COMPUTATIONAL PHYSICS PROJECT 3
Numerical methods for solving integrals

Martin Krokan Hovden

October 22, 2019

Abstract

In this report I have studied different numerical methods for solving integrals. The methods used are Gauss-Legendre, Gauss-Laguerre, brute force Monte Carlo integration, and importance sampling Monte Carlo integration. I have looked at the problem of calculating the ground state correlation energy between two electrons in a helium atom. This was done via using the methods on a six-dimensional integral. For this problem, Monte Carlo integration using importance sampling was the most efficient and accurate method. However, it sometimes suffer from high variance due to numerical effects.

Contents

1 INTRODUCTION

Integrals appears in many mathematical models representing physical phenomena, ranging from modelling waves in the ocean to finding solutions to quantum mechanical problems. Integrals can be solved using both analytical and numerical methods. Unfortunately, most integrals can not be solved analytically. Methods for solving them numerically and efficient are therefore important tools for scientific computing.

In this paper we will look at two different classes of methods for numerical integration; Gauss quadrature and Monte Carlo integration. I will compare the methods accuracy and efficiency on a specific six-dimensional integral used to calculate the ground state correlation energy between two electrons in a helium atom.

I will start by presenting the theory of the methods and look at how they can be implemented using C++ and OpenMP. After that I will discuss how the methods perform on a six-dimensional integral and compare their performance to each other. Lastly, I will conclude on which one was the best method for this specific problem.

2 THEORY AND METHODS

The theory and methods part are based on the lecture notes in *Computational Physics* written by Morten Hjorth Jensen[1] and the book *numerical recipes* by Press et al.[2]. Implementation of gauleg, gauslag and ran0 in main.cpp are found in [1], which is a rewrite of code found in [2].

In general, an integral is given by

$$I = \int_a^b f(x)dx. \tag{1}$$

This can be interpreted as the area under the curve f between a and b. The idea behind solving integrals numerically is to split the domain [a,b] into smaller domains and then add together the contributions from each interval. By reducing the size of each interval, the approximation will get closer to the exact value [2].

2.1 Problem description

In this report we will study the numerical integration methods on a quantum mechanics problem that appears in many different applications. Solving the integral will result in the ground state correlation energy of two electron in a helium atom. The problem can be stated as a six-dimensional integral given by

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

where,

$$\boldsymbol{r_i} = x_i \boldsymbol{e_x} + y_i \boldsymbol{e_y} + z_i \boldsymbol{e_z}. \tag{3}$$

[1]
To model the electrons in the helium atom, we will use $\alpha = 2$. The integral can be interpreted as the quantum mechanical expectation value of the correlation energy between two electrons which interact with each other with Coulomb forces. The exact solution of the integral is given by $5\pi^2/16^2$. Knowing the exact solution will be useful when comparing the different methods.

One problem we have to account for is the case where the denominator is close to or equal to zero. This can be done by just excluding every contribution where denominator is smaller than a small number, say 1e-10. The function can then be implemented in C++ with the code:

```
double integration_function(double x1, double y1, double z1, double x2, double y2, double z2,
    double alpha){
    double r1= sqrt(x1*x1 + y1*y1 + z1*z1);
    double r2= sqrt(x2*x2 + y2*y2 + z2*z2);
    double sep_dist = sqrt(pow((x1-x2),2) + pow((y1-y2), 2) + pow((z1-z2), 2));
    if(sep_dist < 0.000000000001){
        return 0;
    }
    else{
        return exp(-2*alpha*(r1+r2))/sep_dist;
    }
}
```

For some of the methods we use, it will be beneficial to switch to spherical coordinates in equation 2. The function can be rewritten by first noting that

$$d\boldsymbol{r_1} d\boldsymbol{r_2} = r_1^2 dr_1 r_2^2 dr_2 d\cos(\theta_1) d\cos(\theta_2) d\phi_1 d\phi_2 \tag{4}$$

and that

$$r_{12} = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\beta)} \tag{5}$$

where

$$\cos(\beta) = \cos(\theta_1)\cos(\theta_2) + \sin(\theta_1)\sin(\theta_2)\cos(\phi_1\phi_2). \tag{6}$$

We then get that

$$\int_0^{\infty} r_1^2 dr_1 \int_0^{\infty} r_2^2 dr_2 \int_0^{\pi} \sin(\theta_1) d\theta_1 \int_0^{\pi} \sin(\theta_2) d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 \frac{\exp(-2\alpha(r_1+r_2))}{r_{12}}, \tag{7}$$

where I have used that

$$d\cos(\theta_i) = \sin(\theta_i) d\theta_i. \tag{8}$$

Implementation details on the spherical version will be presented later.

Now that the problem is stated on both spherical and cartesian form, we can look closer at the methods used for solving it.

2.2 Gaussian Quadrature

For an introduction to equal step methods for numerical integration, see [2]. The main idea is to approximate the integral on fixed points $x_i \in [a, b]$ so that

$$I = \int_a^b f(x)dx \approx \sum_i^N \omega_i f(x_i) \tag{9}$$

where ω_i are weights given to each point and x_i is points in the grid. In the equal step methods the x_i's are given by

$$x_i = a + \frac{(b-a)}{N} i. \tag{10}$$

The idea behind Gaussian quadrature is to improve the basic numerical integration methods by increasing the degrees of freedom. With Gaussian Quadrature the length between each integration point can vary, and we can freely choose the points. This, together with the choice of weights, will make the methods able to get Gaussian quadrature formulas with the same amount of integration points. Fixed point methods can accurately approximate integrals of polynomial functions of degree N-1 with N points. With the free choice of weights, we will be able to approximate functions of order 2N - 1 with the same N number of points[1]. This is because we now have 2N equations to solve to find the x's and ω's. Finding the optimal weight function is done via orthogonal polynomials. Another feature is that by choosing the weights correctly, we can actually find exact values for integrands equal to a polynomial function times some weight function, given by

$$I = \int_a^b W(x)g(x)dx. \tag{11}$$

[2] where g is smooth. This makes the method able to accurately determine integrals of non-smooth functions [1]. There is some restrictions to the weight function. It has to be non-negative and continuous in the interval we are integrating over.

The Gaussian Quadrature methods often converge faster than the equal steps methods on functions that vary slowly over a large interval. This is because it can choose its mesh points so that it does not take to much care of the part where the integral is almost the same for a large interval. Instead, it focuses on the important parts of the problem. The goal is to increase the accuracy with the same or a smaller amount of integration points.

The Gauss-quadrature methods we will discuss uses orthogonal polynomials to find the weights and abcissas. When those are found, we can approximate the integral by

$$I = \int_a^b W(x)g(x)dx = \sum_{i=1}^{N-1} w_i f(x_i). \tag{12}$$

The main difference is what type of polynomials that are used.

2.2.1 Gauss-Legendre quadrature

In Gauss-Legendre quadrature we use Gauss-Legendre orthogonal polynomials to find the weights ω_i. We will use this method to handle the dimensions of the integral that goes between some fixed values a and b on the form:

$$\int_a^b f(x)dx. \tag{13}$$

The Legendre Polynomials is solutions of the equation

$$C(1-x^2)P - m_l^2 P + (1-x^2)\frac{d}{dx}\left((1-x^2)\frac{dP}{dx}\right) = 0, \tag{14}$$

where C is a constant. We are interested in the case where $m_l = 0$, and the solutions in this case are called the Legendre Polynomials. We thus get

$$C(1-x^2)P + (1-x^2)\frac{d}{dx}\left((1-x^2)\frac{dP}{dx}\right) = 0. \tag{15}$$

The polynomials can be found by the reccurence

$$(j+1)P_{j+1} + jP_{j-1} = (2j+1)xP_j \tag{16}$$

[1] where we typically use

$$P_0 = 1 \tag{17}$$

and

$$P_1 = x \tag{18}$$

The remaining polynomials can be found by using formula 16. The weights and integration points can now be found using the polynomials.

The integration points when using N points are given by the zeros, x_i, of the Legendre polynomial P_{N-1}. When the points are found, we can find the weights from $w_i = 2(P^{-1})_{0i}$ where

$$P = \begin{bmatrix} P_0(x_0) & P_1(x_0) & ... & P_{N-1}(x_0) \\ ... & .... & ... & ... \\ P_0(x_{N-1})... & ... & ... & P_{N-1}(x_{N-1}) \end{bmatrix} \tag{19}$$

[1] For implementation in C++, see the gauleg method in the main.cpp file at the GitHub-adress or [2].

2.2.2 Gauss-Laguerre quadrature

This method will be used to handle integrals on the form

$$\int_0^\infty x^\alpha e^{-x} f(x) dx. \tag{20}$$

The laguerre-polynomials are solutions of the differential equation given by

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2} \right) \mathcal{L}(x) = 0 \tag{21}$$

and are defined in the interval $[0, \infty)$. The reccurence

$$(n+1)\mathcal{L}_{n+1}(x) = (2n+1-x)\mathcal{L}_n(x) - n\mathcal{L}_{n-1}(x) \tag{22}$$

can be used to find the polynomials. The two first ones are given by

$$\mathcal{L}_0 = 1 \tag{23}$$

and

$$\mathcal{L}_1 = 1 - x. \tag{24}$$

[1] The rest can be found using the recurrence formula. The weights and integration points can be found using the same method as for Legendre polynomials. The integration points can be found as the zeros of $\mathcal{L}_{N-1}$ and the weights can be found from the matrix

$$\mathcal{L} = \begin{bmatrix} \mathcal{L}_0(x_0) & \mathcal{L}_1(x_0) & ... & \mathcal{L}_{N-1}(x_0) \\ ... & .... & ... & ... \\ \mathcal{L}_0(x_{N-1})... & ... & ... & \mathcal{L}_{N-1}(x_{N-1}) \end{bmatrix} \tag{25}$$

as $w_i = 2(\mathcal{L}^{-1})_{0i}$, where the x_i's are zeros of $\mathcal{L}_{N-1}$ [1].

Since the method is applied to problems on the form 20, we have to update our function to fit this formula. We will use the polar version and extract the term r and exp term outside and say that

$$f(x) = \frac{\exp(-3(r_1 + r_2))}{r_{12}} \tag{26}$$

where $\alpha = 2$. The spherical equation can now be written as

$$\int_0^\infty dr_1 \int_0^\infty dr_2 \int_0^\pi \sin(\theta_1) d\theta_1 \int_0^\pi \sin(\theta_2) d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 \frac{\exp(-3\alpha(r_1 + r_2))}{r_{12}}. \tag{27}$$

We can then use Gauss-Laguerre to handle the radial parts of the equation, and Gauss-Legendre for the θ's and ϕ's. The function can be implemented in C++ by

```
double integration_function_spherical(double r1, double theta_1, double phi_1, double r2,
    double theta_2, double phi_2)
{
    double cos_beta = cos(theta_1)*cos(theta_2) + sin(theta_1)*sin(theta_2)*cos(phi_1-phi_2);
    double f = exp(-3*(r1+r2))*sin(theta_1)*sin(theta_2)/sqrt(r1*r1+r2*r2-2*r1*r2*cos_beta);
    if(abs(r1*r1+r2*r2-2*r1*r2*cos_beta) > ZERO)
        return f;
    else
        return 0;
}
```

Implementation details for how to solve the complete integral in C++ can be found at the GitHub address in the file main.cpp. The idea is to find the weights and integration points for the r's with the following code

```
double *x_r = new double[N+1];
double *w_r = new double[N+1];
gauslag(x_r, w_r, N, alpha);
```

and for the angular variables with the following

```
double *x_theta = new double[N];
double *w_theta = new double[N];

double *x_phi = new double[N];
double *w_phi = new double[N];

gauleg(0, acos(-1), x_theta, w_theta, N );
gauleg(0, 2*acos(-1), x_phi, w_phi, N);
```

and then use equation 12 to find the integral values.

2.3 Random number generation

Monte Carlo methods uses random numbers. We will use two different variations of Monte Carlo methods. The first one is called the brute force method. This method draws number from a uniform distribution in some given interval. The other method is an improved version and uses something called importance sampling. This method draws numbers from an distribution that resembles the shape of the function we want to integrate. Both methods will be discussed in more detail later.

When drawing random numbers on a computer, the numbers are not actually random. They are often called pseudo random [1]. Algorithms tries to replicate a random distribution as good as possible by making the correlation between the number drawn as small as possible. Since the loop of random numbers eventually will start over, it is also important to let the period of numbers be as long as possible. Generally, the random generators we use will give numbers according to the uniform distribution in the interval [0,1]. The uniform distribution is given by

$$p(x) = \begin{cases} 0 & x \notin [a,b) \\ \frac{1}{b-a} & x \in [a,b) \end{cases} \tag{28}$$

There are different ways of drawing random numbers in C++. One can for example use the built in class random. However, I will use the ran0 random number generator from the text-book numerical recipes [2]. A random number from the uniform distribution in the interval [0,1] can be drawn with the following code

```
long idum = -1;
x = ran0(&idum);
```

where idum is a random seed. It is often needed to draw the number from an general interval [a,b]. This can easily be done with a slight modification of the previous code

```
x = a + (b-a)ran0(&idum);
```

where a and b are pre-specified double values [1].

The uniform distribution can also be used for drawing number from an exponential distribution. We can find a mapping from the uniform numbers in the interval [0,1] to for example the interval $[0, \infty]$. This will be needed for the improved Monte Carlo method. If we assume that x is drawn from a uniform distribution in the interval [0,1], then y is drawn from an exponential distibution if we say that

$$y(x) = -\ln(1-x) \tag{29}$$

[1]. Random numbers from an exponential distribution is implemented in C++ with the following code:

```
y = -log(1-ran0(&idum));
```

We now have the tools to start the discussion on Monte Carlo integration.

2.4 Monte Carlo integration

The main idea behind Monte Carlo integration is to estimate the integral using random numbers. There are a number of ways to do this. In this report we will study a brute force method and a method using

importance sampling. It is important to remember that it is a non-deterministic method, so the results may vary from time to time. In general, Monte Carlo integration work best on multidimensional problems[1]. Due to randomness, we might actually get a more precise value for N=100 than for higher values. Therefore it is important to calculate the variance of the estimates. This way we can see which one is more precise. There exists different variance reducing methods for Monte Carlo Integration, and importance sampling is one of them. Multi-threading is also commonly used for Monte Carlo integration[1].

2.4.1 Brute force

The simplest way to do Monte Carlo integration is by drawing the points from a uniform distribution. This is called the brute force way [1]. The drawback is that the method is not always very accurate as we will see later. In the brute force method we set all the weights equal to one. We can then find an approximation of the integral by

$$I = \int_a^b f(x)dx \approx \frac{(b-a)}{N}\sum_i^N f(x_i)p(x_i) = (b-a)\langle f\rangle, \tag{30}$$

where the brackets indicate the expected value [1]. When expanding to multiple dimension, we get the more general formula

$$I \approx V\langle f\rangle, \tag{31}$$

where V is the volume of the integration domain. V is often called the jacobi determinant. Equation 30 comes from the law of large numbers, which says that

$$\lim_{N\to\infty}\frac{(b-a)}{N}\sum_i^N f(x_i)p(x_i) = I. \tag{32}$$

[1]

The variance for the brute force method is given by

$$\sigma^2 = \frac{V}{N}\sum_i^N (f(x_i) - \langle f\rangle)^2 \tag{33}$$

and the standard deviation is given by

$$\text{std} = \sqrt{\sigma^2} = \sigma. \tag{34}$$

[1] In theory, it looks like the variance will decrease when N increase. It can be shown that

$$\sigma_N \sim \frac{1}{\sqrt{N}} \tag{35}$$

[1]. However, in practice, this is not always the case. This will be discussed later.

The general algorithm for doing Monte Carlo brute force integration is

- Choose the number of points to use, N.
- Loop over N and choose N random numbers drawn from the uniform distribution.
- For each of the N random numbers x_i, evaluate the function, $f(x_i)$, and add the contribution to the mean value and variance.
- At the last step, multiply by the volume V.

For our problem, f is given by equation 2. We have six dimensions, so we need to draw six random numbers for each loop. This can be implemented in C++ with the following code:

```
double V = pow((b-a),6);
for(int i = 0; i < N; i++){
      x1 = a + (b-a)*ran0(&idum);
      y1 = a + (b-a)*ran0(&idum);
      z1 = a + (b-a)*ran0(&idum);
      x2 = a + (b-a)*ran0(&idum);
      y2 = a + (b-a)*ran0(&idum);
      z2 = a + (b-a)*ran0(&idum);
      func_value = integration_function(x1,y1,z1,x2,y2, z2, 2);
      sum+=func_value;
  }
  integral_value = V*sum/((double)n);
```

where

$$V = (b-a)^6 \tag{36}$$

represents the "integration volume".

2.4.2 Importance sampling

Monte Carlo integration using importance sampling is an extension on the brute force method, and using importance sampling will often lead to more stable results which we will see later[1]. The goal is to reduce the variance in the estimations. The idea is to draw numbers from an distribution that resembles the shape of the function that we want to integrate, and thereby get more accurate results.

In equation 7 it is the radial part than can benefit from using important sampling. We can rewrite the equation slightly and do a transformation to find the new expression to be used. If we assume that p(r) is the PDE of an exponential distribution, we can write

$$I = \int_0^\infty p(r)\frac{F(r)}{p(r)}dr = \int_0^\infty \frac{2\alpha\exp(-2\alpha r)r^2\exp(-2\alpha r)}{2\alpha\exp(-2\alpha r)}dr \tag{37}$$

where $p(r) = 2\alpha\exp(-2\alpha r)$ is the exponential distribution as a function of r. We draw our random numbers from an uniform distribution, so we need to do an change of variable. If we say that x is drawn from a uniform distribution, then p(x) is given by equation 28. Now we need to have that

$$p(r)dr = p(x)dx = dx \rightarrow dr = \frac{dx}{p(r)}. \tag{38}$$

By using this, we get that

$$I = \int_0^\infty p(r)\frac{F(r)}{p(r)}dr = \int_0^1 \frac{F(r)}{p(r)}dx \tag{39}$$

where

$$r = r(x) = -\log(1-x). \tag{40}$$

Here I have used the transformation formula from a uniform distribution to exponential. We can now write the problem as

$$\int_0^1 \frac{F(r(x))}{p(r(x))}dx = \int_0^1 \frac{r(x)^2}{4}dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{r(x_i)^2}{4} \tag{41}$$

where I have inserted $\alpha = 2$. The θ_i and ϕ_i, is drawn from the intervals $[0, 2\pi]$ and $[0, \pi]$. This can be implemented in C++ with the following code.

```
for(int i = 0; i < n; i++)
  {
      r1 = -0.25*log(1-ran0(&idum));
      r2 = -0.25*log(1-ran0(&idum));
      theta1 = acos(-1)*ran0(&idum);
      theta2 = acos(-1)*ran0(&idum);
      phi1 = 2*acos(-1)*ran0(&idum);
      phi2 = 2*acos(-1)*ran0(&idum);
      func_value = integration_function_monte_carlo(r1, theta1, phi1, r2, theta2, phi2);
      sum += func_value;

  }
  double V = 4*pow(acos(-1), 4)/16.;
  integral_value = V*sum/((double) n);
```

When implementing Monte Carlo methods, the variance is often a complicated value to get right. For example, using the standard text-book expression for variance

$$\sigma^2 = V\int_a^b (f(x)-\mu)^2 p(x)dx = \left[\left(\frac{1}{N}\sum_i f^2(x)\right) - \mu^2\right]V, \tag{42}$$

can lead to problems. We will sometimes encounter situations where the two terms are almost equal. This can introduce numerical precision errors, and give negative values for the variance which is not possible. Therefore, it is better to use

$$\sigma^2 = [\frac{1}{N}\sum(f(x_i)-\mu)^2]V \tag{43}$$

to calculate the variance. This can be implemented in c++ with the following code

```
for(int i = 0; i < n; i++)
  {
      sum_sigma += (x[i] - integral_value)*(x[i] - integral_value);
  }
sum_sigma = V*sum_sigma/((double)n);
```

where x_i is the function value for the i'th random number.

2.5 Paralleliization

Parallelization of code is a method for improving the speed of algorithms. It means to run a set of calculations on more than one core at the same time, and thereby improve the performance of the code. Due to advancements in computer hardware, parallelization is something that can be done on most modern computers. Not all algorithms will benefit from parallelization, and we have to be aware of what part of the code that can be run in parallel. Monte Carlo integration is one algorithm that typically gets a speed-up from parallization [1].

When we use parallelization, we can measure the improvement by looking at the ratio between the execution time for running on one processor, T_1, versus running on p processors, T_p. This relationship is given by

$$\text{Speed-up} = \frac{T_1}{T_p}. \tag{44}$$

There is actually possible to get a higher speed-up than p, if done correctly. Amhdahls law says that if f is the fraction of the code that is parallelizable, then we can achieve a speed up given by

$$\frac{1}{1-f}. \tag{45}$$

[1] This is not always possible to achieve in practice. There is effects that might slow down the results compared to the optimal value given by equation 45. These effects can be overhead for synchronization, overhead for communication between the cores and extra computations needed as a result of the parallelization[1]. In addition, not all code are able to run in parallel. These effects will lead to sub-optimal speed-up. In the Monte Carlo code, most of the work is done in the for-loop. For-loops of this kind works good for parallelization and we can expect an speed-up.

We will use OpenMP to see what effects we can achieve by parallization of our code for the Monte Carlo Integration using importance sampling. OpenMP is built into most of the modern C++ compilers and are easy to use. With the line

```
#pragma omp parallel
{
    some code...
}
```

we tell the compiler to start running the processes on multiple threads. By using the following code snippet

```
#pragma omp for reduction(+:sum)
for(int i = 0; i < n; i++)
{
    some code...
    sum += something
}
```

we make sure that every loop in the following for-loop is done exactly one time.

3 RESULTS AND DISCUSSION

The code used to generate all plots and tables can be found at the GitHub-repository: `https://github.com/MartinKHovden/CompPhys_project3`.

We will start by looking at the Gauss-Legendre method. For each dimension, the function is integrated between -5 and 5 for varying number of integration points. By looking at the function values outside of this interval, the function was more or less zero, so adding these contributions would not lead to notably better results. In table 1 we can see how it perform. The error is calculated with abs(numerical value - exact value), where the exact value is given by $5\pi^2/16^2 \approx 0.19257$.

N	Gauss-Legendre	Error	Time [s]
5	0.0423	0.1502	0.004
10	0.0111	0.1814	0.132
15	0.3159	0.1233	1.467
20	0.0968	0.0958	8.382
25	0.2401	0.04756	32.1
30	0.1464	0.0462	96.482

Table 1: Results using Gauss-Legendre for different values of N integrated in the interval -5 to 5 for all dimensions.

The method is not converging at the level of three leading digets even for N = 30. Running for higher values up to N=40 does not lead to better results, and the estimate continues to fluctuate around the exact value.

In table 2, the results for Gauss-Laguerre are presented. Now function 7 is used, with $\theta_i \in [0, \pi]$ and $\phi_i \in [0, 2\pi]$ and $r_i \in [0, \infty)$.

N	Gauss-Laguerre	Error	Time [s]
5	0.0626	0.1299	0.008
10	0.1816	0.0110	0.356
15	0.1959	0.0033	4.138
20	0.1956	0.00307	23.546
25	0.1952	0.0027	88.19
30	0.19507	0.0025	261.011

Table 2: Results using Gauss-Lagurre the radial parts and Gauss-Legendre for the angular part for different values of N on function 7. $\theta_i \in [0, \pi]$ and $\phi_i \in [0, 2\pi]$ and $r_i \in [0, \infty)$

When it comes to precision, the method using Gauss-Laguerre is the superior. Even for N = 10, we get an fairly acceptable approximation with an error of 0.0110. By increasing N we can increase the precision, which is what we would expect from the discussion of the methods. Gauss-Legendre does not come close for any N up to 30. When it comes to speed, using Gauss-Legendre is faster for the same N, and using by Gauss-Laguerre the time increases faster in comparison when N increases. This is not surprising, since the brute force method is a lot more straight forward, and requires fewer mapping and other calculations. However, we can study the results closer. By looking at Gauss-Legendre with N = 30, we see that the error is 0.0462, and time used is about 96 seconds. If we look at Gauss-Laguerre with N = 25, it uses approximately the same time, but the error is only 0.0027. We have to move below N = 10 for the Gauss-Laguerre to get worse error than Gauss-Legendre with N = 30. One thing to note is that Gauss-Laguerre seems to get values a bit higher than the exact value. We see that Gauss-Laguerre is the best method of the two, and obtain better results for smaller N and shorter time.

We can now look closer at the different variations of the Monte Carlo methods. The brute force method are ran in the interval -5 to 5 in every dimension, for varying N. We see in figure 1 that the estimate hits the actual value for some N's, and then fluctuates away again. Even for higher N-values than shown in the plots, the estimated values keeps oscillating around the exact value. For lower N, the estimate is very far off, and we can see from figure 2 that the error is approximately 0.2 for N-values up to 10e3. At N = 10e4 we see a drop in the error in figure 2, and then it bounces back up. The brute force method is expected to have mean-values that vary a lot, so this is not unexpected. It seems like for the N-values where we hit the actual value it was pure luck.

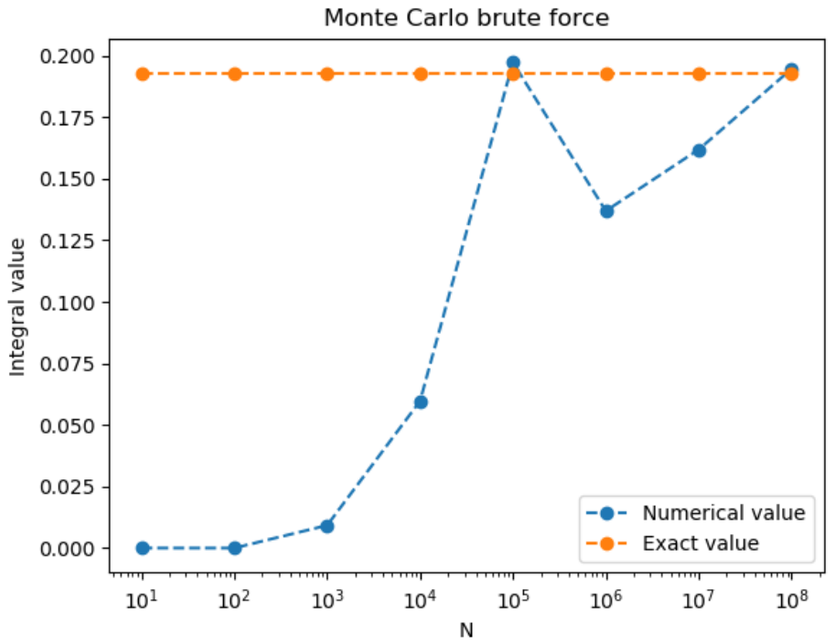

Figure 1: Numerical value vs. exact value for Monte Carlo brute force integration in the interval -5 to 5 in each dimension

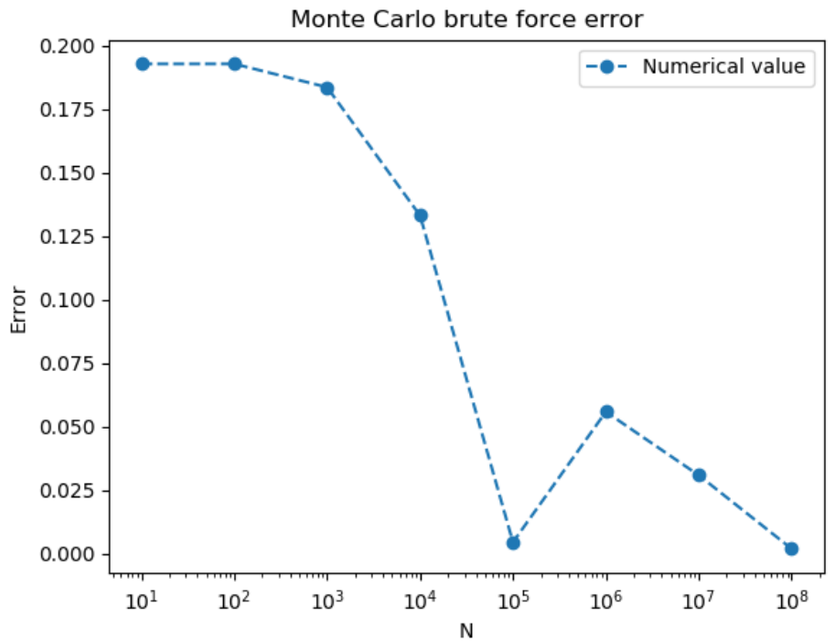

Figure 2: Difference between numerical value and exact value for Monte Carlo integration using brute force in the interval -5 to 5 in each direction.

In figure 3 we see the variance for the brute force Monte Carlo method. Theoretically, the variance is supposed to decrease when we increase N. We see the opposite case in the figure. One explanation is that there occurs numerical problems when subtracting to almost equal number in the calculations of the variance. This will be discussed further for the importance sampling method.

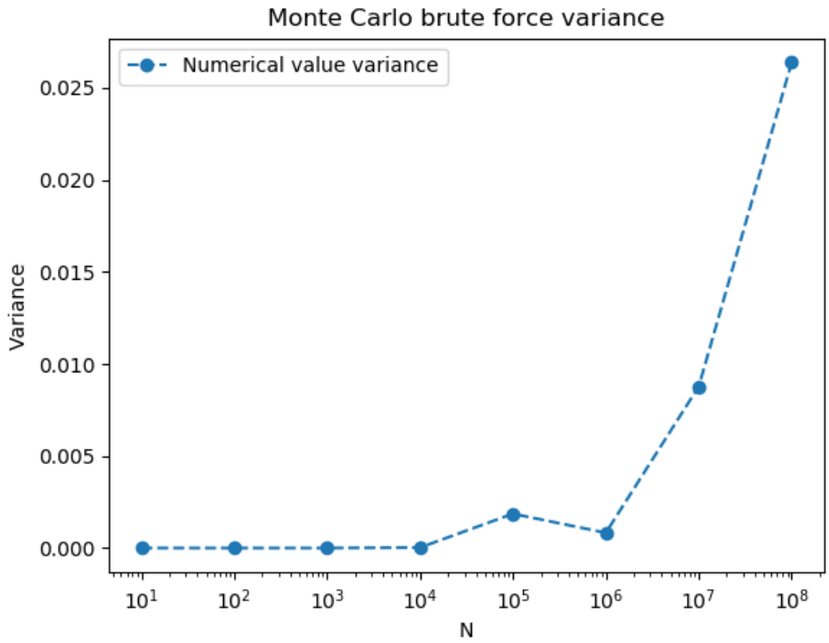

Figure 3: Variance in numerical integral value for Monte Carlo integration using brute force in the intervals -5 to 5 in each direction.

If we look at Monte Carlo integration using importance sampling, we can see that the estimate is better, even for smaller N-values. In addition, it converges much faster to the actual solution and stays at the solution for N larger than 10e5. However, the variance does not behave as one would expect from the theory. We would expect the variance to decrease when we increase N. This is also what we saw for the brute force method. In figure 6 we see that the variance actually increases when N increases. As discussed in the theory part and for the brute force method, this sometimes happens when using Monte Carlo methods. We see that the mean value actually stabilizes a lot faster than the variance, which does not even seems to stabilize. The problem is that the expression for variance involves subtractions of two large numbers that sometimes can get very close to each other. The fluctuations in variance might come from the numerical errors that occurs when this happens. When running the code for different random seeds, the integral values seems to be quite stable, however, the variance varies a lot for different seeds, and does never seems to decrease. For some runs I was able to get results that is closer to what one would expect from theory, but a lot of them had similar behaviour to the one presented in the report.

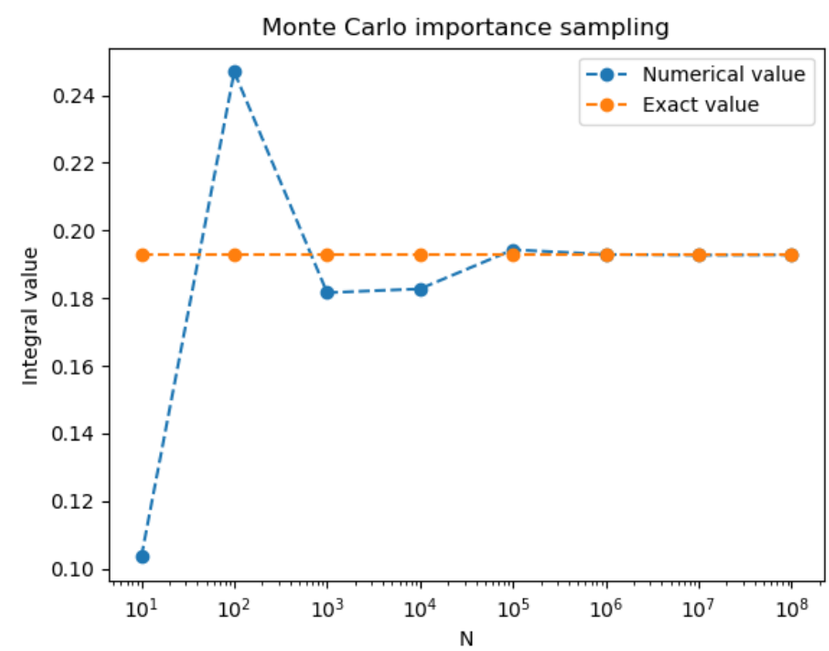

Figure 4: Numerical value vs. exact value for Monte Carlo integration with importance sampling.

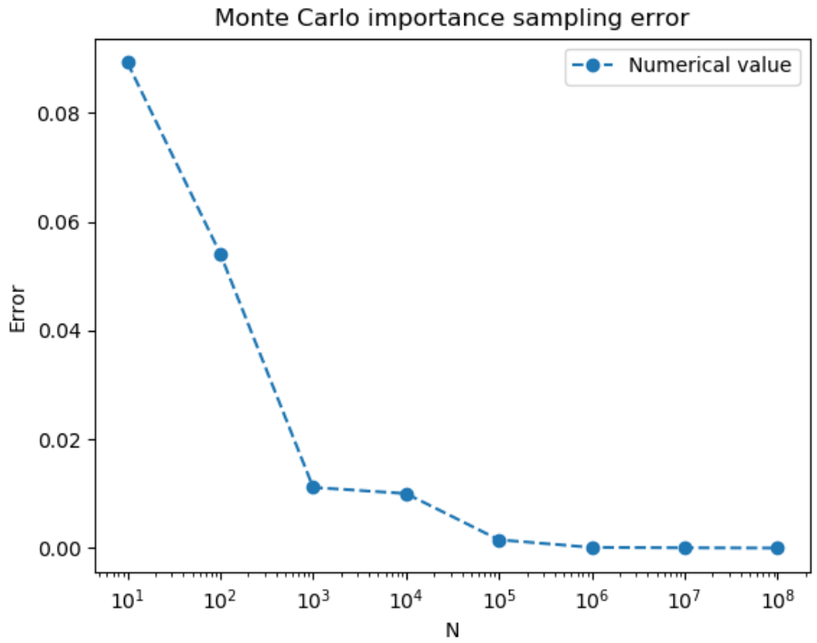

Figure 5: Difference between numerical value and exact value for Monte Carlo integration using importance sampling.

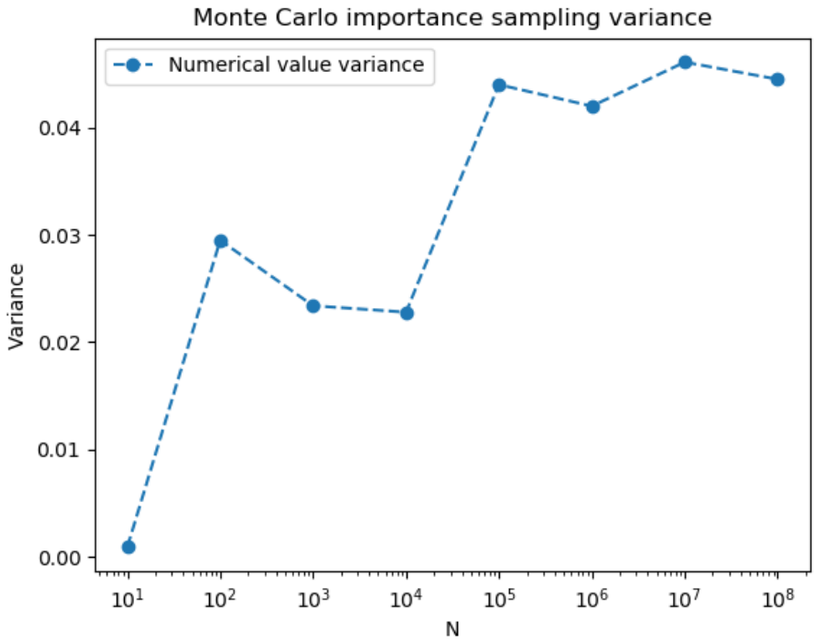

Figure 6: Variance in the numerical integral value for Monte Carlo integration using importance sampling.

In table 3 the error and time used is presented for the brute force method, the importance sampling method and for parallel importance sampling. We see that by using Monte Carlo importance sampling, the time is higher. From table 3 it looks like the time is approximately two times that for brute force. However, from figure 5 the estimates of the integral value seems to stabilize around the exact value with N=10e5. On the other hand, for brute force with N=10e5, the estimate have not stabilized and start to fluctuate away from the actual value.

N	MCBF time[s]	MCBF error	MCIS time[s]	MCISP time[s]	MCIS error	MCISP error
10	0.000	0.19275	0.000	0.004	0.0758	0.0934
10e2	0.000	0.1928	0.000	0.001	0.0679	0.0739
10e3	0.001	0.1835	0.000	0.004	0.0642	0.0563
10e4	0.1331	0.002	0.005	0.003	0.01155	0.0201
10e5	0.030	0.00449	0.060	0.018	0.004397	0.00389
10e6	0.219	0.0558	0.528	0.134	0.00153	0.00126
10e7	2.361	0.03085	5.807	1.182	0.000014	0.000023
10e8	24.537	0.00190	54.444	12.055	0.000066	0.000018

Table 3: Error and time comparison between Monte Carlo brute force (MCBF) and importance sampling (MCIS) and importance sampling with parallel (MCISP) for different values of N.

It is interesting to study how or if the performance of the code increases by using parallelization or compiler flags. My computer have 8 threads, so in theory we could be able to get close to an 8 times speed-up. The estimations of the integral were approximately the same as the ones without parallel code, as can be seen in table 3. In figure 7 we see the time of the code for different values of N. It seems like for N-values up to 10e5, the parallel code does not give an significant improvement. This might be because the overhead takes a lot of the total time, so just a small fraction of the operations are actually parallel. However, for larger N-values, we see that the improvement is quite significant. In figure 8 we see the speed-up factor. The speed-up is actually quite significant for N larger than 10e3. For N = 10e8 we get an speed-up factor of 4. This is a bit smaller than the theoretical value, but it might be due to overhead in the code.

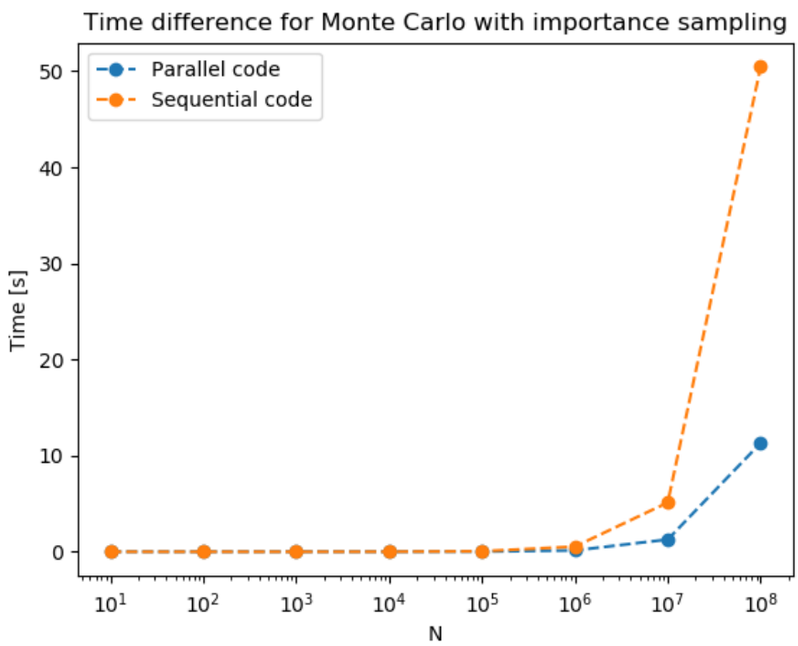

Figure 7: Comparison of time for parallel vs. sequential Monte Carlo integration using importance sampling

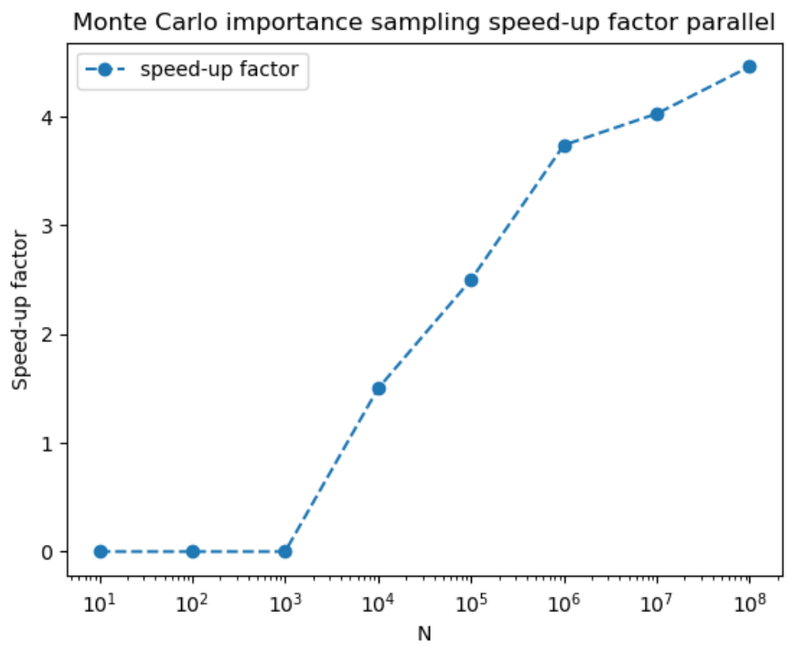

Figure 8: Speed-up factor from using parallel code code vs. sequential code for Monte Carlo integration using importance sampling.

By using the parallel code and the compiler flag -03, we are actually able to get an speed-up factor of eight as we can see in figure 9. This is close to what one could expect when running on eight threads. One can see that for both with and without the compiler flag, the parallel code does not give much effect for N smaller than 10e3. This is probably because that for such low N-values, the overhead is the dominating factor in the calculations, and running the for-loop in parallel does not have that much effect.

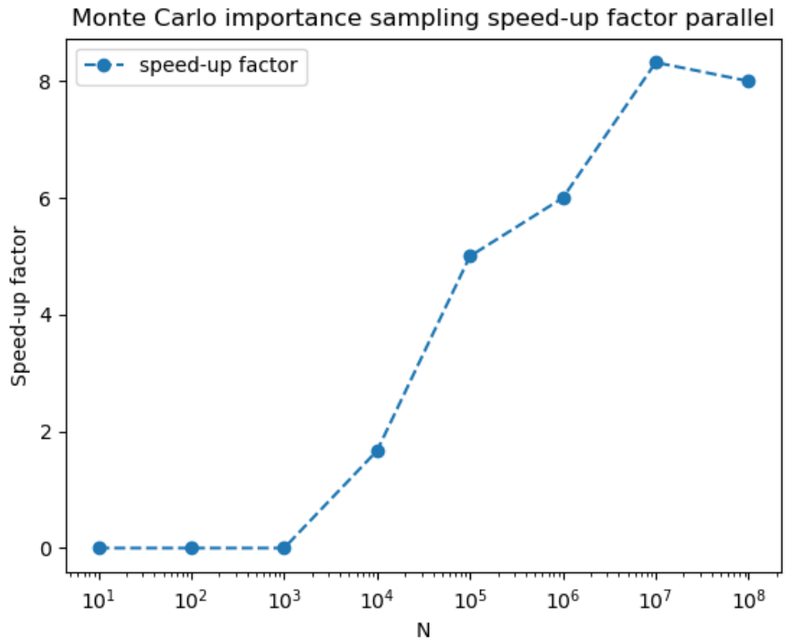

Figure 9: Speed-up factor from using parallel code compiled with compiler flag -03 vs. sequential code for Monte Carlo integration using importance sampling.

We can now look closer at the two best methods from each class. The Gauss-quadrature using spherical coordinates, and the Monte Carlo method using importance sampling and parallel code. I will compare both methods without using compiler flags. From table 3 that we get an error of only 0.000014 in 1.2 second. This is superior to every other methods used. Compared to the brute force Monte Carlo, it is faster for all N's, and the accuracy is much higher for all N, so the brute force method is to no good use. Compared to the Gauss-quadrature method using both Gauss-Laguerra and Gauss-Legendre on the spherical function, we also obtain a higher accuracy and the time is much better with a factor of 200. Even though the variance of the Monte Carlo method using importance sampling is high, the mean value seems to be stable. For this problem, the best method is by far the Monte Carlo method using parallel code and importance sampling.

4 CONCLUSION

After studying the different methods, it seems like the Monte Carlo method using importance sampling and parallel code is the best performing algorithm. The time it takes to obtain a given accuracy is the fastest, and the estimate of the integral converges to the exact solution. One drawback of the method is the high variance for this specific problem. This probably comes from numerical errors after subtraction of two large and almost equal numbers. However, the mean value is very stable. The accuracy is also higher than the Gauss-quadrature. It might be interesting to try to apply other variance reducing techniques to see if the variance of the Monte Carlo method can be better. Trying to run the code on a more powerful computer would also be beneficial, to see if the variance gets more stable when running for even higher N's than my laptop was able to handle. Testing for higher N would be interesting. We could then see if increasing N would actually lead to the mean value stabilizing. One of the main problems with the brute force methods was that they did not converge to the exact results. Using the algorithms that uses info about the data was beneficial and led to the best approximations in this case.

References

[1] Jensen, M. H., *Computational Physics, lecture notes 2015*, 2015.

[2] Press W. H., Teukolsky. S. A., Vetterling W. T., Flannery B. P,
Numerical Recipes, the art of scientific computing, Cambrige University Press 2007.