An Exploration of Numerical Integration Methods for Use in Particle Physics

Emir Muhammad

University of Birmingham, Birmingham, UK

January 24, 2020

Abstract

Efficient algorithms capable of performing numeric multidimensional integrals are highly sought after in Particle Physics. The project aimed to investigate numerical integration methods by implementing quadrature-based methods, Monte Carlo, and recursive stratified sampling-based Monte Carlo methods. Each was implemented in Python and characterized by evaluating well known integrals in one and four dimensions. The results show that adaptive quadrature was best suited for one dimensional integrals, however recursive stratified sampling outperformed other methods in four dimensions due to its fast execution speeds and stable convergence.

1 Introduction

Determining the cross-section of a particular interaction is desirable in multiple particle physics experiments, as the cross-section is an indication of the probability for that particular interaction occurring. Calculating the cross-section of a two-to-two process, a common example being electron scattering, requires the integration over two variables; in general, a two-to-n process requires integrating over 3n-4 variables. As typical LHC events produce hundreds of particles, evaluating the cross-section would involve integrating hundreds of variables [1]. This is an impossible task to perform analytically, thus numerical integration methods are relied upon to perform the calculation.

Several numerical integration methods are available that could be used for this task, each performing differently and with unique characteristics. Traditional quadrature-based methods such as the trapezium rule are simple to implement and are deterministic in nature, allowing for a fast convergence as a function of sampling points. Non-deterministic methods, such as Monte Carlo integration, relies on sampling the function randomly and would therefore perform differently to quadrature methods [2]. Each method could be enhanced by adapting which points are sampled to provide faster convergence. The MISER algorithm [3] and the VEGAS algorithm [4] are two examples of adaptive Monte Carlo integration, where the random points distribution is adapted over several iterations to converge faster in comparison to using a uniform distribution. Generalizing and implementing these methods in multiple dimensions would alter the characteristics of each method.

The project aimed to investigate and characterize multiple numerical integration methods and determine which would be best suited for use in particle physics. The integration methods investigated were the Newton-Cotes quadratures up to the second-degree polynomial, adaptive integration using Newton-Cotes, Monte Carlo integration, and its recursive stratified sampling-based counterpart. The main objectives were to implement these functions in Python, generalize them to multiple dimensions, and characterize its performance by demonstrating convergence and evaluating its execution speed.

2 Physics Review

The Newton-Cotes quadrature rules are a set of rules used to calculate the integral by numerically interpolating the integrand as a polynomial through equally spaced intervals. The simplest of such rules is the midpoint rule, where the function is approximated as a 0th order polynomial evaluated at the midpoint of the limits a and b, and is mathematically expressed as

$$\int_a^b f(x)dx \approx (b-a)f(\frac{(a+b)}{2}). \tag{1}$$

The function could also be approximated as a 1st order polynomial, in which case the quadrature is based on determining the area of a trapezium. Utilizing this approximation is known as the trapezium rule, and is expressed as

$$\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a)+f(b)). \tag{2}$$

Simpsons rule uses a 2nd order polynomial to approximate the function, theoretically increasing the accuracy of the value and increasing the speed of convergence. The rule is expressed as

$$\int_a^b f(x)dx \approx \frac{b-a}{6}(f(a)+4f(\frac{a+b}{2})+f(b)). \tag{3}$$

Higher-order polynomial approximations are available and should theoretically increase the accuracy of the evaluation, however, are rarely used due to the possibility of the approximation suffering from Runge's Phenomenon [5]. When approximating certain functions such as the Runge Function, the approximation would fluctuate wildly near the limits as the degree of the polynomial increases, as shown in Fig. 1. For numerical integration, the fluctuations increase the error of the estimate and would be undesirable. Runge's Phenomenon shows that using higher-order polynomials are not always advantageous; therefore, only Newton-Cotes quadrature rules up to Simpsons rule were analysed for this project.

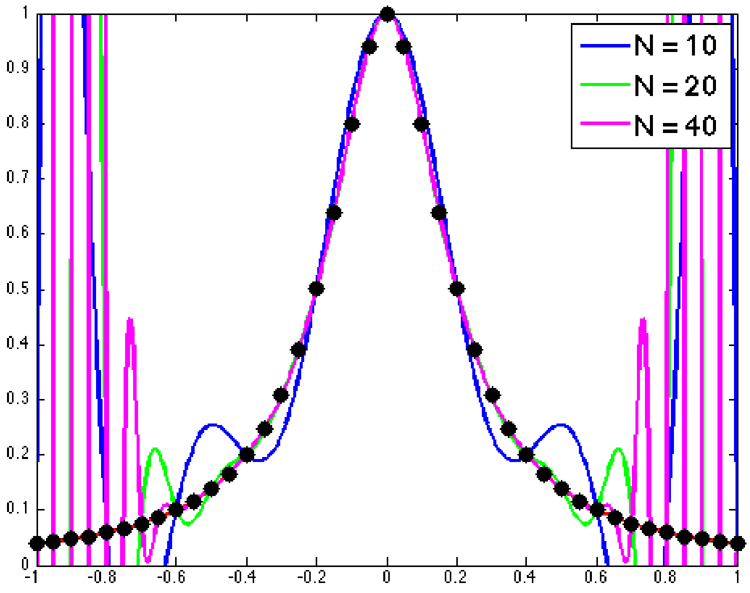


Figure 1: Shows the Runge Function, with interpolating polynomials of different order used to approximate it. [6]

The Newton-Cotes methods could provide a better estimate by dividing the integral to a series of equally spaced intervals and applying the Newton-Cotes rule at each division. Generalizing this process to n sampling points with each method leads to the Newton-Cotes formula,

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

$$x_i = hi + a = \frac{b-a}{n}i + a, \tag{4}$$

where w_i are the weights used for each point and h is the distance between two points. The weights can be derived from the previous definitions of each rule and are shown in Table 1 [7].

Table 1: List of characteristics associated with each quadrature method. M is the actual value of the integral, n is the number of sample points used, and N is the number of dimensions.

Quadrature Method	Weights	Error Bounds (1-D)	Error bounds (N-D)
Midpoint Rule	[0, h, h, ..., h]	$\frac{M(b-a)^3}{24n^2}$	$O(1/n^{\frac{2}{N}})$
Trapezium Rule	$[\frac{h}{2}$, h, h, ..., $\frac{h}{2}]$	$\frac{M(b-a)^3}{12n^2}$	$O(1/n^{\frac{2}{N}})$
Simpsons Rule	[h/3, 4h/3, 2h/3, 4h/3, ..., h/3]	$\frac{M(b-a)^5}{180n^4}$	$O(1/n^{\frac{4}{N}})$

The convergence of each method is mathematically well known and is shown in Table 1 [8]. Due to the higher-order approximation of Simpsons rule, it naturally converges the fastest at $O(1/n^4)$. As the midpoint rule and trapezium rule approximates the function as straight lines, the convergence of the two are of the same order $O(1/n^2)$. Counter-intuitively, the midpoint rule has a better bound compared to the trapezium rule, arising from how the trapezium rule systematically overestimates the value of the integral when the function has a positive curvature and underestimates the value at negative curvature. An illustration of this is found in Appendix 6.1.

The convergence of the Newton-Cotes quadrature rules could be improved using adaptive quadrature by systematically choosing the appropriate intervals to integrate over rather than equally spaced intervals. This process is done by first approximating the value of one division using a Newton-Cotes rule, after which the division is subdivided into two. The value of each subdivision is evaluated using the same Newton-Cotes rule. If the difference between the value of the division and the sum of the two subdivisions are less than an error tolerance, then the division stops. Else, the two subdivisions are divided again, and the process repeats until the error tolerance condition is met. The selective choosing of intervals allows for faster convergence and better execution speed, as it would only perform a function calculation at the most relevant sample points.

Newton-Cotes quadratures could be generalized to N dimensions by applying the quadrature rule in each dimension throughout all sampling points. The sampling points are found from the Cartesian product of the sampling points required in each dimension, illustrated in two dimensions in Fig. 2. Summing over all sampling points lead to the general equation

$$\int_{\Omega} f(\mathbf{x})d\mathbf{x} \approx \sum_{i_N=0}^{n_N} \cdots \sum_{i_1=0}^{n_1} w^{[i_1,i_2,\cdots i_N]} f(\mathbf{x}^{[i_1,i_2,\cdots i_N]})$$
$$w^{[i_1,i_2,\cdots i_N]} = w_1^{[i_1]} w_2^{[i_2]} \cdots w_N^{[i_N]}. \tag{5}$$

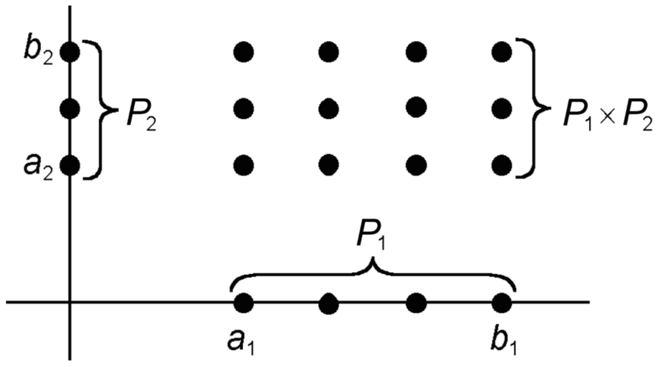


Figure 2: A two dimensional example of sampling in multiple dimensions. $P1 \times P2$ are the Cartesian products of sampling points P1 and P2 [7].

A problem develops when using Newton-Cotes quadratures in multiple dimensions. In Fig. 2, 12 points were used for the sampling points, resulting from the Cartesian product of 3 and 4 points. Analysing Equation 5 reveals that the more dimensions there are, the more summations are required, resulting in a large increase in the total number of sampling points used. This would lead to an exponential increase in the algorithm's run time, leading to a substantial decrease in performance. The error bounds for each Newton-Cotes method is shown in Table 1.

Monte Carlo integration is another numerical integration method, utilising random numbers to sample the integrand within the limits to approximate the value. Monte Carlo takes advantage of the definition of the functions average over an interval, using it to evaluate the integral as

$$\int_\Omega f(\mathbf{x})d\mathbf{x} \approx \sum_{i_m=0}^{n_m} \cdots \sum_{i_1=0}^{n_1} w^{[i_1,i_2,\cdots i_m]} f(\mathbf{x}^{[i_1,i_2,\cdots i_m]})$$

$$\int_a^b f(x)dx \approx \langle f(x) \rangle (b-a) = \frac{1}{N}\sum_{n=1}^{N} f(x_n).$$

The method is easily generalisable to multiple dimensions, whereby the equation is given as

$$\int_\Omega f(\mathbf{x})d\mathbf{x} = \int_\Omega f(u_1, u_2, \cdots, u_d)d^d u \approx \frac{1}{N}\sum_{n=1}^{N} f(\mathbf{x}_n)$$

The convergence of Monte Carlo scales with $O(\frac{1}{\sqrt{N}})$ and is dimensionally invariant [1]. This property is Monte Carlo's most important characteristic, as it shows how the method compares against Newton-Cotes in one dimension and multiple dimensions. Monte Carlo is expected to converge slowly in one dimension in comparison with Newton-Cotes, however, it is expected to surpass Newton-Cotes methods in multiple dimensions.

The convergence of Monte Carlo methods can be improved by implementing variance reduction techniques to increase convergence. One such method is the recursive stratified sampling technique, which the MISER algorithm is based on and is not unlike adaptive quadrature [3]. The integral is divided into several bins, and each bin is sampled using Monte Carlo with a few points, to determine its value as well as its variance. The variance is calculated as

$$\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2.$$

If the variance in a particular bin is larger than a tolerance, the bin is subdivided into two smaller bins and the process is repeated. Similarly, to adaptive quadrature, this method selectively chooses the intervals to use more sampling points at areas where the variance is largest. This not only increases the convergence of the integral by using only the required number of sampling points, but it also increases the execution speed of the algorithm as well.

3 Algorithm and Code Structure

Every integration method was applied in the Integrator class, with each integration method defined in an individual method. Each of the three Newton-Cotes methods was implemented in individual methods, following the definitions provided in Equations 1, 2, and 3, after which they were generalized to multiple sampling points using Equation 4. The performance of individual Newton-Cotes rules and the generalised rule were verified against each other.

Adaptive quadrature required the use of recursion due to the recursive nature of the method as was described before. Pseudocode was provided below to give an overview of how the method was implemented using recursion, where the method calls itself if a certain condition was met. The tolerance was divided in two during the subdivision process to keep the cumulative error of the integral below the tolerance specified by the user.

```
def AI(f, a, b, tolerance):
        Midpoint = (a+b)/2
        Value = NCIntegrate(f, a, b)
        DivValue = NCIntegrate(f, a, midpoint) + NCIntegrate(f, midpoint, b)

        If |Value - DivValue| > tolerance:
                Value = AI(f, a, midpoint, tolerance/2) +
                        AI(f, midpoint, b, tolerance/2)
        return value
```

The implementation of the Monte Carlo method followed the mathematical description provided in Section 2. Producing the sampling points for Monte Carlo required the use of a pseudo-random number generator, which was provided from the *random* module imported into python. A seed was used to initialize the generator to reproduce consistent results as well as debugging purposes.

The algorithm for recursive stratified sampling was relatively complex compared to previous methods, as the interval must be subdivided into two and then evaluated using Monte Carlo. A new inner class RSSBin was created to handle the required information and methods the algorithm required, including performing Monte Carlo, storing the variance, and the subdivision process. This allowed the algorithm to look through all the available bins and find the bin with the largest variance, which would be subdivided first. The new inner class also allowed the implementation of a maximum iteration condition, which terminates the algorithm once the limit was met.

Generalizing each method to N dimensions was achieved by converting the upper and lower limits from accepting floats to accepting arrays with N elements, with each element containing the appropriate limit for one dimension. The Newton-Cotes quadrature rule required implementing another function that generates all the associated Cartesian products required in Equation 6 for indexing the sampling points and the weights, while also generalisable to N dimensions. This required the use of recursion to iterate over all the required for loops throughout all sampling points.

The main challenge in generalizing adaptive quadrature to multiple dimensions was handling the division of the interval into smaller intervals. A simple method to approach this was to divide the interval in half in every dimension, which required the use of the Cartesian product generator created for Newton-Cotes. The error condition was similar to the one-dimensional version, in that the difference between the initial estimate and the sum of the subdivisions must be smaller than the error tolerance for it to terminate.

The Monte Carlo and stratified sampling algorithms were relatively straightforward to generalize, as the algorithm structure is identical to the one-dimensional counterpart. However, implementing stratified sampling in multiple dimension required implementing the subdivision of the bins in multiple dimensions, which has been solved during the generalization of the adaptive quadrature method.

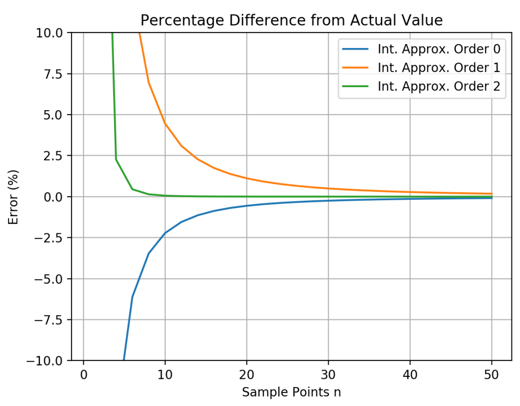
Once all integration methods were implemented, a new class was written that contained methods used to test each method. The methods would iterate over the number of sampling points to characterise is convergence and execution speeds. The Python module *time* was used to perform timing tests, and *matplotlib* was used to display the results in a series of graphs. The functions used for the testing, as well as its associated limits and values, are shown in Table 2. The finished project code is found in Appendix 6.2.

Table 2: List of functions used to test the integration methods

Function Name	Function	Limits of Integration	Value (approx.)
F5	$F_5(x) = x^5 - x^3 + x^2 + x$	[-2, 6]	7536
Gaussian	$G(x) = e^{-x^2}$	[-10, 10]	1.7724538509
SPh	$S(x_1, x_2, x_3, x_4) = 1 - \Theta \sum_{i=1}^{4} x_i^2$	[(-1, -1, -1, -1),(1, 1, 1, 1)]	4.9348022005
Circ	$C(x_1, x_2, x_3, x_4) = \sum_{i=1}^{4} x_i^2$	[(-1, -1, -1, -1),(1, 1, 1, 1)]	21.3333333333

4 Results

Newton-Cotes quadature rules converged quickly in well-behaved polynomials, as shown in Fig. 3a. As expected, Simpsons rule converged fastest due to its higher-order approximation, while both midpoint rule and trapezium rule converges at the same rate. However, there were some cases where Simpsons rule converged slower compared to the others. Integrating the Gaussian function in Fig. 4b shows that Simpsons rule was prone to fluctuations at low numbers of sampling points. Analysing Simpsons rule in Fig. 4a further reveals that the fluctuations were still within theoretical limits described in Table 1, confirming that Simpsons rule converges the fastest in general. The execution speeds and behaviour of all three were similar, which was expected due to the algorithm performing the same calculations with different numerical weights. The general trend of the execution speeds shown in Fig. 3b shows that the speeds have a direct proportionality with the number of sampling points used; more sampling points mean that the method took longer to complete. There were some unexplained spikes in the timing graphs, which could be explained from random processes in the computer's CPU.



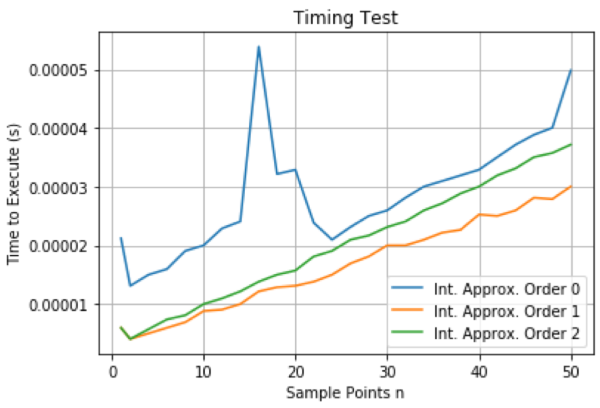
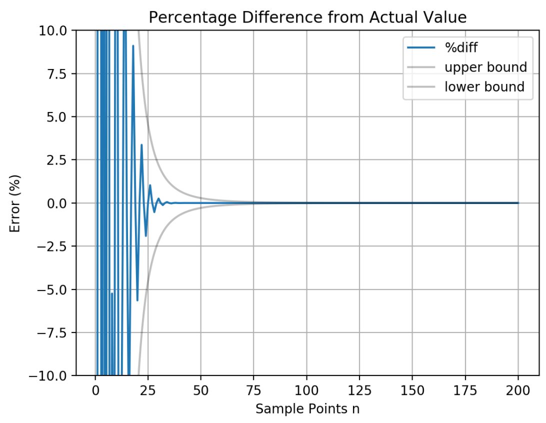


Figure 3: a) Integrating F5 shows the relative error vary for the three different quadrature methods as a function of sample points b) shows the timing performance for the different quadrature methods



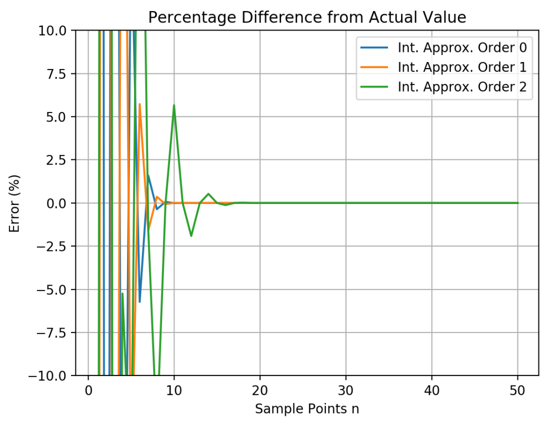


Figure 4: a) Integrating the Gaussian shows the relative error vary for Simpsons rule, while also showing the theoretical convergence limits as discussed in Section 2 b) Integrating the Gaussian shows the relative error vary for each of the three quadrature rules

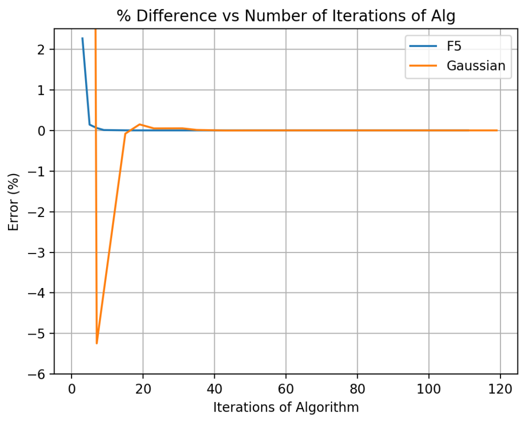


Figure 5: Integrating both F5 and the Gaussian shows how the relative error varies for adaptive quadrature as a function of iterations.

Adaptive quadrature was an improvement over the previous method as it converges quickly both in analytical and non-analytical functions. There were also no wild fluctuations as the value converges to the actual value, as every additional iteration brought the estimate closer to the actual value. Furthermore, less than 20 iterations were required for it to converge to less than 1% error, shown in Fig. 5. Adaptive quadrature exhibited similar timing characteristics as the other Newton-Cotes methods as seen in Appendix 6.3, however performed better due to its fast and stable convergence.

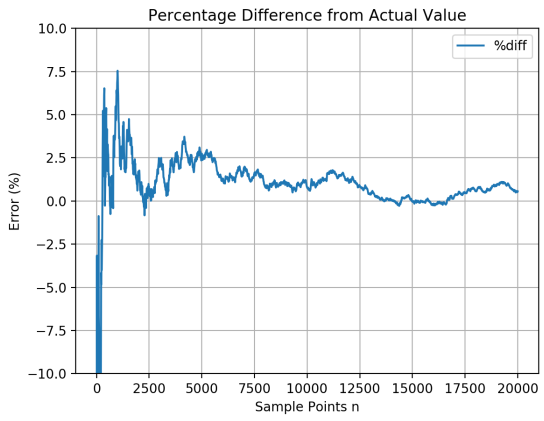


Figure 6: Integrating F5 shows the percentage error vary for Monte Carlo as a function of sampling points

Monte Carlos execution speeds per sampling point were similar to NC quadrature methods, as seen in Appendix 6.4, however Fig. 6 shows that Monte Carlo took much longer to converge. As discussed before, Monte Carlo converges as $O(1/\sqrt{N})$, requiring more sampling points to achieve an accurate result. A slight advantage of Monte Carlo was that only a few sampling points were required to get a decent estimate of the integral, however more was required if the desired estimate needed a higher accuracy. The estimates tend to fluctuate largely at low numbers of sampling points, however the values stabilize at large numbers of sampling points. These undesirable qualities show that Monte Carlo methods are not well suited for one-dimensional integration.

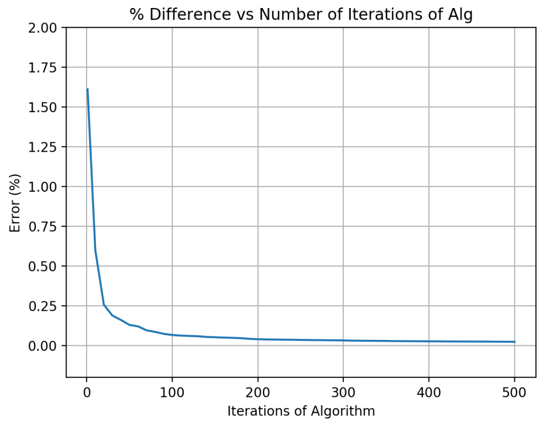
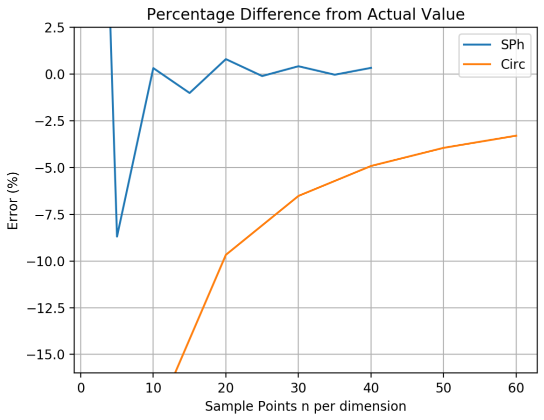


Figure 7: Integrating F5 shows the relative error vary for recursive stratified sampling as a function of sampling points

Recursive Stratified Sampling was a large improvement over Monte Carlo method as Fig. 7 shows the estimate converging quickly and in a stable manner. Similarly to adaptive quadrature, each additional iteration reduced the error of the estimate. However, due to still relying on Monte Carlo methods, a larger amount of iterations was required to converge, thereby increasing the execution times as seen in Appendix 6.5.



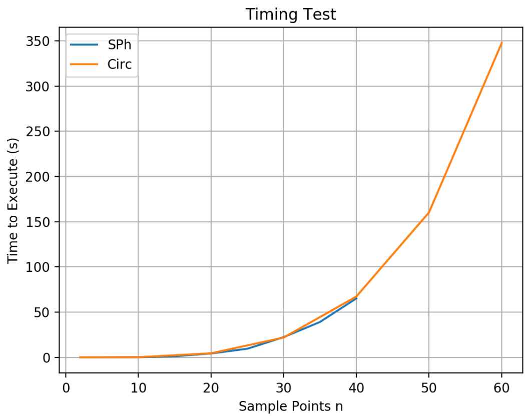


Figure 8: a) Shows the relative error vary for Newton-Cotes in 4 dimensions as a function of sampling points b) Shows the timing characteristics of Newton-Cotes in 4D

Applying Newton-Cotes quadrature rules in four dimensions lead to poor performance. The quadrature rule required more sampling points in multiple dimensions for convergence, and each additional sampling point added to the function increases execution speed exponentially. Furthermore, integrating some functions could still lead to fluctuation in its convergence, as seen in Fig. 8a. The most notable characteristic of this method was the execution speed exponentially increasing as a function of sample points per dimension, clearly shown in Fig. 8b. Extrapolating to hundreds of dimensions, Newton-Cotes quadratures would perform ineffectively due to the aforementioned characteristics.

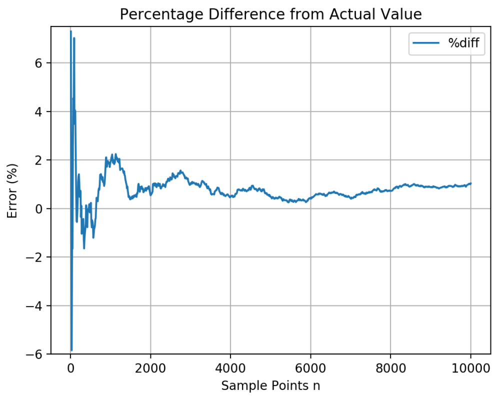


Figure 9: Integrating Circ shows the percentage difference varying for Monte Carlo in 4D

On the other hand, Monte Carlo's performance was relatively similar in four dimensions compared to one dimension. The convergence rate remained of the order $O(1/\sqrt{N})$, which was now favorable over quadrature methods as it converges much faster than Newton-Cotes as seen in Fig. 9. Its execution speed remains similar as seen in Appendix 6.6, allowing it to estimate the integral with relative ease. Furthermore, Monte Carlo only requires relatively few sampling points before a decent estimate is made, which is an advantage over quadrature methods especially in multiple dimensions.

The multidimensional form of recursive stratified sampling has a much faster convergence compared to Monte Carlo as seen in Fig. 10a. Similarly, with before, the algorithm converges quickly and in a stable manner. A few iterations of the algorithm returned a value with less than 0.2% error, which was highly desirable. While its execution speed was much slower than Monte Carlo as seen in Fig. 10b, the fast convergence and accurate result makes it the ideal method for multidimensional integration.

Unfortunately, the implementation of adaptive quadrature in multiple dimensions was not successful. The algorithm seems to converge to an incorrect value of about -50% from the actual values as seen in Appendix 6.7. However, converging at -50% implies that there is a bug in the code causing the algorithm to systematically underestimate the value, but due to time constraints this bug could not be fixed in time.

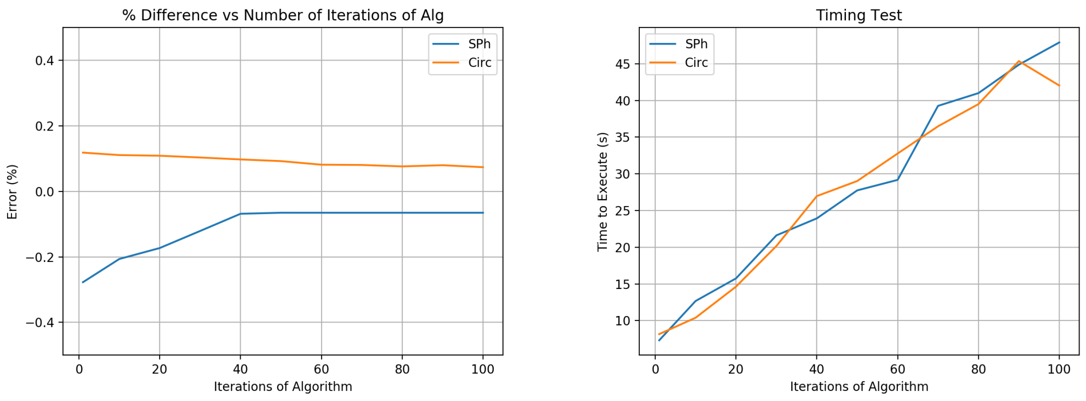


Figure 10: a) Shows the percentage difference varying for Stratified Sampling in 4D b) shows the timing test for stratified sampling in 4D

It was noted that most of the N dimensional methods suffered from a loss of execution speeds, particularly those that required the code used to generate the Cartesian products. The Cartesian product generator could be a computational bottleneck within the code, which could be solved by using an external package or by making the code more efficient. However, due to time constraints, these changes could not be made.

5 Conclusion

In conclusion, the project was a success. The main objectives of implementing Newton-Cotes quadrature and Monte Carlo methods in Python were completed and were successfully characterized using multiple functions. It was shown that Newton-Cotes quadrature methods, especially Simpsons rule and adaptive quadrature, had superior performance compared to Monte Carlo methods in one dimension due to its fast execution speeds and convergence. However, in multiple dimensions, Monte Carlo based methods performed better due to having consistent characteristics regardless of the dimensionality. Recursive stratified sampling provided a large enhancement to Monte Carlo, resulting in a fast yet stable convergence as the number of iterations increased. Based on these findings, Monte Carlo integration methods are recommended for use in particle physics.

However, there were still factors in the project that could be improved. A maximum iteration condition was not implemented in the one-dimensional adaptive quadrature method, as there was a lack of time due to more effort spent in polishing the stratified sampling algorithm. Furthermore, the adaptive quadrature method in multiple dimensions seemed to not work properly and could not be fixed in time due to focus on Monte Carlo methods in higher dimensions. The stratified sampling code, while accurate, was still highly inefficient. Proper variance reduction algorithms such as VEGAS and MISER subdivide the grid into only a particular dimension where the variance reduction would be greatest, unlike this projects implementation of subdividing evenly into all dimensions. The more efficient subdivision has not been implemented yet due to lack of time. Lastly, the analysis of the convergence of Monte Carlo was incomplete due to time constraints preventing the implementation of plotting the theoretical error bounds for this method.

Recommendations for future work would be to perform a similar comparison study between multiple variance reduction techniques in Monte Carlo. This project has established that Monte Carlo methods were ideal for multidimensional integrals, however, only one variance reduction technique was studied. Recommended methods for comparison study include recursive stratified sampling methods, importance sampling, and Latin Hypercubes.

References

[1] Sj ostrand, T. (2020). Monte Carlo Event Generation for LHC. Retrieved 20 January 2020, from `http://home.thep.lu.se/~torbjorn/preprints/th6275.pdf`

[2] Weinzierl, S. (2020). Introduction to Monte Carlo methods. Retrieved from `https://arxiv.org/pdf/hep-ph/0006269.pdf`

[3] Press, W., & Farrar, G. (1990). Recursive Stratified Sampling for Multidimensional Monte Carlo Integration. Computers In Physics, 4(2), 190. doi: 10.1063/1.4822899

[4] Lepage, G. (1980). VEGAS- An Adaptive Multi-dimensional Integration Program, (CLNS-447). Retrieved from `http://cds.cern.ch/record/123074/files/clns-447.pdf`

[5] Epperson, J. (1987). On the Runge Example. The American Mathematical Monthly, 94(4), 329-341. doi: 10.1080/00029890.1987.12000642

[6] The Runge phenomenon. (2020). Retrieved 21 January 2020, from `https://math.boisestate.edu/~calhoun/teaching/matlab-tutorials/lab_11/html/lab_11.html`

[7] Holton, G. Numerical Integration: Multiple Dimensions. Retrieved 20 January 2020, from `https://www.value-at-risk.net/numerical-integration-multiple-dimensions/`

[8] Strang, G., Herman, E., Seeburger, P. (2019). 2.5: Numerical Integration - Midpoint, Trapezoid, Simpson's rule. Retrieved 20 January 2020, from `https://math.libretexts.org/Courses/Mount_Royal_University/MATH_2200%3A_Calculus_for_Scientists_II/2%3A_Techniques_of_Integration/2.5%3A_Numerical_Integration_-_Midpoint%2C_Trapezoid%2C_Simpson%27s_rule`

6 Appendices

6.1 Error Bound Difference Between Midpoint and Trapezium Rule

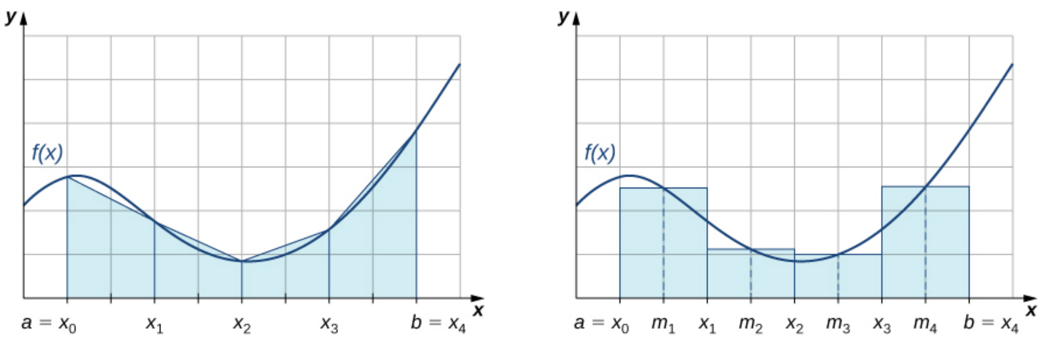


Figure 11: Shows the application of the trapezium rule and midpoint rule, respectively, over an interval.

Note that in the trapezium rule, the approximation systematically overestimates the function when the curvature is positive and underestimates it when the curvature is negative. On the other hand, the midpoint rule both overestimates and underestimates the function over the same interval, as seen between x_0 and x_1, and the errors would tend to average out.

6.2 Project Code

The finished project code is found at GitHub, available at the following link. Documentation was provided detailing the identity and use of the Python file as well as the Jupyter Notebook file.
`https://github.com/xlr91/Integration-Methods-PP`

6.3 Timing test for 1D Adaptive Quadrature

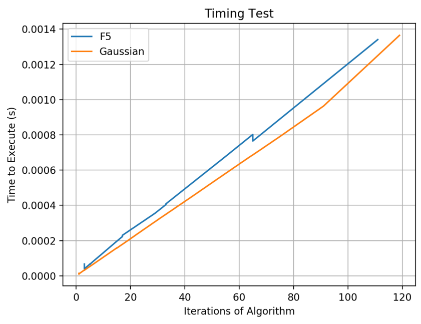


Figure 12: Shows the timing characteristics for adaptive quadrature as a function of algorithm iterations.

6.4 Timing test for Monte Carlo in 1D

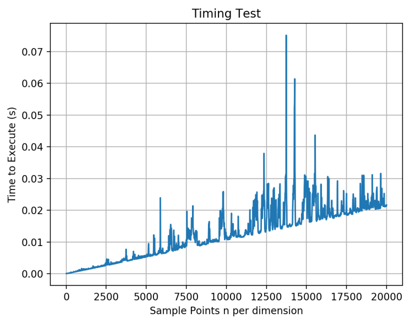


Figure 13: Shows the timing characteristics of Monte Carlo in 1D as a function of number of random sampling points used.

6.5 Timing test for Recursive Stratified Sampling

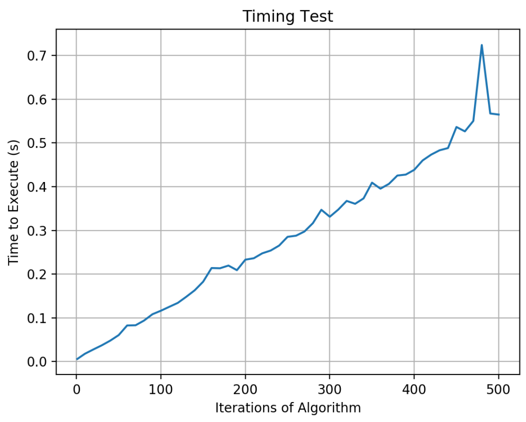


Figure 14: Shows the timing characteristics of recursive stratified sampling in 1D as a function of algorithm iterations.

6.6 Timing test for Monte Carlo in 4D

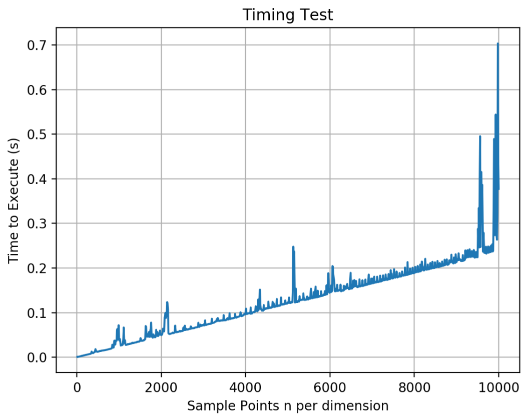


Figure 15: Shows the timing characteristics of Monte Carlo in 4D as a function of number of random sampling points used.

6.7 Performance of Adaptive Quadrature in 4D

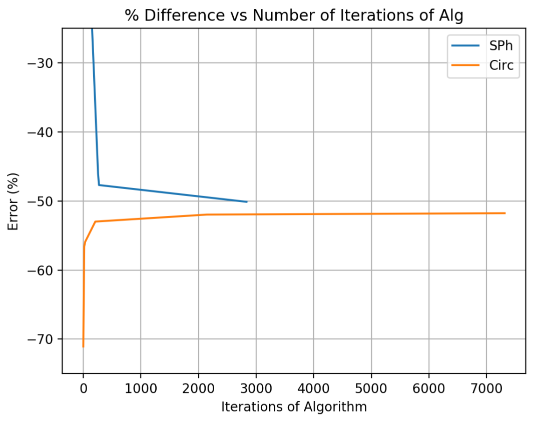


Figure 16: Shows the percentage difference varying for adaptive quadrature in 4D.