Numerical Integration Using Gaussian Quadrature and Monte Carlo Integration

Fevzi Sankaya, Fredrik Leiros Nilsen and Ilse Kuperus

(Dated: October 21, 2019)

In this article we will discuss why and how we have used numerical integration methods such as Gaussian Quadrature and Monte Carlo integration. These methods will be used to compute the expected value for the correlation energy between two electrons repelling each other, which has applications in quantum mechanics. First we look at Gaussian Quadrature with Legendre polynomials, then we rewrite it to use Laguerre polynomials. We find that the Gauss-Legendre method yields an error of order 10^{-3}, while Gauss-Laguerre has en error of order 10^{-4} with 40 points. Afterwards we used Monte Carlo integration, first with uniform distribution, then we improve our method by using an exponential distribution (importance sampling) which fits the function better. The average error of a program with $N = 10^9$ points is around 10^{-3} and $10^{-4.5}$ for brute force Monte Carlo and Monte Carlo with importance sampling respectively. However to achieve this result we have to on average wait 1000 seconds. If we parallelize the code with openMP with $N = 10^7$ points and optimize compilation the average run time with 4 threads gets lowered to ≈ 1 second from ≈ 9 seconds for Monte Carlo with importance sampling.

INTRODUCTION

Integration is one of the most important operations in physics, mathematics and science in general. The integral of a function can represent an abundance of physical observables and even probability. Due to this applicability integration techniques and especially numerical integration methods are frequently used and improved upon to solve difficult integrals. In this article we will focus on some numerical methods for computing integrals. We will primarily use two numerical integration methods. Gaussian Quadrature in the form of Gauss-Legendre and Gauss-Laguerre Quadrature and Monte Carlo integration. For reproducibility all the numerical code used for this article can be found here [1].

The integral we will compute is highly relevant for quantum mechanical applications. It is a 6-dimensional integral that is used to determine the ground state correlation energy between two electrons in a helium atom. To be more specific we need to compute the estimate value for the correlation energy between two electrons repelling each other due to the Coulomb force given as 1,

$$\langle\frac{1}{|\mathbf{r_1}-\mathbf{r_2}|}\rangle = \int_{-\infty}^{\infty} d\mathbf{r_1}d\mathbf{r_2}e^{-2\alpha(r_1+r_2)}\frac{1}{|\mathbf{r_1}-\mathbf{r_2}|}. \tag{1}$$

In this article we will therefore use the different numerical methods to compute this integral and compare their accuracy and speed to eachother. We will discuss which methods are optimal for different types of integrals and how the methods can be improved. We will start by looking at brute force Gaussian quadrature and then move on to monte carlo methods. Lastly we will try to achieve a noticeable speed up of our computations by parallelizing our Monte Carlo code with OpenMP and optimizing with compiler flags.

METHOD

Gaussian Quadrature

In general when we want to numerically solve an integral we can write it as a weighted sum i.e

$$\int f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i), [2]$$

where f(x) is the function we want to integrate, w_i is the i-th weight and x_i is the i-th mesh point. Usually the mesh points are equidistantly spaced, meaning they are the same distance between each other by a step h. The weights also tends to follow a pattern, for instance when using Simpson's rule the weights are

$$w : \{h/3, 4h/3, 2h/3, 4h/3, 2h/3 \dots h/3\}.$$

If we were to use Gaussian quadrature (GQ for short) we would now have to determine both the weights and mesh points through the use of orthogonal polynomials. What this achieves is greater precision for a given amount of numerical work. Due to the nature of the orthogonal polynomials we have to put constraints on the integral. For example Legendre polynomials are orthogonal in the interval $[-1, 1]$, while Laguerre polynomials behave similarly over $[0, \infty)$.

Gauss-Legendre

If we want to solve an integral using the Gauss-Legendre method we have to rewrite it into an integral from -1 to 1, which can then be rewritten into a sum, this can be done in the way,

$$\int_{-1}^{1} f(x)dx = \sum_{0}^{\infty} w_i f(x_i)$$

Where x_i are the integration points and w_i are the weights. The weights are given by [3],

$$w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2},$$

where P_n is the Legendre polynomial of degree n. The integral we want to calculate is the integral

$$\int_{-\infty}^{\infty} e^{-2\alpha(r1+r2)} \frac{1}{|\mathbf{r1}-\mathbf{r2}|} d\mathbf{r_1} d\mathbf{r_2}$$

Since the boundaries in this integral are not -1 and 1 we need to transform the boundaries by using [2]

$$\int_a^b f(x) = \frac{b-a}{2} \sum_0^{\infty} w_i f\left(\frac{b-a}{2}x_i - \frac{b+a}{2}\right).$$

Which can be simplified since we have that the lower boundary equals the negative upper boundary.

$$\int_{-\lambda}^{\lambda} f(x) = \lambda \sum_0^{-\infty} w_i f(\lambda x_i).$$

For our specific integral we then get,

$$\int_{-\lambda}^{\lambda} e^{-2\alpha(r1+r2)} \frac{1}{|\mathbf{r1}-\mathbf{r2}|} d\mathbf{r_1} d\mathbf{r_2}$$

$$= \lambda \sum_0^{-\infty} w_i e^{-2\alpha(\lambda r1_i + \lambda r2_i)} \frac{1}{|\lambda \mathbf{r1_i} - \lambda \mathbf{r2_i}|}.$$

We know that $r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$. We will in this article do numerical integration using the Gauss-Legendre method, but we will make use of the lib library [4] for this boundary change and to find the weights, by doing this we will not have to explicitly calculate the weights for all the different Legendre polynomials. This means that we can disregard the boundary change as it happens in the function, so we an approximation for the integral given in the form,

$$\sum_0^{\infty} \frac{w_i e^{-2\alpha(\sqrt{x_1^2+y_1^2+z_1^2}+\sqrt{x_2^2+y_2^2+z_2^2})}}{\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}}$$

For the this method we need to calculate the sum over the integration points multiplied with the weights, since our integral has six different integration points we will then need to do this in six sums, or numerically in six different loops. The weights are gotten numerically by using the Gauss legendre function in the existing library gotten from[4]. For this exact integral from minus infinity to infinity the analytical solution is $5\pi^2/16^2 \approx 0.19277$. However, we can not numerically make a sum that goes to or from infinity we have to make an estimate for infinity i.e λ. We simply use a value for λ which gives us satisfactory results. In addition, we have the factor $\frac{1}{|\mathbf{r1}-\mathbf{r2}|}$ where the denominator can be zero. This means that as the denominator approaches zero the factor tends to infinity, so we need to account for this and adjust, otherwise the numerical integration will just go to infinity. What we have done is to set the factor to zero if the denominator is zero to avoid this problem.

Gauss-Laguerre

When we use the Gauss-Legendre method for numerical integration we might not get very good results, for instance on account of the change in boundaries due to the constraints and the approximation to infinity with λ. Therefore it can be beneficial to rewrite the integral into spherical coordinates.

$$\int \frac{r_1^2 r_2^2 e^{-2\alpha(r_1+r_2)} \sin\theta_1 \sin\theta_2}{\sqrt{r_1^2+r_2^2-2r_1r_2\cos\beta}} du$$

Where

$$du = dr_1 dr_2 d\theta_1 d\theta_2 d\phi_1 d\phi_2,$$

and,

$$\cos(\beta) = \cos\theta_1\cos\theta_2 + \sin\theta_1\sin\theta_2\cos(\phi_1-\phi_2)$$

we get that the boundaries are $r_1, r_2 \in [0,\infty)$, $\theta_1\theta_2 \in (0,\pi)$, $\phi_1, \phi_2 \in (0, 2\pi)$. Since we here have a boundary from 0 to ∞ for r_1 and r_2 it is useful to use the Gauss-Laguerre polynomials for finding the weights and points for these variables. For the weights and points of θ_1, θ_2, ϕ_1 and ϕ_2 we will still be using the Legendre polynomials for the weights and points.

We can solve the integral with the spherical coordinates in a similar fashion as we did for the Cartesian coordinates. We will again have six sums, or six loops, in the numerical implementation that sums all the weights and the mesh points. The difference being that for the Laguerre method we use different polynomials for finding the weights. Since we use the Laguerre polynomials for the r_1 and r_2 integration variables we have that this approximates the integral using

$$\int_0^{\infty} x^{\alpha} e^{-x} f(x) = \sum_{i=1}^{N} w_i f(x_i) \qquad (2)$$

Therefore we can rewrite the integral into a sum. For our integral to be on the form that the integral 2 we use a substitution $4r_i = r_i'$ and $dr_i' = 1/4 dr_i$ this gives the integral,

$$\int \frac{\frac{1}{16}\sin\theta_1\sin\theta_2 \frac{r_1'^2}{16}\frac{r_2'^2}{16} e^{-(r_1+r_2)}}{\sqrt{\frac{r_1'^2}{16}\frac{r_2'^2}{16} + \frac{2}{16} r_1' r_2' \cos\beta}} d_1' dr_2' d\theta_1 d\theta_2 d\phi_1 d\phi_2$$

$$\int \frac{\frac{1}{4096}\sin\theta_1\sin\theta_2 r_1'^2 r_2'^2 e^{-(r_1+r_2)}}{\frac{1}{4}\sqrt{r_1'^2 r_2'^2 + 2r_1' r_2' \cos\beta}} d_1' dr_2' d\theta_1 d\theta_2 d\phi_1 d\phi_2$$

$$\frac{1}{1024}\int \frac{\sin\theta_1\sin\theta_2 r_1'^2 r_2'^2 e^{-(r_1+r_2)}}{\sqrt{r_1'^2 r_2'^2 + 2r_1' r_2' \cos\beta}} d_1' dr_2' d\theta_1 d\theta_2 d\phi_1 d\phi_2$$

From 2 we see that the $r_1'^2 r_2'^2 e^{-(r_1+r_2)}$ gets absorbed into the weights when using Laguerre polynomials for $\alpha = 2$. This means that the sum we need to calculate will be,

$$\sum_{i,j,k,l,m,n=0}^{N} = w_{r_1 i} w_{r_2 j} w_{\phi_2 k} w_{\phi_2 l} w_{\theta_1 m} w_{\theta_1 n} \frac{\sin\theta_1 \sin\theta_2}{\sqrt{r_1'^2 r_2'^2 + 2r_1' r_2' \cos\beta}}$$

Where the w's are the different weights, numerically we implement these by using the existing library that contains functions for finding, among other things, weights for different numerical integration methods [4]. In the end we need to divide this sum by 1024 to get the approximation of the original integral.

Monte Carlo Integration

The Monte Carlo integration method gives a numerical estimate for an integral, by evaluating the integrand at random points. If we look at a one dimensional integral of a function f, evaluated between 0 and 1, and have N points x_i given by a probability distribution $p(x)$, we can estimate the average of the function by calculating the sum

$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^{N} f(x_i) p(x_i). \tag{3}$$

If we say that $p(x)$ is uniform, so $p(x) = 1$ for $x \in [0,1]$, we get that the average of f is simply given by the sum of all $f(x_i)$. If we imagine an infinite amount of points, we can then rewrite equation (3) as an integral I of $f(x)$ from 0 to 1. Since the sum in equation (3) should approach the analytical expectation value $\langle f \rangle = I$ as $N \to \infty$, we can approximate the integral by evaluating it at N random points x_i, so [5, p. 6]

$$I = \int_0^1 f(x)\,dx \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i). \tag{4}$$

This is also know as the "law of large numbers" and it's the basics of the Monte Carlo integration method.

While the integration method described by equation (4) is specific for integrals from 0 to 1, there are ways of using this on other integrals as well. If we have an integral from a to b, we can use a variable transformation from x to y, given by

$$x(y) = a + (b-a)\,y. \tag{5}$$

If $x \in [a,b]$, we then see that $y \in [0,1]$, since when $y = 0$, $x = a$ and when $y = 1$, $x = b$. This means that we can use equation (4). We do however have to multiply with a Jacobian determinant, which will here simply be given by

$$\frac{dx}{dy} = b - a. \tag{6}$$

So by using the variable transformation given by equation (5), the Jacobian from equation (6) and the Monte Carlo method in equation (4), we can calculate a more general integral as

$$I = \int_a^b f(x)\,dx = \int_0^1 f\left(x(y)\right)(b-a)\,dy = \frac{b-a}{N} \sum_{i=1}^{N} f\left(x(y)\right), \tag{7}$$

where y_i is given by a uniform distribution between 0 and 1. Here we notice that in the numerical calculation we only need to multiply with the Jacobian after the sum, which is 1 floating point operation instead of N.

However this method does not work well for all integrals. If for instance one of the limits is $-\infty$ or ∞, we need to set a finite number as a approximation for infinity. The sum might also converge slowly to the true value, if there is a small area where the function changes quickly, while it is almost constant outside of this area. One will then end up calculating many points in the less important areas of the function, which will lead to slow convergence. One way to get around this is importance sampling. In importance sampling, one uses a probability function $P(x)$ that resembles the integral to do a change of variables $x \to y$, where $x \in [a,b]$, and P is normalized and defined on the same interval. The interval of y will be defined by the interval of x and the transformation, we will assume that $y \in [0,1]$, if this is not the case we need to another change of variables, as given by equation (5). For the transformation we use that the likelihood for a event given by $P(x)\,dx$, must be the same after the variable transformation. So we get [5, p. 31]

$$P(x)\,dx = p(y)\,dy = dy, \tag{8}$$

where we have used that $p(y) = 1$. By integrating both sides of equation (8), we then get

$$y(x) = \int_a^x P(x')\,dx', \tag{9}$$

as our variable transformation. From equation (8) and (9), we can then rewrite our integral as

$$I = \int_a^b F(x)\frac{P(x)}{P(x)}\,dx = \int_0^1 \frac{F(x(y))}{P(x(y))}\,dy, \tag{10}$$

where $x(y)$ is found by inverting equation (9).

One function for $P(x)$ we will look at here is the exponential distribution. This distribution is defined between 0 and ∞ as

$$P(x) = \alpha e^{-\alpha x}, \tag{11}$$

where $\alpha > 0$. By integrating this function from 0 to x, we get

$$y(x) = \int_0^x \alpha e^{-\alpha x}\,dx = e^{-\alpha x} - e^{0\alpha} = e^{-\alpha x} - 1. \tag{12}$$

We then invert this equation to get

$$x(y) = -\frac{1}{\alpha}\ln\left(1-y\right). \tag{13}$$

We note here that for for $y \to 1$, $x \to \infty$, so we do not have to make an estimate for infinity, as we would have to do with the initial Monte Carlo method.

For our integral, we will first evaluate it in Cartesian coordinates, with a uniform distribution. We then have to make an approximation of infinity. This is similar to what we had to do with Gauss-Legendre quadrature. We then have to make six uniform transformations, described by equation (5), and multiply with the corresponding jacobians from equation (6), since we now have six integrals from $-a$ to a, where a is the approximation for infinity. Here we have sat this approximation to 2 for Monte Carlo method.

We will also use importance sampling with the exponential distribution (given by equation (11)) as the distribution function. We will also evaluate the integral in polar coordinates. This mean that we only have to use importance sampling for the variables r_1 and r_2, since the others will have finite boundaries. We then have to divide the integrand by $P(r_1)P(r_2) = \alpha'^2 \exp\left(-\alpha'(r_1 + r_2)\right)$, where we have called the factor α from equation (11) α' to distinguish it from the α in the integral. We see that since $\alpha = 2$, we can make the integrand simpler by setting $\alpha' = 4$. For the change of variables $r_1 \to r_1'$ and $r_2 \to r_2'$ (both described by equation (13)), we then get the integral

$$\begin{aligned} I &= \int r_1^2 r_2^2 \frac{e^{-2\alpha(r_1+r_2)}}{\alpha'^2 e^{-\alpha'(r_1+r_2)}} \frac{\sin\theta_1 \sin\theta_2}{\sqrt{r_1^2 + r_2^2 + r_1 r_2 \cos\beta}}\, du' \\ &= \frac{1}{16} \int \frac{r_1(r_1')^2 r_2(r_2')^2 \sin\theta_1 \sin\theta_2}{\sqrt{r_1(r_1')^2 + r_2(r_2')^2 + r_1(r_1') r_2(r_2') \cos\beta}}\, du'. \end{aligned} \quad (14)$$

Here we have also used $du \to du'$. For the two θs and ϕs we then do a uniform transformation, as we did in the brute force integral, since $\phi_1, \phi_2 \in [0, 2\pi]$ and $\theta_1, \theta_2 \in [0, \pi]$.

When using the Monte Carlo methods, we will also keep track of the variance σ^2 of the estimate. To do this we use that the variance of a variable x is given by [5, p.5]

$$\sigma^2 = \mathbb{E}\left[x^2\right] - \mathbb{E}\left[x\right]^2. \quad (15)$$

So for each integration point $\mathbf{r}_i$ we keep the value $f(\mathbf{r}_i)$, and $f(\mathbf{r}_i)^2$. We then calculate the sum of these two values, and divide by the number of integration points to estimate the expectation values. If we then combine equation (15) with the central limit theorem, we can estimate the standard deviation of the integral σ_m with N integration points as

$$\sigma_m = \sqrt{\frac{\mathbb{E}\left[f(\mathbf{r})^2\right] - \mathbb{E}\left[f(\mathbf{r})\right]^2}{N}}. \quad (16)$$

Here, one important aspect to note about the error in the Monte Carlo method is that since the expectation values for $f(\mathbf{r})$ and $f(\mathbf{r})^2$ are constants, the error scales as $1/\sqrt{N}$, so the error is not dependant on the dimension of the function [5, p. 8].

To test the Monte Carlo method we ran with $10^3, 10^4, \ldots, 10^9$ points 100 times each with and without importance sampling. To get the random numbers we used the 64-bit Mersenne Twister from c++'s random class. We used this to pick random numbers between 0 and 1, and then used the transformations described by equations (5) and (13) to get the numbers in a uniform distribution between $-a$ and a (where a is the approximation of ∞), and in the exponential distribution respectively. For each run we measured the time it took with the computers clock, and calculated the variance for each calculation. To test the variance, we also calculated the standard deviation of the integral directly from the definition as

$$\sigma_m = \mathbb{E}\left[\left(f(\mathbf{r}) - \mu\right)^2\right], \quad (17)$$

with μ as the analytical value of the integral, $5\pi^2/16^2$. To compare with Gauss quadrature we also ran 50 calculations with 5, 10, 15 25 and 30 integration points (here we had to use fewer runs due to time constraints). We then calculated the average time spent by the method, and the error for each number of points. For the comparison, all programs was run using no compiler flags, on a computer with an Intel i7-4790 CPU and 16 GB of RAM.

Monte Carlo Parallelization with OpenMP

By utilizing open multi-processing or openMP we can parallelize our code. In other words we can make parallel regions in our code where the master thread which reads the code in series splits into multiple parallel threads. This means that we can for instance compute several parts of a for loop simultaneously with a different set of random numbers (seeds) which will increase the computations speed if implemented correctly. By default when we create a parallel region in our code we get four parallel threads. We will also try using compiler flags to optimize the code. All of this should in theory speed up the run time of the Monte Carlo integration code.

RESULTS

Gaussian Quadrature

For finding the approximation for infinity for the boundaries of the integral we have plotted $e^{-2\lambda}$ for different λ to see where it will be zero.

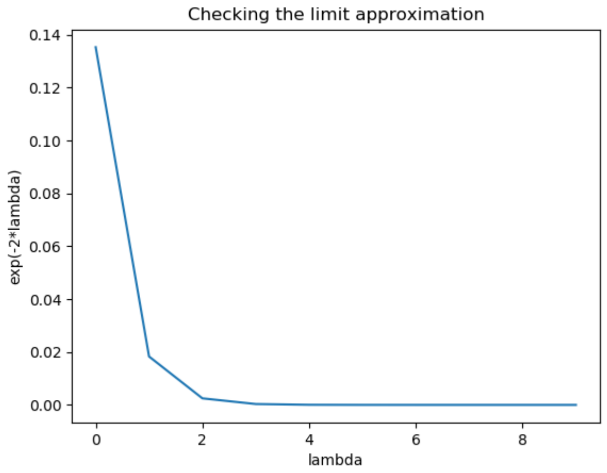

Figure 1. Here we see a graph of the function for different values for lambda, it shows where the function will be zero, which will be a good approximation

Table I. This is a table for the approximation of the integral using different approximations to infinity where the N is constant at 20, meaning the degree of the polynomial is 20.

λ	Integral Approximation	Error
1	0.16142	0.0313502
2	0.177065	0.0157048
3	0.156139	0.0366306
4	0.127513	0.0652571

In table I we see the different integrals and errors when setting N to be a fixed number and varying the λ which is are the boundary conditions, the approximations to infinity.

Table II. This is a table over the different integration approximations using the Gauss-Legendre method with the approximation for infinity set to 2.

N	Integral Approximation	Error
5	0.354602	0.161832
10	0.129834	0.0629358
15	0.199475	0.00670478
20	0.177065	0.0157048
25	0.189110	0.00366047
30	0.185796	0.00697438
40	0.18867	0.00409987

In table II we see the approximation for the integral when using different order polynomials with the same approximation for infinity. We see that the approximation get better for larger polynomials generally, but the error is still quite large.

Table III. This is a table over the different integration approximations using the Gauss-Laguerre method for different N

N	Integral Approximation	Error
5	0.17345	0.0193204
10	0.186457	0.0063127
15	0.189759	0.00300671
20	0.191082	0.0016882
25	0.191741	0.0010247
30	0.192114	0.0006563
40	0.192493	0.0002768

In table III we have the results for doing the brute force method of the integral using spherical coordinates in the integral and finding the weights for two of the parameters by using the Laguerre polynomials instead of the Legendre polynomials. As we see in this table the results get better as we use a higher value for N, i.e. a polynomial of a higher degree and a larger sum, In addition we see the results here are better than the results we got by using the Legendre method.

Monte Carlo integration

We have plotted our results for the standard deviation from the two Monte Carlo methods in figure 2. Here we have plotted the mean standard deviation of the integral estimations, $\overline{\sigma_m}$, calculated from the variance in each run with equation (15), as well as the "true" σ_m, calculated from the estimates of the integral compared with the analytical value using equation (17). We see that the two values are mostly similar, however for 10^3 and 10^9 points the brute force method seems to give a lower estimate for the variance than the actual value. When taking a linear fit of $\overline{\sigma_m}$ in the log-log plot we get the slope -0.42 ± 0.02 for the brute force version and -0.49 ± 0.01 for the importance sampling.

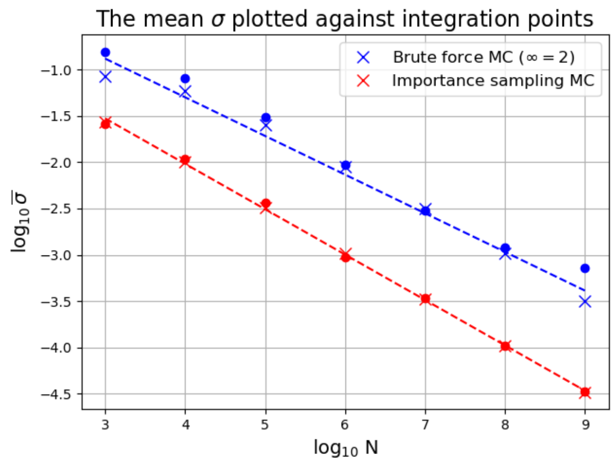

Figure 2. The standard deviation σ_m for the two Monte Carlo methods. The value for $\overline{\sigma_m}$ calculated from the variance of each run is marked as a cross, while the value calculated from the estimates compared with the analytical value is marked with a dot. The dotted lines are a linear fit of the values calculated from the variance of each run.

When comparing the time the two methods we got the results in figure 3. Here we see that the two methods use roughly an equal amount of time. We also notice that the slope in the log-log plot is around 1, which means the amount of time is roughly linearly proportional to the number of integration points.

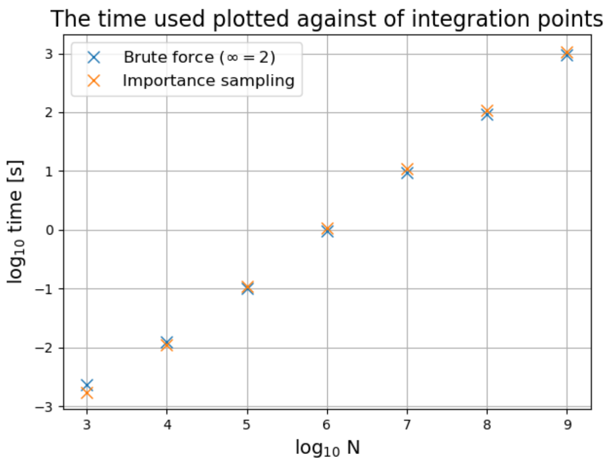

Figure 3. The average time spent on both Monte Carlo methods.

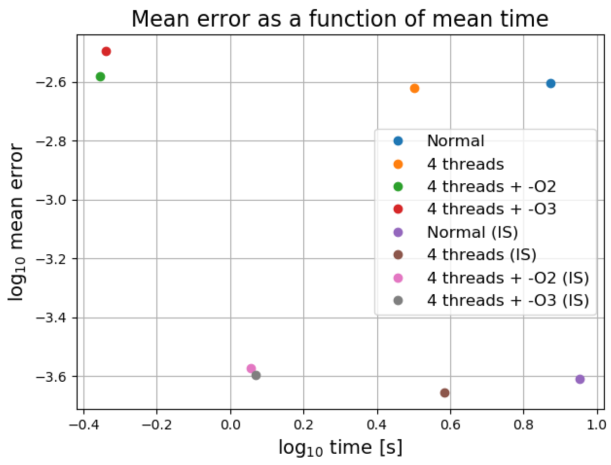

Figure 5. The mean error plotted against the amount of time spent computing the integral for $N = 10^7$ points

When comparing the different methods, we have looked on the time it took to compute each integral, and how accurate the results was. In figure 4 we have plotted these results. We see that in general, the error goes down as the time one spend more time to compute the integral.

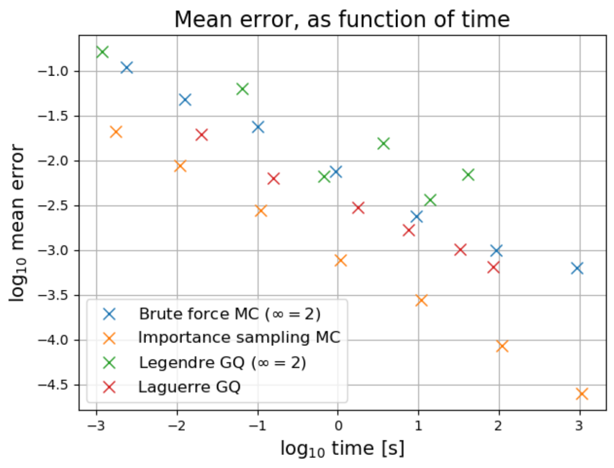

Figure 4. The average error plotted against the amount of time spent computing the integral, for each of the four integration methods used in this report.

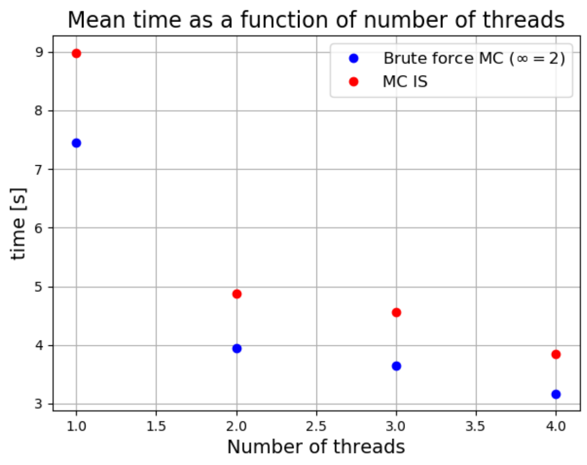

Figure 6. The average time plotted against the number of threads for $N = 10^7$ points (no compiler flags)

	Time [s] MC	Time [s] MC (IS)
Normal(1 thread)	7.4	9.0
2 threads	3.9	4.8
4 threads	3.1	3.8
4threads + optimized	0.4	1.2

Table IV. Table over approximate (rounded off to first decimal) average run time for Monte Carlo code with and without importance sampling.

OpenMP Parallelized Monte Carlo Integration

In figure 5 we have plotted the logarithm of the mean error against the logarithm (both with base 10) of mean time. The plots represents the mean error and time of 100 sample runs with $N = 10^7$ points. Lastly figure 6 shows us mean time against the number of threads used.

Table IV shows some of the average run times for the Monte Carlo method with and without importance sampling.

DISCUSSION

Gaussian Quadrature

We have first used the Gauss-Legendre method to find an approximation for the integral. To do so we have to approximate the infinity and minus infinity boundaries given that we can not numerically use infinity. To find which values might be good approximations to use we have plotted $e^{-4\lambda}$ with different numbers for λ to see where the function becomes 0. At the point where the function becomes zero we have a good approximation. In figure 1 this graph is plotted, from this graph we see that already at $\lambda = 2$ the function will be zeroed out. This gives a good indication for which approximation to try. We therefore use $\lambda = 2$ for finding the approximations to the integral in II. To make sure 2 is a good approximation for infinity here we have also tried with a few other ones to see what the results will be then. In table I we see that when approximating infinity, setting λ with different numbers the error becomes different, we see that a larger λ does not necessarily mean a smaller error. From this table we see that if we use N=20, $\lambda = 2$ is the best approximation, this will vary with the different N's.

By looking at II we see that the integral can be approximated with $\lambda = 2$ for different values of N. A larger N will mean a higher degree polynomial used for the weights and a larger number of mesh points summed over. This table shows that in general the approximation of the integral when using Legendre polynomials becomes better as the number N is higher. We see this from the error that becomes smaller. However this error is still significant for an estimate, it is also important to note that even getting to N = 40 takes quite some time on a normal laptop. Doing this approximation with an even larger N is almost or not doable due to the amount of operations done in a six-double loop with seven multiplications and a function, it just takes a long time to calculate for the computer.

In general we can see that the Gauss-Legendre is not a very accurate method to find the integral, especially not for an integral with six different integration variables. If we had less integration variables the numerical solver would be faster and then we could perhaps get a better result by using a higher order Legendre polynomial. Other drawbacks of this method are also the way we have to approximate the infinity with a value λ as we can not numerically implement infinity. Another factor is that the integrand might diverge in the case when the denominator is zero, this can happen frequently and although we make sure this gives no problems numerically by disregarding these instances this contributes to the error in the approximated number for the integral. The results we got with the Legendre polynomial for this integral are not satisfactory, this is the reason why we chose to explore other methods.

From table III we see that Gauss-Laguerre has better accuracy than Gauss-Legendre. This is in part due to the fact that the Gauss-Laguerre method does not have any approximations in the limits and uses Laguerre polynomials for the 0 to infinity limit and Legendre polynomials for the finite limits. Due to the rewrite and the different use of polynomials part of the integral will also be absorbed into the weights, since the weights are calculated by an existing library this has a good accuracy. However in spite of the Laguerre having better accuracy it has the drawback of being a slower method. This is due to the many more floating point operations that need to be performed in each loop iteration after having rewritten the integral to spherical coordinates. In addition this method uses three different sets of weights, which means that it will have to compute these three arrays for the weights separately compared to the one we used for all the weights in the previous method. Thus we can see that this method has the advantage of being more precise, but not necessarily better since it uses more time. We can see from 4 that the time difference between the two methods will decrease as the number of integration points increases, meaning that the Laguerre method will still be beneficial over the Legendre method for higher N.

Monte Carlo Integration

When calculating the standard deviations of the Monte Carlo estimates in figure 2, we found that for the brute force method, the estimated standard deviation was somewhat lower than the "true" value for two points. This could simply be random, as the estimated standard deviation should be given by some probability distribution. Another explanation for 10^9 points is that since we are not calculating the integral from $-\infty$ to ∞, but -2 to 2, we get a lower estimate for the variance since we have left out a part of the integral, and 10^9 points is enough to start to distinguish between these two integrals. From figure 2 we also see that $\overline{\sigma_m}$ for importance sampling is somewhere between $10^{0.5}$ to 10^1 times lower than that for brute force. Especially for the higher numbers of integration points, we seem to get an improvement of a factor 10. This shows the advantage of importance sampling, with the same amount of points one can here get an answer with σ_m a factor 10 less than the answer from brute force.
When looking at $\overline{\sigma_m}$ we also found slopes of -0.42 and -0.49 in the log-log plot. Both slopes are then close to -0.5. This is as one would expect, as a slope of -0.5 means that σ_m is proportional to $N^{-0.5}$. This fits well with the central limit theorem, which is not surprising given that we used it to calculate σ_m. So what this really tells us is that the variance σ^2 is the same for every run, which it should be, since we used the same distribution and function for each method. The slope for brute force was however about 4σ below the theoretical value of -0.5. If we look at figure 2 this is probably because of the point where we used 10^3 integration points, where the variance seem to be significantly below the trend for the other points. The "real" variance for this point is however closer to the trend. This could mean that with such a relatively low amount of points the range of values sampled is to small to get a good estimate of the variance, and it ends up being underestimated. With importance sampling however, we see that the results fit with the expected value within 1σ, which could be taken as a sign that our random number generator works as it should.

When looking at the time required from the Monte Carlo methods in figure 3, we see that the two methods take about the same amount of time for a given number of points. When we combine this with the accuracy one gets from figure 2, it shows why importance sampling should be preferred, as with almost no change in CPU-cycles, one can get an answer that is around 10 times better. From figure 3, we also see that the slope in the log-log plot is around 1. This means that the time spent scales roughly linearly with the number of points, which is as one would expect since there is a certain amount of floating point operations required for each point, and this is not dependent on the total amount of points.

In figure 4, we have compared the time and mean error of the four different methods discussed here. We see that all methods seem to get a better estimate, as one spends more time to compute the integral, which is as expected. More interesting, is the difference between the four methods. First we see that Monte Carlo with importance sampling seem to give significantly better answers than the other three methods, getting around an order of magnitude lower error than brute force Monte Carlo and Gauss-Legendre quadrature, when spending the same amount of time. After this, Gauss-Laguerre also give a result that is better than brute force MC and Gauss-Legendre quadrature, but by a smaller amount, sometimes barley getting a better result than brute force MC. From figure 4 we then see that for this kind of multi-dimensional integral, the Monte Carlo method with a method like importance sampling is to be preferred as for the same amount of resources (here in the form of CPU cycles) it gives the most accurate results. This is a result we would expect to be more significant if we looked at a function of even higher dimensions, as the error in the Monte Carlo method does not scale with the dimension of the function, while in Gaussian Quadrature it does.

One potential issue with importance sampling however, is how to best chose the distribution function $P(x)$. With this integral we found a distribution that cancelled out one of the terms in our integral, which made it less computationally heavy to solve. For a much more complicated integral, this may not be so easy to achieve, and one may have several candidates for the possible distributions to try. To best chose the transformation of variables is then something one have to think about when using importance sampling. Exactly how to do this for a more complicated function could be something one could look at in further studies.

OpenMP Parallelizsed Monte Carlo Integration

First and foremost, we see that we were successful in our attempt to parallelize our code for Monte Carlo integration. We have made a table to illustrate the average run time as seen by the plot 5, see table IV. We see that with $N = 10^7$ points our normal code takes around ≈ 9 seconds for MC (IS) and after parallelizing and optimizing it takes on average ≈ 1 second to run the code. This is a decrease in run time of almost an order of magnitude. In addition, we see that parallelizing with 2 threads makes the program run approximately twice as fast, from 9 seconds $\rightarrow$ 4.8 seconds, which is to be expected due to the workload being distributed in two the for loop. However, if we look at figure 6 from 2 to 4 threads the time doesn't get halved as one would expect. This may be due to some of the threads finishing before the other causing them to wait for all 4 to be done with their part of the loop. Nonetheless,there is a noticeable speed up between 2-3 and 3-4 threads as well. Furthermore, we can validate that the result actually make sense, as seen by the error. The plotted mean error as function of mean time is meant to illustrate that the error, i.e the result remains around the same order of magnitude $10^{-3.6} \approx 2.5 \cdot 10^{-3}$ for MC (IS) with the parallelized and optimized versions of the code. We do however see a slight deviancy with the brute force Monte Carlo with 4 threads and optimization. The error is a fraction higher perhaps caused by a slight loss in numerical stability, as it gets sacrificed for speed. For future work we could have chosen to test more compiler flags, for instance,-O1 or -Ofast, but this would have been redundant as it would only be slightly slower or faster than using -O2 and -O3. Instead we chose to focus on studying the effect of parallelization with different number of threads more closely.

CONCLUSION

Overall we can see that the Gauss Legendre is not very accurate, but is quick for small number of integration points for a simple integral, Gauss Laguerre has better accuracy, but takes a longer time and will therefore be virtually useless for a complicated integral like this for many integration points or complete accuracy. For both these methods it is the fact that our integral has six integration variables that slows them down. Therefore Gaussian quadrature might be more suited to simple integrals of one or two dimensions. We see that Monte Carlo gives us results improving with the number of points used, and the error in the estimates goes as $1/\sqrt{N}$, where N is the number of points, which is a result that fit well with the central limit theorem. The methods used for this article are quite specific to the integral we have, it might be useful in the future to generalise this method in order to use it on different integrals without having to change it. The least error we were able to achieve was with $N = 10^9$ points, where the average error was 10^{-3} for brute force Monte Carlo and $10^{-4.5}$ for Monte Carlo integration with importance sampling. This took on average about 1000 seconds to run. When comparing the methods, we saw that for the same amount of time spent, Monte Carlo with importance sampling typically had an error that was around $10^{0.5}$ to 10^{1} times lower than the other three methods, showing how the Monte Carlo method is preferable on high dimensional integrals. Lastly we parallelized our Monte Carlo integration with OpenMP with $N = 10^7$ points. This allowed us to split and simultaneously run a for loop on different threads(cores) which drastically sped up the run time. By adding a compiler flag as well we were able to cut our average run time from $t \approx 9$ seconds to $t \approx 1$ seconds for Monte Carlo integration with importance sampling, a remarkable difference. For further exploration it can be interesting to look at different compiler flags when doing the Monte Carlo method, or generalizing all numerical integration methods further in order to make a numerical solver that can be used for different integrals in other situations.

[1] "https://github.com/ilsekup/FYS3150/tree/master/Project%203."

[2] Hjort-Jensen M., "Computational Physics Lectures: Numerical Integration, from Newton-Cotes quadrature to Gaussian quadrature ," *Lecture notes*, pp. 109–149, 2015.

[3] J. Kautsky and S. Elhay, "Calculation of the Weights of Interpolatory Quadratures," *Numerishce Matematik*, vol. 40, pp. 407–422, 1982.

[4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes.* Camebridge University Press, 3rd editio ed., 2007.

[5] Hjort-Jensen M., "Computational Physics Lectures: Introduction to Monte Carlo methods," *Lecture notes*, 2019.