Draft version October 22, 2019
Preprint typeset using LaTeX style emulateapj v. 12/16/11

COMPUTATIONAL PHYSICS - PROJECT 3

Dahl, Jon Kristian
Fløisand, Johan Andreas
Sand, Mats Ola

Draft version October 22, 2019

ABSTRACT

When solving integrals numerically, one can often divide the interval of interest in equally spaced point, for then to evaluate the sum of the function values in these points. Examples are the trapezoidal method and Simpsons method, but such methods are usually not the best, nor the most efficient ways of computing integrals. This report focuses on two general ways to solve integrals. Gaussian quadrature using Legendre and Laguerre polynomials, and Monte Carlo integration with random numbers drawn from a uniform and exponential distribution. We see that Gauss-Legendre quadrature yields errors as low as $\epsilon \approx 10^{-5.28}$ for $N \approx 8.5 \cdot 10^7$ iterations, and that Gauss-Laguerre quadrature has lower errors in all but this specific case. The Monte Carlo integration outperforms the Gaussian quadrature methods by converging much quicker towards the exact result. The brute force Monte Carlo integration maxes out with errors of $\epsilon \approx 10^{-2}$ and goes as low as $\epsilon \approx 10^{-5.93}$. The improved Monte Carlo integration rarely exceeds errors of $\epsilon \approx 10^{-4}$ and goes as low as $\epsilon \approx 10^{-6}$. The Monte Carlo integration has also been parallelized. Parallelization of the code resulted in a speed-up of approximately 6.66 times that of the non-parallelized code.

1. INTRODUCTION

The postulates of quantum mechanics tells us that a system tends to follow the expectation value of an operator, like the position operator or the Hamiltonian (Griffiths & Schroeter 2018, Chapter 3). It is therefore essential to be able to evaluate integrals of all types. For integrals where precision is important, and for multi-dimensional integrals, efficient algorithms are needed to keep the computation time low.

An elegant and quite efficient way of computing integrals are the family of Gaussian quadrature. It allows for non-fixed step lengths and can give exact results when integating polynomials. For lower dimensional integrals, Gaussian quadrature is one of the better integration methods, but it often falls short of Monte Carlo integration. In this report we will study how a six dimensional integral can be evaluated by Gaussian quadrature and by Monte Carlo integration.

We begin our report with a theory section giving a quick introduction to Gaussian quadrature and Monte Carlo integration. The integral we wish to evaluate with the different integration methods is introduced, and a change of variables is given. The method section explains the way we solve and analyse the data generated from the different integration methods. We explain how a variable change and a change of distribution of random numbers can give smaller and more stable errors. A result and discussion section then follow where we show the results of our computation and discuss implications around the different methods. We end this report with a conclusion where we determine which of the algorithms are the best for our purpose.

All code used to generate data in this report is supplied in a GitHub repository at `https://github.com/johanafl/FYS3150-4150/tree/master/project3`.

2. THEORY

2.1. *Gaussian quadrature*

Gaussian quadrature is an elegant integration method which uses roots of orthogonal polynomials instead of equally spaced points on the number line to compute an integral. The main theorem even tells us that we get exact results if the function we want to integrate can be represented by a polynomial.

We shall now take a closer look at the mathematics. The theorems we refer to are all from (Stoer & Bartels 1992, chapter 3.6)[1]. We assume that we are trying to compute an integral

$$I = \int_a^b W(x) f(x)\, dx,$$

where $W(x)$ is called the "weight function". The interval $[a, b]$ can be finite or infinite. The general assumptions are:

(a) $W(x) \geq 0$ is measurable on the finite or infinite interval $[a, b]$.

(b) All moments $\mu_k = \int_a^b x^k W(x)\, dx$, $k = 0, 1, \ldots,$ exist and are finite.

(c) For polynomials $s(x)$ which are non-negative on $[a, b]$, $\int_a^b W(x) s(x)\, dx = 0$ implies $s(x) = 0$.

These conditions are, for instance, met if $W(x)$ is positive and continuous on a finite interval $[a, b]$. It is also possible to show that condition (c) is equivalent to $\int_a^b W(x)\, dx > 0$.

[1] Other textbooks, such as Press et al. (2007) and Hjorth-Jensen (2015), also give an introduction to the subject, but with more focus on the implementation and application. We have chosen to use Stoer & Bartels (1992), because we feel that it gives a more general understanding of the mathematics.

We introduce the notation (same as Stoer & Bartels (1992))

$$\bar{\Pi}_j = \{p \,|\, p(x) = x^j + a_1 x^{j-1} + \ldots + a_j\}$$

for the set of normed real polynomials of degree j and

$$\Pi_j = \{p \,|\, \text{degree}(p) \leq j\}$$

for the linear space of all polynomials whose degree does not exceed j. We also define the inner product

$$\langle f,\, g\rangle = \int_a^b W(x) f(x) g(x)\, dx$$

on the linear space $L^2[a,b]$ of all functions for which the integral

$$\langle f,\, f\rangle = \int_a^b W(x)(f(x))^2\, dx$$

is well defined and finite. We say that the functions $f,\, g \in L^2[a,b]$ are orthogonal if $\langle f,\, g\rangle = 0$[2].

The first theorem is

Theorem: *There exist polynomials $p_j \in \bar{\Pi}_j,\ j = 0,\, 1,\, 2,\, \ldots,$ such that*

$$\langle p_i,\, p_k\rangle = 0 \qquad \text{for } i \neq k.$$

These polynomials are uniquely defined by the recursions

$$p_0(x) = 1,$$
$$p_{i+1}(x) = (x - \delta_{i+1}) p_i(x) - \gamma_{i+1}^2 p_{i-1}(x) \quad \text{for } i \geq 0,$$

where $p_{-1}(x) = 0$ and

$$\delta_{i+1} = \langle x p_i,\, p_i\rangle / \langle p_i,\, p_i\rangle \quad \text{for } i \geq 0,$$
$$\gamma_{i+1} = \begin{cases} 0 & \text{for } i = 0, \\ \langle p_i,\, p_i\rangle / \langle p_{i-1},\, p_{i-1}\rangle & \text{for } i \geq 1. \end{cases}$$

This theorem tells us that our orthogonal polynomials span the space. Hence, since every polynomial $p \in \Pi_k$ is presentable as a linear combination of the orthogonal polynomials $p_i,\ i \leq k$, we must have

Corollary: $\langle p,\, p_n\rangle = 0$ *for all* $p \in \Pi_{n-1}$.

Next we have

Theorem: *The $n \times n$ matrix*

$$A = \begin{bmatrix} p_0(t_1) & \ldots & p_0(t_n) \\ \vdots & & \vdots \\ p_{n-1}(t_1) & \ldots & p_{n-1}(t_n) \end{bmatrix}$$

is non-singular[3] *for mutually distinct arguments $t_i,\ i = 1,\, \ldots,\, n$.*

This theorem shows that the interpolation problem of finding a function of the form

$$p(x) = \sum_{i=0}^{n-1} c_i p_i(x)$$

with $p(t_i) = f_i,\ i = 1,\, \ldots,\, n$ is always uniquely solvable. The main result is the following theorem

Theorem:

(a) *Let $x_1,\, \ldots,\, x_n$ be the roots of the nth orthogonal polynomial $p_n(x)$, and let $w_1,\, \ldots,\, w_n$ be the solution of the (non-singular) system of equations*

$$\sum_{i=1}^{n} p_k(x_i) w_i = \begin{cases} \langle p_0,\, p_0\rangle & \text{if } k = 0, \\ 0 & \text{if } k = 1,\, 2,\, \ldots,\, n-1. \end{cases} \tag{1}$$

Then $w_i > 0$ for $i = 1,\, 2,\, \ldots,\, n$, and

$$\int_a^b W(x) p(x)\, dx = \sum_{i=1}^{n} w_i p(x_i) \tag{2}$$

holds for all polynomials $p \in \Pi_{2n-1}$. The positive numbers w_i are called "weights."

(b) *Conversely, if the numbers $w_i,\ x_i,\ i = 1,\, \ldots,\, n$, are such that (2) holds for all polynomials $p \in \Pi_{2n-1}$, then the x_i are the roots of p_n and the weights w_i satisfy (1).*

(c) *It is not possible to find numbers $x_i,\, w_i,\, i = 1,\, \ldots,\, n$, such that (2) holds for all polynomials $p \in \Pi_{2n}$.*

This means that we can approximate the function $f(x)$ by a polynomial p of degree $\leq 2n-1$, and then compute the integral of this polynomial exactly. The proof of the theorem explains how we can do this by evaluating the remainder polynomial r after polynomial division p/p_n.

For the most common weight function $W(x) = 1$ and the interval $[-1, 1]$, the result of the main theorem are due to Gauss. The corresponding orthogonal polynomials are

$$p_k(x) = \frac{k!}{(2k)!} \frac{d^k}{dx^k}(x^2 - 1)^k, \qquad k = 0,\, 1,\, \ldots,$$

which is nothing more than the Legendre polynomials (up to a factor).

For the weight function

$$W(x) = x^a e^{-x} \tag{3}$$

and the interval $[0, \infty)$, the corresponding orthogonal polynomials are (up to a factor) the Laguerre polynomials (Hjorth-Jensen 2015, chapter 5.3)

$$L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} \left(x^n e^{-x}\right).$$

The numerical implementations we have used for the Gaussian quadrature can be found at Hjorth-Jensen (2019).

2.2. *Monte Carlo*

Let[4] X be a random variable drawn from a probability distribution with probability distribution function

[2] The notion of the L^2-spaces and inner products are very intuitively explained in Lindstrøm (2017). It might be worth checking out this book if you are not familiar with such abstract vector spaces. We will hopefully be able to explain without to much foreknowledge required.

[3] A non-singular matrix is equivalent to an invertible matrix. The other way around, a singular matrix is the same as non-invertible matrix (David C. Lay 2016, chapter 2).

[4] Most of the notation and language is taken form Devore & Berk (2011).

(PDF) $p_X(x)$, for $x \in (-\infty, \infty)$[5]. The expectation value of X is defined as

$$E[X] = \int_{-\infty}^{\infty} x p_X(x)\, dx.$$

Similarly, the expectation value of a function of a random variable $f(X)$ is defined as

$$E[f(X)] = \int_{-\infty}^{\infty} f(x) p_X(x)\, dx.$$

An other important value is the variance of a random variable,

$$V[X] = \int_{-\infty}^{\infty} (x - E[X])^2 p_X(x)\, dx.$$

This quantity gives us a value for how much the randomly drawn variable X varies from the mean. As for expectation value, we define the variance of a function of a random variable $f(X)$ as

$$\begin{aligned} V[X] &= \int_{-\infty}^{\infty} (f(x) - E[f(X)])^2 p_X(x)\, dx \\ &= \int_{-\infty}^{\infty} f(x)^2 p_X(x)\, dx - 2E[f(X)] \int_{-\infty}^{\infty} f(x) p_X(x)\, dx \\ &\quad + E[f(X)]^2 \int_{-\infty}^{\infty} p_X(x)\, dx \\ &= E[f(x)^2] - 2E[f(X)]E[f(X)] + E[f(X)]^2 \\ &= E[f(x)^2] - E[f(X)]^2 \end{aligned}$$

Now, given an integral $\int_a^b f(x)\, dx$, we can use the fact that the PDF of the uniform distribution on the interval $[a, b]$ is given by $u(x) = \frac{1}{b-a}$. Hence,

$$\int_a^b f(x)\, dx = \int_a^b f(x) \frac{b-a}{b-a}\, dx = (b-a) \int_a^b f(x) u(x)\, dx.$$

Thus, interpreting our function $f(x)$ as a function of a random variable, this integral is nothing else than the expectation value of the uniformly distributed, random variable, $f(x)$[6],

$$\int_a^b f(x)\, dx = (b-a)E[f(X)] = (b-a)\langle f(x) \rangle.$$

In the same manner, we can define the variance as

$$\begin{aligned} V &= \int_a^b ((b-a)f(x) - E[(b-a)f(X)])^2 \frac{1}{b-a}\, dx \\ &= (b-a)^2 V[f(X)]. \end{aligned}$$

[5] If the probability for observing X only exist on the interval $[a, b]$, then the PDF $\widetilde{p}_X$ is defined such that

$$\widetilde{p}_X(x) = \begin{cases} 0 & \text{for } x > b \text{ and } x < a \\ p_X(x) & \text{for } x \in [a, b] \end{cases}$$

[6] The expectation value, often referred to as the mean value, have many different notations. The most used ones are $E[X]$, μ and $\langle X \rangle$. We will mostly be using the last notation for most of the remainder of the text.

The mean can now easily be estimated by taking the average of N randomly drawn numbers x_i:

$$\frac{1}{N} \sum_{i=1}^{N} f(x_i).$$

Solving the integral by finding the expectation value for the function evaluated at randomly drawn points is called Monte Carlo integration. This gives us a way to estimate the integral, and by computing

$$\begin{aligned} &(b-a)^2 \left(\langle f(x)^2 \rangle - \langle f(x) \rangle^2 \right) = \\ &(b-a)^2 \left(\frac{1}{N} \sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N} \sum_{i=1}^{N} f(x_i) \right)^2 \right), \end{aligned} \tag{4}$$

we are also able to estimate how much the computed value might vary from the exact value. All though this is a biased estimator for the variance Devore & Berk (2011), we choose this instead of the unbiased one

$$\frac{(b-a)^2}{N-1} \left(\sum_{i=1}^{N} f(x_i)^2 - \frac{1}{N} \left(\sum_{i=1}^{N} f(x_i) \right)^2 \right)$$

due to easier implementation when programming.

The final trick of this section is called importance sampling. If we assume that our original function $f(x)$ contains a known PDF $p(x)$, that is $f(x) = g(x)p(x)$, then we do not need to introduce the uniform distribution:

$$\int_a^b f(x)\, dx = \int_a^b g(x) p(x)\, dx = E[g(X)].$$

In other words, we can rather compute the expectation value of the function $g(x)$ than $f(x)$. This can be thought of as drawing random variables that fits the function better.

2.3. *The quantum integral*

The single-particle wave function for an electron i in the $1s$ state is given in terms of a dimensionless variable

$$\mathbf{r}_i = x_i \mathbf{e}_x + y_i \mathbf{e}_y + z_i \mathbf{e}_z$$

as

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

where α is a parameter and

$$r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}.$$

We will fix α to represent the charge of the helium atom $Z = 2$, i.e., $\alpha = 2$.

The ansatz for the wave function for two electrons is then given by the product of two $1s$ wave functions:

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

The integral we want to solve is

$$\left\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \right\rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2 \cdot 2(r_1 + r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}, \tag{5}$$

which is the quantum mechanical expectation value of the correlation energy between two electrons that repel each other via the classical Coulomb interaction. The keen reader might notice that there is a normalization factor missing. We have excluded this factor since we are more interested in computing a difficult integral using the methods explained above.

Since the integrand is spherically symmetric, we might also want the integral in spherical coordinates:

$$\int_{-\infty}^{\infty}\cdots\int_{-\infty}^{\infty} e^{-2\alpha(r_1+r_2)}\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|}d\mathbf{r}_1d\mathbf{r}_2 \tag{6}$$

$$=\int_0^{2\pi}\int_0^{2\pi}\int_0^{\pi}\int_0^{\pi}\int_0^{\infty}\int_0^{\infty} e^{-2\alpha(r_1+r_2)}\frac{1}{\sqrt{r_1^2r_2^2-2r_1r_2\cos(\beta)}}$$
$$r_1^2dr_1r_2^2dr_2\sin(\theta_1)d\theta_1\sin(\theta_2)d\theta_2d\phi_1d\phi_2.$$

The motivation for this coordinate transformation is that we recognize the $e^{-\widetilde{x}}$ term as a both part of the weight function from the Laguerre polynomials and a part of the exponential distribution function. Since the radial coordinate in spherical coordinates resides in the interval $[0,\infty)$, and the Laguerre polynomials are defined in this very interval, a change of coordinates is wanted.

The exact result of the integral is $15\pi^2/16^2$.

3. METHOD

3.1. *Gauss-Legendre quadrature*

As the the theory section states, Gauss-Legendre quadrature approximates an integral of a function as a weighted sum of function values at specified points within integration domain using Legendre polynomials. Since the Legendre polynomials are defined in the domain $[-1,1]$, and the integral we wish to evaluate is integrated over the limits $(-\infty,\infty)$, we need to make a scaling/mapping of the limits to make them fit the domain of the Legendre polynomials. Since we are unable to use infinity, we will approximate $\pm\infty$ with a value λ.

The six dimensional integral has the exact same integral limits in the cartesian coordinate representation, so only a single integral limit scaling and infinity approximation is needed. Thus, by looping over number of grid points, N_g and integral limits/infinity approximation values, λ, we can make a contour plot of the calculation error for each pair for N_g and λ and decide where the best results are located.

Another point of interest is the amount of time that is needed for the Gauss-Legendre quadrature method to reach a sufficiently low error. We have run the Gauss-Legendre quadrature method for a set of grid points to see which number of grid points, along with a suitable infinity approximation, λ, produces the lowest error at the lowest amount of time. This is simply done by looping over a suitable amount of grid values while writing the time spent on each N_g to a file. The resulting data file is then read and analysed with a Python script, as we will see in the results section. All timing values are averaged over a loop of 10 runs.

The domain mapping, Legendre points and weights are all handled in the function gauss_legendre_points() which is, except for a name change, an exact copy of the supplied C++ function gauleg() Hjorth-Jensen (2019).

3.2. *Gauss-Laguerre quadrature*

Gauss-Laguerre quadrature approximates an integral of a function as a weighted sum of function values at specified points within integration domain using Laguerre polynomials. An important distinction from the Legendre polynomials is that the Laguerre polynomials are defined in the domain $[0,\infty)$, and they are therefore better suited for the spherical coordinate representation of the integral, since it contains the weight function (3) of the Gauss-Laguerre quadrature.

A coordinate transformation from cartesian to spherical coordinates gives us polar, azimuthal and radial components θ,ϕ,r. We use Laguerre polynimials for the radial coordinate and Legendre polynomials for the angular coordinates. Since none of the angular integral limits are infinity, we dont need to think about any infinity approximations like we did in the previous section.

In the coordinate transformation to spherical coordinates we get a Jacobi determinant along for the ride, containing a r^2 term. This term, along with the e^r term is absorbed in the weight function of the Gauss-Laguerre quadrature and is therefore not present in the integrand in the code.

We can then easily run the Gauss-Laguerre quadrature method for a set of grid values, for then to compare the times and errors for the different values.

The Laguerre points and weights are all handled in the C++ program gauss_laguerre.cpp, function gauss_laguerre() which is an exact copy of the supplied C++ file gauss-laguerre.cpp Hjorth-Jensen (2019).

3.3. *Monte Carlo Integration*

Monte Carlo integration is a particular Monte Carlo method that numerically computes a definite integral using random numbers. As seen in the theory section, we can approximate the integral by calculating the expectation value of the integrand using randomly drawn numbers as function input values. The numbers are drawn using the pseudo-random number generator (PRNG) Mersenne Twister based on the Mersenne prime $2^{19937}-1$. A seed based on the UNIX epoch is given to the number generator to easily give a new unique seed for every run unless the program is compiled and run twice within a second.

There is no need to pick points far outside of the function domain of interest, since these points produce function values which are for our purpose, approximately zero. This applies for the numbers drawn from the PRNG. The integration scope is defined by the interval of the PDF. To find a suitable interval we will use the same method of determining the infinity approximation as we did with the Legendre polynomials in subsection 3.1. We loop over different numbers of Monte Carlo iterations, N, and for each N, we loop over a set of infinity approximations, λ, and present the result as a contour plot.

We are using a brute force Monte Carlo integration which means that we draw all the random numbers from a uniform distribution. As we will see in the next subsection, a uniform distribution gives us a bad match of random function values for the integrand, and by choosing another distribution we can draw random numbers more suited for the integrand.

An important notice for the later comparisons of the different integration methods is that N for Monte Carlo refers to the number of iterations, and N_g for Legendre and Laguerre refers to grid values, or more specifically, number of integration points for each of the six integrals. Number of iterations for Legendre and Laguerre will then be $N = N_g^6$.

3.4. *Monte Carlo Integration improved*

The Monte Carlo Integration described in subsection 3.3 uses random numbers drawn from a uniform distribution. By choosing a distribution which fits the integrand better we can increase the accuracy of the calculation. This is due to the fact that a uniform distribution favors no number over the other in its interval, and therefore yields a lot of numbers which resides in the non-interesting part of the function domain.

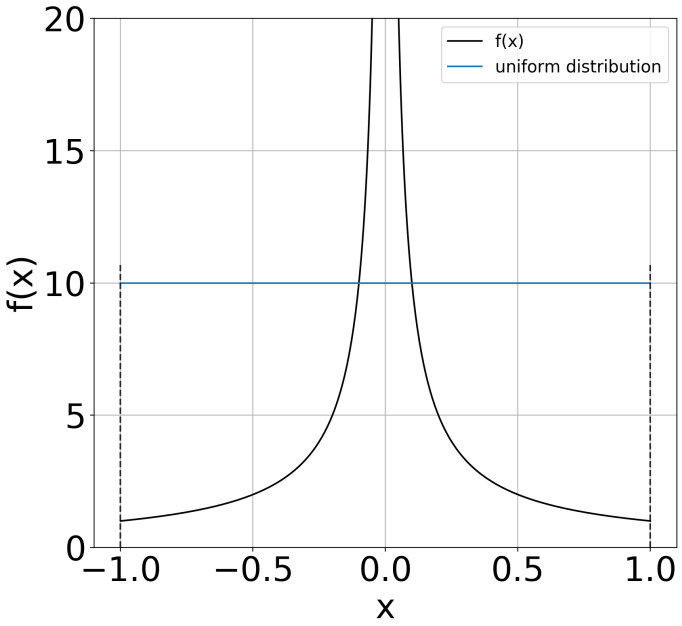

FIG. 1.— A function $f(x)$ plotted together with a uniform distribution interval. This illustration shows that the edges of the uniform distribution overlaps areas of the function which has negligible function values and are therefore a waste of computer resources compared to drawing more suited random numbers.

In figure 1 we can see how the interval of a uniform distribution overlaps a function $f(x)$. The function $f(x)$ has a form very similar to the integrand, but it is a bit tweaked for better illustration. At the edges of the distribution we find low function values which are negligible or at least, less important than function values further towards the center. This means that drawing random numbers from a uniform distribution will give a lot of non-important function values. By changing the distribution of the random numbers we can get lower computing errors by choosing more relevant numbers. There is no need in using $x = 10^{10}$ since the function value of this point is basically zero.

In figure 2 we can see the same function $f(x)$ as in figure 1, but now with a distribution function

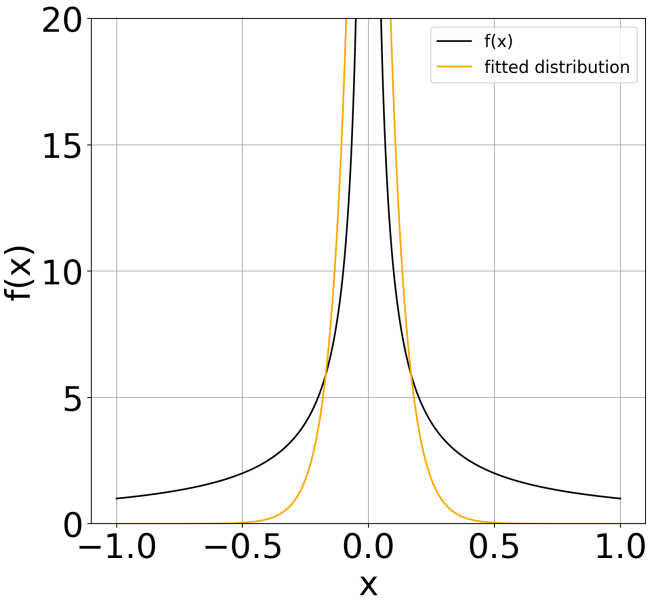

FIG. 2.— A function $f(x)$ plottet together with a better fitting distribution function. The better fitting distribution interval follows the form of the function and will therefore yield random numbers more suited for the function domain.

$$P(x) = \Lambda/2 \begin{cases} e^{-\Lambda x} & \text{for } x \geq 0 \\ e^{\Lambda x} & \text{for } x < 0 \end{cases}.$$

By choosing an appropriate value Λ we can fine-tune the distribution to match the integrand, which ultimately leads to drawing of random numbers that are more appropriate for the given integrand.

Using the spherical coordinate representation of the integrand (6), we recognize the exponential distribution function. The angular coordinates on the other hand are well suited for a uniform distribution in the interval $[0, \pi]$ and $[0, 2\pi]$ for the θ and ϕ coordinates respectively.

This is also known as importance sampling as described in the theory section.

3.5. *Monte Carlo Integration improved and parallelized*

Monte Carlo integration is based on computing the expectation value of an integrand evaluated in randomly drawn points. An important thing about this process is that no term in the sum is dependent on the result from another term. This means that all the terms, that is all the N different evaluations of the integrand for randomly drawn numbers, are independent of each other. They can be summed in any order we want them to. This kind of behaviour is well suited for parallelization, where we can let the computer run several evaluations of the integrand at the same time.

For parallelization we have used the tool MPICH2[7]. MPICH2 allows us to run the MC integration program

[7] The process of installing, setting up and using MPICH2 is described quite well in this tutorial: https://mpitutorial.com/tutorials/

(or any C++ program) in parallel by assigning the program to several threads. Threads are a way for a program to divide itself into two or more simultaneously running tasks. We are then effectively running the program in parallel n times, where n is the number of threads. MPICH2 allows us to define a master thread which gather all the individual sums calculated by every single thread, add the individual sums together, and divide the sum by the number of threads. In theory, running the program with 10 threads would speed up the computation by a factor of 10, but in reality this will vary depending on the type of processor and the amount of other work the processor has to deal with at the same time as running the MC integration.

4. RESULTS AND DISCUSSION

4.1. *Gauss-Legendre and Gauss-Laguerre quadrature*

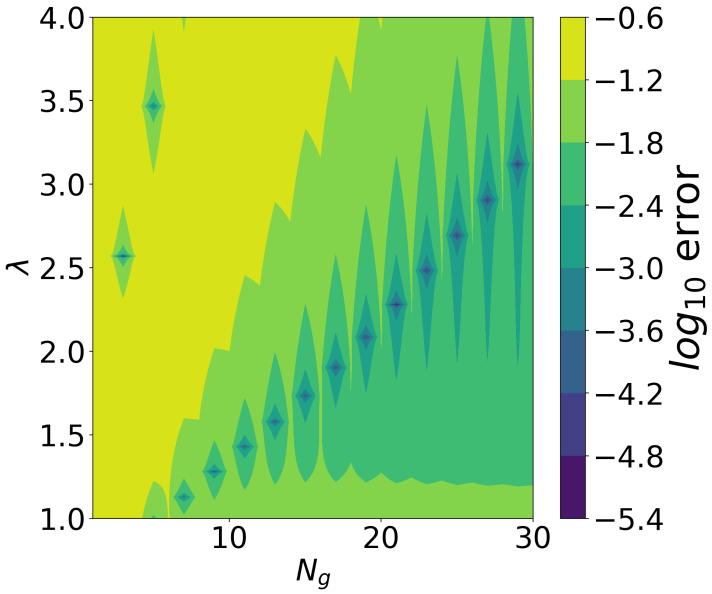

FIG. 3.— Contour plot showing grid values N_g on the x axis and infinity approximations/integral limits, λ on the y axis. The height (color) values represent the error for each pair of N_g and λ values, and are scaled with $\log_{10}$ to better distinct the small variations. The error is calculated from the known exact integral value of $5\pi^2/16^2$. The best pair of values are $N_g = 21$, $\lambda = 2.28$ with an error of $\approx 10^{-5.28}$.

Figure 3 shows a contour plot with grid valuess, N_g on the x axis and infinity approximations/integral limits, λ on the y axis. The height (color) values represent the error for each pair of N_g and λ values, and are scaled with $\log_{10}$ to better distinct the small variations. From the contour plot we can see that a higher N_g generally gives a better error, but only if we pair it with a correct λ. Consider now $\lambda \approx 1.5$. For this value of λ and $N_g \in [20, 30]$ we get approximately the same error value even though we have a larger grid value N_g, and a larger N_g grid value costs CPU time. The number of iterations is linked to the grid value by $N = N_g^6$. We see that Gauss-Laguerre quadrature has a faster rising computation time than Gauss-Legendre quadrature, but this is only one side of the story. In figure 5 we see the computation time as a function of the error.

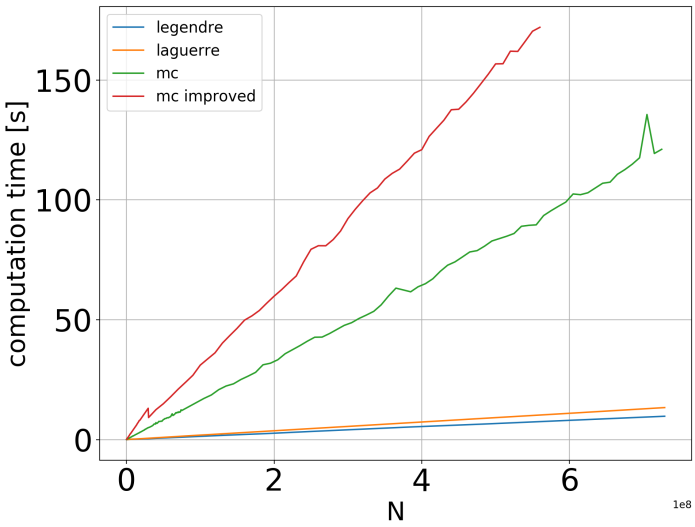

FIG. 4.— Gauss-Legendre quadrature, Gauss-Laguerre quadrature, MC, and MC improved plotted for number of iterations N as a function of computation time. The x axis is scaled with 10^7. All graphs rise linearly. Note that the number of iterations for Legendre and Laguerre is calculated by $N = N_g^6$.

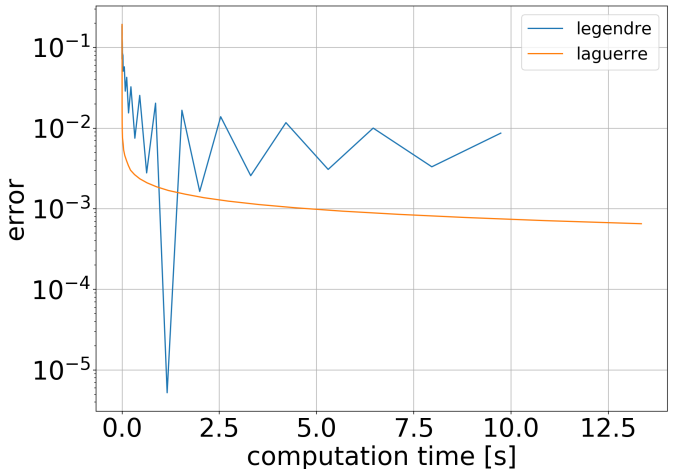

FIG. 5.— The computation time as a function of the error for Legendre and Laguerre. Legendre has in general a larger error for the same computation time as Laguerre, but has a significantly lower error of $\approx 10^{-5.28}$ in the point $N_g = 21$, $\lambda = 2.28$ as described in figure 3. Worth noting is that Laguerre has a much more consistent error than Legendre.

Gauss-Legendre quadrature has in general a larger error for the same computation time as Gauss-Laguerre quadrature, but with a significant drop in the error at $N_g = 21$, $\lambda = 2.28$ as described in figure 3. This error of $\approx 10^{-5.28}$ is where the "perfect" combination of N_g and λ is located. This shows that Gauss-Legendre quadrature can indeed be quicker at finding a better approximation of the integral than Gauss-Laguerre quadrature, but only

with considerable beforehand data analyzation which is only possible since we know the analytical result of the integral. With its error more consistent, and lower than the error of Gauss-Legendre quadrature, Gauss-Laguerre quadrature is the preferred method of the two, since we know that Laguerre will in general always have a lower error with the same computation time as Legendre.

There are three reasons why we do not want to use $\pm\infty$ as the integral limits, and the obvious one is that a computer has no way to represent infinity. A solution to this is to pick a sufficiently large number which is large enough to make the integrand behave as if the large number is infinity. To briefly illustrate this, consider the function $f(x) = 1/x^2$. The function value will quickly decrease as x increases, and at a finite value x_i the function value will be approximately zero. We have decided that an appropriate x_i value is one where the function value has decreased at least a thousandfold from the highest function value, and we started our search for λ with that in mind.

The second reason as why not to use infinity or just any very large number for the integral limits is that a lot of the domain values of the function produce non-interesting function values. By non-interesting we mean that $f(x) = 1/x^2$ for $x = 10^{10}$ produces a so small value compared to $x = 1$ that we dont even consider it. Spending time and resources on these values as compared to actually interesting function values is a waste.

The third reason as why not to use infinity or just any very large number for the integral limits is that the resolution of the grid of points which we use to evaluate the integral will be very poor. If we have a fixed amount of grid valuess N_g, and the interesting part of the function domain is $[-2, 2]$, we get fewer points in the interesting part of the domain if we choose the integral limits to be $\pm 10^{10}$ as compared to ± 4.

4.2. *Monte Carlo Integration*

Figure 6 shows a comparison of Gauss-Legendre quadrature, Gauss-Laguerre quadrature, MC, and MC improved plotted for iterations as a function of error. Even with its very best combination of N_g and λ as described in figure 3, Legendre is not better than the improved Monte Carlo integration at much fewer iterations. Both MC integrations have a much lower error than Legendre and Laguerre. It is apparent that Monte Carlo integration is the best option due to its much lower computation time.

In figure 4 it is also revealed that the improved Monte Carlo integration is in fact a bit slower than the non-improved, but from figure 6 we see that improved Monte Carlo consistently yields lower errors at a factor of 10^{-1} and lower.

4.2.1. *Approximating infinity*

As mentioned in the method section, we need to find an appropriate interval for the uniform distribution. Remember that this defines the integral limits and therefore effectively is an approximation to infinity. With the same technique as as for the Laguerre integral limits, we get the contour plot in figure 7. In contrast with figure 3 there are no apparent patterns in the plot. For values $\lambda \approx 1.5$ and below we seem to get the same error regardless of number of iteration. These λ values are therefore a poor approximation to infinity, and a poor choice of integral limits. We have extracted the best error values with the accompanying number of iterations and integral limits, and the numbers are $N = 7.8 \cdot 10^6$, $\lambda = 2.5$ with an error of $\approx 10^{-5.93}$. This value λ was the chosen value in the previous study of the non-improved Monte Carlo integration.

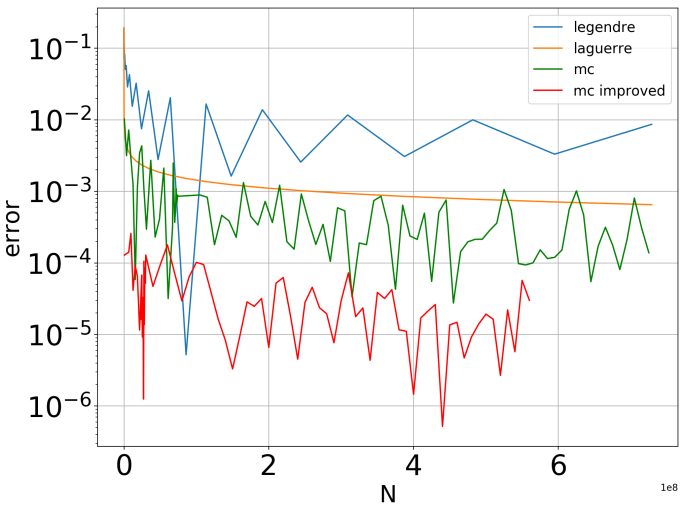

FIG. 6.— Gauss-Legendre quadrature, Gauss-Laguerre quadrature, MC, and MC improved plotted for iterations, N as a function of error. The x axis is scaled with 10^8. The MC integrations have consistently lower errors than Legendre and Laguerre at fewer iterations. The "perfect" point of $N_g = 21$, $\lambda = 2.28$ for Legendre is visible as a major dip in the Legendre graph. At its best, Legendre is not better than the improved Monte Carlo integration at fewer iterations.

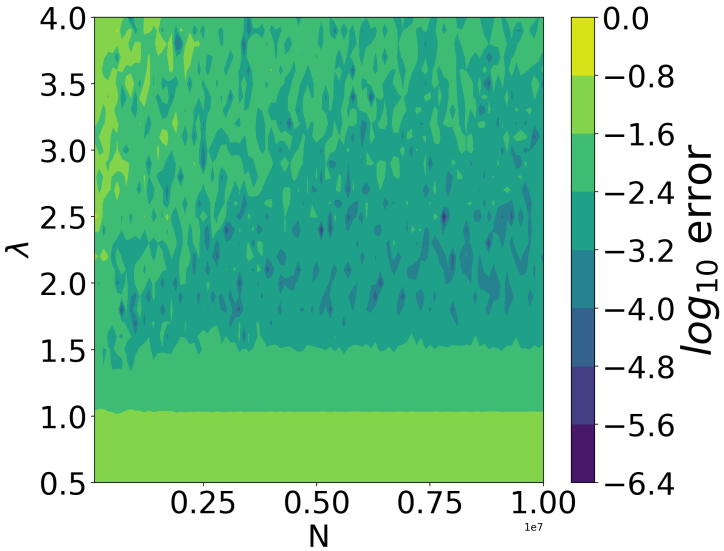

FIG. 7.— Contour plot showing number of iterations N on the x axis and infinity approximations/integral limits, λ on the y axis. The x axis is scaled with 10^7. The height (color) values represent the error for each pair of N and λ values, and are scaled with $\log_{10}$ to better distinct the small variations. The error is calculated from the known exact integral value of $5\pi^2/16^2$.

4.2.2. *Variance*

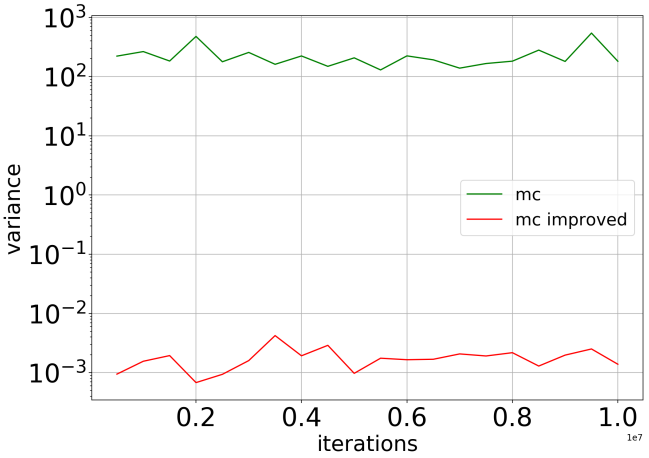

FIG. 8.— Both Monte Carlo implementations have a fairly constant variance, and the improved lies several orders of magnitude below the non-improved. We can see that the variance does not depend in the number of iterations. The x axis is scaled with 10^7.

From figure 8 it is apparent that the variance for both implementations of Monte Carlo integration is fairly constant. The variance is calculated using equation (4). The improved Monte Carlo implementation has a lower variance of a factor of approximately 10^{-5} for all iterations. We can see from the figure that the variance is not dependent on the number of iterations. This is an indication that we have a good estimator for the variance. This means that we can use the central limit theorem Devore & Berk (2011) to find how many iterations we need to get as low variance as we want:

$$\sigma^2_{\text{mean}} = \sigma^2/M,$$

where M is the number of iterations we need to run.

4.3. *Parallelization*

By running the improved Monte Carlo on 10 threads in parallel, we got a significant speed-up, as seen in figure 9. The ratio of the two graphs is approximately constant at ≈ 0.15, which means that the speed-up about 6.6 times, so not quite the theoretical 10. For both calculations, we used the optimization level -O3.

The theoretical speed-up of 10 is probably never achieved, due to several factors. The computer running the program has a lot of other processes running which takes CPU time. Depending on the type of processor, the speed-up may be capped at a lower amount of threads. Consider the content of figure 10. It displays the number of iterations as a function of time, where each thread is given the same amount of work. This is displayed for 2, 4, 6, 8, and 10 threads, and we can see that there is a noticeable speed penalty for running the same operation on several threads at once. Even though running the same operation several times in parallel is slower than running the operation just once on one thread, the overall computation is much faster which we saw in figure 9. 10 threads running the computation for 10^5 iterations is the same as running a single thread for 10^6 iterations, and

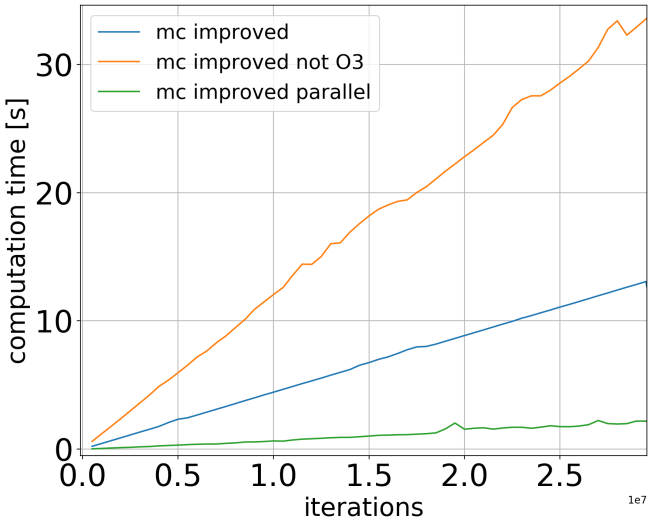

FIG. 9.— Monte Carlo improved with and without parallelization. By running the program simultaneously in 10 threads, the computation time is almost 10 times less than the non-parallelized implementation. A non-parallel run without optimization level -O3 is also displayed. The x axis is scaled with 10^7.

the parallelization makes this a lot faster. Figure 9 also displays an improved Monte Carlo integration without using optimization level -O3. Without optimization, the calculations are much slower than the optimized ones.

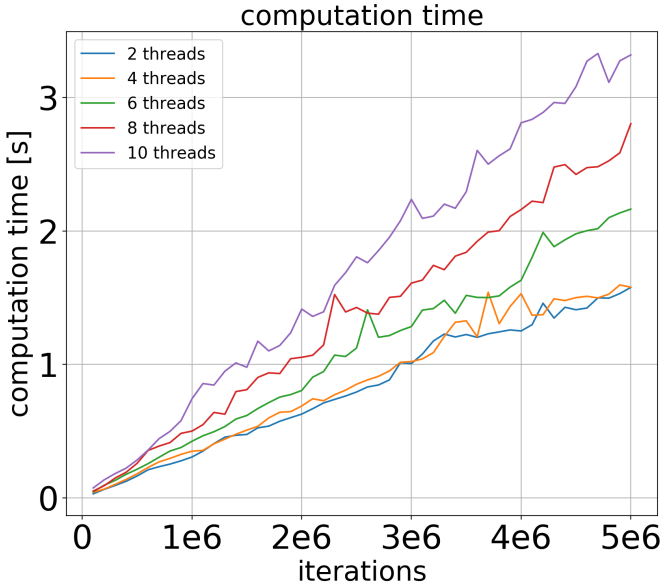

FIG. 10.— The computation time for 2, 4, 6, 8, and 10 threads where all threads do the same amount of iterations, indicated on the x axis.

More data analysis can be done to determine the perfect amount of threads, since it depends on both the saved time of running processes in parallel, but also the time penalty of running several threads at once. This is an area for a future study.

5. CONCLUSION

By doing quite some data analysis beforehand, we were able to get better results from Gauss-Legendre quadrature, but only because we know the analytical result of the integral. In situations where the answer is unknown, Gauss-Laguerre quadrature is preferred due to its lower error for the same amount of computation time. Gauss-Legendre quadrature achieved its lowest error of $\approx 10^{-5.28}$ at $\approx 8.5 \cdot 10^7$ iterations. Only at this very point did Legendre beat Laguerre, while Laguerre is the winner in all other number of grid points.

Monte Carlo integration yields lower error than both Legendre and Laguerre, and quickly converges to the correct answer for relatively few iterations. The Monte Carlo integration implementation requires more calculations per iteration, thus requiring longer computation time for a given number of iterations compared to Legendre and Laguerre, but ultimately wins due to more accurate results.

The improved Monte Carlo integration is more precise due to the usage of an exponential distribution, and yields lower errors from the very beginning. The error of the brute force Monte Carlo integration maxes out at about $\epsilon \approx 10^{-2}$ and goes as low as $\epsilon \approx 10^{-5.93}$ revealed by the contour plot analysis. The computation time of the improved Monte Carlo is somewhat larger than the brute force Monte Carlo, but the improved implementation wins the battle due to its lower errors. The improved errors rarely exceed $\epsilon \approx 10^{-4}$ and goes as low as $\epsilon \approx 10^{-6}$. Even though we did data analysis on the problem beforehand for the brute force Monte Carlo integration to determine good integration limits, it could not beat the improved Monte Carlo.

By parallelizing the Monte Carlo integration, we were able to get a speed-up by a factor of ≈ 6.6 with 10 threads running the program in parallel.

All code used to generate data in this report is supplied in a GitHub repository at https://github.com/johanafl/FYS3150-4150/tree/master/project3.

REFERENCES

David C. Lay, Steven R. Lay, J. J. M. 2016, Linear Algebra and its Applications, 5th edn. (Pearson)

Devore, J. L., & Berk, K. N. 2011, Modern Mathematical Statistics with Applications, 2nd edn., Springer Texts in Statistics (Springer)

Griffiths, D. J., & Schroeter, D. F. 2018, Introduction to Quantum Mechanics, 3rd edn. (Cambridge University Press)

Hjorth-Jensen, M. 2015, Computational Physics, Lecture Notes Fall 2015

—. 2019, ComputationalPhysics, https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Projects/2019/Project3/CodeExamples, GitHub

Lindstrøm, T. L. 2017, Spaces: An Introduction to Real Analysis, 1st edn. (American Mathematical Society)

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 2007, Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd edn. (Cambridge University Press)

Stoer, J., & Bartels, R. 1992, Introduction to Numerical Analysis, Texts in Applied Mathematics (Springer-Verlag)