

# Project 3 in Computational Physics, Numerical Integration

George Cowie and Jonas Thoen Faber

October 21, 2019

## Abstract

In this project we will solve a six dimensional integration numerically using four different methods. That is the Gauss-Legendre and Gauss-Laguerre quadrature, brute force Monte Carlo and improved Monte Carlo with importance sampling. Since the quadrature methods depends on orthogonal polynomials such as Legendre and Laguerre polynomials, they must be integrated over six dimensions which leads to six for-loops in our code. This means that the program is very slow per iteration, but in return they quickly increase in accuracy per iteration. Monte Carlo simulation is based on random numbers and does not need to cover six dimensions. This method needs a much larger number of iterations to reach a satisfactory accuracy, but is in return much faster per iteration.

Our infinity is estimated to be 2.3 when using Gauss-legendre. We achieve result at the level of three leading digits when  $N = 25$  or higher. Only  $N = 10$  or higher is needed for Gauss-Laguerre.  $N = 10^8$  and  $N = 10^4$  is needed for brute force Monte Carlo and importance sampling respectively. The time needed for achieving this precision are approximately 15.68 seconds (Legendre), 0.49 seconds (Laguerre), 18.75 seconds (brute Monte Carlo) and 0.004 seconds (importance sampling). After parallelized the importance sampling method, the time needed was downsized even further. Approximately  $1.28 \cdot 10^{-3}$  seconds,  $1.21 \cdot 10^{-3}$  seconds and  $9.51 \cdot 10^{-4}$  seconds for plain parallelization, parallelization with compiler flag -O2 and parallelization with compiler flag -O3 respectively.

Our code can be found from our github page: [https://github.com/JonasTFab/Project\\_3](https://github.com/JonasTFab/Project_3)

## 1 Introduction

The main focus in this project is to integrate some function using different kinds of numerical methods. The function we want to integrate is based on two electron orbiting a helium atom and the integral is often used in quantum mechanical applications. We already know the analytical answer to this integration so our mission is to make our methods as accurate and fast as possible.

We will start with two different Gauss quadrature methods where we basically split a function into two,  $f(x) \rightarrow W(x)g(x)$  where  $W(x)$  is the weight function and  $g(x)$  is a smooth function. The weight functions depends on which quadrature method that is being used for solving the integral. The first method is called the Gauss-Legendre quadrature which uses Cartesian coordinates to describe the system. Since we are working with two particles, the integral is therefore six dimensional and will firstly be solved in a brute force manner. The size of this integration leads to some major complications which we will soon notice later in this project. The second method is the Gaussian-Laguerre quadrature which instead uses spherical coordinates. By doing this change of variable makes the integration somewhat faster and more accurate. We will then move over to stochastic Monte Carlo methods that implement random numbers to determine the location of the particles. As in the quadrature methods, we will start off with plain Cartesian coordinates before moving over to spherical coordinates. Again, we will notice some improvements by doing so. Finally, we want to parallelize our last method for the sake of increasing the efficiency even more.

## 2 Methods

The main focus on this project is to solve the integration which corresponds to the correlation energy between two electrons. We assume that we may model the wave function of the electrons as single-particle wave function of an electron around a hydrogen at the  $1s$  state  $\psi_{1s}(\mathbf{r}_i) = \exp(-\alpha r_i)$ . The vectors  $\mathbf{r}_i = x_i\mathbf{e}_x + y_i\mathbf{e}_y + z_i\mathbf{e}_z$  is the location of each electron and so  $r_i = (x_i^2 + y_i^2 + z_i^2)^{1/2}$ .  $\alpha$  is a parameter which corresponds to the charge of the helium atom  $Z = 2$ . Take notice that the wave function for each electron is not properly normalized. So with this in mind, we will do the ansatz that the product of the wave functions of each electron is given as  $\Psi(\mathbf{r}_1, \mathbf{r}_2) = \exp(-\alpha(r_1 + r_2))$ . This leads us then to the expectation value:

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

So this is the integral we want to work with during this project. This integral has the exact solution  $5\pi^2/16^2$  which is the value we want our algorithm to obtain. The expression to remember is then:

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

### 2.1 Gauss-Legendre quadrature

The first two methods we will look at are based on the Gaussian quadrature. The reason for using the Gaussian quadrature is to not be dependent on equally spaced

integration points. This will lead to a better precision of numerical work by extracting an orthogonal function out of the original function. The quadrature formula is given as following:

$$I = \int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=1}^N \omega_i g(x_i) \quad (3)$$

where  $W(x)$  is the weight function and  $g(x_i)$  is a smooth function with  $N$  distinct quadrature points. We want to achieve arbitrary weights  $\omega_i$  through the use of orthogonal polynomials within some interval. The  $x_i$  is then carefully chosen within this interval. If it happens to be that the right side of Equation 3 is equal to the left side, that is if all polynomials  $p \in P_{2N-1}$  is integrated exactly, the formula is called Gaussian quadrature:

$$\int_a^b W(x)g(x)dx = \sum_{i=1}^N \omega_i g(x_i) \quad (4)$$

So the first method we will begin with is the Gauss-Legendre quadrature remembering that we are working with six integrating dimension. We need to find the weight function and the mesh points. The mesh points are the points we get from solving the orthogonal polynomials. For orthogonal Legendre polynomials, it is stated from [1] that the weight function is  $W(x) = 1$  within the interval  $x \in [-1, 1]$  and the mesh points is the solution of the Legendre polynomials. So now our integral looks like this:

$$I = \int_{-1}^1 dx_1 \int_{-1}^1 dy_1 \int_{-1}^1 dz_1 \int_{-1}^1 dx_2 \int_{-1}^1 dy_2 \int_{-1}^1 dz_2 \cdot \omega_{i,x_1} \omega_{i,y_1} \omega_{i,z_1} \omega_{i,x_2} \omega_{i,y_2} \omega_{i,z_2} f(x_1, y_1, z_1, x_2, y_2, z_2) \quad (5)$$

$$= \int_{-1}^1 dx_1 \int_{-1}^1 dy_1 \int_{-1}^1 dz_1 \int_{-1}^1 dx_2 \int_{-1}^1 dy_2 \int_{-1}^1 dz_2 f(x_1, y_1, z_1, x_2, y_2, z_2) \quad (6)$$

We see that the integral now becomes long and difficult to investigate. So for the sake of simplicity, we will just keep using one dimension variable since the integral would still be the same more or less. Nevertheless, the Gauss-Legendre function is not necessarily limited from -1 to 1 but rather from  $-\lambda$  to  $\lambda$  where  $\lambda \rightarrow \infty$ . We therefore need to perform a change of variable. Since  $t \in [-\lambda, \lambda]$ , our new variable becomes:

$$t = \frac{\lambda - (-\lambda)}{2}x + \frac{\lambda + (-\lambda)}{2} = \lambda x \quad (7)$$

and the following integral becomes:

$$I = \int_{-\lambda}^{\lambda} f(t)dt = \frac{\lambda - (-\lambda)}{2} \int_{-1}^1 f(\lambda x)dx = \lambda \int_{-1}^1 f(\lambda x)dx \quad (8)$$

The `gauss_legendre` function in the code<sup>1</sup> will automatically take care of the limits. So what it does is calculating both the weights and mesh points of some implemented function  $f(x)$ , or in our case [Equation 2](#). The code would then look something like this:

```
// first of all we calculate the mesh points x and weights
// W based on integration points N and the limits lambda.
// We do so by sending all the variables into the
// gauss_legendre function
gauss_legendre(-lamb, lamb, x, W, N);

// Runs through N^6 operations to calculate the integral
for (int i1 = 0; i1 < N; i1++){
  for (int i2 = 0; i2 < N; i2++){
    for (int i3 = 0; i3 < N; i3++){
      for (int i4 = 0; i4 < N; i4++){
        for (int i5 = 0; i5 < N; i5++){
          for (int i6 = 0; i6 < N; i6++){
            integral += W[i1]*W[i2]*W[i3]*W[i4]*W[i5]*W[i6]*
            func(x[i1],x[i2],x[i3],x[i4],x[i5],x[i6]));
          }
        }
      }
    }
  }
}
```

The number of operations is then  $N^6$  since we are dealing with 6 dimensional integration.

## 2.2 Gauss-Laguerre quadrature

The difference between the Gauss-Legendre quadrature and the Gauss-Laguerre quadrature, is that we are now changing our function to spherical coordinates. We will come back to why this method should give us some better results. So in this case, the Laguerre polynomials are defined for  $x \in [0, \infty)$  which means the radial parts. The angles are defined within the interval  $\theta \in [-\pi/2, \pi/2]$  and  $\phi \in [-\pi, \pi]$ . The weight function for orthogonal Laguerre polynomials is defined, according to [\[1\]](#), as  $W(x) = x^a e^{-x}$ . So what we want to do is to extract this part out of our original function ([Equation 2](#)) as following:

---

<sup>1</sup>The `gauss_legendre` code can be found at <https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Projects/2019/Project3/CodeExamples/exampleprogram.cpp>

$$e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} = e^{-r_1} e^{-r_2} \frac{e^{2\alpha}}{|\mathbf{r}_1 - \mathbf{r}_2|} \quad (9)$$

The change of variables leads to the Jacobi determinants that must be included in the equation. The transformation from Cartesian coordinates to spherical coordinates replaces the infinitesimals to:

$$d\mathbf{r}_1 d\mathbf{r}_2 = r_1^2 r_2^2 \sin(\theta_1) \sin(\theta_2) dr_1 dr_2 d\theta_1 d\theta_2 d\phi_1 d\phi_2 \quad (10)$$

and the distance between the two electrons are now defined as:

$$r_{12} = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\beta)} \quad (11)$$

whereas

$$\beta = \cos(\theta_1) \cos(\theta_2) + \sin(\theta_1) \sin(\theta_2) \cos(\phi_1 - \phi_2) \quad (12)$$

So the dimensions we need to extract the Laguerre polynomial out of, is just the radial part because these variables are the ones that goes from zero to infinity. The integration we now are working with goes as follows:

$$I = \int_0^\infty dr_1 \int_0^\infty dr_2 \int_0^\pi d\theta_1 \int_0^\pi d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 r_1^2 e^{-r_1} r_2^2 e^{-r_2} \sin(\theta_1) \sin(\theta_2) \frac{e^{2\alpha}}{r_{12}} \quad (13)$$

The Gauss-Laguerre will now with the term  $r^2 e^{-r}$  transforms this integral into six sums including the weights and mesh points. This is quite a big expression so for simplicity,  $\omega_{x,y}$  is defined as the weight that corresponds the variable  $x$  and the mesh points  $y$ :

$$I = \sum_{i_1=1}^N \sum_{i_2=1}^N \sum_{i_3=1}^N \sum_{i_4=1}^N \sum_{i_5=1}^N \sum_{i_6=1}^N \omega_{r_1,i_1} \omega_{r_2,i_2} \omega_{\theta_1,i_3} \omega_{\theta_2,i_4} \omega_{\phi_1,i_5} \omega_{\phi_2,i_6} \cdot \sin(\theta_{i_1}) \sin(\theta_{i_2}) g(r_{i_1}, r_{i_2}, \theta_{i_3}, \theta_{i_4}, \phi_{i_5}, \phi_{i_6}) \quad (14)$$

This is the fully Gauss-Laguerre method that we will implement in our code and should look something like this:

```
// since the radial part is the only thing that
// goes to infinity, these will use the Gauss-
// Laguerre function to find its weights and
// mesh points. Legendre is used on both the
// theta's and phi's
gauss_legendre(0, pi, xt, Wt, N);
```

```

gauss_legendre(0, 2*pi, xp, Wp, N);
gauss_laguerre(xgl, Wgl, N, alf);

// Runs through N^6 operations to calculate the integral
for (int i1 = 1; i1 <= N; i1++){
  for (int i2 = 1; i2 <= N; i2++){
    for (int i3 = 0; i3 < N; i3++){
      for (int i4 = 0; i4 < N; i4++){
        for (int i5 = 0; i5 < N; i5++){
          for (int i6 = 0; i6 < N; i6++){
            integral +=
              W_r1[i1]*W_r2[i2]*W_t1[i3]*W_t2[i4]*W_p1[i5]*W_p2[i6]*
              func_sphere(r1[i1],r2[i2],t1[i3],t2[i4],p1[i5],p2[i6]);
          }
        }
      }
    }
  }
}

```

We are still operating with  $N^6$  operations, so the methods calculations increases really fast for a minor change in  $N$ .

Notat: The error in the Monte Carlo integration is however independent of  $d$  and scales as  $\sigma \sim 1/\sqrt{N}$ , always! Hvorfor det, gjør den det i vårt tilfelle?

### 2.3 Brute force Monte Carlo Integration

The third integration method we wish to use is a so called brute force Monte Carlo method. We will approximate the integral using the average of the function  $f$  for a given PDF  $p(x)$  from [2].

$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^N f(x_i) p(x_i) \quad (15)$$

$$I \approx \langle f \rangle \quad (16)$$

In this case the PDF is just the uniform distribution.  $x_i$  is a pseudo-random number generated in the range  $[0,1]$  multiplied with the range in which we wish to integrate.

$$X_i = x * (b - a) + a$$

we have used the random number generator `rand()` from `cstdlib` used in C++. The numbers are initialized by the time at which the program is run. Here  $a$  and  $b$  are the start and end of the integration area. This method is not optimized for our specific function as it samples evenly from the whole distribution. Our function may have a peak or be close to zero for certain values of  $x$ . Sampling equally from the whole range may therefore lead to a large number of samples in uninteresting areas. This means that we could be wasting a lot of cpu cycles on calculations that hardly contribute to the integral at all, hence the name brute force Monte Carlo. The algorithm for the brute force integration is implemented as follows

```
double a = lambda_start;
double b = lambda_end;

double brute_monte_carlo(int N, double a, double b){
    srand(time(NULL)); // seed RNG with current time
    double Jacobi_det = (b-a)**6
    double integral = 0;

    for (int i=0; i<N; i++){
        double x1,y1,z1,x2,y2,z2 = ran()*(b-a)+a; //initialize the
                                                    random variables

        integral += func(x1,y1,z1,x2,y2,z2)
    }

    integral = integral/N*Jacobi_det
```

```

    return I
;}

```

Since our integral is just the average value of the distance between the electrons, it should be obvious that we should implement the variance in the code as well. The standard deviation is just the square root of the variance. The variance of a function  $f$  is defined from [3] as following:

$$\sigma_f^2 = \sum_{i=1}^N f(x_i)^2 - \left( \frac{1}{N} \sum_{i=1}^N f(x_i) \right)^2 \quad (17)$$

This will be trivial for the sake of understanding how much our original function deviates from its average over the region of integration.

## 2.4 Importance sampling, improved Monte Carlo

Our fourth and last method will be the Monte Carlo method including both change of variables and importance sampling. We will go back into spherical coordinates as we did with Gauss-Laguerre method. There are a few conditions that needs to be met for us to perform a variable change. That is  $p$  is normalize-able, analytically integrate-able and invert-able, that is to express a new variable in terms of an old one. If we take a look back to our original [Equation 2](#), we notice that there are two exponential functions existing. We will therefore perform a variable change that includes the exponential distribution, that is  $p(r) = ae^{-ar}$  where  $r = r(x)$ . Since  $x \in [0, 1]$ , we want  $r \in [0, \infty)$ . Remembering that we are working with probability distribution, it follows that  $p(r)dr = p(x)dx = dx$ . With this in mind, it also follows from [3] that  $r = -\ln(1 - x)$  for the desired domain. After the variables change, our integral becomes:

$$I = \int_0^\infty dr_1 \int_0^\infty dr_2 \int_0^\pi d\theta_1 \int_0^\pi d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 r_1^2 r_2^2 \sin(\theta_1) \sin(\theta_2) \frac{e^{-4(r_1+r_2)}}{r_{12}} \quad (18)$$

We see that in the exponential that there exist a 4 which means  $a = 4$ . We therefore have to add in  $4/4$  in the integrand so that we get the required PDF of the radial part. Also, before we can perform a Monte Carlo evaluation on this integral, We must perform a sampling on the  $\theta$  and  $\phi$  parts as well. We know that  $\theta \in [0, \pi]$ . The integral of the probability must be preserved, that is  $\int_0^\pi p(\theta)d\theta = 1$ . From this we get that  $p(\theta) = 1/\pi$ . For the  $\phi$  parts, we get  $p(\phi) = 1/2\pi$ . Now all the variable are generated from the original distribution  $p(x)$  through change of variable. Let's define our original function as  $F(r_1, r_2, \theta_1, \theta_2, \phi_1, \phi_2) = F$ . With these changes, our integral may now be written, using the Monte Carlo evaluation, as:



$$I = \int_0^\infty dr_1 \int_0^\infty dr_2 \int_0^\pi d\theta_1 \int_0^\pi d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 r_1^2 r_2^2 \sin(\theta_1) \sin(\theta_2) F \quad (19)$$

$$I = \int_0^\infty dr_1 \int_0^\infty dr_2 \int_0^\pi d\theta_1 \int_0^\pi d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 r_1^2 r_2^2 \sin(\theta_1) \sin(\theta_2) F$$

$$\cdot \frac{p(r_1)p(r_2)p(r_{\theta_1})p(r_{\theta_2})p(r_{\phi_1})p(r_{\phi_2})}{p(r_1)p(r_2)p(r_{\theta_1})p(r_{\theta_2})p(r_{\phi_1})p(r_{\phi_2})} \quad (20)$$

$$I = \frac{4\pi^4}{N} \sum_{i=1}^\infty \frac{r_{1,i}^2 r_{2,i}^2 \sin(\theta_{1,i}) \sin(\theta_{2,i}) \tilde{F}}{4e^{-4r_{1,i}} 4e^{-4r_{2,i}}} \quad (21)$$

$$I = \frac{\pi^4}{4N} \sum_{i=1}^\infty \frac{r_{1,i}^2 r_{2,i}^2 \sin(\theta_{1,i}) \sin(\theta_{2,i}) \tilde{F}}{e^{-4r_{1,i}} e^{-4r_{2,i}}} \quad (22)$$

where  $\tilde{F} = F(r_1(x_i), r_2(x_i), \theta_1(x_i), \theta_2(x_i), \phi_1(x_i), \phi_2(x_i))$  and  $r_{1,i} = r_1(x_i)$  and so on. This is relatively easy to implement in the code. Remember that `ran()` is a RNG between 0 and 1. The code would look somehow like this:

```
double r(x){          // RNG for r
    return -log(1-x);
}

double p_r(r){        // PDF of radius
    return 4*exp(-4*r);
}

double improved_monte_carlo(int N){
    double p_tp = 4*pi*pi*pi*pi;    // PDF of theta and phi
    double integral = 0;

    for (int i=0; i<N; i++){
        r1 = r(ran());
        r2 = r(ran());
        theta1 = ran()*pi;          // RNG for theta
        theta2 = ran()*pi;
        phi1= ran()*2*pi;           // RNG for phi
        phi2= ran()*2*pi;
        integral += r1*r1*r2*r2*sin(theta1)*sin(theta2)*
                    func_sphere(r1,r2,theta1,theta2,phi1,phi2) /
                    (p_r(r1)*p_r(r2));
    }
}
```

```

    }
    integral *= p_tp/N;
}

```

We will here also implement the variance as described in [subsection 2.3](#) using [Equation 17](#).

## 2.5 Parallization and optimization

In order to increase computational performance we have parallelized our improved Monte Carlo algorithm using MPI. This means we can run the Monte Carlo simulations simultaneously on multiple processors. Monte carlo simulations lend themselves very well to parallization since every simulation is independent of the others. The machine we used to test our code has two processors. With a parallelized code we can keep the number of simulations constant and have the same acurracy, but reduce the real time used as we split the tasks between the processors. With two processors the number of simulations handled by each processor is  $N/2$ .

To further increase performance we can add various compiler flags when running the program. These flags identify parts of the code which can be optimized, for example through automatic vectorization, procedure inlining, replacing division with a reciprocal and a multiplication or moving constants inside loops outside loops.[\[4\]](#)

### 3 Results

#### 3.1 Gauss-Legendre and Gauss-Laguerre

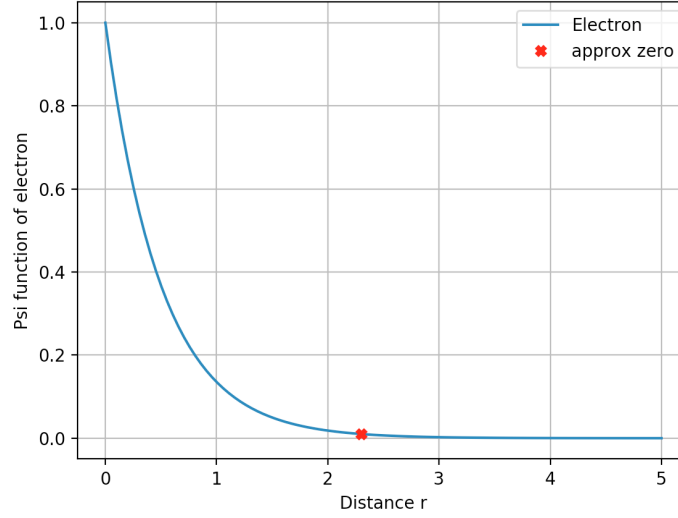


Figure 1: This plot visualizes the fact that the single particle wave function quickly approaches zero as  $r$  increases.

From [Figure 1](#) we may now determine  $\lambda \approx \infty$ . This is done by looking at the graph to approximate where the function becomes zero, that is where the function is basically zero from that point and to infinity.  $r = 2.3$ , denoted by the red cross, was found to be a satisfactory approximation as it gave us good integration results. This is the value we have used as the integration limits  $\pm\lambda$ . With this  $r$ -value in mind, we may try our code for different values of integration points  $N$  with the given limits  $\lambda = \pm 2.3$ . This is shown in [Figure 2](#) as well as the efficiency of the method in the sense of error per time taken for different  $N$ 's. In this figure we have also included the Gauss-Laguerre method. Because of the Laguerre polynomials, we don't have to find the limits as we did with the Gauss-Legendre method. We only need the number of integration points.

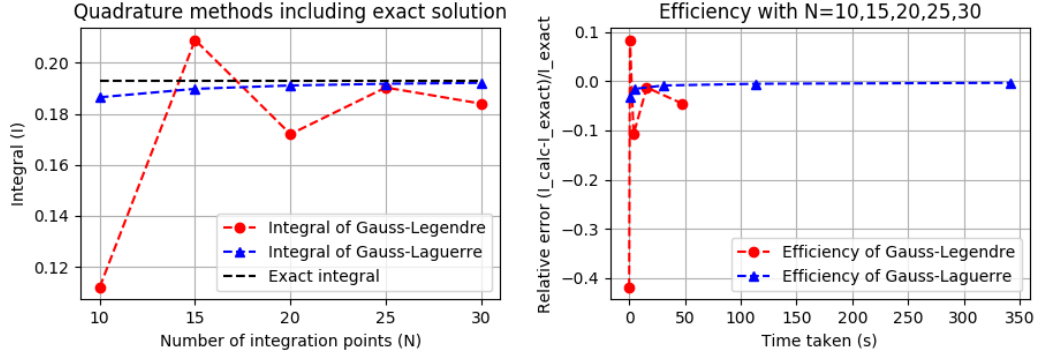


Figure 2: The left plot shows the calculated integral with the two different methods for increasing integration points. The exact integral ( $I = 5\pi^2/16^2$ ) is included in that plot. The right plot shows how fast the different methods is operating and how large the difference is from the exact solution.

Figure 2 shows us how the integration develops as the number of integration points increases. The left plot shows us that the Legendre method is more or less oscillating around the exact solution and also not converging towards the analytical solution. The Laguerre method does on the other hand converge to the solution really fast. On the right plot we see that the Legendre method is much faster than Laguerre, but the absolute error is a lot bigger.

### 3.2 Brute force Monte Carlo and Importance Sampling

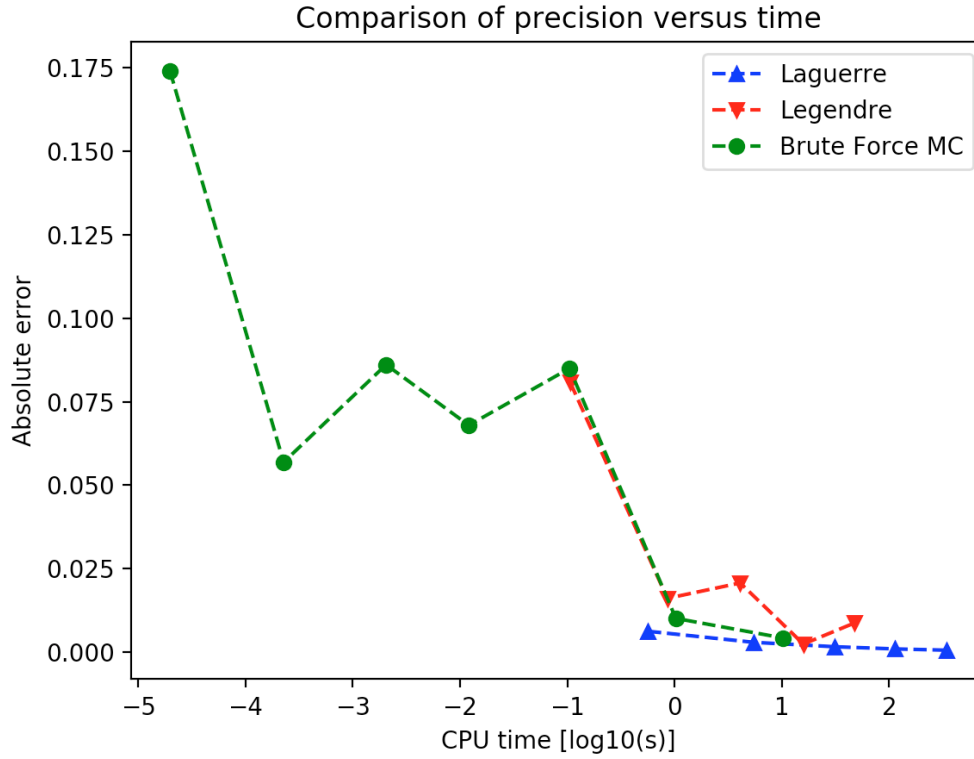


Figure 3: This plot shows a comparison of the CPU time spent to achieve a given absolute error. The absolute error is the difference between the numerical integral and the actual value of  $\frac{5\pi^2}{16^2}$ . Each point marked by a green circle on the brute force Monte Carlo line marks interval of  $10^x$  where  $x = [1, 2, 3, 4, 5, 6, 7, 8]$ . In order to achieve an accuracy of three leading digits the brute force method need  $10^8$  iterations. At this accuracy there is no significant speed up for the Monte Carlo simulation compared to Laguerre and Legendre. In fact it uses roughly 2x longer to reach the same precision as Laguerre.

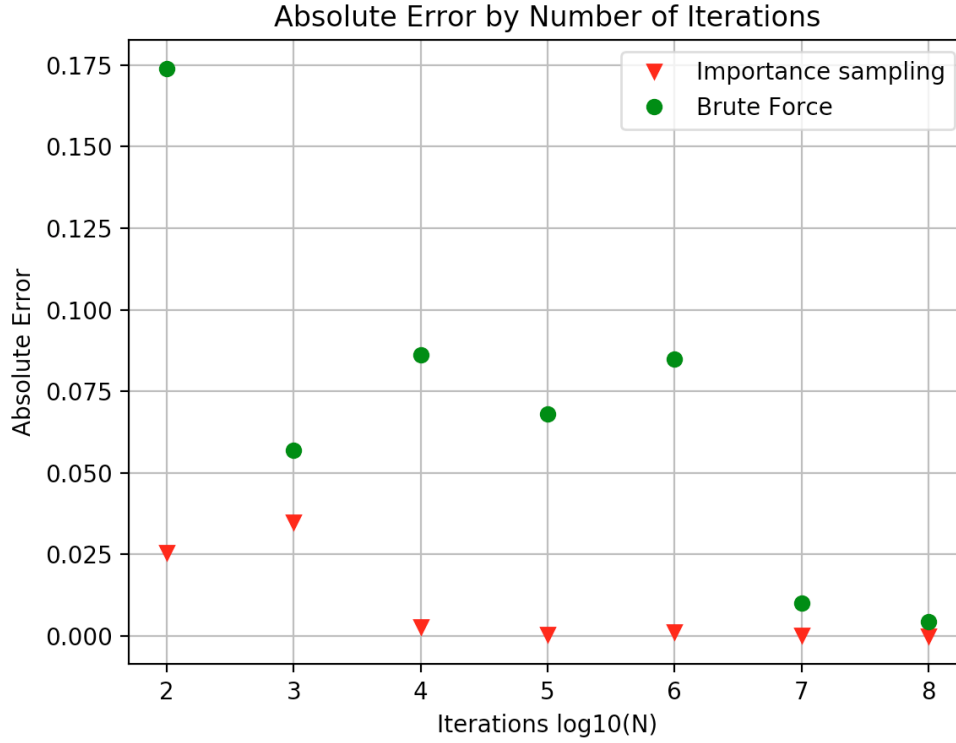


Figure 4: This plot compares the absolute error in the brute force method (Green dots) and the method using importance sampling (Red triangles). The error is plotted against the number of iterations needed to achieve the given accuracy. The number of iterations are plotted on a logarithmic scale. The error is calculated by taking the difference between the numerical integral and a known analytical value for this integral, in this case  $\frac{5\pi^2}{16^2}$ . The method utilizing importance sampling gains accuracy by the number of iterations at a much higher rate than the brute force method.

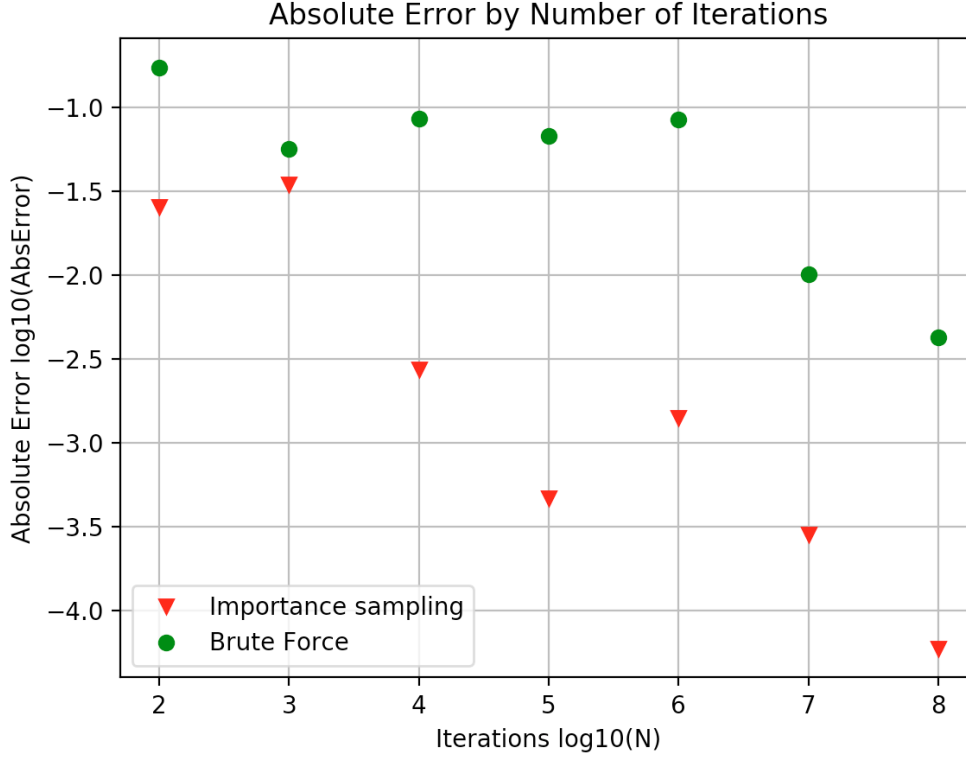


Figure 5: This plot compares the absolute error in the brute force method (Green dots) and the method using importance sampling (Red triangles). This plot uses the same data as Figure 4 but has a logarithmic scale on the y axis. The logarithmic scale allows us to observe that the error continues falling for an increasing number of iterations.

In Figure 4 we see that the methods using importance sampling clearly need less iterations than the brute force method to achieve a good accuracy. The error in the Brute force method decreases from  $N = 10 \rightarrow N = 10^2$ , but between  $N = 10^3 \rightarrow N = 10^6$  the error fluctuates about 0.075 before it finally becomes more accurate. For an accuracy of three leading digits the brute force method needs  $10^8$  iterations and 10.2804s of cpu time, while the method using importance sampling only needs  $10^4$  and 0.00352412s. The difference in iterations is by a factor of  $10^4$ , and in cpu time by a factor of 3000.

### 3.3 Parallization and optimization

Table 1: This table shows the time spent for a given number of iterations for the improved Monte Carlo method. The first column shows the number of iterations  $\mathbf{N}$ , the second column is the improved Monte Carlo without any optimization or parallization, the third column is the same method run in parallel on two processors, the fourth and fith columns are parallelized and optimised with different compiler flags.

<b>N</b>	<b>Importance [s] sampling [s]</b>	<b>Parallelized [s]</b>	<b>Parallelized + -O2 [s]</b>	<b>Parallelized + -O3 [s]</b>
$10^2$	6.673e-05	1.04346e-04	8.971e-05	9.1195e-05s
$10^3$	0.000516783	2.19668e-04	1.30941e-04	1.52967e-04
$10^4$	0.00352412	1.2813e-03	1.2098e-03	9.51482e-04
$10^5$	0.0265326	1.22907e-02	1.02255e-02	8.49046e-03
$10^6$	0.247553	1.27039e-01	9.31483e-02	8.80818e-02
$10^7$	2.47919	1.22464	8.52854e-01	8.71797e-01
$10^8$	24.7062	12.3421	8.45107	8.39174



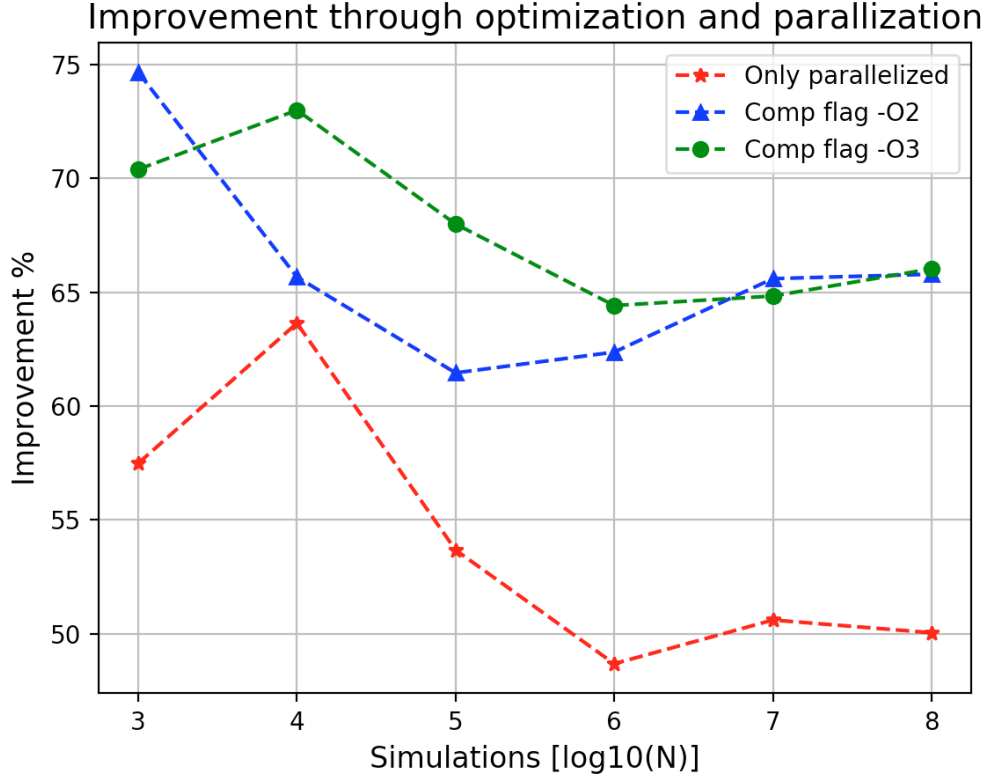


Figure 6: This figure shows the reduction in CPU time using parallization and compiler flags. The benchmark used to measure the improvement is the improved Monte Carlo method. The y axis shows the percentwise improvement from the benchmark  $\frac{T_{optimized}}{T_{benchmark}}$ . The x axis shows the number of simulations on a logarithmic scale. The red line show the case were the algorithm is parallelized to run on two processors without any optimization. The blue and green line show the parallelized algorithm also including optimization flags.

In [Figure 6](#) and [Table 1](#) we see that parallization and optimization in our case yield a significant speedup of our program. Running two processes in parallel gives us roughly a 50% increase in efficiency. Additional optimization with compiler flags -O2 and -O3 also increase the performance. The percentwise increase in performance due to optimization decreases with a growing N. For large values of N the compiler flags both approach the same value and increase the performance by an additional 15%.

## 4 Discussion

### 4.1 Gauss-Legendre and Gauss-Laguerre quadrature

From [Figure 2](#) we will notice the difference of accuracy of Gauss-Legendre and Gauss-Laguerre for increasing values of  $N$ . By looking at the left plot, we may see that the Legendre method is actually bouncing around the exact solution. It also looks like it slowly but surely somewhat converge to the exact solution. But let us remember, that at the right most points corresponds to integration points  $N = 30$ . And since we are integrating over six dimensions, the method requires  $N^6$  operations, or in this case  $30^6 \approx 730$  million iterations. So for this huge amount of calculations, it is safe to say that this method is really bad. The Laguerre method is included in the same plot and we see that it converge towards the exact solution way faster. Still, we are integrating over six dimensions but now with spherical coordinates. The method is therefore better but still very heavy.

If we look at the right plot on [Figure 2](#), we notice how time consuming each method is and the relative error from the exact answer. The leftmost dot and triangle corresponds to  $N = 10$  for Legendre and Laguerre respectively. The rightmost points is then obviously for  $N = 30$ . But whats interesting here, is the fact that the Legendre method is quite faster than the Laguerre for larger  $N$ . Nevertheless, if we take closer look at the graph, we notice that the Laguerre method hits almost the exact solution for smaller  $N$ . We could therefore say that Gauss-Laguerre is faster and more precise than Gauss-Legendre because  $N$  can be smaller for this method.

The conclusion for the Gauss-Legendre method is that we need at least a number of mesh points  $N = 25$  before our results converge at a level of third level digit. For the Gauss-Laguerre, we only need  $N = 10$  which obviously saves a lot of computation time.

### 4.2 Brute force Monte Carlo and importance sampling

Since the Monte Carlo method is based on random numbers we only need to iterate through one loop to evaluate the integral. This is because the method is not dependent on integrating over every possible mesh point throughout the three dimensional space we are working with. Instead it takes the average of those random points. With this implementation, the algorithm is able to run through a much higher value of mesh points.

From [Figure 3](#) we see the brute force Monte Carlo compared to both of the Gauss quadrature methods. First of all, we notice that Monte Carlo is faster. This should not come as a surprise by the fact that we are running a single for-loop instead of six. This lets the algorithm run for mesh points equally to a million or even more. But still, the results aren't quite that good. As we see from the same plot, the error is almost half the size of the integral for  $N = 100 \rightarrow 10^5$ . After hitting the point that corresponds to  $N = 10^5$ , we see that the accuracy is more or less the same

as the Gauss-Legendre method before it gets a little better for higher values of  $N$ . Even though it is more accurate than the Legendre method, it is still worse than the Laguerre method. There is then no point in using the brute force Monte Carlo Gauss-Laguerre quadrature is already implemented in the code.

Now let's take a look at [Figure 3](#) where the plotted green dots and the red triangle shows the absolute error of the brute force Monte Carlo and the importance sampling respectively. While the brute force demands a high number of mesh points, the importance sampling method is quite accurate after just  $N = 10^4$  mesh points. This might come from how we define our random variables between these two methods. From the brute force method, the random values are evenly spread out within some interval for all the dimension. For instance  $X_i = x * (b - a) + a$  where  $x$  is a random number between 0 and 1. The important sampling method uses an exponential distribution with  $r \in [0, \infty)$ . We found that  $r(x) = -\ln(1 - x)$  where  $x$  is the same random number. This means that  $r(x)$  is more of a compact distribution around 0 instead of  $\infty$ . We are therefore more likely to achieve a random number close to zero which might be the cause of getting smaller error much faster than the brute force. If we look at [Figure 5](#), we may notice that the error in importance sampling are more or less linearly on a logarithmic scale.

### 4.3 Parallization and optimization

From [Figure 6](#) we see that the parallization reduced the CPU time dramatically compared to the non parallelized case. This is not surprising as we are splitting heavy lifting of the program in two across two processors. It is however surprising that it actually exceeds this increase with a maximum at 64% for  $N = 10^4$ . This could however be a false result caused by inaccuracy in the time measurement as we are operating at tiny time intervals. The improvement seems to converge around 50% for large values of  $N$ . This shows that there is significant potential for large increases in computational efficiency through parallization. It would be interesting to see if the efficiency increases linearly with an increased number of cores, but we did not have any available machines for this project. An increased number of cores may however be somewhat slowed down by overhead of synchronization or communication

- O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

- O2 Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.

- O3 Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload and -ftree-vectorize options.

## 5 Conclusion

The code for parallization is a bit quick and dirty. It only allows for two processors because this is the number of available processors on our test machine. It would of course be much better to make a more general code that can run more processes if they are available.

We have not taken into consideration that we could reduce the number of floating point operations per second. This could be heavily reduced especially with the Gauss quadrature methods since we are running a six dimensional for-loops. The difference of operation is then a stunning factor of  $n^6 - (n - 1)^6$  if we reduce our FLOPS with just a value of one.

## References

- [1] Hjorth Jensen M. Computational Physics Lectures: Numerical integration; 2019. Available from: <http://compphysics.github.io/ComputationalPhysics/doc/pub/integrate/pdf/integrate-print.pdf>.
- [2] Hjorth Jensen M. Computational Physics, Lecture Notes; 2015. Available from: <https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf>.
- [3] Hjorth Jensen M. Computational Physics Lectures: Introduction to Monte Carlo methods; 2019. Available from: <http://compphysics.github.io/ComputationalPhysics/doc/pub/mcint/pdf/mcint-print.pdf>.
- [4] Hjorth Jensen M. Computational Physics Lectures: How to optimize your code, from compiler flags to vectorization and parallelization; 2019. Available from: <http://compphysics.github.io/ComputationalPhysics/doc/pub/codeoptimization/html/codeoptimization-reveal.html>.