Abstract

Efficient algorithms capable of performing numeric multidimensional integrals are highly sought after in Particle Physics. The project aimed to implement quadrature-based methods, Monte Carlo, and recursive stratified sampling-based Monte Carlo and characterize each method. This was done using Python, after which they were investigated 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 the others in four dimensions due to its fast execution and stable convergence.

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 hundreds of variables (**1** REFERENCE). 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 its characteristics. Traditional quadrature-based methods such as the midpoint rule are simple to implement and are deterministic in nature, allowing for a fast convergence as a function of sampling points. Random number-based methods, such as Monte Carlo integration, relies on random sampling and would perform differently to quadrature methods (Reference). Each method could be enhanced by adapting which points are sampled to provide faster convergence. The MISER algorithm (REFERENCE) and the VEGAS algorithm (REFERENCE) are two examples of adaptive Monte Carlo integration, where the random number 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 quadrates 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.

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

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. This rule is the trapezium rule and is expressed as

Simpson’s 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

.

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 (reference). 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 FIGURE SOMETHING. Runge’s Phenomenon shows that using higher-order polynomials are not always advantageous; therefore, only Newton-Cotes quadrature rules up to Simpson’s rule were analysed for this project.

FIGURE SOMETHING

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 each method leads to the Newton-Cotes formula,

where wi are the weights used for each point. The weights can be derived from the previous definitions of each rule and are shown in table something (REFERENCE).

TABLE SOMETHING List of weights associated with each quadrature method

|  |  |
| --- | --- |
| Quadrature Method | Weights |
| Midpoint Rule | [0, h, h, …, h] |
| Trapezium Rule | [h/2, h, h, …, h/2 |
| Simpsons Rule | [h/3, 4h/3, 2h/3, 4h/3, 2h/3, … , h/3] |

The convergence of each method is well known and are shown in TABLE SOMETHING. Due to the higher-order approximation of Simpson’s 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). Counterintuitively, 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 SOMETHING

TABLE SOMETHING. Absolute error scaling for each quadrature method. M is the actual value of the integral, and n is the number of sampling points used.

|  |  |
| --- | --- |
| Quadrature Method | Error Scales as |
| Midpoint Rule | M(b-a)^3 / 24n^2 |
| Trapezium Rule | M(b-a)^3 / 12n^2 |
| Simpsons Rule | M(b-a)^5/180n^4 |

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 at 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 the calculations 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. Summing over all sampling points lead to the general equation

A close up of a logo

Description automatically generated

.

The two-dimensional case is illustrated in FIGURE SOMETHING, where the sampling points shown are the cartesian product of the sampling points P1xP2.

A close up of a logo

Description automatically generated

FIGURE SOMETHING. A two dimensional example of sampling in multiple dimensions P1xP2 are the cartesian products of sampling points P1 and P2.

A problem develops when using Newton-Cotes quadratures in multiple dimensions. In FIGURE SOMETHING, 12 points were used for the sampling points, resulting from the Cartesian product of 3 and 4 points. The number of total sampling points used for quadrature would increase exponentially as the dimensions increase, decreasing the performance of the method. The error scaling for each method is shown in TABLE SOMETHING.

TABLE SOMETHING. Error scaling for each quadrature method as a function of dimension d

|  |  |
| --- | --- |
| Quadrature Method | Error Scales as |
| Midpoint Rule | O(1/n^(2/d)) |
| Trapezium Rule | O(1/n^(2/d)) |
| Simpsons Rule | O(1/n^(4/d)) |

Monte Carlo Integration is another numerical integration method, utilising random numbers to sample the integrand within the limits to determine its actual value. Monte Carlo takes advantage of the definition of the function’s average over an interval, using it to evaluate the integral as

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

The convergence of Monte Carlo scales with O(1/√N) and is dimensionally invariant (REFERENCE). This property is Monte Carlo’s most important characteristic, as it shows how the method compares against Newton-Cotes in one dimension and multiple dimension. Monte Carlo is expected to converge slowly in one dimension in comparison with Newton-Cotes, however, 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 (REFERENCE) and is not unlike adaptive quadrature. 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

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.

Algorithms 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 SECTION SOMETHING, after which they were generalized to multiple dimensions using EQUATION GENERAL NCINT. 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. Pseudocode was provided below to give an overview of how the method was implemented using recursion, in which the method called itself if a certain condition was met. It should be noted that as the interval was subdivided into two, so too must the tolerance to keep the overall 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 SOMETHING. 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 Bin 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 will terminate the algorithm once the limit was met.

Generalizing each method to N dimensions was achieved mostly by converting the upper and lower limits from accepting floats to accepting arrays with N elements, 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 for indexing the sampling points and the weights.

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, code was written that would test each method. The code would iterate over the number of sampling points to characterize its convergence and execution speed. The time Python module was used to perform timing tests, and matplotlib was used to display the result in a series of graphs. The functions used for the testing, as well as its associated limits and values, are shown in TABLE SOMETHING.

|  |  |  |  |
| --- | --- | --- | --- |
| Function name | Function | Limits of integration | Value (approx.) |
| F5 |  | [-2, 6] | 7536 |
| Gaussian |  | [-10, 10] | 1.7724538509 |
| SPh |  | [(-1, -1, -1, -1), (1, 1, 1, 1)] | 4.9348022005 |
| Circ |  | [(-1, -1, -1, -1), (1, 1, 1, 1)] | 64/3 |

Results

F5

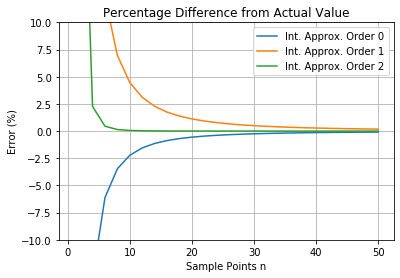
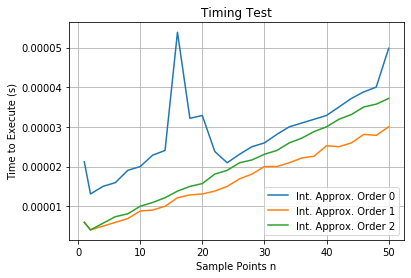
 

Fig a) Integrating F5 shows the Error % vary for the three different quadrature methods as a function of sample points b) shows the timing performance for the different quadrature methods

Gauss

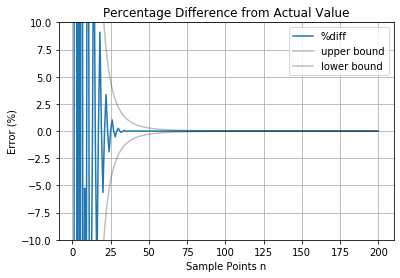
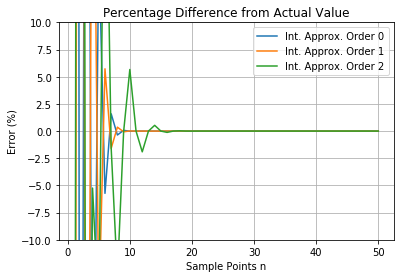
 

Fig a) Integrating the Gaussian shows the Error % vary for Simpson’s rule, while also showing the theoretical convergence limits as discussed in SECTION SOMETHING

b) Integrating the Gaussian shows the Error % vary for each of the three quadrature rules

Newton-Cotes quadrature rules converged quickly in well-behaved polynomials, as shown in FIGURE SOMETHING. As expected, Simpson’s 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 Simpson’s rule converged slower compared to the others. Integrating the Gaussian function in FIGURE SOMETHING shows that Simpson’s rule was prone to fluctuations at low numbers of sampling points. Analysing Simpson’s rule further reveals that the fluctuations were still within theoretical limits SHOWN IN FIGURE SOMETHING, confirming that Simpson’s rule converges 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 FIGURE SOMETHING 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.

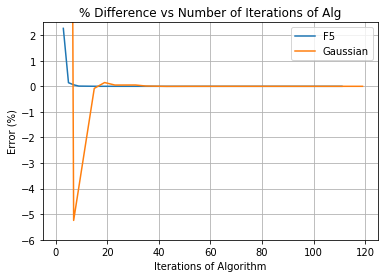
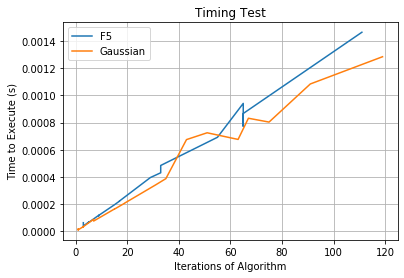
 

Fig a) Integrating both F5 and the Gaussian shows how Error % varies for adaptive quadrature as a function of iterations.

b) Shows the timing characteristics for adaptive quadrature as a function of algorithm 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, showing that 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 FIGURE SOMETHING. Due to its implementation, adaptive quadrature is marginally slower than Newton-Cotes as shown in FIGURE SOMETHING, however, the fast and stable convergence exhibited by adaptive quadrature makes it more desirable for integration.

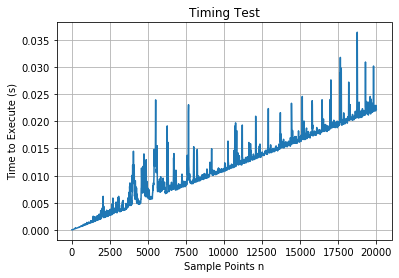
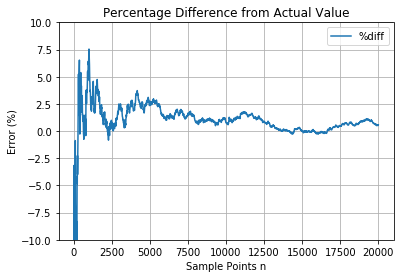


Fig a) Integrating F5 shows the Error % vary for Monte Carlo as a function of sampling points

b) Shows the timing characteristics of Monte Carlo

Monte Carlo’s execution speeds per sampling point were similar to NC quadrature methods yet were much slower to converge. As discussed before, Monte Carlo converges as O(1/√N), meaning that it requires much more sampling points to achieve an accurate result. One slight advantage of Monte Carlo methods was that only a few sampling points were required to get a decent estimate of the integral, however much more sample points were required to get a value with low errors. The values also fluctuate as the sampling points increase, eventually stabilizing when the number of sampling points becomes large enough. These undesirable qualities show that Monte Carlo methods are not well suited for one-dimensional integration.

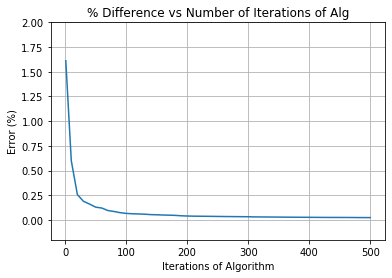
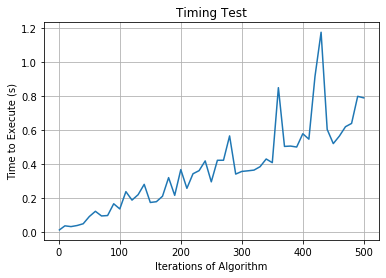
 

Fig a) Integrating F5 shows the Error % vary for recursive stratified sampling as a function of sampling points

b) Shows the timing characteristics of recursive stratified sampling

Recursive Stratified Sampling is a large improvement over Monte Carlo method as it converges 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, this method’s execution speed was slower than adaptive quadrature.

N dimensions

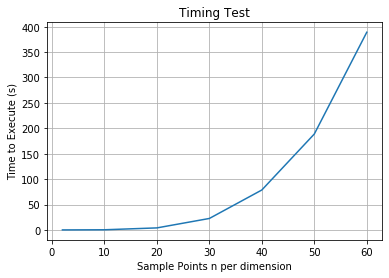
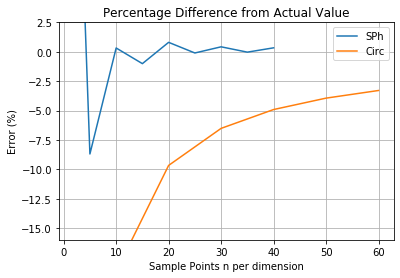


Fig a) Shows the Error % vary for Newton-Cotes in 4 dimensions as a function of sampling points

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 requires more sampling points in multiple dimensions for convergence as discussed in sECTION SOMETHING, 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 fIGURE SOMETHING. The most notable characteristic of this method was the execution speed exponentially increasing as a function of sample points per dimension. Extrapolating to hundreds of dimensions, Newton-Cotes quadratures would perform ineffectively due to the aforementioned characteristics.

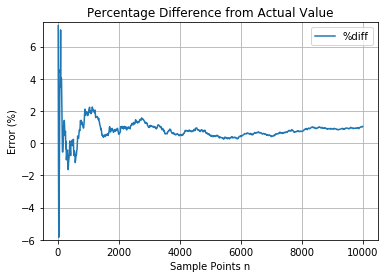
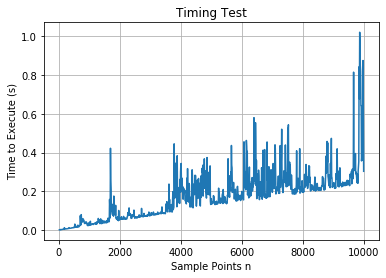
 

Fig a) Integrating Circ shows the percentage difference varying for Monte Carlo in 4D b) shows the timing test for Monte Carlo in 4D

On the other hand, Monte Carlo’s performance was relatively similar in four dimensions compared to one dimension. The convergence rate remains of the order O(1/√N), which is now favourable over quadrature methods. While it may still take tens of thousands of sample points to return an accurate value as seen in fIGURE SOMETHING, the speed of the algorithm enables it to accomplish this task with relative ease.

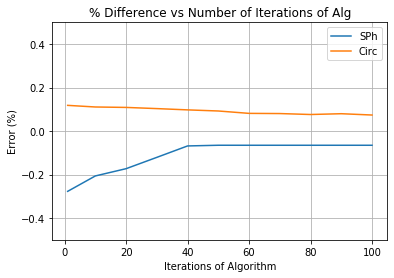
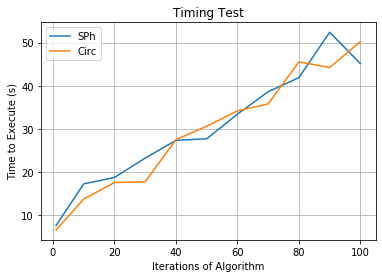
 

Fig a) Shows the percentage difference varying for Stratified Sampling in 4D b) shows the timing test for stratified sampling in 4D

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

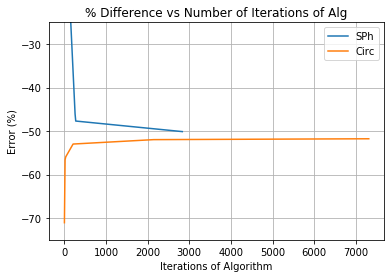


Fig shows the percentage difference varying for adaptive quadrature in 4D

Unfortunately, the implementation of adaptive quadrature in multiple dimensions was not successful. The algorithm seems to converge to an incorrect value, which should not happen. However, an unusual behaviour from this method was that it seems to converge at an error of -50% for both functions, which seems to imply that there is a bug in the code that causes the algorithm to systematically underestimate the value of the integral. However, due to time constraints, it is not possible to fix the code in time.

Conclusion

In conclusion, the project was a success. The main objectives of implementing Newton-Cotes quadrature methods and Monte Carlo methods in Python were completed and were successfully characterized using multiple functions. It was shown that Newton-Cotes quadrature methods, especially Simpson’s 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 project’s implementation of subdividing evenly into all dimensions. The more efficient subdivision has not been implemented yet due to lack of time.

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.

Appendix

Midpoint and trapezium error bits

A close up of a screen

Description automatically generated

FIGURE SOMETHING 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.

Code for the thingy

The finished project code is found at GitHub, available at the following link. Please read the documentation, as the page contains both the Python file containing the project code as well as a Jupyter Notebook file where the stable generation of the graphs used for this report were made.

https://github.com/xlr91/Integration-Methods-PP

To do for Monday

* Get all references in order (done)
* Get all the plots fixed and ready to ship (done) (extra done)
* Get rid of all notes (done)

Tuesday

* Put it all into latex tomorrow
* Fix any pressing bugs in the code (whatevs)
* Do all documentations

Wednesday

* Revise the lab report again
* Fix git stuffs