FYS4150: Project 3

Jonathan Brakstad Waters

Øyvind Engebretsen Elgvin

October 21, 2019

Abstract

In this article, we present four different numerical methods for calculating integrals, namely Gaussian Quadrature with Legendre polynomials, Gaussian Quadrature with Laguerre polynomials, Brute Force Monte Carlo, and Monte Carlo with Importance Sampling. We introduce a six dimensional example integral, and solve this using the four different methods. Furthermore, we present the results yielded by these methods, and compare their precision and efficiency against each other.

The main takeaway from this article should be that Monte Carlo integration is by default superior to Gaussian Quadrature for a six dimensional integral. Whilst Gaussian Quadrature yields an error of order 10^{-3} for our integral, Monte Carlo integration yields an error of order 10^{-5}, a hundred times better.

Also, it is shown how a bit of tweaking can improve the most basic application of both Gaussian Quadrature and Monte Carlo - in our case by changing coordinate system and applying methods more tailored to our specific problem, i.e. Gaussian Laguerre and Monte Carlo Importance Sampling.

1 Introduction

Numerical integration is an important aspect of modern physics. The goal of this report is to present a number of methods for performing numerical integration, namely variations of Gaussian Quadrature and Monte Carlo Integration. To motive our analysis, we will use an example integral derived from Quantum Mechanics:

$$\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-4(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}$$

where

$$r_i = \sqrt{x_i^2 + y_i^2 + z_i^2} \qquad \text{and} \qquad |\mathbf{r_1} - \mathbf{r_2}| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

The exact solution of this integral is $5\frac{\pi^2}{16^2} \approx 0.192766$. In the following, we will present four numerical methods for solving this integral (Gaussian Quadrature with Legendre polynomials, Gaussian Quadrature with Laguerre polynomials, Brute Force Monte Carlo and Monte Carlo with Importance Sampling), briefly explain our implementation of these methods, and finally discuss their performance with regards to precision and runtime.

2 Methods

2.1 Gaussian Quadrature

The basic idea behind Gaussian Quadrature is to approximate an integral I as

$$I = \int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=1}^{N} \omega_i g(x_i)$$

Here, $W(x)$ is the so-called weight function, having the purpose of emphasizing different parts of the interval $[a, b]$ differently. The integration points x_i and integration weights ω_i are determined by a set of orthogonal polynomials. Such sets could for example be the Legendre, Laguerre or Hermite polynomials. Now, there is quite a bit of theory involved to derive x_i and w_i, presented well by Hjorth-Jensen (2015), but the bottom line is as follows.

The points x_i are given by the zeros of an orthogonal polynomial of degree N, we call it L_N. This would be the highest order polynomial of our set. To find the weights w_i, we define a matrix

$$L = \begin{bmatrix} L_0(x_0) & L_1(x_0) & \cdots & L_{N-1}(x_0) \\ L_0(x_1) & L_0(x_1) & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ L_0(x_{N-1}) & L_1(x_{N-1}) & \cdots & L_{N-1}(x_{N-1}) \end{bmatrix}$$

The weights are then given by

$$\omega_i = 2(L^{-1})_{0i}$$

Given the methods for determining $I \approx \sum_{i=1}^{N} \omega_i g(x_i)$, the only thing left to decide is what set of orthogonal polynomials to use.

2.1.1 Gaussian Quadrature with Legendre polynomials

We are integrating our example integral for six variables, (x_1, y_1, z_1) and (x_2, y_2, z_2). In principle, all of these take values in the range $[-\infty, \infty]$, but we can approximate reasonable minimum and maximum values, let us call them $\pm\lambda$ (see subsection further down). In that case, we need a set of orthogonal polynomials that lets us integrate an integral $\int_{-\lambda}^{+\lambda} f(x)dx$. We have chosen to use the Legendre polynomials.

Legendre polynomials are orthogonal on the interval $x \in [-1, 1]$. To use these polynomials to solve an integral on $[a, b]$, we need to perform a change of variables so that we can integrate instead from -1 to 1. With

$$t = \frac{b-a}{2}x + \frac{b+a}{2}$$

we can transform an integral like this:

$$\int_a^b f(t)dt = \frac{b-a}{2}\int_{-1}^{1} f(\frac{b-a}{2}x + \frac{b+a}{2})dx$$

From this transformation, we can also map our weights and integration points on to new values.

In our case, the variable transformation from $[-\lambda, \lambda]$ to $[-1, 1]$ is handled by the function `gauleg(a,b,x,w,N)`. This function takes the interval end points and the number of integration points as arguments, and returns arrays for x_i and ω_i. Then the only thing that remains is to perform the sum

$$I = \int_{-\lambda}^{\lambda} f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i)$$

Our code for the implementation of this method is found in `Gaussian_Legendre.ccp`. Since our integral is six dimensional, we multiply with six weights in the for-loop. Note that we have assumed our integrand function to be symmetrical around the origin. This means that we only need one array for our integration points, and one array for the weights. If our coordinates existed in different intervals, we would need to call the `gauleg` function once for every coordinate. We have run our Gaussian Legendre solver for a range of λ and N (number of integration points). The results are shown and discussed in the results section (see table 1).

Finding λ to approximate infinity

The main argument for approximating infinity by some λ, is that our integrand $f = \frac{1}{|\mathbf{r_1}-\mathbf{r_2}|}e^{-2\alpha(r_1+r_2)}$ goes to zero as r_i goes to infinity. We need then to find a value λ where the integrand is reasonably small. To do this, we have plotted the exponential part of f as a function of r_1 and r_2, as illustrated in figure 1. We have seen where f seems to go to zero, and used the definition of r_i to find reasonable limits for (x_i, y_i, z_i).

2.1.2 Gaussian Quadrature with Laguerre polynomials

If an integral can be rewritten as

$$\int_0^{\infty} f(x)dx = \int_0^{\infty} W(x)g(x)dx = \int_0^{\infty} x^{\alpha}e^{-x}g(x)dx,$$

then generally it is better to use Laguerre polynomials, which are orthogonal on the whole interval $[0, \infty]$. Amongst other things, this rids us of the need for approximating infinity with some λ, as in the Legendre case.

The observant reader will notice that our integral is actually not on the Laguerre form. To achieve this, we perform a change of variables to spherical coordinates, as follows:

$$d\mathbf{r_1}d\mathbf{r_2} = r_1^2 r_2^2 \sin\theta_1 \sin\theta_2 \quad dr_1 dr_2 d\theta_1 d\theta_2 d\phi_1 d\phi_2$$
$$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos\beta}}$$
$$\cos\beta = \cos\theta_1 \cos\theta_2 + \sin\theta_1 \sin\theta_2 \cos(\phi_1 - \phi_2)$$

With this change of variables, our integral can be written:

$$I = \int_0^\infty \int_0^\infty \int_0^\pi \int_0^\pi \int_0^{2\pi} \int_0^{2\pi} \frac{r_1^2 r_2^2 \sin\theta_1 \sin\theta_2}{r_{12}} e^{-4(r_1+r_2)} \; dr_1 dr_2 d\theta_1 d\theta_2 d\phi_1 d\phi_2$$
$$= \int_0^\infty \int_0^\infty \int_0^\pi \int_0^\pi \int_0^{2\pi} \int_0^{2\pi} e^{-(r_1+r_2)} g(r_1, r_2, \theta_1, \theta_2, \phi_1, \phi_2) \; dr_1 dr_2 d\theta_1 d\theta_2 d\phi_1 d\phi_2$$

with

$$g(r_1, r_2, \theta_1, \theta_2, \phi_1, \phi_2) = \frac{r_1^2 r_2^2 \sin\theta_1 \sin\theta_2}{r_{12}} e^{-3(r_1+r_2)}$$

We see that this is the desired form for integrating with Laguerre polynomials, with $\alpha = 0$. Note that we also could have extracted the factor $r_1^2 r_2^2$ from g, but in our program we have not done so.

Now that we have rewritten our integral, we can proceed to solving it, using Gaussian Laguerre for the radial part, and Gaussian Legendre for the angular part. This is implemented in the code `Gaussian_Laguerre.cpp`. Note how `gauleg` is called twice - once for the θ-weights, and once for the ϕ-weights. The weights and integration points for the radial part are obtained from the function `gauss_laguerre`. This function takes as input the number of integration points N, and the exponent of x in the weight function $W(x)$, α. Since we have not extracted the $r_1^2 r_2^2$-factor, we call the function with $\alpha = 0$.

When the weights and integration points have been determined, we perform the same calculation of the sum as we did with the Legendre polynomials. Note that the weights are not all the same anymore, rather they are different for the different coordinates. The results of this method can be seen in table 2.

2.2 Monte Carlo integration

Monte Carlo Integration is a method that uses randomness to obtain numerical results, by performing random sampling N number of times, and is often used to solve three types of problems; optimization, numerical integration, and drawing from a probability distribution.

The main idea is to evaluate the function to be integrated, $f(x)$ at N random points x_i, drawn from a discretized Probability Distribution Function (PDF). The PDF gives the probability (or relative frequency), P, for each of these values to occur, in way of the integral:

$$P(X \in [a, b]) = \int_a^b p(x)dx$$

where $p(x)$ is the probability density of the given PDF.

Our integral is then approximated as the expectation value of the function values at the random points:

$$I \approx \langle f \rangle = \frac{1}{N} \sum_{i=1}^{N} f\left(x_i\right) p\left(x_i\right),$$

An important quantity in the Monte Carlo calculation is the variance, σ^2, which measures the extent to which the function f deviates from its average over the region of integration, and is given by:

$$\sigma_f^2 = \frac{1}{N} \sum_{i=1}^{N} \left(f\left(x_i\right) - \langle f \rangle\right)^2 p\left(x_i\right)$$

Here, f is the function we are integrating and $p(x_i)$ is already implemented in the sampling of our coordinates, $\mathbf{r_1}$ and $\mathbf{r_2}$. Defining the variance this way we avoid the chance of getting a negative value. The standard deviation, σ, is then defined as :

$$\sigma = \frac{\sqrt{\sigma^2}}{\sqrt{N}}$$

In general, the Monte Carlo method is exceptionally efficient in higher dimensions as the error (in opposition to Gaussian quadrature) scales independently from the number of dimensions, with a factor of $1/\sqrt{N}$. With the six dimensions in this integral it should be much faster and more accurate than the Gaussian quadrature methods using Legendre and Laguerre polynomials.

In this study, we are using two versions of Monte Carlo Integration, namely Brute Force Monte Carlo and Monte Carlo with Importance Sampling.

2.2.1 Brute force Monte Carlo integration

The brute force method uses the uniform distribution for its random sampling. One downside of this method is that if the function to be integrated peaks in a narrow interval, many of our samples could be drawn from areas where there is no contribution to the integral, leaving fewer samples to add meaningful contributions.

For the uniform distribution we have:

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x)$$

with

$$\theta(x) = 0 \qquad x < 0$$
$$\theta(x) = \tfrac{1}{b-a} \quad \in [a,b]$$

Now, the random number generator we use in our code generates numbers from an inherently uniform distribution. The distribution gives a domain $D = \{x\}$ with numbers in the interval $[0,1]$. When $a = 0$, $b = 1$, and $x \in [0,1]$, the probability density simply turns into:

$$p(x) = 1$$

The interval we are interested in is not really $x \in [0,1]$, rather it is $[-2,2]$ (see justification of this interval under the Gaussian Quadrature results section). We therefore perform a mapping onto this interval, resulting in the need to multiply with the Jacobi determinant, which is defined for our mapping as

$$\texttt{jacobi} = \prod_{i=1}^{d} (b_i - a_i)$$

In our case the dimension of the integral is $d = 6$, and the jacobi is therefore:

$$\texttt{jacobi} = (b_i - a_i)^6$$

In our code, this multiplication is introduced at the very end of our calculations.

The program `MC_1.cpp` calculates the integral using Brute Force Monte Carlo Integration. The program generates random numbers for the coordinates with a uniform distribution from the random number generator, and maps them with the uniform distribution, $p(x) = 1$. The coordinates are

then run through the function, `func_brute`, summed, and averaged to obtain the numerical integral results, as shown in the results section below.

The uniform sampling distribution is not the smartest way to do the calculation, as we will see in the next section where we are using importance sampling with an exponential distribution.

2.2.2 Monte Carlo with Importance Sampling

The more advanced version of the Monte Carlo method is the importance sampling method, which implements two new features compared to the brute force method, namely importance sampling and a change of variables.

Importance sampling samples points from the probability distribution described by the function f so that the points are concentrated in the regions that make the largest contribution to the integral. Since there is an exponential expression in the integrand, an exponential distribution $p(x)$ should be similar to the original expression and make a good fit. The exponential distribution $p(y) = exp(-y)$ is manipulated to:

$$y(x) = -\ln(1 - x)$$

Since the integrand has a factor of -4 in the exponential expression, we have added a factor of $1/4$ in front of the y-expression, yielding:

$$y(x) = -\frac{1}{4}\ln(1 - x)$$

The change of variables is implemented with a transform from cartesian to spherical coordinates as described in the Gaussian Laguerre method section, but here without the exponential component in g:

$$g(r_1, r_2, \theta_1, \theta_2, \phi_1, \phi_2) = \frac{r_1^2 r_2^2 \sin\theta_1 \sin\theta_2}{r_{12}}$$

The absence of exponential factors in g are due to the exponential factor being taken care of by the exponential PDF.

The exponential distribution is only implemented on the radial variables $r1$ and $r2$, while leaving the angles θ and ϕ with the uniform distribution. These new coordinates are then evaluated with `func_important_samp`, summed and averaged to an $f(x)$, like in the brute force method described

earlier. The variance and the standard deviation are calculated the same way as in the brute force method.

Like any other change of variables, we need to multiply the sum of $f(x)$ with the Jacobi determinant. For the transformation to spherical coordinates, the Jacobi determinant is defined as:

$$\texttt{jacobi} = 4 * pow(pi, 4)/16.0$$

The results of the importance sampling can be seen in table 5.

To make our program even more efficient, we have used the library `openmp` to parallelize our for-loop for the Monte Carlo Integration. This is done by having our loop inside an `openmp`-environment. Inside the environment, the loop is split into four segments, calculated by separate CPU threads. The program can be read in full in the file `mc_2_para.cpp`, and the runtime results for this procedure are shown in table 6.

3 Results and discussion

3.1 Gaussian Quadrature

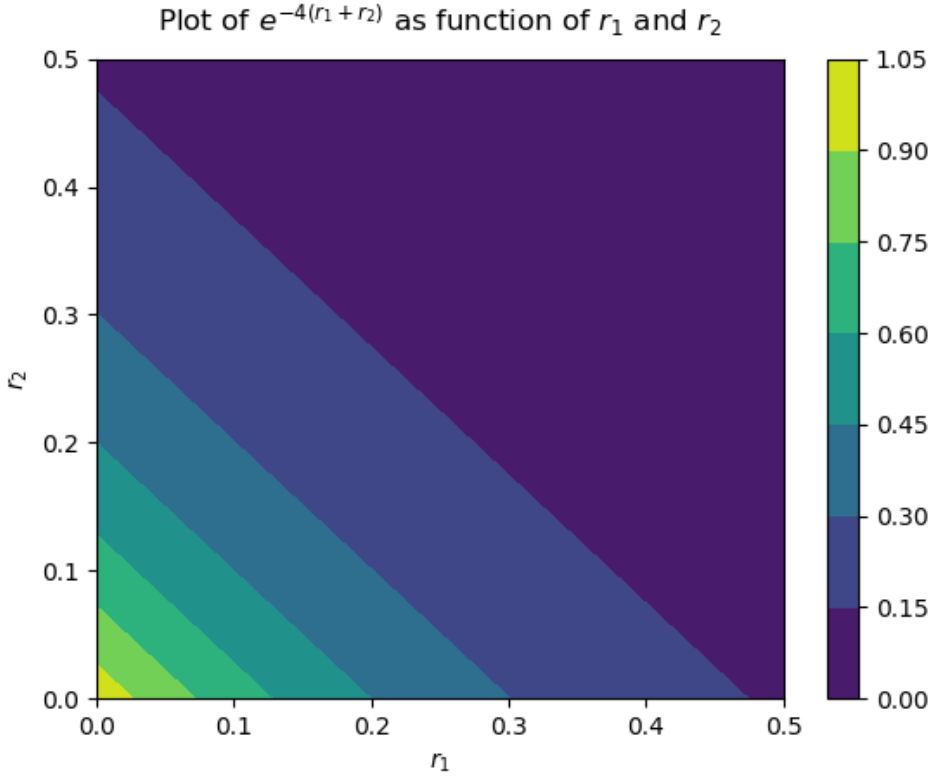

Figure 1: *Plot of* $f = e^{-4(r_1+r_2)}$ *as function of* r_1 *and* r_2*. The values of* f *are shown by colour, as indicated by the colour bar.*

The plot in figure 1 indicates that f should go to zero somewhere in the interval $r_i \in [0.5, 1]$. If we were to choose x, y and z-coordinates in the interval $[-\lambda, \lambda] = [-1, 1]$, the maximum value of r_i is $\sqrt{3}$, which seems sufficient. However, to be on the safe side, we have also run our program for the intervals $[-2, 2]$ and $[-3, 3]$.

The results of our Gaussian Quadrature with Legendre polynomials are shown in table 1. From the table one can discern that the smallest error, 0.0028, occurs with $\lambda = 2$ and $N = 50$. It is to be expected that the highest N gives the best precision. With regards to the choice of λ, there are two things to consider. Firstly, we need an interval that is wide enough to include all significant contributions to the integral. Secondly, we need to keep it narrow enough so that our integration points end up being where these contributions are found. This second consideration would be the

		Computed Value	Error
L = 1	N = 10	0.15109	0.04167
	N = 20	0.16142	0.03135
	N = 30	0.16321	0.02955
	N = 40	0.16383	0.02894
	N = 50	0.16411	0.02865
L = 2	N = 10	0.12983	0.06293
	N = 20	0.17707	0.01570
	N = 30	0.18580	0.00697
	N = 40	0.18867	0.00410
	N = 50	0.18995	0.00281
L = 3	N = 10	0.07198	0.12079
	N = 20	0.15614	0.03663
	N = 30	0.17728	0.01548
	N = 40	0.18442	0.00835
	N = 50	0.18756	0.00521

Table 1: *Results for integration with Gaussian quadrature and Legendre polynomials. The integration points are in the interval* $[-L, +L]$

reason why $\lambda = 2$ yields better results than $\lambda = 3$. However, it could be that if we included more integration points (that is, a larger value for N) we would get better results for the wider interval of $\lambda = 3$, since we hopefully would include more of the significant contributions to the integral.

Table 2 shows the results of our program when we have changed coordinates and used Laguerre polynomials for the radial part. As described above, the approximation to infinity, λ, does not factor in here. Note that already at $N = 20$, our error is 0.002, which is the same as we got for $N = 50$ in the Legendre case. Comparing runtimes (table 3), we see that this difference in N corresponds to a difference of approximately factor 50 in runtime, meaning that we can get good results much quicker when using Laguerre polynomials. Keep in mind that this is due to the fact that we are integrating an exponential function - Laguerre polynomials are not always better than Legendre, it depends on what your integrand is. Note also that for a particular value of N, the Laguerre solver needs much more time than the Legendre solver.

In figure 2 we have plotted the runtimes as functions of N logarithmically. This lets us read

	Computed Value	Error	Runtime
N = 10	0.17708	0.01568	0.35900
N = 20	0.19479	0.00202	20.45900
N = 30	0.19478	0.00201	224.38000
N = 40	0.19467	0.00190	1243.70000
N = 50	0.19452	0.00175	4710.40000

Table 2: *Results for integration with Gaussian quadrature, using Laguerre polynomials for the radial part, and Legendre polynomials for the angular part. Runtimes are given in seconds.*

N	Legendre Runtimes	Laguerre Runtimes
10	0.07	0.36
20	4.41	20.46
30	46.94	224.38
40	240.59	1243.70
50	905.59	4710.40

Table 3: *Runtimes in seconds, for Gaussian Quadrature with Legendre and Laguerre polynomials, respectively.*

of the N-dependancy of the runtimes as the slope of our interpolated line. Since the line goes as $\log_{10}(t_R) \sim 6\log_{10}(N)$, our actual runtime scales as $t_R \sim N^6$. This is to be expected, since our calculation consists of six nested loops of length N. The increase in runtime for the Laguerre solver should mainly be because the integrand in spherical coordinates contains more floating point operations.

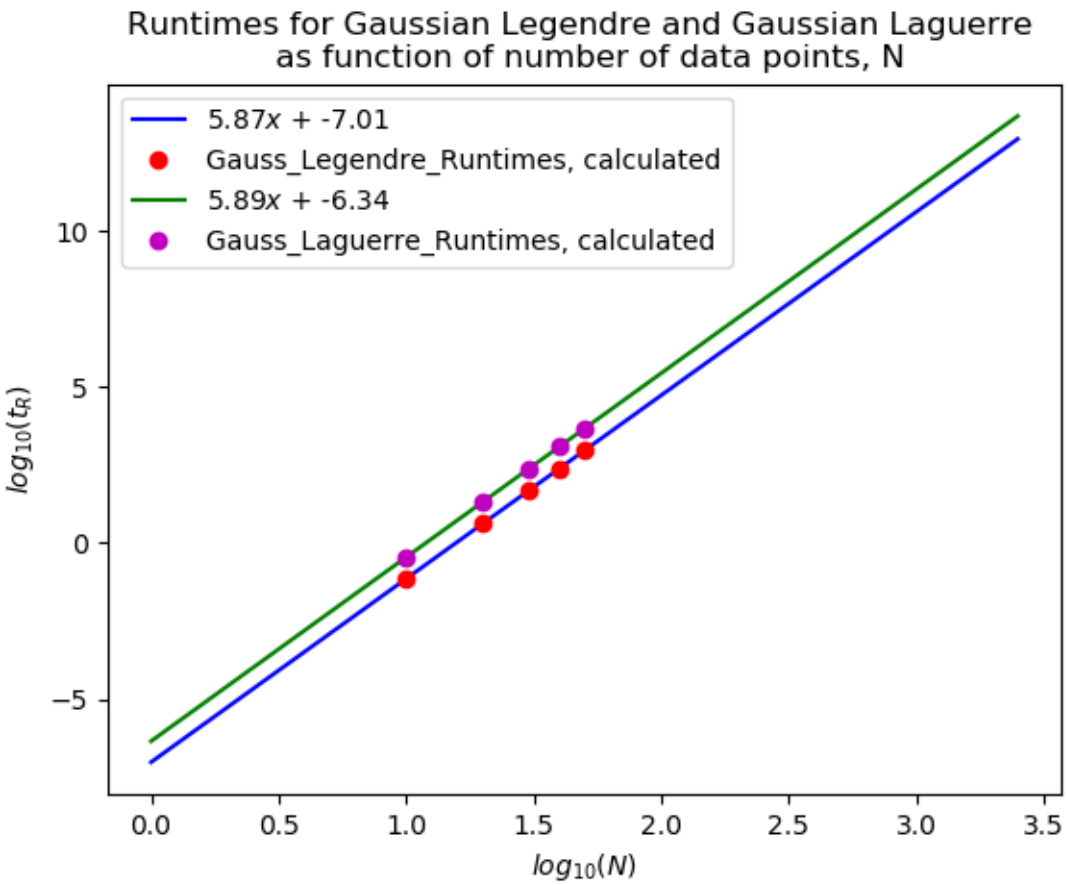

Figure 2: *Logarithmic plot of interpolated runtimes for Gaussian Quadrature with respectively Legendre and Laguerre polynomials.*

3.2 Monte Carlo Integration

The results for Brute Force Monte Carlo and Monte Carlo with Importance Sampling are shown in tables 4 and 5 respectively. In this study, we chose not to focus on the standard deviation as it is simply calculated from the variance, as shown in methods. In contrast to the variance, runtimes, and absolute error, which have been averaged over ten rounds in both tables, the computed value for the integral is not averaged, rather it is the last computed value out of the ten.

For the brute force method, we see that the variance is converging towards approximately 0.023 and that the error is approaching an order of 10^{-4} for $N = 10^8$. Comparing these results with the results for importance sampling in table 5, we see that the brute force method is more than twice as fast, but also gives a more significant error than the importance sampling, which gives an error of order 10^{-5}. The longer runtime and smaller error are somewhat expected as the nature of the importance sampling

suggests a better accuracy and a smaller variance through a more extensive calculation method and thus a longer runtime. Note however, that Monte Carlo with Importance Sampling with $N = 10^6$ yields equally good results as the Brute Force method with $N = 10^8$. Comparing the runtimes for these two values, we see that Importance Sampling runs about fifty times as fast.

The variance is, strangely enough, smaller for the brute force method than for importance sampling. The variance for the importance sampling method is approaching 0.044, almost twice as much as for Brute Force Monte Carlo, but this seems to be consistent with other results in the literature, i.e. Hjorth-Jensen (2015).

	Computed Value	Average Error	Average Variance	Average Runtime
N = 1e+03	0.89427	0.1317570	1.7430e−02	0.00010
N = 1e+04	0.11525	0.0697091	1.0759e−03	0.00170
N = 1e+05	0.17025	0.0231700	2.6523e−02	0.01520
N = 1e+06	0.19929	0.0091984	3.0791e−02	0.18150
N = 1e+07	0.19361	0.0018188	2.0608e−02	1.70080
N = 1e+08	0.19152	0.0007018	2.3278e−02	24.95600

Table 4: *Results for Brute Force Monte Carlo Integration, in the interval* $[-2, 2]$. *The average values are aqcuired from* 10 *runs for each* N. *The computed value for the integral is the lastly calculated value out of the ten. Runtimes are given in seconds.*

	Computed Value	Average Error	Average Variance	Average Runtime
N = 1e+03	0.20785	0.0151080	3.5463e−02	0.00010
N = 1e+04	0.18423	0.0058078	3.5042e−02	0.00650
N = 1e+05	0.19743	0.0038186	4.6489e−02	0.05610
N = 1e+06	0.19258	0.0007887	4.1822e−02	0.49250
N = 1e+07	0.19310	0.0002734	4.3766e−02	4.85740
N = 1e+08	0.19273	0.0000837	4.3753e−02	57.01820

Table 5: *Results for Monte Carlo Integration with Importance Sampling. The average values are aqcuired from* 10 *runs for each* N. *The computed value for the integral is the lastly calculated value out of the ten. Runtimes are given in seconds.*

Table 6 shows the runtimes for our parallelized version of the Importance Sample code. We choose

to show only the results for higher values of N, since these tend to be more trustworthy (the runtimes for lower values are very small, and prone to significant changes from one run to the next). The table suggests that the speed up of the parallelized code is somewhere between four and five times. The expected result here is a speed up of four times, since we are running on four CPU threads. It is then a bit curious that we seem to have more than optimal improvement on our parallelized runtimes. In way of explanation, we have seen that runtimes might actually vary quite a bit, even for bigger N. Also, our parallelized and non-parallelized codes have been run on two different computers, where one might be slightly faster than the other. In short, our result is plausible, but prone to influences from a number of disturbing factors.

N	No parallelization	Parallelization	pR / npR
	Runtimes, npR	Runtimes, pR	
100000	0.06	0.01	0.20
1000000	0.49	0.11	0.23
10000000	4.86	1.11	0.23
100000000	57.02	11.64	0.20

Table 6: *Runtimes in seconds for Monte Carlo Integration with Importance Sampling, both with and without parallelization. Note how the runtime is greatly reduced by the parallelization, due to the four CPU threads.*

4 Conclusion

In this report we have solved the six dimensional integral

$$I = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-4(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}$$

using four different numerical integration methods - Gaussian Quadrature with Legendre polynomials, Gaussian Quadrature with Laguerre polynomials, Brute Force Monte Carlo Integration, and Monte Carlo Integration with Importance Sampling. The theory behind the methods and their implementation have been explained briefly, and we have gone on to discuss their precision and efficiency.

First we showed how Gaussian Quadrature with Legendre polynomials offered only precision to the order of 10^{-2}. Then we performed the integration with Laguerre polynomials for the radial part, and showed how this gave better results for less integration points (and hence a lower runtime). We also noted how the runtime for Gaussian Quadrature scales as N^6, whcich makes sense, considering the six nested for loops in the algorithm.

We then shifted our focus to Monte Carlo Integration, and showed how the precision was much better than with Gaussian Quadrature, to a factor of 100 or even 1000. For approximately the same runtime, Brute Force Monte Carlo yielded an error of $\sim 10^{-4}$, and the Importance Sampling algorithm yielded an error of $\sim 10^{-5}$. Comparing the two different Monte Carlo methods, we see that with Importance Sampling the calculation becomes much more accurate for much fewer integration points. A curious find is that the Importance Sampling algorithm had a larger value for the variance than the Brute Force algorithm, approximately by a factor two.

Finally, we showed how parallelizing our code using Open MP provided faster run times, by a factor of between four and five.

Further research could involve testing the integration methods on lower dimensional integrals, and see if the superiority of Monte Carlo integration also applies there. Also, one could run the Monte Carlo algorithms for a larger number of N, and see if the results could get even better. Another thing to look at could be to look for reasons for the greater variance yielded by Importance Sampling. Alternatively, one could refine the program to make more accurate measurements of runtime, and so get better estimates for the improvement provided by `openmp`.

References

Hjorth-Jensen, M (2015) Lecture Notes. Department of Physics, University of Oslo. Downloaded 07.09.2019 from:
https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Lectures

Developed Code files

This report, code files and selected output files can be found in github repository at: https://github.com/HolyWaters95/FYS4150_Project3

- `Gaussian_Laguerre.cpp`
- `Gaussian_Legendre.cpp`
- `mc_1.cpp`
- `mc_2.cpp`
- `MC_Functions.cpp`
- `p3_functions.cpp`
- `Project_3_plotting.py`
- `Py_Functions.py`