

FYS4150 - Project 3

Kristoffer Langstad
krilangs@uio.no

Abstract

In this project we want to solve an integral of the quantum mechanical expectation value of the correlation energy between two electrons which repel each other via the classical Coulomb interaction. For this integral we neglect the normalization factor. This integral can be solved in closed form and has an answer $(5\pi^2/16^2)$ that we will try to get from our numerical methods. We will use different numerical integration methods to see the difference in both the methods, and the difference in brute force calculation versus more thought out methods. Then we will compare the convergence to the answer of the methods as of how many mesh points are needed to get at the level of third leading digit, and the CPU time used for the different methods to reach this answer. First we use a brute force Gaussian-Legendre quadrature integration to solve the integral. Then we improve the method by using Gaussian-Laguerre and change to spherical coordinates. The last three methods are variants of Monte Carlo integration; where we first use brute force with a uniform distribution, then improve by using importance sampling with an exponential distribution and spherical coordinates and lastly we will use parallelization of Monte Carlo. The brute force Gaussian-Legendre method is as we would expect quite slow, and we reach the analytic answer at third leading digit around 15 mesh points with CPU time 1 102 ms. For the improved Gaussian-Laguerre method we reach the third leading digit around 10 mesh points with CPU time 309 ms. So the improved method is slower for the same number of mesh points, but seems to be more accurate and reaches the goal for fewer mesh points. For the brute force Monte Carlo we reach the third leading digit around 10^6 mesh points with CPU time 411 ms. For the improved Monte Carlo we reach the goal after around 10^5 mesh points with CPU time around 74 ms. So we can conclude that the Monte Carlo is the better method when we reach high number of mesh points compared to the Gaussian quadrature methods. For high number of mesh points the Monte Carlo has similar accuracy as the Gaussian quadrature methods, while being much faster. When we parallelize the Monte Carlo with the best tested compiler flag **-Ofast** in C++, we get similar results as for the improved method only with 10^4 mesh points and faster CPU time around 4 ms.

1 Introduction

The efficiency of numerical integration methods have a great importance to us when solving physical problems. Today, there are many numerical integration methods that can be used. So to find the optimal method to solve our problem is important. That is why

we will study different numerical methods to see the importance of optimization of the methods versus brute force integrations.

In this project we solve a six-dimensional integral, which is used to determine the ground state correlation energy between two electrons in a helium atom. For this project we will first look at different versions of Gaussian quadrature before we move over to different Monte Carlo integration methods. The Gaussian quadrature integration methods are mostly used in low-dimensional cases, while Monte Carlo are mostly used in multidimensional cases. The numerics of this project is done in C++ with QT Creator on Windows.

In the methods section we look at the theory of the physical problem and the different numerical algorithms we are using. For the integral we are solving in closed form, the quantum mechanical expectation value of the correlation energy between two electrons without a normalization factor, we compare our numerical results with a known solution. We will then compare the results we get between the different numerical integration methods to see the difference between brute force methods and more thought out methods. The methods we are using are brute force Gaussian-Legendre Quadrature, Gaussian-Laguerre quadrature in spherical coordinates, brute force Monte Carlo, improved Monte Carlo by use of importance sampling and parallelized Monte Carlo. In the results we present and compare our results of the numerical integration methods and discuss the methods used. Then in the conclusion section we come up with a conclusion to the project, and which method is the best for this task.

2 Method

2.1 Integration problem

We first look at a six-dimensional integral that determines the ground state correlation energy between two electrons in a helium atom. Then we assume that the wave function for each of the electrons can be modeled like a single-particle wave function of an electron in the hydrogen atom. The single particle wave function for an electron i in the 1st state for a dimensionless variable $\mathbf{r}_i = x_i\mathbf{e}_x + y_i\mathbf{e}_y + z_i\mathbf{e}_z$ and $r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$, can be expressed as

$$\psi_{1s}(\mathbf{r}_i) = e^{-\alpha r_i}. \quad (1)$$

Then we fix the constant $\alpha = 2$, which represents the charge of the He-atom ($Z=2$).

The wave function for two electrons can then be written as the product of two single-particle (1s) wave functions as

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = e^{-\alpha(r_1+r_2)}. \quad (2)$$

This two interacting electrons problem in the helium atom has no analytical solution to the Schrödinger equation.

So the integral we need to solve is the quantum mechanical expectation value of the correlation energy between two electrons which repel each other via the classical Coulomb interaction:

$$\left\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \right\rangle = \int_{-\infty}^{\infty} e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} d\mathbf{r}_1 d\mathbf{r}_2 \quad (3)$$

The wave function is not normalized, so there is a normalization factor that we neglect in this project. The above integral can be solved in closed form with answer $\frac{5\pi^2}{16^2}$, which is used to compare with later.

2.2 Gaussian-Legendre quadrature

First we look at the brute force method for Gaussian-Legendre quadrature. The full explanation of the derivations are found in Hjorth-Jensen [1] (ch. 5.3). The main idea is to approximate the integral with a weight function W as

$$I = \int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=1}^N w_i g(x_i), \quad (4)$$

where w and x are weights and g is smooth. The weights are obtained through orthogonal polynomials. In this case we use Legendre polynomials which are orthogonal on some interval $x \in [-1, 1]$. This polynomial is represented through polynomial division as

$$P_{2N-1}(x) = L_N(x)P_{N-1}(x) + Q_{N-1}(x),$$

where $L_N(x)$ is the Legendre polynomial of order N and $P_{N-1}(x)$ and $Q_{N-1}(x)$ are some polynomials of degree $N - 1$. For Legendre polynomials we have the weight function as $W(x) = 1$.

For the integral we want to solve in equation 3, we have the limits $-\infty$ and ∞ . To be able to solve this numerically we have to rewrite the integral with appropriate limits $[a, b]$. So we use a change of variable

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

to rewrite the integral. Since we integrate a 6-dimensional using Cartesian coordinates, then we need to have that the limits of all the 6 coordinate integrals are $a = -\lambda$ and $b = \lambda$, such that for $\lambda \rightarrow \infty$ we get that the single-particle wave function

$$e^{-\alpha r_i} \quad (5)$$

is more or less zero at $r_i \approx \lambda$. So by plotting the single-wave function we can find the satisfactory limits λ .

The integration weights are found by inversion of a matrix involving the orthogonal polynomials used such that the weights w_i and mesh points x_i are zeros of the polynomial. The algorithms for finding these are presented in Press et al. [2], and implemented into the **weights.cpp** file in the GitHub repository in Appendix A. For this brute force Gaussian-Legendre quadrature case, we use the *gauleg*-function to calculate the weights and mesh points with finite integration limits. The calculation of the integral in equation 3 is done as an integration sum for approximating the integral as a sixfold loop, with $\alpha = 2$, for all the six coordinates in the *gauleg_quad*-function in the **main.cpp** program:

$$\begin{aligned} I &= \int_{-\infty}^{\infty} e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} d\mathbf{r}_1 d\mathbf{r}_2 \\ &\approx \sum_{i,j,k,l,m,n=0}^{N-1} w_i w_j w_k w_l w_m w_n f(x_{1,i}, x_{2,j}, y_{1,k}, y_{2,l}, z_{1,m}, z_{2,n}) \end{aligned} \quad (6)$$

The function f is implemented in the *int_function* which calculates

$$f = e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}. \quad (7)$$

In this case we could end up in the situation with a singularity in the fraction part for $|\mathbf{r}_1 - \mathbf{r}_2| = 0$. To account for this we say that if this integrand is smaller than a tolerance 10^{-10} , we just return $f = 0$. If the integrand is bigger than the tolerance, then we just return f in equation 7. Then we run the program for different number of mesh points N to reach our goal of the numerical results to converge to the analytical at the level of the third leading digit.

2.3 Gaussian-Laguerre quadrature

The brute force Gaussian-Legendre quadrature give us quite unsatisfactory results. So to get better results, we improve the method we used. So we will change to spherical coordinates and now use Laguerre polynomials defined for $x \in [0, \infty)$ (Hjorth-Jensen [1] ch. 5.3.5). The integral is now on the form

$$I = \int_0^\infty f(x) dx = \int_0^\infty x^\alpha e^{-x} g(x) dx, \quad (8)$$

where the weight function is now $W(x) = x^\alpha e^{-x}$. When using Laguerre polynomial we get that the exponential is absorbed in the weights, such that we get a more accurate approximation. The weights and mesh points are derived and found like for the Gaussian-Legendre case, and are once again implemented in the **weights.cpp** file in the *gauss_laguerre* function with the algorithms from Press et al. [2].

Now we have to change to spherical coordinates such that $(x, y, z) \rightarrow (r, \theta, \phi)$ where $r \in [0, \infty)$, $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. We then get

$$\begin{aligned} d\mathbf{r}_1 d\mathbf{r}_2 &= r_1^2 dr_1 r_2^2 dr_2 d\cos(\theta_1) d\cos(\theta_2) d\phi_1 d\phi_2 \\ &= r_1^2 dr_1 r_2^2 dr_2 \sin(\theta_1) d\theta_1 \sin(\theta_2) d\theta_2 d\phi_1 d\phi_2, \end{aligned}$$

and

$$\frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} = \frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\beta)}}$$

with

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

Then we use change of variable with $s = \alpha r$, $du = \alpha dr$. The integral then looks like this

$$I = \frac{1}{32\alpha^5} \int_0^\pi \int_0^\pi \int_0^{2\pi} \int_0^{2\pi} \int_0^\infty \int_0^\infty f ds_1 ds_2 d\theta_2 d\theta_1 d\phi_1 d\phi_2, \quad (9)$$

with

$$f = f(s_1, s_2, \theta_2, \theta_1, \phi_1, \phi_2) = \frac{s_1^2 s_2^2 \sin(\theta_1) \sin(\theta_2) e^{-(s_1+s_2)}}{\sqrt{s_1^2 + s_2^2 - 2s_1 s_2 \cos(\beta)}}.$$

$\frac{1}{s_{12}} = \frac{1}{s_1^2 + s_2^2 - 2s_1s_2 \cos(\beta)}$ is calculated in the *int_func_spherical*-function in the **main.cpp** file. There we once again have to take into consideration the singularity cases when $\frac{1}{s_{12}} = 0$. So when $\frac{1}{s_{12}}$ is smaller than the tolerance 10^{-10} , we return the value 0. If it is bigger than the tolerance, we return $\frac{1}{\sqrt{s_{12}}}$.

The integral in spherical coordinates (eq. 9) is calculated in the *gaulag_quad*-function in the **main.cpp** file. For the angular parts we use the *gauleg*-function to calculate the weights and mesh points with limits for θ and ϕ . For the radial part we use the *gauss_laguerre*-function, but this Gauss-Laguerre integration that calculates the mesh points and weights starts the arrays differently than for Gauss-Legendre. So for the radial parts we declare the arrays of lengths $N + 1$ instead of length N for Legendre.

Then we run the program for different number of mesh points N to reach our goal of the numerical results like for the brute force Gaussian-Legendre method. Then we compare our numerical results and the CPU times we get for doing the calculations for the two Gaussian quadrature methods.

2.4 Monte Carlo integration

Monte Carlo integration methods are widely used for computing multidimensional integrals. This is often described as a statistical simulation method that utilizes sequences of random numbers to perform simulations (Hjorth-Jensen [1], p. 337). Further information on Monte Carlo methods are found in Hjorth-Jensen [1] ch.11. We want to evaluate the integrated function at N random points from a discretized probability distribution function (PDF). The PDF gives us the probability/ relative frequency that the stochastic variable X occur on the given domain such that

$$P(a \leq X \leq b) = \int_a^b p(x)dx,$$

where $p(x)$ is a continuous function that gives us the probability density. For this brute force Monte Carlo we have that

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

is a uniform probability distribution with $\theta(x) = 0$ for $x < 0$ and $\theta = \frac{1}{b-a}$ for $x \in [a, b]$. Since the uniform distributions only returns values between zero and one, the integral can then be approximated as the expectation value at random points of the function values as

$$I \approx \langle (b-a)f(x) \rangle = (b-a) \int_a^b f(x)p(x)dx \approx \frac{b-a}{N} \sum_{i=0}^{N-1} f(x_i).$$

Then we can calculate an important aspect with Monte Carlo integration, the variance σ^2 . This is used to estimate the accuracy of the integration as

$$\sigma_f^2 = (b-a) \left(\langle f^2(x) \rangle - \langle f(x) \rangle^2 \right) = (b-a) \left(\frac{1}{N} \sum_{i=0}^{N-1} f^2(x_i) - \left(\frac{1}{N} \sum_{i=0}^{N-1} f(x_i) \right)^2 \right). \quad (10)$$

From the variance and the central limit theorem (Hjorth-Jensen [1] p.343) we have that the standard deviation is given as

$$\sigma_N^2 \approx \frac{1}{N} \left(\langle f^2(x) \rangle - \langle f(x) \rangle^2 \right) = \frac{\sigma_f^2}{N}.$$

For solving the 6-dimensional integral in equation 3 we use the same limits as for the Gaussian-Legendre method. We then end up with solving a 6-dimensional expectation value with brute force for uniformly distributed coordinates in the limit $[-\lambda, \lambda]$ as

$$I \approx \frac{(b-a)^6}{N} \sum_{i=0}^{N-1} f(x_{1,i}, x_{2,i}, y_{1,i}, y_{2,i}, z_{1,i}, z_{2,i}) \quad (11)$$

This is done in the *monte_carlo*-function in the **main.cpp** program.

For the random numbers generation we use the pseudo-random number engine Mersene Twister 19937 (64 bit) generator (*mt19937_64*), in combination with a non-deterministic random number generator (*random_device*) to make sure that the random number seed changes as the program goes on. Then for the uniform distribution we use *uniform_real_distribution* to make uniform numbers in the domain $[a,b]$. Since we use the same limits as for the Gaussian-Legendre quadrature, we use the same *int_function* in Cartesian coordinates for computing the function f in equation 11. After evaluating the integral, we calculate the variance in 6-dimensions. We also take the CPU time of the calculations to compare with other methods.

2.5 Improved Monte Carlo integration

Now we want to improve on the brute force Monte Carlo integration. This we do by introducing importance sampling, exponential distribution and changing to spherical coordinates. For the spherical coordinate change in the Gaussian-Laguerre quadrature we got an exponential expression in the integrand. For this factor we will use exponential distribution denoted as $p(y) = e^{-y}$. Then we will use that the probability is conserved when performing a change of variable. So for a change of variable we now get

$$p(y)dy = e^{-y}dy = p(x)dx = dx$$

for the uniform PDF $p(x)$ defined in $x \in [0,1]$. Then we integrate this new change of variable $x \rightarrow y$

$$x(y) = \int_0^y e^{-y'} dy' = 1 - e^{-y}.$$

By inverting this we now get from the uniform PDF $y(x) = -\ln(1-x)$ to exponential PDF. The exponential part can now be absorbed into the expectation value to get

$$I = \int_0^\infty f(r)dr = \int_0^\infty \frac{f(r)}{p(r)} p(r)dr = \int_0^1 g(r(x))dx \approx \frac{1}{N} \sum_{i=0}^{N-1} g(r(x_i)).$$

Here we use change of variables for the radial distance $r = y \in [0, \infty)$ with $p(r) = e^{-r}$ and $p(r)dr = dx$.

Now we can solve the integral numerically in the function *mc_improved* in the **main.cpp** program. We will now draw uniform distributions for the angular coordinates θ_i and ϕ_i with finite limits $[0, \pi]$ and $[0, 2\pi]$ respectively. For the radial coordinates r_i , we will use the importance sampling with exponential distribution. We use the same type of random generators as for the brute force, but now we will also implement the exponential distribution by *exponential_distribution*. From the change of variables we also get a Jacobi determinant as a factor $\frac{\pi^4}{8\alpha^5}$. For calculating the integrand we use the same *int_func_spherical* as for the Gaussian-Laguerre quadrature. The integral to be evaluated numerically is now:

$$I \approx \frac{\pi^4}{8\alpha^5} \frac{1}{N} \sum_{i=0}^{N-1} \frac{(r_{1,i})^2 (r_{2,i})^2 \sin(\theta_{1,i}) \sin(\theta_{2,i})}{\sqrt{(r_{1,i})^2 + (r_{2,i})^2 - 2r_{1,i}r_{2,i} \cos(\beta)}} \quad (12)$$

We also calculate the variance, but this calculation is equal to the brute force variance (eq. 10). Again, we will compare our numerical results with the analytical and the other methods after evaluating the final integral (eq. 12). Then we compare the variances and CPU times for different number of mesh points with the results we got in the other methods.

2.6 Monte Carlo parallelized

For the last part we will parallelize the improved Monte Carlo integration using the library openMP and various compiler flags. This is done in the *mc_parallelization*-function in the **main.cpp** program. The only difference to this and the improved Monte Carlo is that we have added two lines for initializing the openMP parallelization on the for-loop. Before we evaluate the integral, we try to find the optimal speed-up by testing different number of threads and compiler flags. After finding the optimal compiler flag and thread number, we evaluate the integral in equation 12. Then we do the same comparing as we have done for the other methods.

3 Results

The code is run in C++ with QT Creator on a 64-bit Windows 10 laptop with 8GB RAM. For the random generator for the Monte Carlo methods we use the Mersenne Twister generator (64 bit) **mt19937_64**, and we make sure that the random generator seed change so we don't use the same seed multiple times. For the openMP we use 4 threads.

For the brute force Gaussian-Legendre quadrature we first look at the figure of the single-wave function (eq. 5) in Figure 1 to determine the integration limits from $\pm\infty$ to $\pm\lambda$. We find the integration limits where the function is more or less zero. That is why we choose $\lambda = 1.5$ as our limit, since this seems like satisfactory limits for our case. This choice is also the same for the brute force Monte Carlo calculation. In Table 1 we have the results for the Gaussian-Legendre quadrature. There we compare the numerical results we get with the analytical answer, where we look at the absolute difference between them, and the CPU time in milliseconds of the numerics. We see that after 15 mesh points with CPU time about 1.1 second, we have reached our goal for the numerical results to converge to the analytical at the level of third leading digit. In the table we have run every

5 mesh points up to 30 mesh points for comparison we other methods. We can see that the difference between the numerical and analytical answers does not always decrease, but seem to oscillate a bit up and down as for 15-20-25 mesh points. This is because the brute force method is not a very good method. Overall the difference is getting smaller, while the CPU time is getting higher. For bigger mesh points than 30, the difference decreases less and less, while the CPU time increases a lot each time. That is why we have stopped at 30 mesh points, since the results does not get much better and the time of the calculations takes a lot of time.

For the improved Gaussian quadrature with Laguerre polynomial and in spherical coordinates, we see the results in Table 2. There we see the same type of results as for the brute force case. In this improved case we can see from the table that we reach three leading digits now within 10 number of mesh points with around 310 ms. So the improved case seem to be better than the brute force since we reach our goal within both fewer mesh points and in less CPU time. If we compare the difference of the numerical and analytical results, we now see that the differences decrease more stably than for the brute force. The differences for the corresponding mesh points are better for the improved than for the brute force. So we can conclude that the accuracy of the improved Gaussian quadrature is better than the brute force. The CPU time on the other hand is worse for the corresponding mesh points. The improved method takes longer to calculate the numerical answers for the same number of mesh points as the brute force takes. For 30 mesh points we see that the improved method gives more accurate results, but takes almost 5 times as long as the brute force for the same number of mesh points.

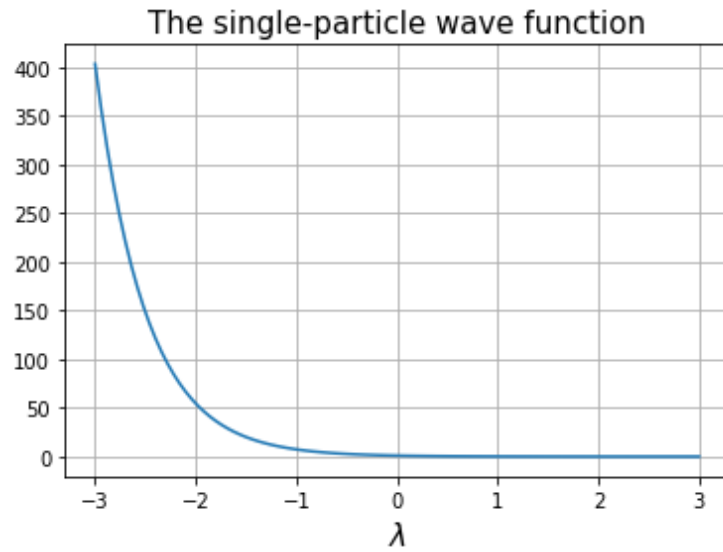


Figure 1: This is a figure of the single-wave function in equation 5. This figure is used to determine the appropriate integration limits λ . Here we see that the function is more or less zero for the chosen interval we look at. So we choose the integration limits of the function as $\lambda = 1.5$.

| Mesh points (N) | Numerical | Analytical | Difference | CPU time [ms] |
|-----------------|-----------|------------|------------|---------------|
| 5 | 0.308692 | 0.192766 | 0.115926 | 1 |
| 10 | 0.154052 | 0.192766 | 0.0387138 | 97 |
| 15 | 0.187130 | 0.192766 | 0.00563558 | 1102 |
| 20 | 0.180469 | 0.192766 | 0.0122967 | 5729 |
| 25 | 0.185593 | 0.192766 | 0.00717284 | 22936 |
| 30 | 0.185011 | 0.192766 | 0.00775459 | 71586 |

Table 1: Table for brute force Gaussian-Legendre quadrature integration results with integration limit $\lambda = 1.5$. The numerical results are what we get for the given number of mesh points. The differences are means of a number of sets of absolute differences between the numerical and analytical results. The absolute difference decreases and the CPU time, in milliseconds, increases as the mesh points increases. We see that the difference does not decrease all the time as it converges towards zero, but seem to go up and down as the mesh points increases.

| Mesh points (N) | Numerical | Analytical | Difference | CPU time [ms] |
|-----------------|-----------|------------|-------------|---------------|
| 5 | 0.174353 | 0.192766 | 0.0184125 | 6 |
| 10 | 0.186544 | 0.192766 | 0.00622209 | 309 |
| 15 | 0.189757 | 0.192766 | 0.00300848 | 4139 |
| 20 | 0.191066 | 0.192766 | 0.00169936 | 23450 |
| 25 | 0.191724 | 0.192766 | 0.00104137 | 96382 |
| 30 | 0.192099 | 0.192766 | 0.000666809 | 354909 |

Table 2: Table for Gaussian-Laguerre quadrature results for spherical coordinates. For this case we have a more stable and faster decrease in the absolute difference than in Table 1. So for the comparison between the numerical and analytical results goes, this improved Gaussian quadrature integration method with Laguerre polynomial and spherical coordinates is better than the brute force with Legendre polynomial. The CPU time, in milliseconds, on the other hand, is a lot higher in this case than for the brute force.

For the brute force Monte Carlo integration method we used the same $\lambda = 1.5$ integration limits as for the brute force Gaussian quadrature. In Table 3 we have the same type of results as for the Gaussian quadratures, but now we have included the variances as well. For the Monte Carlo method we have to use mesh points in powers of ten to be able to do the same analysis as before. For this method we get CPU time in microseconds instead of in milliseconds as for the other two methods. So even though the Monte Carlo method needs much higher number of mesh points, the CPU time is much quicker. For lower powers of ten for the mesh points, the numerical results are not as good as for the Gaussian quadratures with low mesh points. As the mesh points get bigger, 10^4 and up, the numerical results gets better. For around 10^6 mesh points with CPU time around 411 ms, we reach our goal for third leading digit of the analytical result. If we compare the CPU time for when we reach our goal with the other two methods, we see that the brute force Monte Carlo method is faster than the brute force Gaussian quadrature but slower than the improved Gaussian quadrature. The variance in the Monte Carlo method is not that low for low number of mesh points, but decreases as the number of mesh points increase. It goes a little up and down for $10^2 - 10^4$ like the difference does. This is the same behavior we see in the brute force Gaussian-Legendre quadrature. The CPU time for our chosen number of mesh points are still under half a second even for one million mesh points.

| Mesh points (N) | Numerical | Analytical | Difference | Variance | CPU time [μ s] |
|-----------------|-----------|------------|------------|-------------------------|---------------------|
| 10 | 0.366922 | 0.192766 | 0.174156 | 0.103689 | 0 |
| 10^2 | 0.0888911 | 0.192766 | 0.103875 | $1.38006 \cdot 10^{-3}$ | 5 |
| 10^3 | 0.462171 | 0.192766 | 0.269405 | 0.0737342 | 964 |
| 10^4 | 0.211879 | 0.192766 | 0.0191138 | $1.18462 \cdot 10^{-3}$ | 4376 |
| 10^5 | 0.178392 | 0.192766 | 0.0143734 | $1.17708 \cdot 10^{-4}$ | 42083 |
| 10^6 | 0.187786 | 0.192766 | 0.00497958 | $3.20708 \cdot 10^{-5}$ | 410568 |

Table 3: Table for brute force Monte Carlo integration results with integration limit $\lambda = 1.5$. For this method we see that for mesh points up to around 10^4 , the numerical results are worse than for the Gaussian quadrature methods. With higher mesh points than this, the numerical results becomes better. The CPU time, in microseconds, for this brute force Monte Carlo for reaching our goal, is better than the brute force Gaussian quadrature but worse than the improved. For around $N=30$ the CPU time were in seconds for Gaussian quadrature, but for this Monte Carlo method we use around half a second for $N=10^6$. The variance is not very low at first, but decreases as the mesh points increases.

For the improved Monte Carlo method we changed to spherical methods and used importance sampling. This gave us the results in Table 4. As for the brute force Monte Carlo case the improved gave a bad result for very low number of mesh points. For the improved we now get faster better results with increasing mesh points. So now we reach our goal within 10^5 number of mesh points with CPU time around 74 ms. As for the Gaussian quadratures, we now get that the improved method is more accurate for lower number of mesh points than the brute force. Again, the brute force is quicker than the improved if we only look at the same number of mesh points. The variance for the

improved Monte Carlo is lower than the brute force method. For higher mesh points the variance for the improved method decreases more than the brute force method.

| Mesh points (N) | Numerical | Analytical | Error | Variance | CPU time [μ s] |
|-----------------|-----------|------------|-------------|-------------------------|---------------------|
| 10 | 0.0809260 | 0.192766 | 0.111840 | $1.42888 \cdot 10^{-3}$ | 0 |
| 10^2 | 0.280483 | 0.192766 | 0.0877171 | $1.39761 \cdot 10^{-2}$ | 47 |
| 10^3 | 0.220718 | 0.192766 | 0.0279521 | $6.88288 \cdot 10^{-4}$ | 1258 |
| 10^4 | 0.205860 | 0.192766 | 0.0130945 | $1.48367 \cdot 10^{-4}$ | 8529 |
| 10^5 | 0.189125 | 0.192766 | 0.00364114 | $8.72397 \cdot 10^{-6}$ | 73793 |
| 10^6 | 0.192029 | 0.192766 | 0.000737069 | $1.05234 \cdot 10^{-6}$ | 693108 |

Table 4: Table for improved Monte Carlo integration results for spherical coordinates. This improved Monte Carlo method gives a better absolute difference between the numerical and analytical results for the same number of mesh points as the brute force Monte Carlo in Table 3. As in the Gaussian quadrature cases, the improved method with spherical coordinates seem to use longer CPU time than the brute force method. For the Monte Carlo methods, the time increase between the methods is not that high and are still in microseconds. The variance for the improved Monte Carlo is lower for lower mesh points, and decreases more as the mesh points increases. For $N=10^6$ mesh points the variance in the improved method is around 10 times lower than the brute force.

For the final improvement we parallelize the improved Monte Carlo method using openMP and different compiler flags. Since the CPU time is heavily dependent on the background processes of the CPU, the time it takes to run the calculations may vary from time to time. That is why we have taken an estimated average of the CPU times to try and find the fastest compiler flag to run with the parallelization. So in Table 5 we have tested various compiler flags on the improved Monte Carlo method with 10^6 number of mesh points. There we tested the C++ compiler flags -O1, -O2, -O3 and -Ofast with 4 number of threads for the openMP. From the table we get the best average CPU times for -Ofast with -O3 very close second. We get slightly different results each time we run the code since we have implemented the random number generator in such a way that the seed changes. In Table 6 we used compiler flag -Ofast for optimal speed-up. Since we have used parallelization on the improved Monte Carlo method, we get almost the same numerical results. The thing that changes the most is the CPU times. From the table we see that the optimal compiler flag and the parallelization makes the calculations a lot faster, especially for higher number of mesh points. For this method we now reach our goal of converging to third leading digit of the analytical answer for 10^4 number of mesh points with CPU time around 4 ms. For 10^6 mesh points the calculations now take around 280 ms, while it took around 693 ms for the improved method and around 410 ms for the brute force. So the parallelization with 4 number of threads and with optimal compiler flag give us the best accuracy of the numerical results and the best CPU time for calculating them.

| Compiler flags | -O1 | -O2 | -O3 | -Ofast |
|---------------------|--------|--------|--------|--------|
| CPU time [μ s] | 453982 | 432725 | 356079 | 348569 |

Table 5: Table for parallelized Monte Carlo integration results for 4 number of threads and $N = 10^6$ number of mesh points. Here we have compared the CPU time for different C++ compiler flags. Here we see that -Ofast seems to be the fastest in our case.

| Mesh points (N) | Numerical | Analytical | Error | Variance | CPU time [μ s] |
|-----------------|-----------|------------|------------|-------------------------|---------------------|
| 10 | 0.0430971 | 0.192766 | 0.149669 | $4.90837 \cdot 10^{-4}$ | 0 |
| 10^2 | 0.132091 | 0.192766 | 0.0606752 | $2.39315 \cdot 10^{-3}$ | 8 |
| 10^3 | 0.179565 | 0.192766 | 0.0132003 | $5.40965 \cdot 10^{-4}$ | 998 |
| 10^4 | 0.183520 | 0.192766 | 0.00924560 | $7.32929 \cdot 10^{-5}$ | 4036 |
| 10^5 | 0.198329 | 0.192766 | 0.00556329 | $2.35229 \cdot 10^{-5}$ | 31919 |
| 10^6 | 0.195437 | 0.192766 | 0.00267126 | $1.15460 \cdot 10^{-6}$ | 279342 |

Table 6: Table for parallelized Monte Carlo integration results for spherical coordinates and with 4 number of threads. This is run with the compiler flag **-Ofast** in C++, since this gave the best CPU times (as seen in Table 5). The results we get are close to the ones in Table 4 for the improved Monte Carlo, except that in this case the CPU times, in microseconds, are much better. This is exactly what we wanted with the parallelization. For chosen higher number of threads than what we have chosen, the CPU time decreases but then the numerical results get worse. For lower number of threads the numerical results are closer to the analytical, but has higher CPU time. So the chosen 4 number of threads is a compromise between the accuracy and CPU time. For the parallelization we reach our third leading digit goal around 10^4 with CPU time around 4 ms.

When running the C++ program directly in QT Creator we get different results and CPU times then if we run the program in a terminal window. This happens especially for the Monte Carlo methods. In the terminal window the results stay the same if we run the same task with the same values several times, while the results change a little when run directly in the QT Creator. In the terminal window, changing the number of threads does not seem to change anything for the parallelized Monte Carlo. When we run the program directly in QT Creator, then the results and CPU times change when changing the number of threads for openMP. That is why we choose 4 number of threads, since this seems to give a good compromise between the numerical results and the CPU time. For lower number of threads we get better numerical results, but longer CPU time. For higher number of threads we get better CPU time, but worse numerical results. So it seems like the terminal does not manage to take into account the random generator and the number of threads each time a task is run, while the QT Creator manages to do so if run there.

4 Conclusion

For this project we wanted to test various numerical integration methods to see which method is best, and how our results change for more brute force methods versus more improved and thought out methods. We have used brute force Gaussian-Legendre quadrature, improved Gaussian quadrature with Laguerre polynomial and spherical coordinates, brute force Monte Carlo integration with uniform distribution, improved Monte Carlo integration with importance sampling, exponential distribution and spherical coordinates and lastly parallelized Monte Carlo integration with openMP and compiler flags.

We have studied the differences in the methods by looking at the difference in the numerical and analytical results, variances of the methods and CPU times with increasing number of mesh points. From these results we have come to the conclusion that the parallelized Monte Carlo is the best method in accuracy to the analytic answer, with lowest variance and lowest CPU time of the calculations for high number of mesh points. This seems to be as expected since Monte Carlo is expected to work best for multidimensional cases, like our case here where we look at a six-dimensional integral. We also expect the CPU time to decrease when we parallelize the code. For both the Gaussian quadrature and Monte Carlo, the improved methods are more accurate than the brute force, but the brute force methods seem to be faster for the same number of mesh points.

For the integral we had to work with the possibility of singularities and dividing by zero. So to avoid that, we just said that the integrand got the value zero when the denominator got a value smaller than a given tolerance. An improvement in the future could be to rewrite the integral to avoid this kind of trouble with dividing by zero. We have also excluded a normalization factor in the final integral in equation 3 that we solved. Future work could be to include this normalization factor in the calculations.

A Appendix

Link to GitHub repository:

<https://github.com/krilangs/FYS4150/tree/master/Project3>

References

- [1] M. Hjorth-Jensen. **Computational Physics - Lecture notes Fall 2015.** 2015.
URL <https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf>.
- [2] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.