	Windows 10
Python version	3.6	3.8	3.8	3.7
Tensorflow version	2.2.0	2.2.0	2.2.0	2.2.0

The difference in specifications across the CPU testing devices did not result in significant performance differences, with the results for devices 2 and 3 being similar to the results for device 1 that have been presented throughout the thesis.

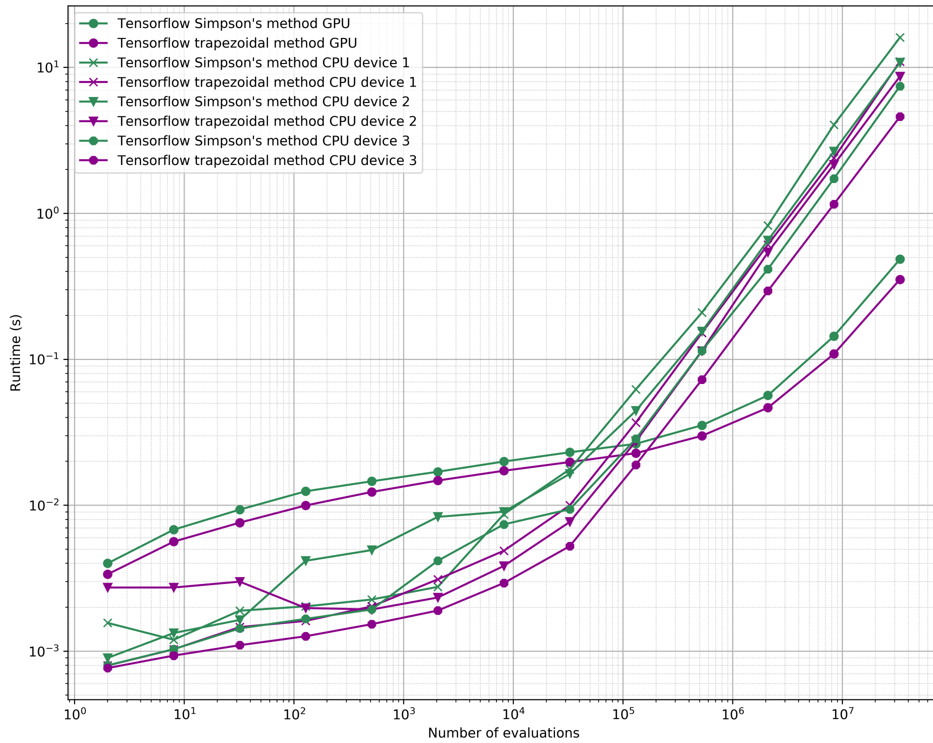


Figure A.1: Run-time vs Number of evaluations comparison of the testing devices for $\int_{4\pi/6}^{5\pi/6} \sin(x)\tan(x) + \frac{3}{x^2} + \ln(x^3)$

A.2 Implementation of adaptive methods

As mentioned in subsection 2.2.2 recursion is a powerful option for the implementation of these methods, below is an example of a recursive Simpson's rule implemented in Python.

```
import numpy as np

def simpsonarea(function,a,b):
    h = (b-a)
    area = (2*h*function((a+b)/2)+h/2*(function(a)+function(b)))/3
    return area

def adaptintsimp(function, a, b,max, tol,iter=1):
    h = (b - a)
    m = (b + a) / 2
    area = 0
    areatot = simpsonarea(function, a, b)
    nextareatot = simpsonarea(function, a, m) + simpsonarea(
function, m, b)
    err = np.abs(areatot - nextareatot)
    if iter<max:
        iter += 1
        if err < tol:
            return areatot
        else:
            arealeft = adaptintsimp(function, a, m, max,tol,iter)
            arearight = adaptintsimp(function, m, b, max,tol,iter)
            area = area + arealeft + arearight
        return area
    else:
        return areatot
```

The implementation is quite elegant, we simply check whether the difference between the results of two iterations is below our tolerance. If it is not, then we split the interval in half and run the function for each sub-interval until we obtain the desired result.

To implement a similar method in Tensorflow we need to convert to iteration, which makes using the difference between two iterations as a criterion for splitting inefficient. However, from Equation 2.1 we could see that the error is dependent on the derivative and by extension the absolute difference between the value of the function evaluated at two points.

The relation between the absolute error and the absolute difference is not known exactly, thus we have to determine suitable values through trial and error.

```
1  import tensorflow as tf
2
3  @tf.function(autograph=False)
4  def integrate(func, lower, upper):
5      return (func(lower) + 4*func((lower+upper)/2) +  func(upper)) /
       6 * (upper - lower), func(lower),func(upper)
6
7  @tf.function(autograph=False)
8  def diff_body(integral,increase,tol,lower,upper,plot_points,
     rejected_points,iterations):
9      integrals,low,up = integrate(integrand,lower,upper)
10     abs_diff = tf.abs(up-low)
11     abs_diff_bound = tf.cond(tf.math.greater(tol,tf.cast(1e-10,
     dtype=tf.float64))
12     ,lambda :tf.cond(tf.math.greater(tol,1e-6)
13      ,lambda: tf.abs((tf.cast(16, dtype=tf.float64) * tol**(1/4))),
       lambda : tf.abs(1250 * ((tol ** (1 / 2)))))
14     ,lambda :tf.abs(5000* ((tol ** (2 / 3)))))
15     too_big = tf.greater(abs_diff,abs_diff_bound)
16     points  = tf.where(too_big)[:, 0]
17     integral += tf.reduce_sum(tf.boolean_mask(integrals, mask=tf.
     logical_not(too_big)), axis=0)
18     increase = tf.reduce_sum(tf.boolean_mask(integrals, mask=tf.
     math.equal(too_big,True)), axis=0)
19     lower_to_redo = tf.gather(lower, points, axis=0)
20     upper_to_redo = tf.gather(upper, points, axis=0)
21     new_middle = (upper_to_redo + lower_to_redo) / 2
22     new_lower = tf.concat([lower_to_redo, new_middle], axis=0)
23     new_upper = tf.concat([new_middle, upper_to_redo], axis=0)
24     iterations += 1
25     return integral, increase,tol ,new_lower, new_upper,iterations
26 @tf.function(autograph=False)
```

The previous code is responsible for the evaluation process, it needs to be placed in a loop with a termination condition to form a complete algorithm.

```
def cond(integral,increase,tol,lower,upper,plot_points,
  rejected_points,iterations):
    cond = tf.cond(tf.equal(integral,tf.cast(0,dtype=tf.float64))
                   ,lambda: tf.constant(True)
                   ,lambda :tf.greater(tf.abs(increase),tol))
    return cond

@tf.function(autograph=False)
def adaptive_simpson_diff(l,u,tol,iter):
    initial_points = tf.linspace(tf.constant(l,dtype=tf.float64),
  tf.constant(u,dtype=tf.float64), num=75)
    result = tf.while_loop(cond=cond, body=diff_body, loop_vars=[tf
  .constant(0.,dtype=tf.float64),tf.constant(0, dtype=tf.float64),
  tf.constant(tol,dtype=tf.float64),initial_points[:-1],
  initial_points[1:],tf.constant(1,dtype=tf.float64) ],
    shape_invariants=[tf.TensorShape(()),
  tf.TensorShape(()), tf.TensorShape(()),
tf.TensorShape((None,)),tf.TensorShape((None,)),tf.TensorShape((
  None,)),tf.TensorShape((None,)),tf.TensorShape((None,))],
  maximum_iterations=iter)
    integral = result[0] + result[1]
    iterations = result[7]
    return integral,iterations
```

A.3 Improving error-tolerance matching

The precision of the error-tolerance matching is dependent on the termination criterion. Through evaluating the integral at different tolerance ranges and observing the agreement between the error and the tolerance, we can obtain weights that allow us to improve the matching.

Obtaining optimal matching in all cases is a time consuming task and was not one of our goals during the development of this project. An ad-hoc solution was implemented to allow for proper testing of the methods. Below is a comparison between the initial termination condition and the improved one.

```
@tf.function(autograph=False)
def romberg_cond(l,u,tol,a,min_iter):
    k = tf.shape(a)[0]
    cond = tf.cond(tf.less(tf.shape(a)[0],min_iter),
                   lambda: tf.constant(True),
                   lambda:tf.cond(tf.math.greater_equal(tf.shape(a)
  [0],25),
                                  lambda: tf.constant(False),
                                  lambda: tf.math.less(tol,tf.abs(
  tf.subtract(a[k-1,k-1],a[k-2,k-2])))
                                  ))

    return cond
```

```
12 @tf.function(autograph=False)
13 def romberg_cond(l,u,tol,a,min_iter):
14     k = tf.shape(a)[0]
15     cond = tf.cond(tf.less(tf.shape(a)[0],min_iter),
16                    lambda: tf.constant(True),
17                    lambda:tf.cond(tf.math.greater_equal(tf.shape(a)
    [0],25),
18                                   lambda: tf.constant(False),
19                                   lambda:tf.cond(tf.math.
    greater_equal(tol,1e-8),
20                                                  lambda:tf.cond(tf.
    math.greater_equal(tol,1e-6),
21
    lambda:tf.math.less(tol,tf.abs(tf.subtract(a[k-1,k-1],a[k-2,k
    -2])/200)),
22
    lambda: tf.math.less(tol,tf.abs(tf.subtract(a[k - 1, k - 1], a[
23
                               k - 2, k - 2]) ))),
24                                                  lambda: tf.math.
    less(tol,tf.abs(tf.subtract(a[k-1,k-1],a[k-2,k-2])/20))
25                                   )))
26
27     return cond
```

It is possible to substantially improve the criterion. The current approach is flawed when applied to integrals with small numerical values, thus adjusting the criterion to account for the size of the integral is a priority. Furthermore, the approach is only split at three tolerance values, namely $10^{-6}, 10^{-8}$ and 10^{-10}. Running tests at more tolerance values may allow for the derivation of a general condition, which would improve the error-tolerance matching and the run-time of the methods.

References

[1] J. N. Lyness. "An algorithm for Gauss-Romberg integration". *BIT Numerical Mathematics* 12 (1972), 194–203.

[2] W. H. Press et al. *Numerical recipes in C: the Art of Scientific Compuation.* Cambridge University Press, 1992.

[3] D. Calvetti et al. "Computation of Gauss-Kronrod quadrature rules". *Mathematics of Computation* 69.231 (2000), 1035–1052.

[4] M. Mori and M. Sugihara. "The double-exponential transformation in numerical analysis". *Journal of Computational and Applied Mathematics* 10 (2001), 287–296.

[5] N. Hale and A. Townsend. "Fast and accurate computation of Gauss-Legendre and Gauss-Jacobi quadrature nodes and weights". *SIAM Journal on Scientific Computing* 35.2 (2013), 652–674.

[6] M. Thomson. *Modern Particle Physics*. Cambridge University Press, 2013.

[7] A. Lazorenko. *TensorFlow performance test: CPU vs GPU*. 2017. URL: https://medium.com/@andriylazorenko/tensorflow-performance-test-cpu-vs-gpu-79fcd39170c.

[8] Jeff Miller. *Earliest Known Uses of Some of the Words of Mathematics.* 2017. URL: http://jeff560.tripod.com/mathword.html.

[9] W. Notling. *Theoretical Physics 6: Quantum Mechanics: Basics.* Springer International Publishing, 2017.

[10] J. White. *The advantages of GPU acceleration in computational finance.* 2017. URL: https://elsen.co/blog/the-advantages-of-gpu-acceleration-in-computational-finance/.

[11] *Better performance with tf.function.* 2020. URL: https://www.tensorflow.org/guide/function.

[12] C. Cornella et al. "Hunting for B+→K+τ+τ^- imprints on the B+→K+μ+μ^- dimuon spectrum" (2020).

[13] *Integration and ODEs (scipy.integrate).* 2020. URL: https://docs.scipy.org/doc/scipy/reference/integrate.html.

[14] *Nodes and Weights of Gauss-Kronrod Calculator*. 2020. URL: https://keisan.casio.com/exec/system/1289382036.

[15] *Nodes and Weights of Gauss-Legendre Calculator*. 2020. URL: https://keisan.casio.com/exec/system/1280624821.

[16] *numpy.polynomial.legendre.leggauss*. 2020. URL: https://numpy.org/doc/stable/reference/generated/numpy.polynomial.legendre.leggauss.html.

[17] E. W. Weisstein. *First Fundamental Theorem of Calculus*. 2020. URL: https://mathworld.wolfram.com/FirstFundamentalTheoremofCalculus.html.

[18] E. W. Weisstein. *Fundamental Theorem of Gaussian Quadrature*. 2020. URL: https://mathworld.wolfram.com/FundamentalTheoremofGaussianQuadrature.html.