A comparison of numerical integration methods for solving quantum mechanical expectation value problems

Erik Alexander Sandvik

October 22, 2019

Abstract

We study four methods of numerical integration based on Gaussian quadrature and Monte Carlo integration and how they compare by solving a quantum mechanical expectation value problem. A six dimensional integral to find the expectation value of the potential energy due to the interactions of two electrons is computed. We find that the CPU time using the Gaussian quadrature methods increases in an exponential manner with the number of mesh points, while the CPU time using the Monte Carlo methods increases linearly. A precision of four leading digits in an analytical expression is reached after 3 CPU seconds using Monte Carlo integration. Such a precision was not reached using Gaussian quadrature methods. A suggestion on how to make the Monte Carlo methods converge even faster is given.

1 Introduction

In this report we will examine four selected methods of numerical integration and test them against a quantum mechanical expectation value problem to see how they compare. The first two methods are Gaussian quadrature rules that uses orthogal polynomials, Legendre polynomials and Associated Laguerre polynomials respectively. In the last two methods we will use Monte Carlo integration that is based on probability distributions and random number generators. In the first one we'll use the uniform distribution. In the last one we'll use a composite probability distribution consisting of the uniform and the exponential distribution. We will briefly study a system of two electrons over a helium nucleus interacting with the Coloumb potential, and use the various methods of numerical integration to compute the expectation value of the potential energy due to this interaction. Such a system contains six degrees of freedom requiring a six dimensional integral. We will first go through the theory behind each integration method, then discuss how we can apply them to this specific problem. Finally we will compute the six dimensional integral using the respective methods of numerical integration and study how fast they converge to an analytical expression in terms of the amount of mesh points and CPU time required.

2 Theory

2.1 Newton-Cotes rules of numerical integration

The basic philosophy of all numerical integration methods is that one can approximate the integral of a function $f(x)$ by

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} f(x_i)w_i \tag{1}$$

where the x_i's are called the mesh points and the w_i's are called the weights. The various integration methods differ by how the mesh points and the weights are chosen. For example, the most elementary integration method, the rectangle method, uses the weights

$$w_i = \frac{b-a}{N}$$

and the mesh points

$$x_i = a + iw_i$$

A slightly less elementary integration method is the so-called trapezoidal rule which uses the weights and mesh points

$$w_{i\neq 1,\ N} = \Delta x,\ \ w_1 = w_N = \frac{\Delta x}{2}$$

$$x_i = a + i\Delta x$$

where $\Delta x = (b-a)/N$. There is also the midpoint-method and Simpson's rule. What these methods all have in common is that the mesh points are evenly spaced. Such methods are collectively called Newton-Cotes rules.

2.2 Gauss-Legendre quadrature

But there is no reason why we can't have unevenly spaced mesh points, and those integration methods that do generally turn out to be more powerful than Newton-Cotes rules.

The main ingredient of Gaussian quadrature methods are orthogonal polynomials. One example of such polynomials are the so called Legendre-polynomials $L_n(x)$ given by the Rodriguez formula [1]

$$L_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}(x^2-1)^n$$

The Legendre-polynomials are orthogonal in the sense that

$$\int_{-1}^{1} L_n(x)L_m(x)dx = \frac{2}{2n+1}\delta_{nm}$$

The first N Legendre polynomials span the space $\mathbb{P}_N[-1,\ 1]$ of all real polynomials up to degree N on the interval $[-1,\ 1]$, which is where the Legendre polynomials live.

To integrate a function $f(x)$ on $[-1,\ 1]$ the idea is to approximate $f(x)$ by a polynomial* P_{2N-1} of degree $2N-1$ where N is the number of mesh points. This will help us determine the mesh points and the weights in Eq. (1). Such a polynomial may be written as

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

where $P_{N-1}(x)$ and Q_{N-1} are some polynomials of degree $N-1$. Thus our integral can be approximated by

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} P_{2N-1}dx = \int_{-1}^{1} L_N P_{N-1}(x)dx + \int_{-1}^{1} Q_{N-1}(x)dx \tag{3}$$

Using that the first $N-1$ Laguerre polynomials span the space $\mathbb{P}_{N-1}$, we can write P_{N-1} as

$$P_{N-1}(x) = \sum_{i=0}^{N-1} c_i L_i(x)$$

Thus

$$\int_{-1}^{1} L_N P_{N-1}(x)dx = \int_{-1}^{1} L_N \sum_{i=0}^{N-1} c_i L_i dx = \sum_{i=0}^{N-1} c_i \int_{-1}^{1} L_N L_i dx = 0 \tag{4}$$

Similarly

$$Q_{N-1}(x) = \sum_{i=0}^{N-1} \alpha_i L_i(x) \tag{5}$$

Thus

$$\int_{-1}^{1} Q_{N-1}dx = \int_{-1}^{1} \overbrace{L_0}^{=1} Q_{N-1}dx = \sum_{i=0}^{N-1} \alpha_i \int_{-1}^{1} L_0 L_i dx = 2\alpha_0 \tag{6}$$

Putting the results from Eq. (4) and (6) into Eq. (3) we get

$$\int_{-1}^{1} f(x)dx \approx 2\alpha_0$$

*We don't actually have to approximate $f(x)$ by a polynomial, but pretending that we do for the moment might get the point across easier.

So Gauss-Legendre quadrature ultimately comes down to finding α_0 in Eq. (5).

But it (luckily) turns out that we don't care what P_{2N-1}, P_{N-1} and Q_{N-1} even are. Let x_k for $k = 0,\ 1,\ \ldots,\ N-1$ be the zeros of $L_N(x)$ (they just so happens to all be in the interval $\langle -1,\ 1\rangle$). Then

$$f(x_k) \approx P_{2N-1}(x_k) = Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) \tag{7}$$

If we substitute $L_i(x_k) \to L_{ik}$ and $f(x_k) \to f_k$ (and sloppily throw away the $\approx$-sign) Eq. (7) becomes

$$f_k = \sum_{i=0}^{N-1} \alpha_i L_{ik}$$

which is the index notation of the matrix equation $\hat{f} = \hat{L}\hat{\alpha}$ with $\hat{f} = (f(x_0),\ \ldots,\ f(x_{N-1}))$, $\hat{\alpha} = (\alpha_0,\ \ldots,\ \alpha_{N-1})$ and

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

Because of the orthogonal properties of the Legendre polynomials, each row is linearly independent. So $\hat{L}$ has an inverse $\hat{L}^{-1}$ such that

$$\hat{\alpha} = \hat{L}^{-1}\hat{f} \tag{8}$$

and since we only care about α_0 we arrive at

$$\alpha_0 = \sum_{i=0}^{N-1} L_{0i}^{-1} f_i$$

where $f_i = f(x_i)$ with x_i for $i = 0,\ \ldots,\ N-1$ being the zeros of $L_N(x)$. Thus our integral can be approximated by

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=0}^{N-1} 2L_{0i}^{-1} f(x_i) \tag{9}$$

which is on the form of Eq. (1) with the mesh points being the zeros of $L_N(x)$ and the weights being $2L_{0i}^{-1}$ where L_{0i}^{-1} are the elements of the matrix $\hat{L}^{-1}$.

To integrate our function $f(x)$ on an arbitrary interval $[a,\ b]$ we simply do a change of variables $x \to t$ where

$$t = \frac{b-a}{2}x + \frac{b+a}{2} \tag{10}$$

Eq. (9) for an arbitrary interval becomes

$$\int_a^b f(t)dt = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right)dx \approx \frac{b-a}{2}\sum_{i=0}^{N-1} 2L_{0i}^{-1} f\left(\frac{b-a}{2}x_i + \frac{b+a}{2}\right)$$

which corresponds to choosing the weights in Eq. (1) to be $(b-a)L_{0i}^{-1}$ and the mesh points to be $(b-a)x_i/2 + (b+a)/2$.

2.3 Gauss-Laguerre quadrature

If our function $f(x)$ can be written on the form $f(x) = x^\alpha e^{-x} g(x),\ \alpha > -1$ and our integration limits are $[0,\ \infty\rangle$, we can do something similar to the previous section by using the associated Laguerre polynomials given by the Rodgriguez formula

$$\mathcal{L}_n^{(\alpha)}(x) = \frac{x^{-\alpha}e^x}{n!}\frac{d^n}{dx^n}\left(x^{n+\alpha}e^{-x}\right) = \frac{x^{-\alpha}}{n!}\left(\frac{d}{dx} - 1\right)^n x^{n+\alpha}$$

The reason why is that they happen to live on the interval $[0,\ \infty\rangle$ and because they are orthogonal in the sense that

$$\int_0^\infty x^\alpha e^{-x}\mathcal{L}_n^{(\alpha)}(x)\mathcal{L}_m^{(\alpha)}(x)dx = \frac{(n+\alpha)!}{n!}\delta_{nm}$$

The first N associated Laguerre polynomials span the space $\mathbb{P}_N[0,\ \infty\rangle$. In this case $g(x)$ is approximated by a polynomial $P_{2N-1}(x)$ of degree $2N-1$ which we may write on a form analogous to Eq. (2)

$$f(x) \approx x^\alpha e^{-x} P_{2N-1}(x) = x^\alpha e^{-x}\mathcal{L}_N^{(\alpha)}(x)P_{N-1}(x) + x^\alpha e^{-x}Q_{N-1}(x)$$

By repeating all the steps in the previous section it can be shown that

$$\int_0^\infty x^\alpha e^{-x} g(x)dx \approx \alpha!\sum_{i=0}^{N-1}\mathcal{L}_{0i}^{(\alpha)-1} g(x_i)$$

where $\mathcal{L}_{ki}^{(\alpha)-1}$ are the elements of the inverse of the matrix

$$\hat{\mathcal{L}}^{(\alpha)} = \begin{bmatrix} \mathcal{L}_0^{(\alpha)}(x_0) & \mathcal{L}_1^{(\alpha)}(x_0) & \dots & \mathcal{L}_{N-1}^{(\alpha)}(x_0) \\ \mathcal{L}_0^{(\alpha)}(x_1) & \mathcal{L}_1^{(\alpha)}(x_1) & \dots & \mathcal{L}_{N-1}^{(\alpha)}(x_1) \\ \vdots & \vdots & \vdots & \vdots \\ \mathcal{L}_0^{(\alpha)}(x_{N-1}) & \mathcal{L}_1^{(\alpha)}(x_{N-1}) & \dots & \mathcal{L}_{N-1}^{(\alpha)}(x_{N-1}) \end{bmatrix}$$

and the x_i's are the zeros of $\mathcal{L}_N^{(\alpha)}(x)$.

2.4 Monte Carlo integration

As always we consider the integral $I = \int_a^b f(y)dy$

We may, for no apparent reason, multiply and divide the integrand by some probability distribution $P(y)$ on the interval $[a,\ b]$.

$$\int_a^b f(y)dy = \int_a^b P(y)\frac{f(y)}{P(y)}dy$$

Now change variables from y to some variable $x \in [0,\ 1]$ such that $P(y)dy = dx$

$$\int_a^b P(y)\frac{f(y)}{P(y)}dy = \int_0^1 \frac{f(y(x))}{P(y(x))}dx \tag{11}$$

Now consider the average $\langle f \rangle$ of a function $f(x)$ on an interval $[a,\ b]$.

$$\langle f \rangle = \frac{1}{b-a}\int_a^b f(x)dx$$

We can rewrite this to

$$\int_a^b f(x)dx = (b-a)\langle f \rangle$$

Thus from Eq. (11) the next step is

$$\int_0^1 \frac{f(y(x))}{P(y(x))}dx = \left\langle \frac{f(y(x))}{P(y(x))} \right\rangle \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(y(x_i))}{P(y(x_i))}$$

So Monte Carlo integration basically comes down to

$$\int_a^b f(y)dy \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(y(x_i))}{P(y(x_i))} \tag{12}$$

where $P(y)$ is some probability distribution or other. Why does it have to be a probability distribution? You don't have to think of it that way, but since $P(y)dy = dx$ where $x \in [0,\ 1]$ there are conditions on $P(y)$, namely

- It has to be integrable

- It has to satisfy $\int_a^b P(y)dy = 1$
- It has to be positive definite so that there is a one to one correspondence between x and y
- The integral $x(y) = \int_{y_0}^{y} P(y)dy$ has to be invertible, allowing us to express one variable in terms of the other.

So it sounds a lot like a probability distribution (except for the thing about necessarily being positive definite, that's just very convenient when we shift from one variable to another).

And to address the elephant in the room: The choice of $P(y)$ does affect the final result. Since the right side of Eq. (12) corresponds to the neanderthal rectangle method it pays to pick a $P(y)$ that resembles $f(y)$ in their derivatives such that the variance of the points $f(y(x_i))/P(y(x_i))$ is as small as possible. And also because we're literally going to pick the points $x_i \in [0,\ 1]$ at random using the uniform distribution and a random number generator. So if we make a very inappropriate choice of $P(y)$ we might get approximations to the integral that deviates wildly from each other each time we run the calculations.

In other words, since the right side of Eq. (12) is essentially the sample mean of the quantity $f(y(x))/P(y(x))$ on the interval $y \in [a,\ b]$, we should ideally pick some $P(y)$ such that the variance

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{f(y(x_i))}{P(y(x_i))}\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}\frac{f(y(x_i))}{P(y(x_i))}\right)^2$$

is as small as possible. As we've seen we could've just approximated the integral by taking the average of $f(y)$ on $[a,\ b]$ and multiply by the length of the interval. But if $f(y)$ is some complicated function it might have a very large standard deviation on this interval, meaning that we need a large number N of points to get to a satisfactory precision. $f(y(x))/P(y(x))$ should ideally be constant on $x \in [0,\ 1]$ and then we would only need a single point to get an exact answer!

2.5 Two electrons over a helium nucleus

We will consider two electrons over a helium nucleus both being in the $1s$-state and assume that their respective wavefunctions can be modelled like the wavefunction of an electron over the hydrogen nucleus. For $i = 1,\ 2$ the wavefunction of electron i is

$$\psi_i\left(\vec{r}_i\right) \propto e^{-\alpha r_i}$$

where α is a parameter where $\alpha = 2$ corresponds to the helium atom. So the total wavefunction of the two electrons is

$$\Psi\left(\vec{r}_1,\ \vec{r}_2\right) \propto e^{-\alpha(r_1+r_2)}$$

Our mission is to calculate the expectation value $\langle V\rangle$ of the potential energy V due to the Coloumb interaction between the two electrons. It is given by

$$\langle V\rangle = \langle\Psi|V|\Psi\rangle$$

By inserting the identity operator $\int_{-\infty}^{\infty} d\vec{r}_1 d\vec{r}_2 |\vec{r}_1,\ \vec{r}_2\rangle\langle\vec{r}_1,\ \vec{r}_2|$ between $\langle\Psi|$ and $V|\Psi\rangle$ and using that $\langle\Psi|\vec{r}_1,\ \vec{r}_2\rangle = \Psi^*(\vec{r}_1,\ \vec{r}_2)$ and $\langle\vec{r}_1,\ \vec{r}_2|V|\Psi\rangle \propto \Psi(\vec{r}_1,\ \vec{r}_2)/r_{12}$ [2] where $r_{12} \equiv 1/|\vec{r}_1 - \vec{r}_2|$ we arrive at

$$\langle V\rangle \propto I = \int_{-\infty}^{\infty} d\vec{r}_1 d\vec{r}_2 \frac{e^{-2\alpha(r_1+r_2)}}{r_{12}} \tag{13}$$

which we can write in a more compact manner as

$$\langle V\rangle \propto I = \int_{-\infty}^{\infty} d\vec{r}_1 d\vec{r}_2 f(\vec{r}_1,\ \vec{r}_2) \tag{14}$$

with $f(\vec{r}_1,\ \vec{r}_2) = e^{-2\alpha(r_1+r_2)}/r_{12}$.

This six dimensional integral can be calculated analytically and turns out to be $5\pi^2/16^2 \approx 0.192765711$. These are the digits we would like to see over and over again in the following sections.

3 Methods

3.1 Gauss-Legendre quadrature with cartesian coordinates

The first method we will use to tackle the integral in Eq. (13) is Gauss-Legendre quadrature with cartesian coordinates $x_i,\ y_i,\ z_i$ for $i = 1, 2$. With these coordinates our integral becomes

$$I = \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dy_1 \int_{-\infty}^{\infty} dz_1 \int_{-\infty}^{\infty} dx_2 \int_{-\infty}^{\infty} dy_2 \int_{-\infty}^{\infty} dz_2 \frac{e^{-2\alpha(r_1+r_2)}}{r_{12}}$$

where $r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$ and $r_{12} = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2}$.

There are three main challenges here, the first one being the integration limits. The Legendre polynomials live on the interval $x \in [0,\ 1]$ and we can use them to integrate a function on any arbitrary interval $[a,\ b]$ through a change of variables $x \to t$ in Eq. (10), but that obviously doesn't work for the interval $\langle -\infty, \infty\rangle$. But we don't expect the potential energy due to the Coloumb interaction between the electrons to be infinite, which corresponds to them being infinitely close together all the while repelling each other with the second largest force in nature. Thus the integrand in Eq. (13) approaches zero when $r_1,\ r_2 \to \infty$. Somewhere away from the origin the integrand is essentially zero and we can use this to replace the integration limits with something that can be handled by Eq. (10).

The sloppy way to accomplish this is to consider the single particle wavefunction $\psi e^{-\alpha r}$ and find the lowest $r = \lambda$ such that $\psi(\lambda) \approx 0$.

Figure 1 shows the single particle wavefunction ψ as a function of the distance r away from the origin. We see that $\psi(r)$ is essentially zero when $r = 3.0$ so we can choose $\pm\lambda = \pm 3.0$ as our integration limits.

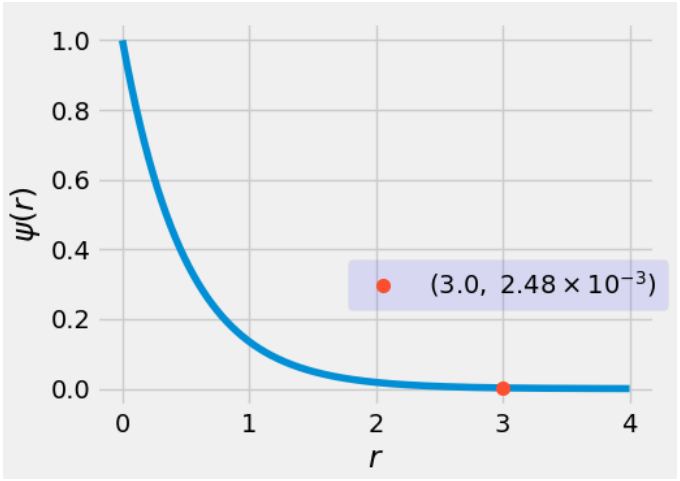

Figure 1: Plot of the single particle wavefunction $\psi(r) \propto e^{-\alpha r}$. We see that the wavefunction is essentially zero beyond $r = 3$.

The second challenge is that we have a six-dimensional integral instead of a one-dimensional integral. Thus there is a set of weights and mesh points for each variable that is to be integrated over. But by choosing the same number N of mesh points and the same integration interval $\pm\lambda$ for each variable all variables have the exact same weights and mesh points. Let the weights and mesh points be denoted w_i and x_i respectively. Our integral is to be approximated by the sixtuple sum

$$I \approx \sum_{i=1}^{N}\sum_{j=1}^{N}\dots\sum_{n=1}^{N} w_i w_j \dots w_n f(x_i,\ x_j,\ \dots,\ x_n) \tag{15}$$

But this leads us to the third challenge. If each variable has the exact same set of mesh points, r_{12} is going to be zero more than a handful of times and then we divide by zero. One way to deal with this is to set $f = 0$ when $r_{12} = 0$. Actually we should set $f = 0$ when $r_{12} < \varepsilon$ for some small number ε. The reason being that $f\left(\vec{r}_1,\ \vec{r}_2\right)$ blows up in this region and that f is essentially being approximated by a (six dimensional) polynomial of degree $2N - 1$. To find a polynomial that mimics such behaviour you have to choose N to be very large, larger than any reasonably sized N for running the calculations on a laptop. I've decided to use $\varepsilon = 10^{-4}$.

3.2 Gauss-Laguerre quadrature with spherical coordinates

By switching to spherical coordinates $(r,\ \phi,\ \theta)$ we can write our integral as

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

where

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

and

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

We can use Gauss-Laguerre quadrature on this integral by making the change of variables $r_i \to R_i = -2\alpha r_i$. This lets us write our integral as

$$I = \frac{1}{(2\alpha)} \int_0^\infty dR_1 \int_0^\infty dR_2 \int_0^\pi d\theta_1 \int_0^\pi d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 R_1^2 R_2^2 \sin\theta_1 \sin\theta_2 \frac{e^{-(R_1+R_2)}}{r_{12}}$$

where

$$r_{12} = \frac{1}{2\alpha}\sqrt{R_1^2 + R_2^2 - 2R_1R_2\cos\beta}$$

This is so our integral is analogous to an integral on the form $\int_0^\infty x^\alpha e^{-x} g(x) dx$. The part of our integrand that corresponds to $g(x)$ is $\sin\theta_1 \sin\theta_2 / r_{12}$. All the other parts will be absorbed into the weights when we use Gauss-Laguerre quadrature!

But we're ony going to be using Gauss-Laguerre quadrature when we integrate over the R_i's. When we integrate over the ϕ_i's and the θ_i's we're going to be using Gauss-Legendre quadrature, the reason being that the limits are not 0 and ∞.The weights $w_r^{(i)}$ for the R_i's will be the same and so will the weights $w_\phi^{(i)}$ for the ϕ_i's and the weights $w_\theta^{(i)}$'s. The approximation to the integral is

$$I \approx \sum_{i=1}^N \sum_{j=1}^N \dots \sum_{n=1}^N w_r^{(i)} w_r^{(j)} \dots w_\theta^{(n)} g\left(R_1^{(i)},\ R_2^{(j)},\ \dots,\ \theta_2^{(n)}\right)$$

3.3 Monte Carlo integration with the uniform distribution

The uniform distribution $P(y)$ on an interval $[a,\ b]$ satisfies

$$\int_a^b P(y)dy = 1$$

so it is given by $P(y) = 1/(b-a)$ when $y \in [a,\ b]$. Since $dx = P(y)dy$ where $x \in [0,\ 1]$, we can find a relation between x and y by solving the differential equation

$$\int_0^x dx = \int_a^y \frac{dy}{b-a}$$

$$y = a + (b-a)x \tag{16}$$

We're going to be doing something pretty stupid, namely solving the integral in Eq. (14) using cartesian coordinates and the uniform distribution. This is stupid because the integrand only resembles the uniform distribution far away from the origin where it is essentially zero anyway!! And also because with cartesian coordinates the limits are $\pm\infty$, so we have no choice but to approximate infinity with some λ which unavoidably contributes to the error.

To be more precise the six dimensional uniform distribution we're going to be using is given by

$$P\left(\vec{r}_1,\ \vec{r}_2\right) = \begin{cases} \frac{1}{(2\lambda)^6} & x_1,\ y_1,\ z_1,\ x_2,\ y_2,\ z_2 \in [-\lambda,\ \lambda] \\ 0 & \text{else} \end{cases}$$

Our approximation to the integral in Eq. (14) is then going to be

$$I \approx \frac{(2\lambda)^6}{N} \sum_{i=1}^{N} f\left(x_1^{(i)},\ \ldots,\ z_2^{(i)}\right)$$

where $x_1^{(i)},\ \ldots,\ z_2^{(i)}$ are found by using Eq. (16) with random variables on the interval $[0, 1]$. Why is there no sixtuple sum this time? Because the points $x_1^{(i)},\ \ldots,\ z_2^{(i)}$ are supposed to be chosen at random. If we for example had some points $(x_i,\ y_j)$ and x_i is held constant for some set of y-values, these points would be far from random!!

The approximation to the variance σ^2 of $f\left(x_1^{(i)},\ \ldots,\ z_2^{(i)}\right)/P\left(x_1^{(i)},\ \ldots,\ z_2^{(i)}\right)$ is given by

$$\sigma^2 \approx \frac{(2\lambda)^{12}}{N} \sum_{i=1}^{N} f\left(x_1^{(i)},\ \ldots,\ z_2^{(i)}\right)^2 - \frac{(2\lambda)^{12}}{N^2} \left[\sum_{i=1}^{N} f\left(x_1^{(i)},\ \ldots,\ z_2^{(i)}\right)\right]^2 \tag{17}$$

3.4 Monte Carlo integration with the exponential distribution

We saw in section 3.2 that by switching to spherical coordinates that parts of the integrand will be on the form of e^{-x}. In that case we can try using the exponential distribution in Eq. (12). The exponential distribution is given by

$$P(y) = e^{-y}$$

Solving the differential equation $P(y)dy = dx$ with the limits of the y-integral being 0 and y we can express the variable y in terms of the random variable $x \in [0,\ 1]$ as

$$y = -\ln(1 - x) \tag{18}$$

The six dimensional distribution that we'll be using is given by

$$P\left(\vec{r}_1,\ \vec{r}_2\right) = \begin{cases} \frac{e^{-(r_1+r_2)}}{4\pi^4} & \phi_1,\ \phi_2 \in [0,\ 2\pi]\ \wedge\ \theta_1,\ \theta_2 \in [0,\ \pi] \\ 0 & \text{else} \end{cases}$$

It is a product of exponential distributions for the r_i's and uniform distributions for the ϕ_i's and the θ_i's. Random values for the r_i's are found using Eq. (18) and for the ϕ_i's and θ_i's by using Eq. (16). So the approximation to our integral is

$$I \approx \frac{4\pi^4}{N} \sum_{i=1}^{N} r_1^{(i)2} r_2^{(i)2} \sin\theta_1^{(i)} \sin\theta_2^{(i)} e^{\left(r_1^{(i)} + r_2^{(i)}\right)} f\left(r_1^{(i)},\ \ldots,\ \theta_2^{(i)}\right)$$

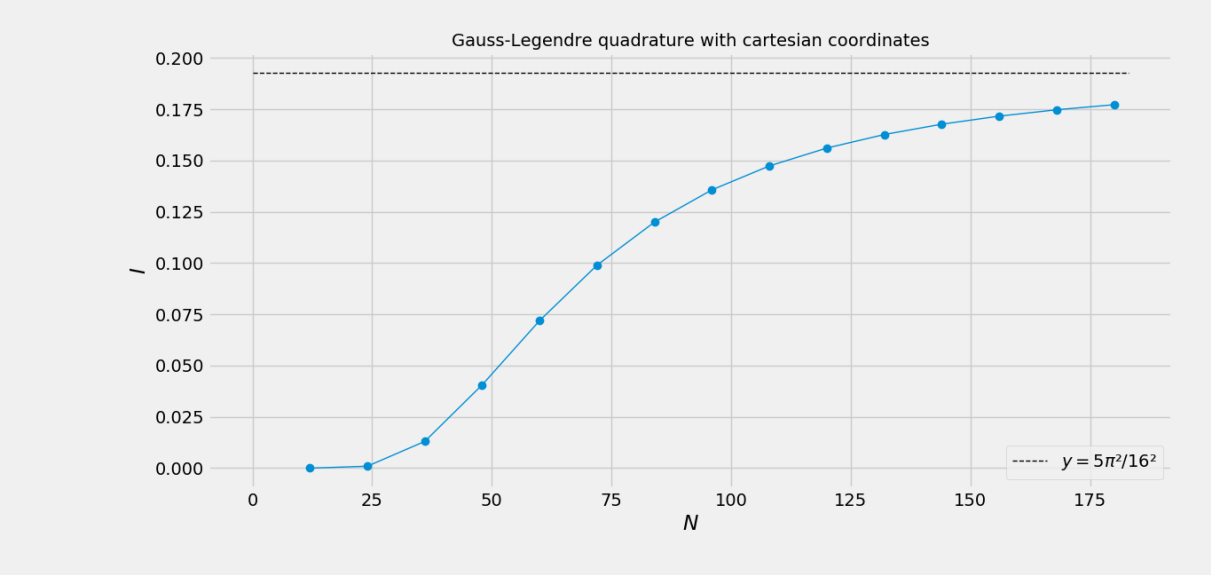

Figure 2: Numerical approximations to the integral in Eq. (14) as a function of the number of mesh points N using Gauss-Legendre quadrature and cartesian coordinates. The black dashed horizontal line is a plot of $y = 5\pi^2/16^2$, the analytical value of the integral.

Since it is the variance of the integrand that ultimately matters with Monte Carlo integration the approximation to the variance is

$$\sigma^2 = \frac{16\pi^8}{N}\sum_{i=1}^{N}\left(r_1^{(i)2}r_2^{(i)2}\sin\theta_1^{(i)}\sin\theta_2^{(i)}e^{\left(r_1^{(i)}+r_2^{(i)}\right)}f\left(r_1^{(i)},\ \ldots,\ \theta_2^{(i)}\right)\right)^2 - \frac{16\pi^8}{N^2}\left(\sum_{i=1}^{N}r_1^{(i)2}r_2^{(i)2}\sin\theta_1^{(i)}\sin\theta_2^{(i)}e^{\left(r_1^{(i)}+r_2^{(i)}\right)}f\left(r_1^{(i)},\ \ldots,\ \theta_2^{(i)}\right)\right)^2 \tag{19}$$

4 Results

Gauss-Legendre quadrature

Figure 2 shows a plot of the numerical approximation to the integral in Eq. (14) using Gauss-Legendre quadrature and cartesian coordinates. We see that the numerical approximation converges to the line $y = 5\pi/16$ as the number of mesh points N increases. In figure 3 we see that the slope of the absolute error graph evens out as N increases similarly to a function that goes like $\sim \ln N^{-1}$. This means that the difference in the amount of mesh points that each leading digit

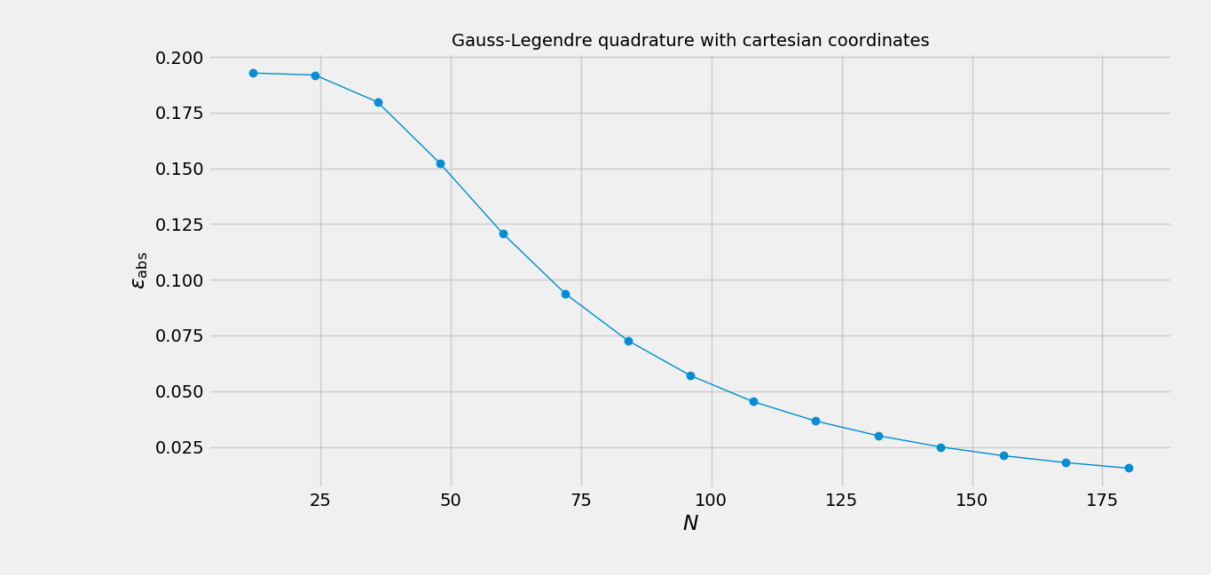

Figure 3: The absolute error ε_{abs} in the numerical approximations in figure 2 as a function of the number of mesh points N.

requires is going to increase, making it more and more expensive to produce the next leading digit. We see from figure 3 that a precision of two leading digits was reached around $N = 70$ mesh points, but a third leading digit was not produced even though we used up to $N = 180$ mesh points.

Meanwhile we see from figure 4 that the CPU time used to produce the results increases even more and more. Polynomial regression using the method of least squares was attempted on this data set for a polynomial of degree 2 up to 9. The statistical error in the coefficient of the leading term was found from the covariance matrix and decreases with the degree of the polynomial.
A linear regression model was also attempted by taking the natural logarithm of the CPU time. The coefficient of the leading term was found to have a higher statistical error than the coefficient in the leading term of the second degree polynomial model. This suggests that the CPU time does not increase exponentially with N. A proper model for the behaviour of the curve in figure 4 is yet to be found.

Gauss-Laguerre quadrature

Figure 5 shows a behaviour similarly to figure 2, although the curve seems to be converging faster to the analytical expression. This is supported by figure 6 which shows that a precision of two leading digits was reached at $N = 50$ mesh points as opposed to $N = 70$ mesh points for Gauss-Legendre quadrature. We see that a third leading digit was close to be produced at

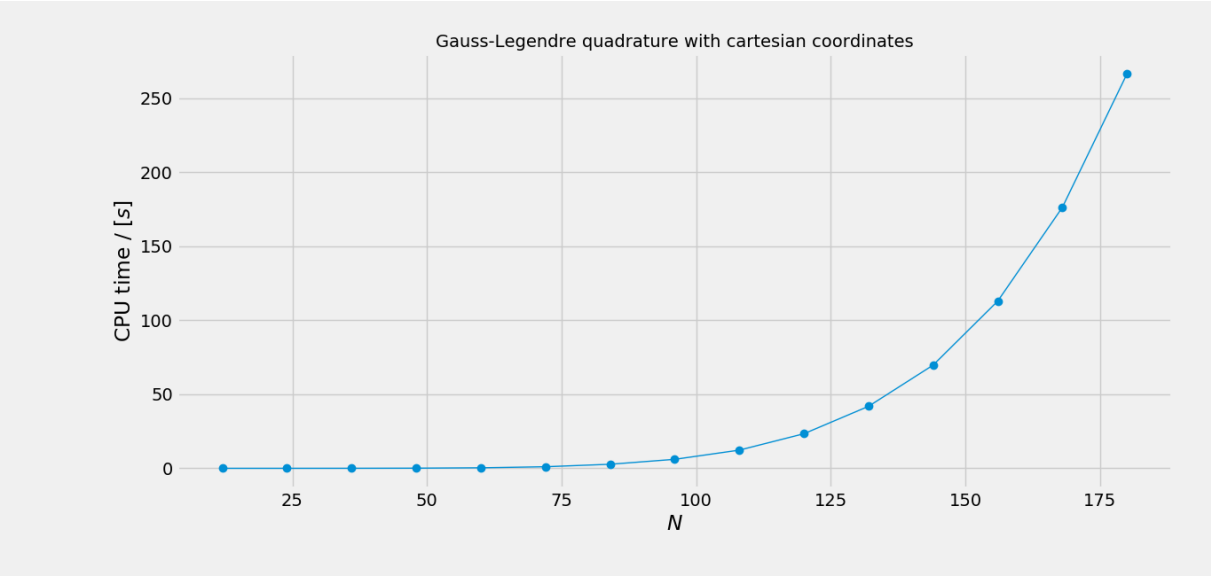

Figure 4: The CPU time used to calculate the numerical approximations in figure 2 as a function of the number of mesh points N.

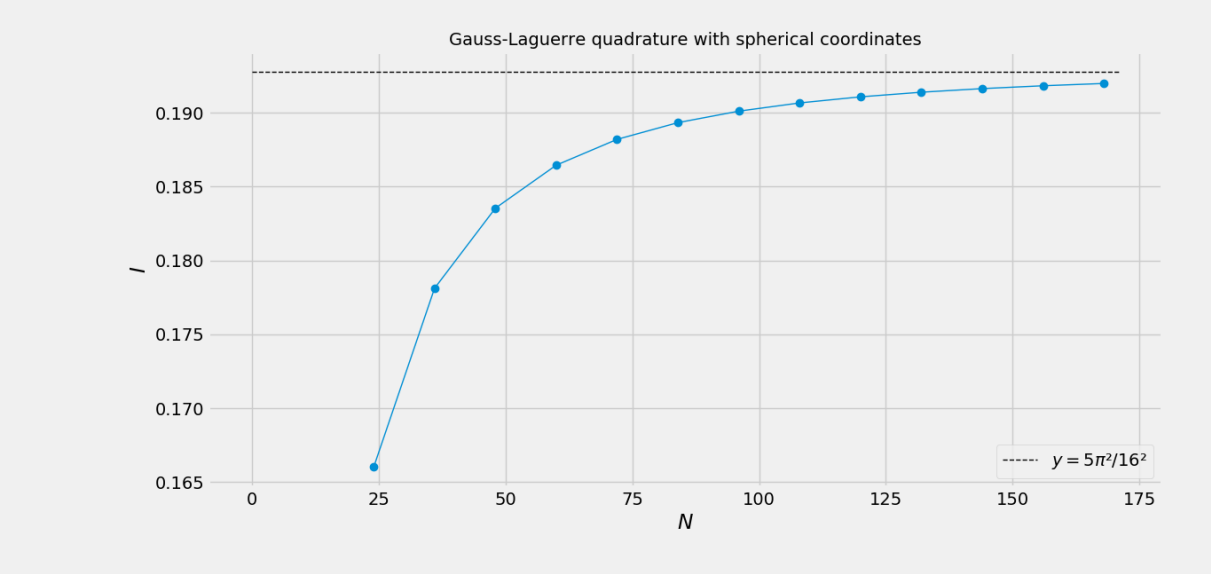

Figure 5: Numerical approximations to the integral in Eq. (14) as a function of the number of mesh points N using Gauss-Laguerre quadrature and spherical coordinates. The black dashed horizontal line is a plot of $y = 5\pi^2/16^2$, the analytical value of the integral.

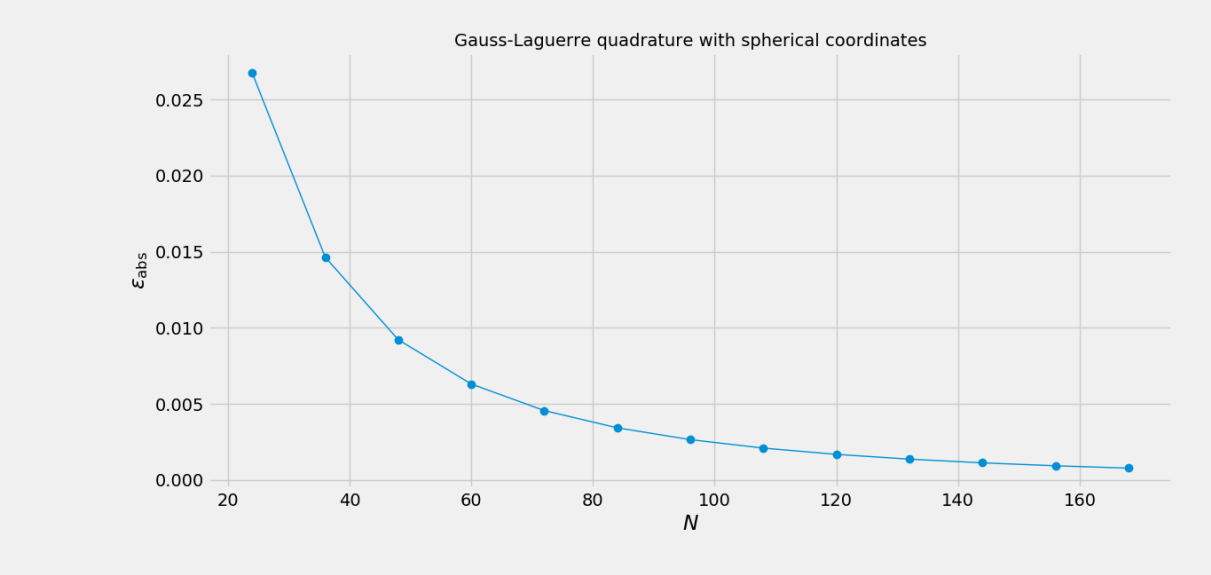

Figure 6: The absolute error ε_{abs} in the numerical approximations in figure 5 as a function of the number of mesh points N.

$N = 170$ mesh points. In figure 7 we see that although the behaviour of the curve is similar to the curve in figure 4, the CPU time does not increase as quickly as for Gauss-Legendre quadrature. To produce even more leading digits in the approximation to the integral in Eq. (14) these figures suggests that Gauss-Laguerre quadrature is the preferable integration method of the two Gaussian quadrature methods discussed in this report.

Monte Carlo integration with the uniform distribution

Figure 8 shows that the Monte Carlo integration method using the uniform distribution converges to the analytical expression in a more disorderly fashion. By only examining the left half of figure 8 and noting the scaling of the x-axis one would've thought that this integration method performs extremely poorly compared to the Gauss-Legendre and Gauss-Laguerre quadrature. But the strength of this method is expressed in figure 10 of the CPU time used to do the calculation. The CPU time increases linearly with N. Even 25 million grid points only takes about half a minute of CPU time. In the right half of figure 8 we see that this integration method consistenly reaches a precision level of two leading digits after a calculation of 20 seconds of CPU time. To produce even more leading digits this method of numerical integration is by far preferable to Gauss-Legendre and Gauss-Laguerre quadrature. In the days of Gauss and pen and paper the opposite would've been true, but in the age of computers Monte Carlo beats Gauss out of the park.

However figure 11 shows that we can do even better. It shows the standard deviation in Eq. (17) versus the number of mesh points N. The goal of Monte Carlo integration is to reduce the standard deviation as much as possible and there is clearly room for improvement here. We can easily find another probability distribution that resembles the integrand more than the uniform distribution such that the approximations converge even more quickly and with more consistency.

Monte Carlo integration with the exponential distribution

Figure 13 shows that this is indeed what happens. While we never reached a precision level of three leading digits in the previous integration methods (we did with Monte Carlo and the uniform

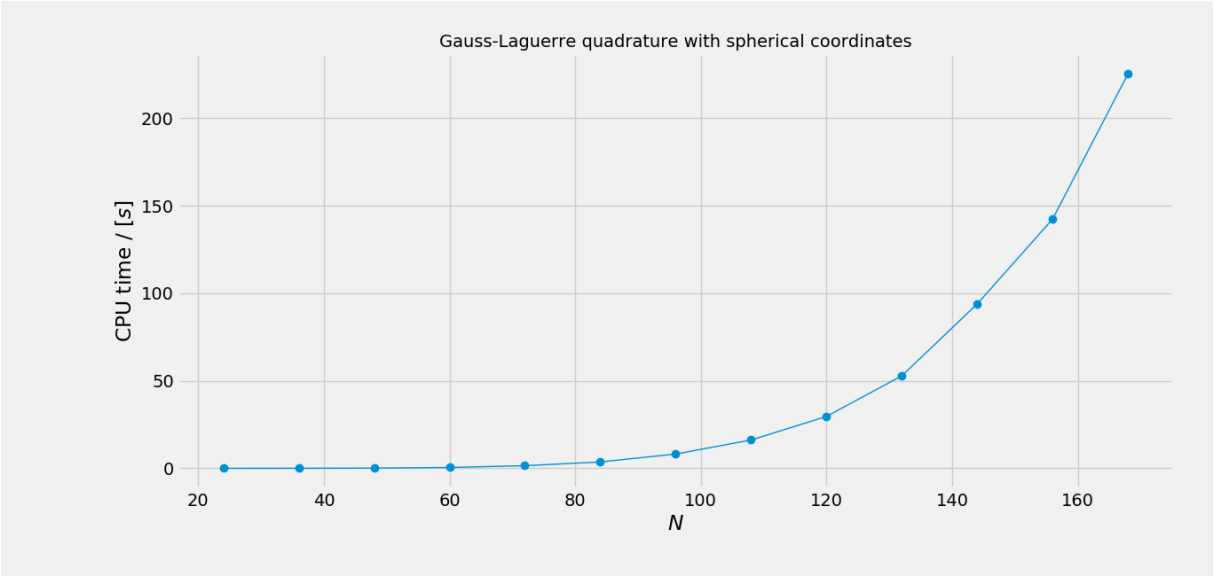

Figure 7: The CPU time used to calculate the numerical approximations in figure 5 as a function of the number of mesh points N.

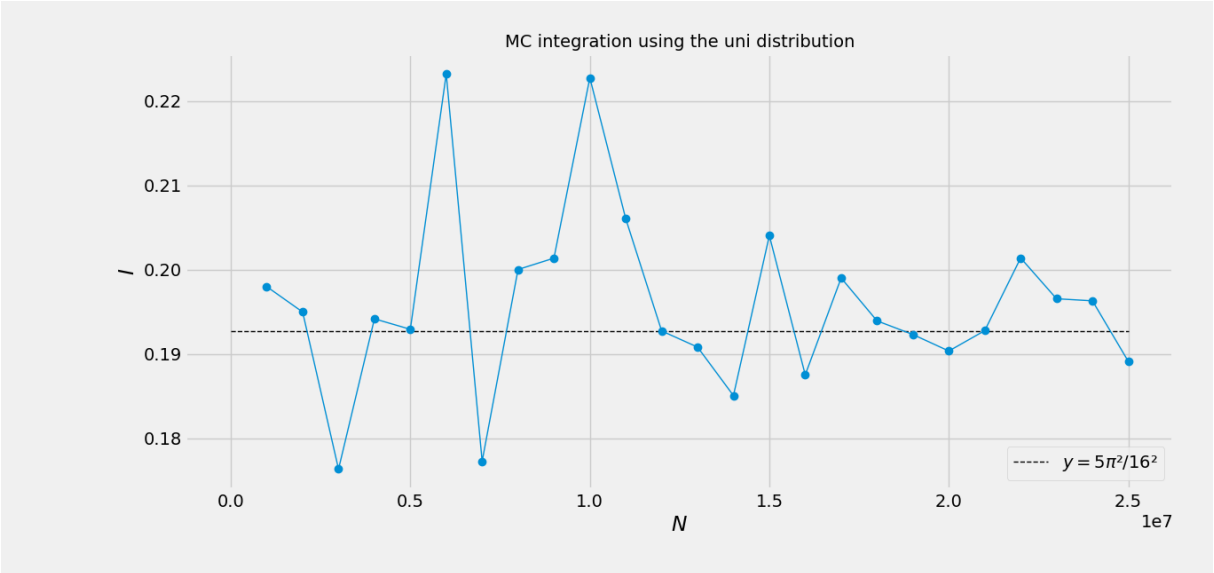

Figure 8: Numerical approximations to the integral in Eq. (14) function of the number of mesh points N using Monte Carlo integration and the uniform distribution. The black dashed horizontal line is a plot of $y = 5\pi^2/16^2$, the analytical value of the integral.

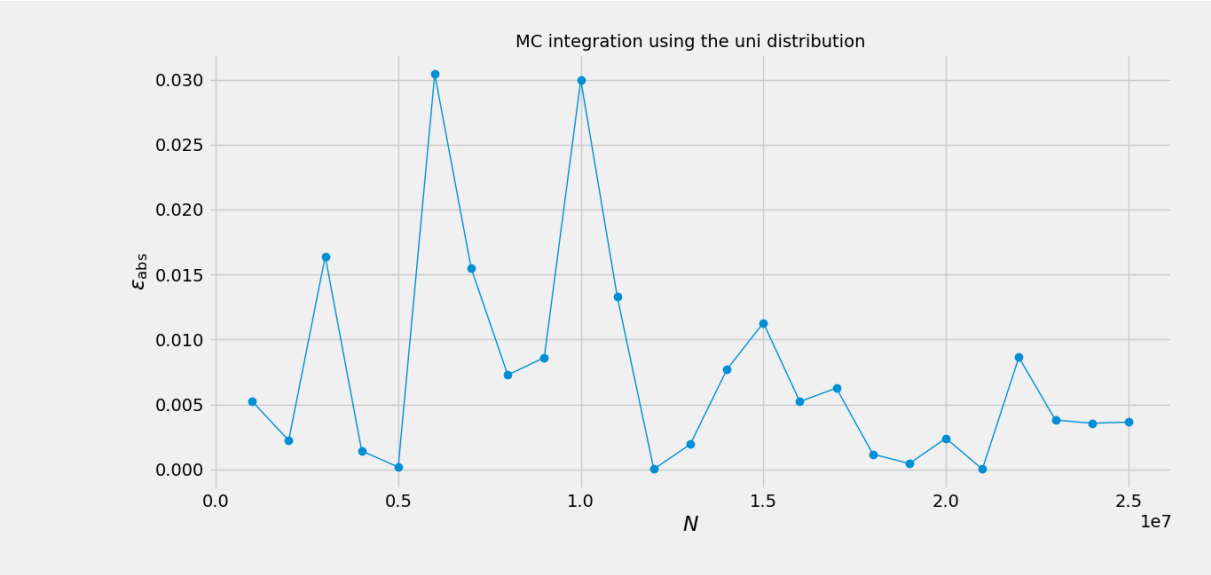

Figure 9: The absolute error ε_{abs} in the numerical approximations in figure 8 as a function of the number of mesh points N.

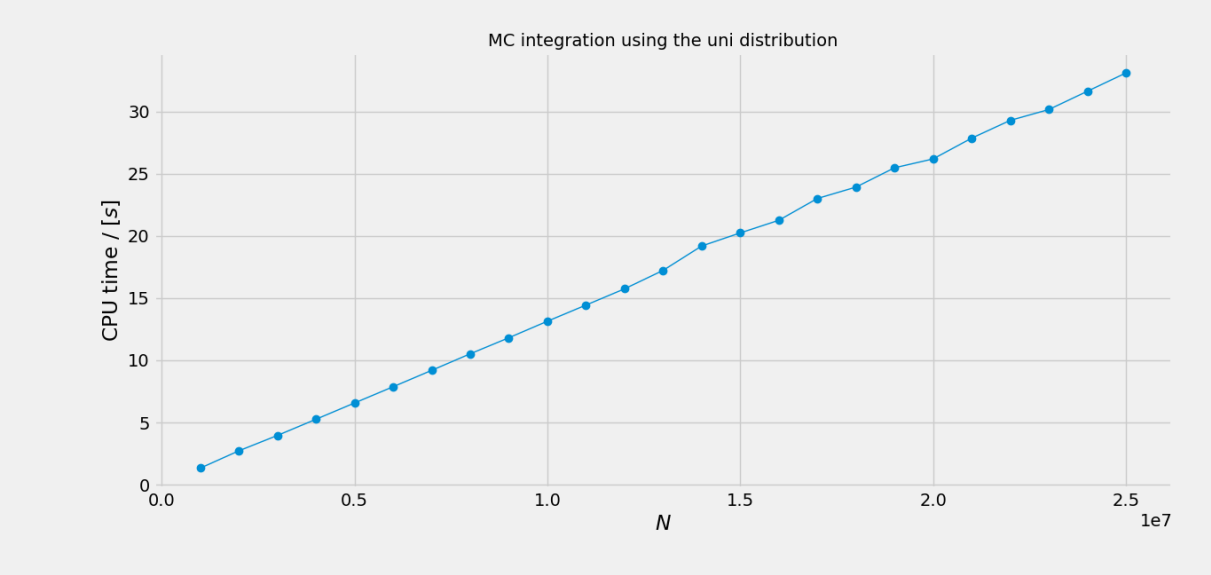

Figure 10: The CPU time used to calculate the numerical approximations in figure 9 as a function of the number of mesh points N.

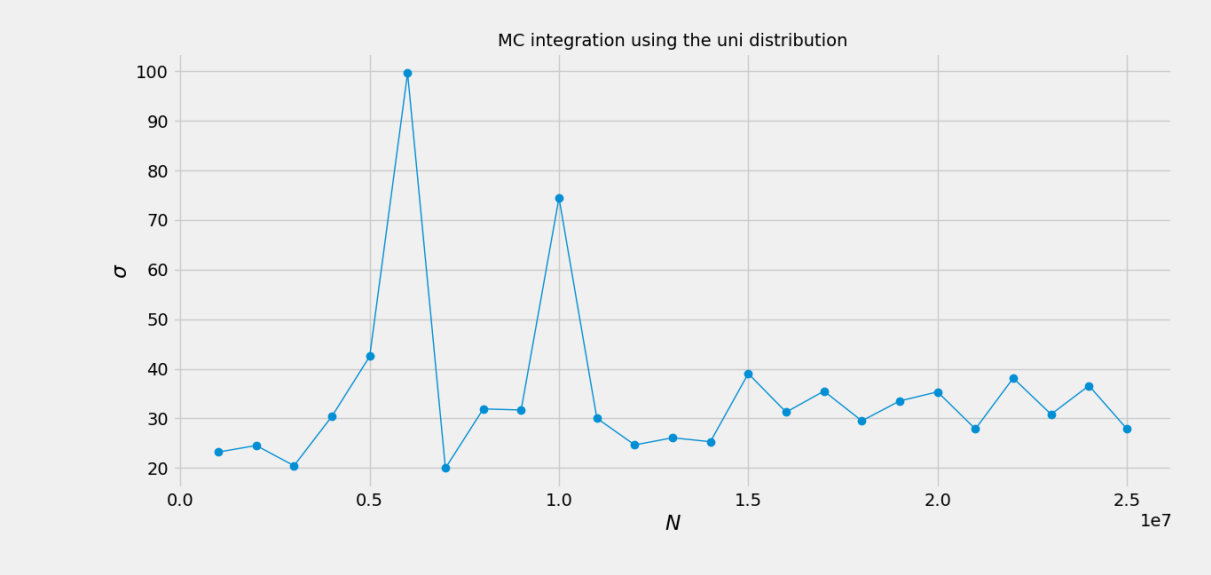

Figure 11: The standard deviation in Eq. (17) as a function of the number of mesh points N.

distribution, but by pure chance. Consistency matters with Monte Carlo methods.) we see that we consistently reach a precision level of four leading digits from the very first data point! And how much CPU time did that cost? Only a few seconds! Even though the slope of the curve in figure 14 is slightly steeper than in figure 10, this integration method totally makes up for it by how fast it converges to the analytical expression. Comparing figures 15 and 11 we see that the standard deviation is significantly reduced meaning the results of this integration method will far and away be more consistent. Out of all the integration methods examined in this report there is no doubt that this should be the method of choice for computing the integral in Eq. (14) numerically.

5 Conclusion

We've examined four methods of numerical integration and seen how they perform when solving a quantum mechanical expectation value problem. These being Gauss-Legendre quadrature, Gauss-Laguerre quadrature, Monte Carlo integration using the uniform distribution and Monte Carlo integration using the exponential distribution. While the Gaussian quadrature methods converges to the analytical expression in an orderly fashion as opposed to the Monte Carlo methods, we've seen that by tolerating small inconsistencies in the computed approximations one can get to a higher precision of leading digits with Monte Carlo a lot faster than with Gaussian quadrature. We also saw that we could minimize these inconsistencies quite a bit, by choosing a probability distribution that resembles the integrand. As a prospect for future work, one could try to design a composite function of probability distributions that resembles the integrand even more, as the exponential distribution only resembles parts of the integrand. In theory the perfect probability distribution requires only a single point to evaluate the integral exactly, so it would seem like only the sky is the limit to how precise and how fast we can evaluate multidimensional integrals numerically.

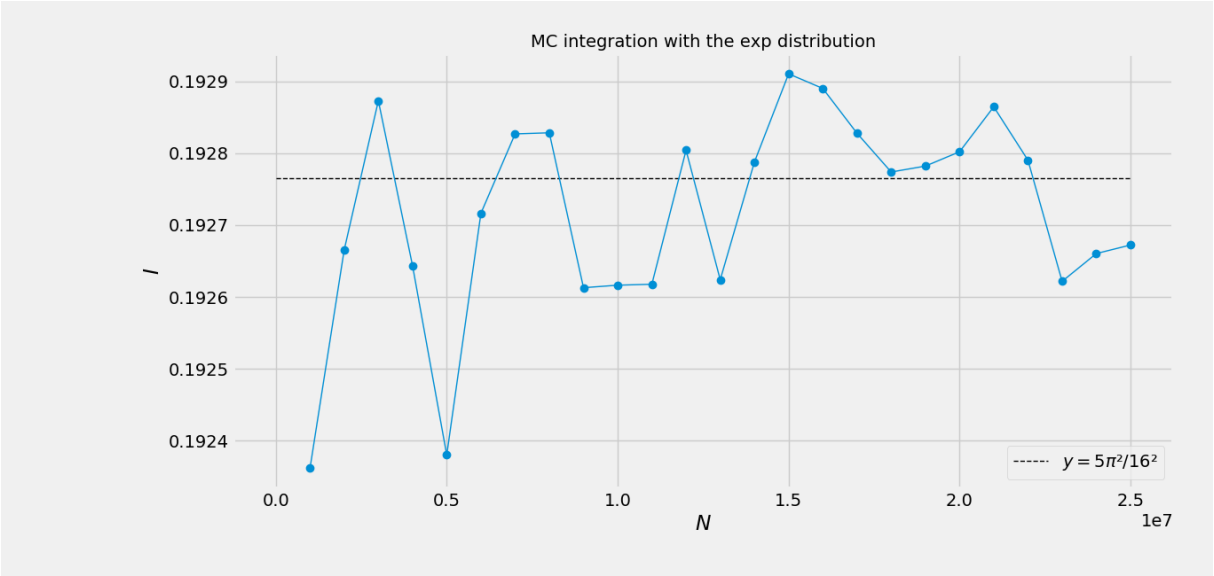

Figure 12: Numerical approximations to the integral in Eq. (14) function of the number of mesh points N using Monte Carlo integration and the exponential distribution. The black dashed horizontal line is a plot of $y = 5\pi^2/16^2$, the analytical value of the integral.

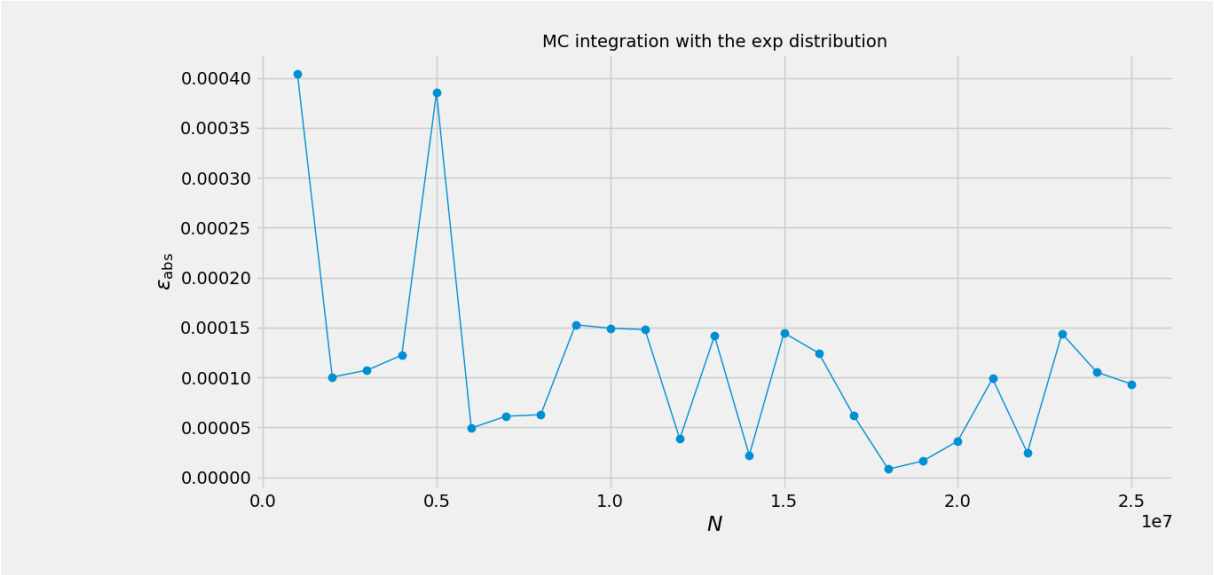

Figure 13: The absolute error ε_{abs} in the numerical approximations in figure 12 as a function of the number of mesh points N.

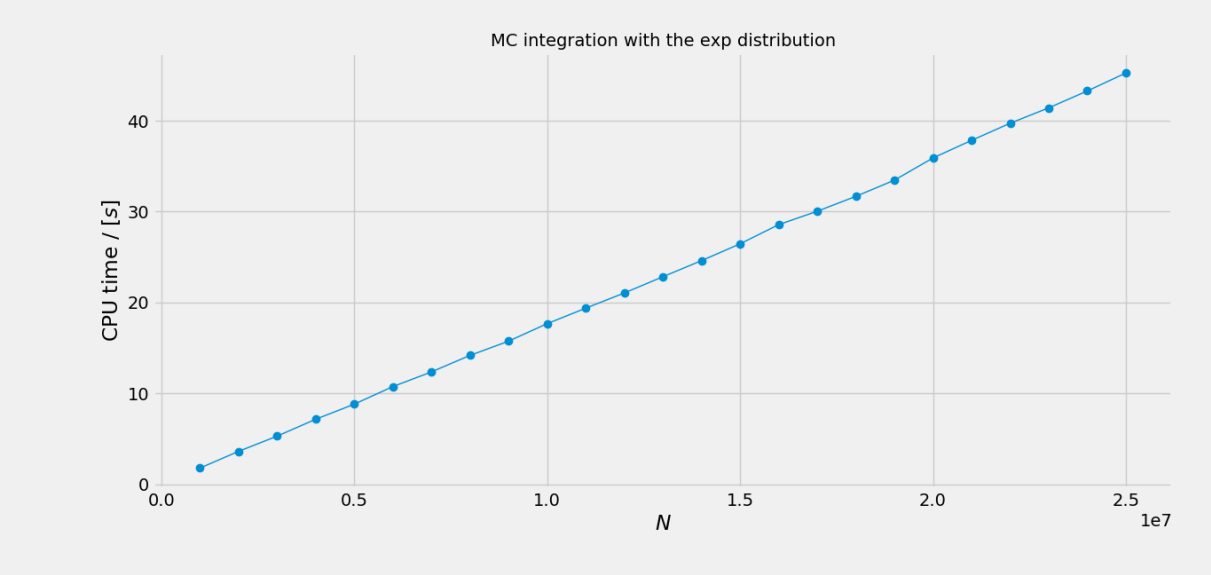

Figure 14: The CPU time used to calculate the numerical approximations in figure 12 as a function of the number of mesh points N.

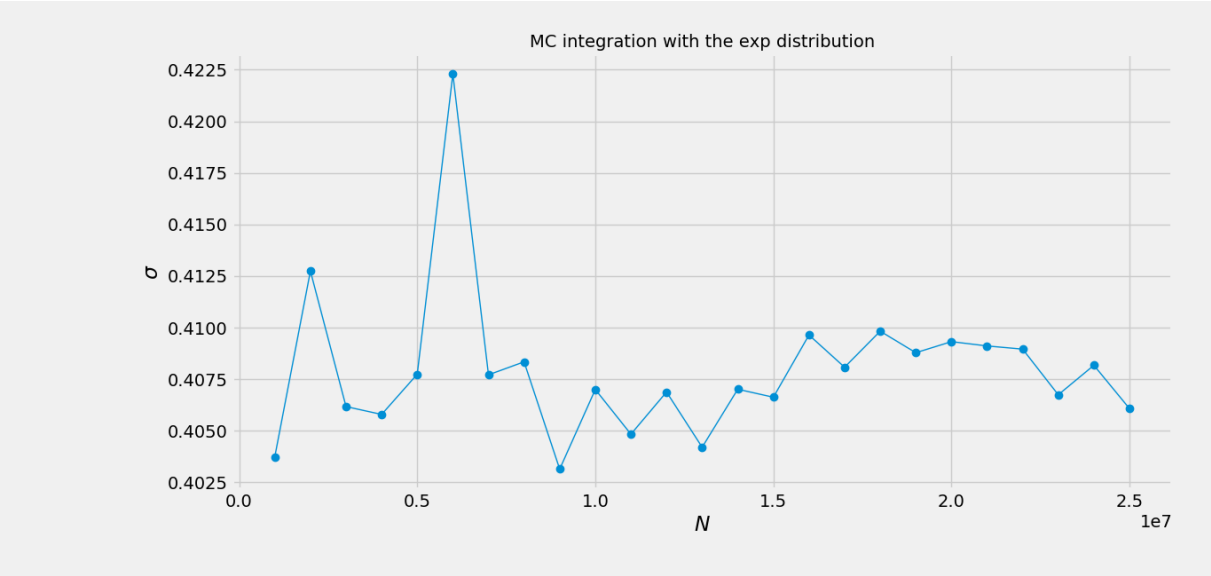

Figure 15: The standard deviation in Eq. (19) as a function of the number of mesh points N.

Appendices

A Implementation of the integration methods in C++

A.1 gauquad.cpp

Thanks to Morten Hjorth-Jensen for letting me copy the functions `gauleg()`, `gammln()` and `gaulag()` from his github for use in this project. According to him the algorithms are taken from Numerical Recipes, The Art of Scientific Computing, 3rd ed.

```
#include <cmath>
#include <iostream>
#include <fstream>
```

```
#include <iomanip>
#include <cmath>
#include <stdlib.h>
#include <stdio.h>
#define EPS 3.0e-14
#define MAXIT 10
#define   ZERO        1.0E-10
using namespace std;


/*
** The function
** gauleg()
** takes the lower and upper limits of integration x1, x2, calculates
** and return the abcissas in x[0,...,n - 1] and the weights in w[0,...,n - 1]
** of length n of the Gauss--Legendre n--point quadrature formulae.
*/

void gauleg(double x1, double x2, double x[], double w[], int n)
{
   int          m,j,i;
   double       z1,z,xm,xl,pp,p3,p2,p1;
   double       const  pi = 3.14159265359;
   double       *x_low, *x_high, *w_low, *w_high;

   m  = (n + 1)/2;                  // roots are symmetric in the interval
   xm = 0.5 * (x2 + x1);
   xl = 0.5 * (x2 - x1);

   x_low  = x;                      // pointer initialization
   x_high = x + n - 1;
   w_low  = w;
   w_high = w + n - 1;

   for(i = 1; i <= m; i++) {          //loops over desired roots
      z = cos(pi * (i - 0.25)/(n + 0.5));

          /*
                ** Starting with the above approximation to the ith root
          ** we enter the mani loop of refinement bt Newtons method.
          */

      do {
         p1 =1.0;
         p2 =0.0;

              /*
                ** loop up recurrence relation to get the
          ** Legendre polynomial evaluated at x
          */
```

```
            for(j = 1; j <= n; j++) {
                p3 = p2;
                p2 = p1;
                p1 = ((2.0 * j - 1.0) * z * p2 - (j - 1.0) * p3)/j;
            }

                    /*
                    ** p1 is now the desired Legrendre polynomial. Next compute
              ** ppp its derivative by standard relation involving also p2,
              ** polynomial of one lower order.
              */

            pp = n * (z * p1 - p2)/(z * z - 1.0);
            z1 = z;
            z  = z1 - p1/pp;                    // Newton's method
        } while(fabs(z - z1) > ZERO);

             /*
              ** Scale the root to the desired interval and put in its symmetric
             ** counterpart. Compute the weight and its symmetric counterpart
             */

        *(x_low++)  = xm - xl * z;
        *(x_high--) = xm + xl * z;
        *w_low      = 2.0 * xl/((1.0 - z * z) * pp * pp);
        *(w_high--) = *(w_low++);
    }
} // End_ function gauleg()

double gammln( double xx)
{
        double x,y,tmp,ser;
        static double cof[6]={76.18009172947146,-86.50532032941677,
                24.01409824083091,-1.231739572450155,
                0.1208650973866179e-2,-0.5395239384953e-5};
        int j;

        y=x=xx;
        tmp=x+5.5;
        tmp -= (x+0.5)*log(tmp);
        ser=1.000000000190015;
        for (j=0;j<=5;j++) ser += cof[j]/++y;
        return -tmp+log(2.5066282746310005*ser/x);
}

// end function gammln

void gaulag(double *x, double *w, int n, double alf)
{
```

```
        int i,its,j;
        double ai;
        double p1,p2,p3,pp,z,z1;

        for (i=1;i<=n;i++) {
                if (i == 1) {
                        z=(1.0+alf)*(3.0+0.92*alf)/(1.0+2.4*n+1.8*alf);
                } else if (i == 2) {
                        z += (15.0+6.25*alf)/(1.0+0.9*alf+2.5*n);
                } else {
                        ai=i-2;
                        z += ((1.0+2.55*ai)/(1.9*ai)+1.26*ai*alf/
                                (1.0+3.5*ai))*(z-x[i-2])/(1.0+0.3*alf);
                }
                for (its=1;its<=MAXIT;its++) {
                        p1=1.0;
                        p2=0.0;
                        for (j=1;j<=n;j++) {
                                p3=p2;
                                p2=p1;
                                p1=((2*j-1+alf-z)*p2-(j-1+alf)*p3)/j;
                        }
                        pp=(n*p1-(n+alf)*p2)/z;
                        z1=z;
                        z=z1-p1/pp;
                        if (fabs(z-z1) <= EPS) break;
                }
        if (its > MAXIT) cout << "too many iterations in gaulag" << endl;
        x[i]=z;
        w[i] = -exp(gammln(alf+n)-gammln((double)n))/(pp*n*p2);
        }
}
// end function gaulag
```

A.2 cartesian_gauleg.cpp

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <string>
#include <ctime>

#include "gauquad.cpp"

using namespace std;

double f ( double x1, double y1, double z1,
```

```
            double x2, double y2, double z2 );

ofstream outfile;

int main()
{

  bool append = 1;

  int      N    ;
  double lam = 3;

  const double    pi = 3.14159265358979323846;
  const double exact = 5*pi*pi/(16*16);

  double *x = new double [N],
         *w = new double [N];

  string filename = "cartesian_gauleg.dat";

  if (append) { outfile.open(filename, ofstream::app) ;}
  else        { outfile.open(filename)                ;}

  double I;

  clock_t start, stop;

  while (cin >> N) {

    start = clock();

    gauleg(-lam, lam, x, w, N);

    I = 0;

    for (int i = 0; i != N; ++i) {
              for (int j = 0; j != N; ++j) {
              for (int k = 0; k != N; ++k) {
              for (int l = 0; l != N; ++l) {
              for (int m = 0; m != N; ++m) {
              for (int n = 0; n != N; ++n) {

    I += w[i]*w[j]*w[k]*w[l]*w[m]*w[n]
      *f(x[i],x[j],x[k],x[l],x[m],x[n]);

    }}}}}}

    stop = clock();

    outfile << setw(8) << setprecision(10)
```

```
    << 6*N << ' ' << I << ' '

    << abs(I - exact) << ' ' << (double) (stop - start)/CLOCKS_PER_SEC

    << endl;

  }

  outfile.close();

  delete [] x;
  delete [] w;

  return 0;

}

double f ( double x1, double y1, double z1,
           double x2, double y2, double z2 )
{

  double r1  = sqrt(pow(x1, 2) + pow(y1, 2) + pow(z1, 2));
  double r2  = sqrt(pow(x2, 2) + pow(y2, 2) + pow(z2, 2));

  double r12 = sqrt(pow(x2 - x1, 2)
                  + pow(y2 - y1, 2)
                  + pow(z2 - z1, 2));

  double eps = 1e-4;
  if ( abs(r12) < eps ) { return 0                    ;}
  else                  { return exp(-4*(r1 + r2))/r12 ;}

}
```

A.3 spherical_gaulag.cpp

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <string>
#include <ctime>

#include "gauquad.cpp"

using namespace std;

double f( double     r1, double     r2,
          double   phi1, double   phi2,
```

```
          double theta1, double theta2 );

ofstream outfile;

int main()
{

  bool append = 1;

  int N;

  double alpha = 2;

  const double      pi = 3.14159265358979323846,
               two_pi = 2*pi;

  double exact = 5*pi*pi/(16*16);

  string        filename = "spherical_gaulag.dat";

  if (append) { outfile.open(filename, ofstream::app) ;}
  else        { outfile.open(filename)                ;}

  double I;

  clock_t start, stop;

  while (cin >> N) {

    double *       r = new double [N+1],
           *     phi = new double [ N ],
           *  theta = new double [ N ],

           *      wr = new double [N+1],
           *    wphi = new double [ N ],
           * wtheta = new double [ N ];

    start = clock();

    gaulag(r, wr, N, alpha);

    gauleg(0, two_pi, phi, wphi, N);

    gauleg(0, pi, theta, wtheta, N);

    I = 0;

    for (int i = 1; i != N+1; ++i) {
              for (int j = 1; j != N+1; ++j) {
              for (int k = 0; k != N  ; ++k) {
```

```
                for (int l = 0; l != N  ; ++l) {
                for (int m = 0; m != N  ; ++m) {
                for (int n = 0; n != N  ; ++n) {

    I += wr[i]*wr[j]*wphi[k]*wphi[l]*wtheta[m]*wtheta[n]
      * f(r[i], r[j], phi[k], phi[l], theta[m], theta[n]);

    }}}}}}

    I /= pow(2*alpha, 5);

    stop = clock();

    outfile << setw(8) << setprecision(10)

    << 6*N << ' ' << I << ' '

    << abs(I - exact) << ' ' << (double) (stop - start)/CLOCKS_PER_SEC

    << endl;

    delete []      r;
    delete []    phi;
    delete [] theta;

    delete []     wr;
    delete []   wphi;
    delete [] wtheta;

  }

  return 0;

}

double f( double     r1, double     r2,
          double   phi1, double   phi2,
          double theta1, double theta2 )
{

  double cosbeta = cos(theta1)*cos(theta2)
                 + sin(theta1)*sin(theta2)*cos(phi1 - phi2);

  double r12    = sqrt(pow(r1, 2) + pow(r2, 2) - 2*r1*r2*cosbeta);

  if (r12 <= 1e-4 || isnan(r12)) { return 0 ;}
  else                           { return sin(theta1)*sin(theta2)/r12 ;}

}
```

A.4 bruteforceMC.cpp

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <string>
#include <ctime>
#include <random>

using namespace std;

double f ( double x1, double y1, double z1,
           double x2, double y2, double z2 );

ofstream outfile;

int main()
{

  bool append = 1;

  int N;
  double lam = 3;

  double x1, y1, z1,
         x2, y2, z2;

  double I, u, sigma, F;

  const double     pi = 3.14159265358979323846;
  const double exact = 5*pi*pi/(16*16);

  random_device rd;
  mt19937_64 gen(rd());
  uniform_real_distribution<double> RandomNumberGenerator(0.0,1.0);

  clock_t start, stop;

  string filename = "bruteforceMC.dat";

  if (append) { outfile.open(filename, ofstream::app) ;}
  else        { outfile.open(filename)                ;}

  while (cin >> N) {

    I = 0; sigma = 0;

    start = clock();
```

```
    for (int i = 0; i != N; ++i) {

      u = RandomNumberGenerator(gen); x1 = lam*(2*u - 1);
      u = RandomNumberGenerator(gen); x2 = lam*(2*u - 1);
      u = RandomNumberGenerator(gen); y1 = lam*(2*u - 1);
      u = RandomNumberGenerator(gen); y2 = lam*(2*u - 1);
      u = RandomNumberGenerator(gen); z1 = lam*(2*u - 1);
      u = RandomNumberGenerator(gen); z2 = lam*(2*u - 1);

      F = f(x1, y1, z1, x2, y2, z2);
      I += F;
      sigma += F*F;

    }

    I *= pow(2*lam, 6);
    I /= N;

    stop = clock();

    sigma *= pow(2*lam, 12)/N;
    sigma -= I*I;
    sigma = sqrt(sigma);

    outfile << setw(8) << setprecision(10)

    << N << ' ' << I << ' '

    << abs(I - exact) << ' ' << (double) (stop - start)/CLOCKS_PER_SEC

    << ' ' << sigma << endl;

  }

  outfile.close();

  return 0;

}

double f ( double x1, double y1, double z1,
           double x2, double y2, double z2)
{

  double r1  = sqrt(pow(x1, 2) + pow(y1, 2) + pow(z1, 2));
  double r2  = sqrt(pow(x2, 2) + pow(y2, 2) + pow(z2, 2));

  double r12 = sqrt(pow(x2 - x1, 2)
                  + pow(y2 - y1, 2)
                  + pow(z2 - z1, 2));
```

```
  double eps = 1e-4;
  if ( abs(r12) < eps ) { return 0                    ;}
  else                  { return exp(-4*(r1 + r2))/r12 ;}

}
```

A.5 improvedMC.cpp

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <string>
#include <ctime>
#include <random>

using namespace std;

double f( double     r1, double     r2,
          double   phi1, double   phi2,
          double theta1, double theta2 );

ofstream outfile;

int main()
{

  bool append = 1;

  int N;

  const double     pi = 3.14159265358979323846,
               two_pi = 2*pi;
  const double exact  = 5*pi*pi/(16*16);

  random_device rd;
  mt19937_64 gen(rd());
  uniform_real_distribution<double> RandomNumberGenerator(0.0,1.0);

  double     r1,     r2,
           phi1,   phi2,
         theta1, theta2;

  double I, u, sigma, F;

  clock_t start, stop;

  string filename = "improvedMC.dat";
```

```
  if (append) { outfile.open(filename, ofstream::app) ;}
  else        { outfile.open(filename)                ;}

  while (cin >> N) {

    I = 0; sigma = 0;

    start = clock();

    for (int i = 0; i != N; ++i) {

      u = RandomNumberGenerator(gen); r1     = -log(1 - u);
      u = RandomNumberGenerator(gen); r2     = -log(1 - u);
      u = RandomNumberGenerator(gen); phi1   =   two_pi*u ;
      u = RandomNumberGenerator(gen); phi2   =   two_pi*u ;
      u = RandomNumberGenerator(gen); theta1 =       pi*u ;
      u = RandomNumberGenerator(gen); theta2 =       pi*u ;

      F = r1*r1 * r2*r2
        *  sin(theta1)*sin(theta2)
        *  exp(r1 + r2)
        *  f(r1,  r2, phi1, phi2, theta1, theta2);

      I += F;
      sigma += F*F;

    }

    I *= 4*pi*pi*pi*pi;
    I /= N;

    stop = clock();

    sigma *= 16*pow(pi, 8)/N;
    sigma -= I*I;
    sigma = sqrt(sigma);

    outfile << setw(8) << setprecision(10)

    << N << ' ' << I << ' '

    << abs(I - exact) << ' ' << (double) (stop - start)/CLOCKS_PER_SEC

    << ' ' << sigma << endl;
  }

  return 0;
}

double f( double      r1, double      r2,
```

```
            double    phi1, double    phi2,
            double theta1, double theta2 )
{

  double cosbeta = cos(theta1)*cos(theta2)
                 + sin(theta1)*sin(theta2)*cos(phi1 - phi2);

  double r12    = sqrt(pow(r1, 2) + pow(r2, 2) - 2*r1*r2*cosbeta);


  if (r12 <= 1e-4 || isnan(r12)) { return 0                    ;}
  else                           { return exp(-4*(r1 + r2))/r12 ;}

}
```

References

[1] Mary L. Boas. *Mathematical Methods in the Physical Sciences, 3rd ed.* 2006.

[2] Nouredine Zettili. *Quantum Mechanics Concepts and Applications, 2nd ed.* 2009.