FYS3150 Computational Physics - Project 3

Minju Kum

October 21, 2019

The expectation value of correlation energy between two electrons in helium atom was calculated using Gaussian quadrature and Monte Carlo method, and compared with its closed-form solution. Gauss-Laguerre quadrature outperformed Gauss-Legendre quadrature in terms of accuracy and efficiency, but the Gaussian quadrature itself was still inefficient to perform multi-dimensional integration due to its dimension-dependent errors. Monte Carlo method, especially the improved one, was best suitable for multi-dimension integration for its high accuracy with few integration points and its convergence without much fluctuation. Task parallelization and optimization made it faster twice and three times each.

I Introduction

When dealing with subatomic systems in quantum mechanics, one cannot predict the exact value of a physical quantity such as position, momentum, Energy, etc. Instead, one can only know some probabilities of possible results. Therefore, we often come to calculating the expectation values of physical quantities in interest, based on the known wave functions of the system. In this project, we particularly focus on a system of two electrons in a helium atom. The main goal is to calculate the expectation value of the ground state correlation energy between the electrons using Gaussian quadrature and Monte Carlo method. The integral dealt with in this project appears in many quantum mechanical applications, but it is impossible to find closed-form or analytical solution for the case. Therefore, we make assumptions that the wave function of each electron can be modeled like a single-particle wave function in the *1s* state in a hydrogen atom, and the wave function for two electrons is given by the product of those two wave functions. With the assumptions, the integration appears in the form

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

In Gaussian quadrature part, Legendre polynomials and Laguerre polynomials are utilized for numerical integration. The results are compared in the aspects of accuracy and efficiency. Overall aspects of errors and possible error sources are also discussed. With

Monte Carlo method, the integration is first approached in somewhat brute force way, without any coordinate transformation or importance sampling. Later we utilize both, and see if the results improve in terms of standard deviation. We also compare the CPU time. The latter case of Monte Carlo method is also parallelized with MPI, with some combinations of code optimization option.

II Method

2.1 Derivation and general settings

We first assume that that the wave function of each electron can be modeled like a single-particle wave function in the *1s* state in a hydrogen atom. The corresponding wave function is in the form of

$$\psi_{1s}(\boldsymbol{r}_i) = e^{-\alpha r_i}, \qquad \text{while } \ \boldsymbol{r_i} = x_i \boldsymbol{e}_x + y_i \boldsymbol{e}_y + z_i \boldsymbol{e}_z.$$

r_i is the magnitude of the vector $\boldsymbol{r}_i$. Here we fix $\alpha = 2$, which is given as the charge of the helium atom. Then we assume that the wave function for two electrons is given by the product of those two wave functions of each electron, namely,

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

The correlation energy can be written as

$$E_{corr} = \frac{1}{|\boldsymbol{r_1} - \boldsymbol{r_2}|}$$

omitting constant multiplications. Then the expectation value of E_{corr} can be calculated as below.

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

The wave function is not normalized here. The integral has closed-form solution $5\pi^2/16^2$. We compare our numerical results with this value.

2.2 Integration with Gaussian quadrature

The idea of Gaussian Quadrature is to approximate the integral with following relation

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

We factor out the weight function $W(x)$ from the original integrand $f(x)$, while ω_i and x_i are weights and mesh points of selected orthogonal polynomial of order N. If $g(x)$ is a polynomial under the order of 2N-1, then the integration is exactly equalized with the summation. If $g(x)$ is not the case, then we get the approximated result with the error of

$$\int_a^b W(x)g(x)dx - \sum_{i=1}^{N} \omega_i g(x_i) = \frac{g^{2N}(c)}{(2N)!} \int_a^b W(x)[q_N(x)]^2 dx,$$

where q_N is the selected orthogonal polynomial and c is a number in [a, b]. The virtue of Gaussian quadrature is that we can approximate the function to a polynomial of degree 2N-1 with only N points. It is appreciable compared to Newton-Cotes method, which approximates the function to a polynomial of degree N-1. In this project we check if Gaussian quadrature is still appreciable in high-dimensional integrations.

2.2.1 Gauss-Legendre quadrature

In Gauss-Legendre quadrature, weight function $W(x)$ is given as 1. The Cartesian coordinate was used here, with no variable transformation. Therefore, the original integrand

$$f = e^{-4(\sqrt{x_1^2+y_1^2+z_1^2}+\sqrt{x_2^2+y_2^2+z_2^2})} \frac{1}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}}$$

is used straightforwardly. Mathematically, the mesh points x_i are calculated as the roots of Nth Legendre polynomial. The weights ω_i are calculated as the first row of $\hat{L}^{-1}$ multiplied by 2, where $L_i(x)$ stands for i-th order Legendre polynomial and $\hat{L}$ being

$$\hat{L} = \begin{pmatrix} L_0(x_0) & L_1(x_0) & \dots & L_{N-1}(x_0) \\ L_0(x_1) & L_1(x_1) & \dots & L_{N-1}(x_1) \\ \dots & \dots & \dots & \dots \\ L_0(x_{N-1}) & L_1(x_{N-1}) & \dots & L_{N-1}(x_{N-1}) \end{pmatrix}.$$

In the program code, library functions provided from this course with lib.cpp were used to calculate the abscissas with appropriate scaling. The functions were originally written in Fortran (Press et al., 2007) and later translated to C++ (Hjort-Jensen, 2015). Library files are available in GitHub repository, mentioned in references.

There are some aspects to point out. First, the integration ranges from $(-\infty, \infty)$ but the Legendre polynomial is defined in [-1, 1], therefore infinity was approximated with finite number λ. In this project, λ was set to 2.5. A plot of probability density for two electrons (wave function squared, the exponential part) is presented below, and it shows that the function sufficiently goes to zero at $\lambda = 2.5$. However, even if the contribution of excluded part is small, it still contributes to the error to some extent.

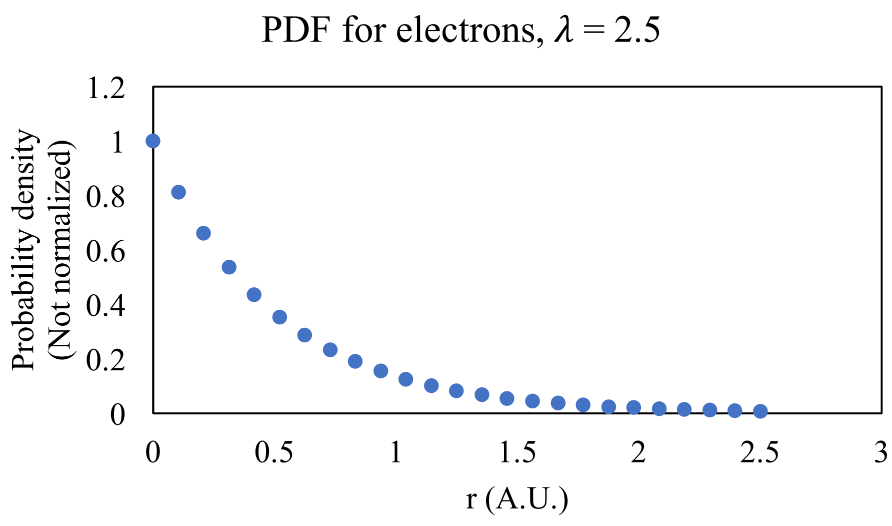


Figure 1. Probability density function(PDF) of two electrons. Here we can see that the function sufficiently goes to zero at λ=2.5.

Second, there is a possibility of denominator going to zero, which makes the integral diverge. In order to avoid this, the denominator was checked before evaluating the function and the function was simply set to zero when it was the case.

2.2.2 Gauss-Laguerre quadrature

Next we used Laguerre polynomial. Laguerre polynomials are defined for x in $[0, \infty)$, so if we transform the variables from Cartesian coordinates to spherical coordinates, we do not need to approximate infinity with finite number. The transformation goes as

$$dr_1 dr_2 = r_1^2 dr_1 r_2^2 dr_2 sin(\theta_1) d\theta_1 sin(\theta_2) d\theta_2 d\phi_1 d\phi_2,$$

$$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\beta)}},$$

while

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

Now abscissas for r_i and corresponding weights can be calculated in the same manner with Laguerre polynomials. For θ_i and ϕ_i, we can again use Legendre polynomials, with integration limits $[0,\ \pi]$ and $[0,\ 2\pi]$ for each. In the program code, the library functions in gauss-laguerre.cpp file from (Hjort-Jensen, 2015) were used to calculate the abscissas.

Again there are some points to note. First, the weight function is given as $W(x) = x^\alpha e^{-x}$ so we can choose the value of α. In the program $\alpha = 2$ was chosen so that the r_i^2 factors can be also absorbed into the weights together with $e^{-4(r_1+r_2)}$ factor. In order to do this, r_i was transformed as $4r_i = r_i{}'$ and $dr_i = \frac{1}{4} dr_i{}'$. Therefore, the actual integration turns out as

$$\int_0^\infty dr_1{}' \int_0^\infty dr_2{}' \int_0^\pi d\theta_1 \int_0^\pi d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 \frac{sin\theta_1 sin\theta_2}{1024\sqrt{r'^2_1 + r'^2_2 - 2r'_1 r'_2 \cos(\beta)}}.$$

We keep $cos(\beta)$ here for simplicity.

Second, different condition for avoiding division by zero was used. In this case (and in further case of 2.3.3) the function was set to zero if the value in the root of denominator was smaller than 10^{-10}. The reason for this is because the integral diverged for smaller thresholds such as $10^{-20}, 10^{-15}$.

2.3 Integration with Monte Carlo method

In Monte Carlo method, integration points are randomly chosen with corresponding PDF. The function to be integrated is evaluated with those random points and averaged. The result can be improved in accuracy by choosing an appropriate PDF for the integrand and increasing the number of samples N. The standard deviation of the result using Monte Carlo method is independent of dimension. It is only related to the size of samples N, namely,

$$STD \sim N^{-1/2}.$$

For Newton-Cotes quadrature which carries a truncation error $\sim O(h^k)$ with k being larger or equal to 1, integration in multi-dimension is highly inefficient. If we calculated the integral with dimension d, the error scales as

$$error \sim N^{-k/d}.$$

Therefore, we need to increase N to get the satisfactory result. But the number of nested *for* loops grows with the dimension, increasing the number of floating point operations rapidly. So, if we are doing integration in higher dimensions, we see the virtue of Monte

Carlo method. We simply increase the size of samples, not worrying about nested *for* loops, and choose the appropriate PDF for stochastic variables. Note that here we are describing STD in the sense of error, but it can often be not the case, strictly speaking.

In the experiment, together with the integral(*I*), the error and standard deviation(STD) was calculated as below

$$I = \frac{1}{N}\sum_{i=0}^{N-1} f_i$$

$$error = \frac{5\pi^2}{16^2} - \frac{1}{N}\sum_{i=0}^{N-1} f_i$$

$$STD = \sqrt{(< f^2 > - < f >^2)/N}$$

where *f* being the integrand and N being the sample size. CPU time was also measured with clock() function, which was situated in right above the start of sampling and right below the end of STD calculation. For CPU time, it was measured 3 times each and averaged.

2.3.1 Random Number Generator(RNG)

Before diving into the details of two Monte Carlo methods used here, we first discuss on the random number generator. Proper randomness with appropriate distribution is the key factor of Monte Carlo method. Since all we can generate is not real random numbers but pseudo-random numbers, the numbers should have certain properties that are good enough to consider the numbers random. The generated numbers should be identically distributed within the range, have very low correlation with previous numbers, and have sufficiently long enough period. In this project, Mersenne Twister pseudo-random generator(64-bit) from C++ library <random> was used. The RNG generates 64-bit numbers with a period of 2^{19937}, which is long enough for this project, and it has passed several tests (Matsumoto et al., 1998) regarding the properties mentioned above. The number of ticks for a short time duration was given as a seed, which varied every time when measured. The generated numbers were then distributed with uniform/exponential distribution functions included in <random>.

2.3.2 Brute-force Monte Carlo method

Here we first implement somewhat "brute-force" Monte Carlo method. In this part, we calculate the integral with the same integrand as shown in 2.2.1, with the same Cartesian coordinates and the same value of λ. Note that we also ignore the contribution of division by zero in the same way as in 2.2.1. The only difference here is that the variables are chosen randomly with Mersenne Twister RNG, and distributed with uniform distribution function in the range of $(-\lambda, \lambda)$. The integration follows as

$$(2\lambda)^6 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{e^{-4(\sqrt{x_1^2+y_1^2+z_1^2}+\sqrt{x_2^2+y_2^2+z_2^2})} dx_1' dx_2' dy_1' dy_2' dz_1' dz_2'}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}}.$$

With $x_1 =-\lambda + 2\lambda x_1'$ and so on.

2.3.3 Improved Monte Carlo method

Next, we improve the previous result by transforming variables from the Cartesian coordinates to spherical coordinates and changing variables with appropriate PDF, which is often called importance sampling. The coordinate-transformed version of integration is

$$\int_0^\infty dr_1 \int_0^\infty dr_2 \int_0^\pi d\theta_1 \int_0^\pi d\theta_2 \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 \frac{e^{-4(r_1+r_2)} r_1^2 r_2^2 sin\theta_1 sin\theta_2}{\sqrt{r_1^2+r_2^2-2r_1r_2\cos(\beta)}}.$$

What we want here is to factor out the exponential part of the integrand. Consider that we have an exponential distribution of r_i

$$p(r_i) = 4e^{-4r_i}.$$

We introduce another variable x_i which follows uniform distribution in [0, 1] and with the relation

$$p(r_i)dr_i = dx_i$$

$$x_i(r_i) = \int_0^{r_i} p(r_i')dr_i' = \int_0^{r_i} 4e^{-4r_i'}dr_i'$$

we can rewrite

$$r_i = -\frac{1}{4}\ln(1-x).$$

Other angular variables are distributed in uniform PDF in the range of $[0, \pi]$ and $[0, 2\pi]$ each. Then we get our integration formula

$$4\pi^4 \int_0^1 dx_1 \int_0^1 dx_2 \int_0^1 dx_3 \int_0^1 dx_4 \int_0^1 dx_5 \int_0^1 dx_6 \frac{r_1^2 r_2^2 sin\theta_1 sin\theta_2}{16\sqrt{r_1^2+r_2^2-2r_1r_2\cos(\beta)}},$$

with

$$r_i = -\frac{1}{4}\ln(1-x), \qquad \theta_i = \pi x, \qquad \phi_i = 2\pi x.$$

What we have just done is to remove the exponential factor, but rather to pick r_i with exponential distribution. This will contribute to the reduction of STD of the integral.

2.3.4 Parallelization and optimization

For the last part of this project, task parallelization of 2.3.3 was done with some optimization options combined. Since the CPU time is mainly used for sampling Monte Carlo variables and the task is not related to one another, it can be easily parallelized. The task was distributed to two processors of a laptop computer (1.3 GHz Intel Core m7) using MPI.

All codes for this project were developed with C++, and after completely debugged, were again compiled and linked with optimization flag -O3 of c++ compiler. All codes were also tested and confirmed that -O3 does not make any inconsistency in the results. But for this section, MPI-wrapped compiler mpic++ was used. For the very last part, the improved Monte Carlo was tested under the following conditions; (1) with both -O3 flag and parallelization, (2) with -O3 flag and without parallelization, (3) without -O3 flag and with parallelization.

2.4 Source codes and files

Source codes and files related to this project can be found in GitHub repository:

`https://github.com/minjukum/FYS3150-Computational-Physics`

and library functions can be found in:

`https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Programs/LecturePrograms/programs/cppLibrary.`

III Results and Discussion

3.1 Gaussian quadrature

Here we present and compare the results from Gauss-Legendre and Gauss-Laguerre quadrature. Convergence and fluctuation of two methods are shown visually in Figure 2.

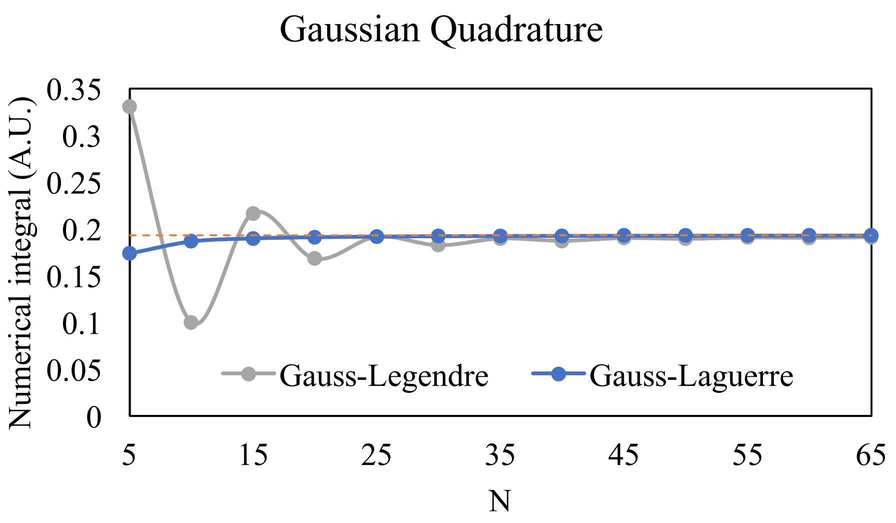


Figure 2. Results from Gauss-Legendre and Gauss-Laguerre quadrature, with N varying from 5 to 65. Gauss-Laguerre quadrature shows less fluctuation and converges faster to the exact integral (~0.192766) which is marked with an orange dotted line.

Apparently, Gauss-Laguerre method shows less fluctuation and converges faster to the exact integral, with less number of mesh points N. Results from Gauss-Legendre quadrature fluctuate largely around the exact value with small N, and gradually converge. On the other hand, results from Gauss-Laguerre quadrature normally lie below the exact value, and also gradually converge.

Next, we precisely look into the data with the table below.

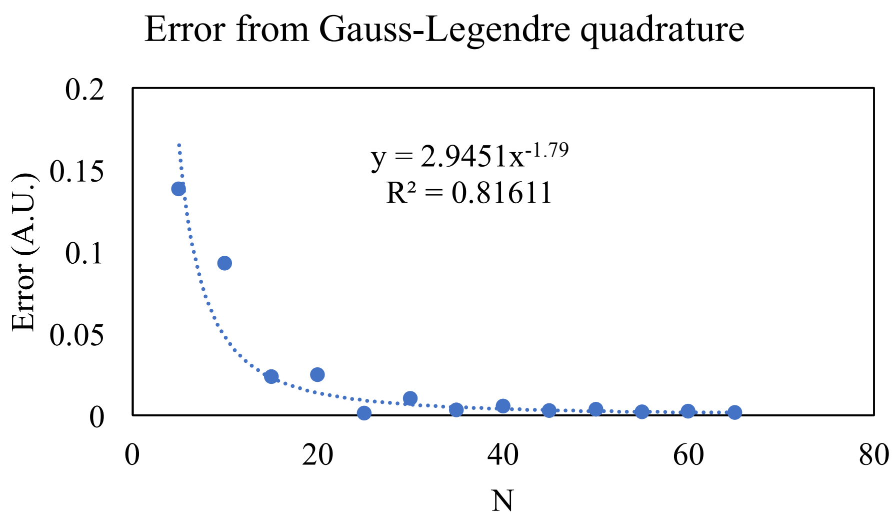
Table 1. Results from Gauss-Legendre and Gauss-Laguerre quadrature are shown with numbers. Using Legendre polynomial, it needs more that 65 mesh points to converge to the exact integral ~0.192766 at the level of the third leading digit, and it converges very slowly. Gauss-Laguerre quadrature shows convergence to the third leading digit with only 30 mesh points, and to the fourth with 55 mesh points.

N	Legendre	Error	Laguerre	Error
10	0.100058	0.0927074	0.186457	0.00630837
15	0.216501	-0.0237353	0.189759	0.00300673
20	0.167971	0.024795	0.191082	0.00168393

25	0.191366	0.00139959	0.191741	0.00102497
30	0.182323	0.0104423	0.192114	0.000651999
35	0.189474	0.00329181	0.192343	0.00042241
40	0.187062	0.00570344	0.192493	0.000272466
45	0.18996	0.00280598	0.192596	0.000170168
50	0.189158	0.00360747	0.192668	9.80E-05
55	0.190566	0.00220017	0.192720	4.58E-05
60	0.190264	0.00250136	0.192758	7.23E-06
65	0.191038	0.00172787	0.192787	-2.17E-05

The closed-form result is calculated as ≈ 0.192766. Gauss-Legendre quadrature needs more that 65 mesh points to converge at the level of the third leading digit, and it converges very slowly. On the other hand, integral using Gauss-Laguerre quadrature shows convergence to the third leading digit with only 30 mesh points, and to the fourth with 55 mesh points. We can see that Legendre polynomial is very inappropriate for this integration. The errors of Gauss-Legendre quadrature can result from several reasons. First, approximating the exponential with polynomials makes the infinitely high-ordered polynomials contribute to the integration. And it is omitted when approximated with finite degree. This can also be the reason for slow convergence of Gauss-Legendre quadrature. Second, approximating infinity with finite number, here with $\lambda = 2.5$, results in error, because it ignores the contribution of the integral outside of the limited range ± 2.5. Third, we simply ignored the contribution when the denominator was 0. However, this can be dealt in better way. The better treatment for singular integral is presented in (Hjort-Jensen, 2015), but was not implemented in this project. Further study with the implementation of this method is needed. Gauss-Laguerre quadrature does not approximate exponential part, but only the inverse square root part of the integrand. It also does not approximate infinity with finite number. Those aspects contributes to the better results, but it still ignores the contribution when the denominator is close to 0.

Next, we look into the relation between errors and number of mesh points N. Below are Error graphs for both method. To see how much the results differed from the exact value, the absolute values of errors were used. We see that the errors from two method follow different form of regression. For Gauss-Legendre quadrature, errors are approximately in the order of $O(N^{-2})$. However, for Gauss-Laguerre quadrature, errors decrease in exponential manner with increasing N. The difference arises from the different properties of orthogonal polynomials.



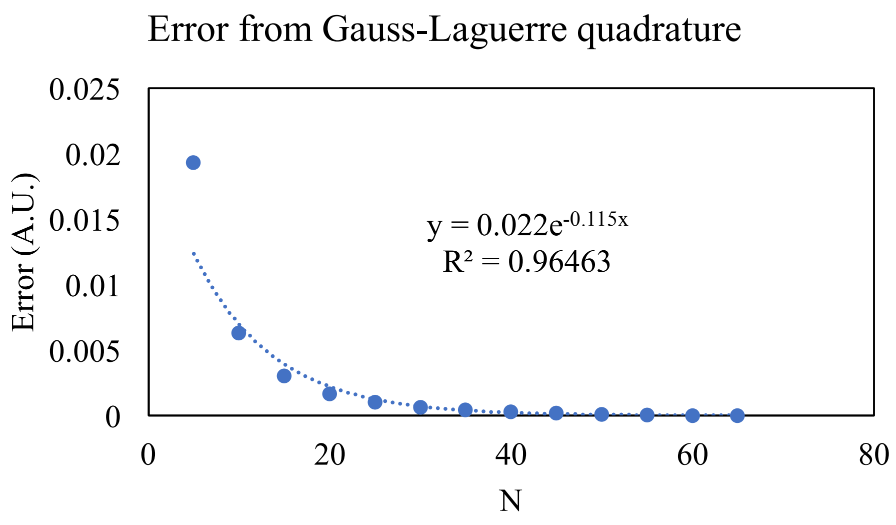


Figure 3. Errors from both Gauss-Legendre(above), Gauss-Laguerre(below) quadrature are presented with degree of polynomial N. Regression formula with the maximum R^2 was chosen. For Gauss-Legendre quadrature, R^2 is not close to 1 due to large fluctuations. Here we see that errors of Gauss-Legendre quadrature follows approximately $E \sim O(N^{-2})$, while errors of Gauss-Laguerre quadrature decrease exponentially with increasing N.

From the results, we can apparently see that Gauss-Laguerre quadrature approximates the exact integral much better than Gauss-Legendre quadrature for the integration in matter. It absorbs the exponent to the weights, so the integrand is made smoother and converges faster. But it still costs a lot of time (several minutes to an hour) to get the precision of say fourth leading digits or more. In addition, it is inevitably confronted with situations that denominators become zero. In the next part, we implement Monte Carlo method to deal with these problems.

3.2 Monte Carlo method

We first present the results from "brute-force" Monte Carlo method. Figure 4 shows the results of integration evaluated with varying sample size N. With small N, the numerical evaluation never reaches the exact value. From N=10000, the result gradually stabilizes and reaches around the exact value. Even though the integrand is centered about the origin, the method still picks integration points uniformly within the integration limits. So, the chance of function values of picked points being evaluated close to the average is quite low. This makes the numerical result with small N so small, which thence makes the STD also small. Besides, STD appears to be so small that it does not include the exact value. Around N = 10000, STD hits the maximum and then decreases gradually.

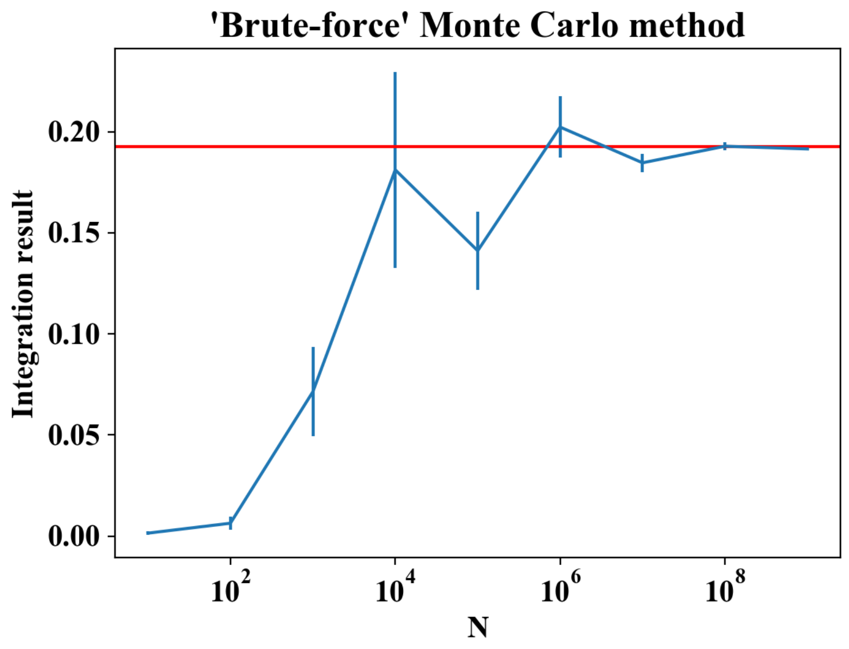


Figure 4. Integration evaluated with brute-force Monte Carlo method. Error bar shows the standard deviation of integral. Here we can see that with small N, the result of numerical integration is very small compared to the exact result (red line), and the exact value does not lie in the range of error bars. From N = 10000, The result gradually stabilizes around the red line. Standard deviation is very small for small Ns, hits the maximum at N=10000, then gradually decreases.

The behavior of error can be interpreted as following. When N is small, the scale of result is also small, because the chance of picking integration points close to 0 so that the function values evaluated close to the average is low. Therefore, the STD is also small. However, when N reaches large enough to evaluate some big enough function values, the variance of f itself hits the maximum. But as N grows further, the variance of f may stay the same, but dividing with larger $\sqrt{N}$ makes the variance decrease. This argument is justified by Figure 5, which shows that from N = 10^6, the STD is in the order of $O(N^{-1/2})$.

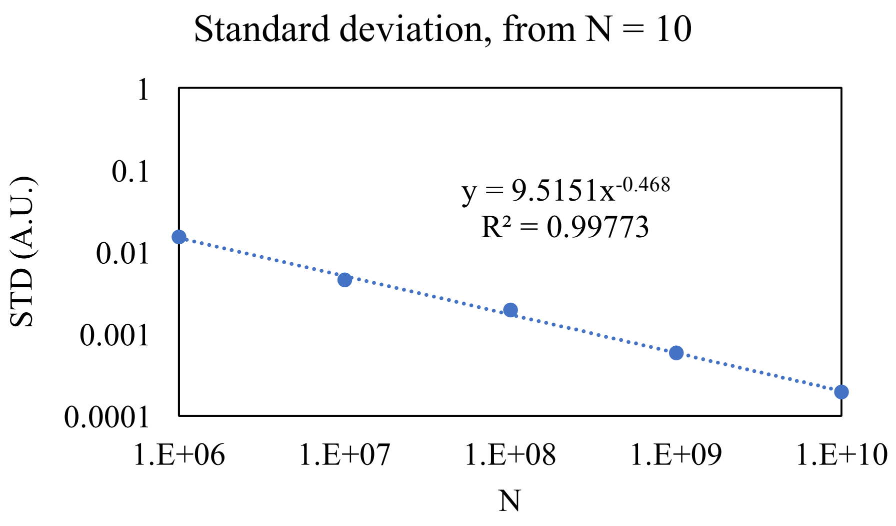


Figure 5. Standard deviation of the result from selected point N = 10^6. The regression shows that STD is in the order of $O(N^{-1/2})$.

However, decreasing STD does not necessarily mean decreasing error. For N = 10^5, the STD decreased but the exact result does not lie within the range. This is the weak point of Monte Carlo method. There are some chances that the exact value lies within the range of STD = 1, this time being 0.68, but that also means that there are some chances that it does not lie within the range. Growing STD would increase the chance, but the result will appear to be fuzzier.

Next, we look into the result with numbers, and see how it is accurate.

Table 2. Here we see the result of brute-force Monte Carlo method with numbers. Compared with previous results, it does not seem to be improved in accuracy. Even with large number of points, it sometimes deviates from the exact result ~0.192766.

Log(N)	**Result of brute-force MC**	**error**	**STD**
1	0.001424007	0.1913417	0.000777337
2	0.006325716	0.18643999	0.003294475
3	0.071319746	0.12144596	0.022088027

4	0.1811166	0.011649113	0.048436045
5	0.14108632	0.051679388	0.019256886
6	0.20225613	-0.009490416	0.015140764
7	0.18458629	0.008179426	0.004529017
8	0.19279268	-2.70E-05	0.001928652
9	0.19143793	0.00132778	0.000575275
10	0.19263794	0.000127772	0.000194019

Above table shows that using brute-force Monte Carlo does not really improve the accuracy, compared to the previous Gaussian quadratures. Even with large number of integration points, it sometimes deviates from the exact result.

Next, we present the results of improved Monte Carlo method using importance sampling and coordinate transformation. Comparisons are made with the brute-force one in terms of standard deviation, CPU times and errors. The figure below shows the result with standard deviation of each method with varying N.

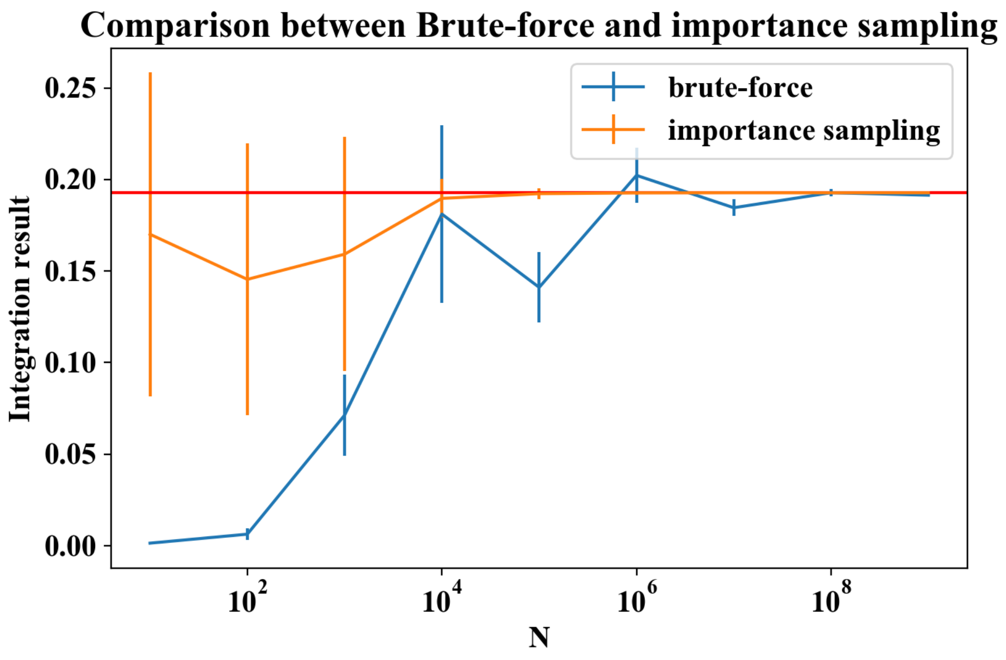


Figure 6.

Here we clearly see the results are quite improved. Now the exact result gets into the range of standard deviation with only 10 points. Variance for small N is now much bigger that the previous method, because the integration points are picked near the origin (which means the function values are not close to 0) and N is yet too small to divide out STD.

But as N increases, STD becomes even smaller than previous. Note that the decreasing rate is the same as before, with $1/\sqrt{N}$.

With CPU times needed for calculating N points, we can compare the efficiency in terms of time. The CPU time consumption of both method is presented in Table 3. Let's compare a case, for example, for brute-force it needs 10^8 integration points to reach the red line in above graph, while importance sampling needs only 10^5. From the table, it reads 8.85 seconds for brute-force and 0.02 second for importance sampling. That shows quite an improve.

Table 3. Here we see the CPU time of both method, the brute-force and importance sampling.

Log(N)	**CPU time of Brute-force MC(s)**	**CPU time of MC with importance sampling(s)**
1	2.17E-05	7.73E-05
2	4.37E-05	9.33E-05
3	2.03E-04	3.61E-04
4	2.23E-03	3.07E-03
5	1.51E-02	2.01E-02
6	9.67E-02	1.61E-01
7	8.94E-01	1.48E+00
8	8.85E+00	1.51E+01
9	8.82E+01	1.95E+02
10	9.69E+02	-

3.3 Parallelization and Optimization

Here presented the CPU time with parallelization and optimization options.

Table 4. CPU time of parallelization and optimization options. It is shown from the second and third column that parallelization need certain amount of time, which cannot be reduced. Comparing first-second column and second-third column, we can see that the optimization option reduce the time more that parallelization.

Log(N)	**With -O3, not parallelized**	**With -O3, parallelized**	**Without -O3, parallelized**
1	7.73333E-05	0.000272	0.000297
2	9.33333E-05	0.000013	0.00004
3	0.000360667	0.000102	0.000349
4	0.003072	0.001067	0.0028625

5	0.020131	0.008959	0.025391
6	0.160737333	0.0842835	0.2172455
7	1.475665667	0.7845595	2.1921895
8	15.06568833	7.8143725	25.621169
9	195.02196	96.988972	260.521875

From the second and the third column, we can see that parallelization itself needs certain amount of time which cannot be reduced. In the table, it reads around ~10^{-4}. However, the minimum time was reached at N=100, not N=10 in both columns.

By comparing the first-second columns and the second-third columns, we can see how much time was reduced by parallelization and optimization. Without considering first rows, we can see that parallelization approximately halves the time. However, it appears that optimization option -O3 does more that parallelization this time. The time with optimization was almost one third of the one without optimization. Since Monte Carlo loop is considered to be done independently, it can easily be vectorized.

V Conclusion

This project aimed to evaluate the integral which calculates the expectation value of the correlation energy of two electrons. We studied mainly two kinds of numerical methods, Gaussian quadrature and Monte Carlo method.

In Gaussian quadrature, we used two kinds of orthogonal polynomials to approximate the integral. Gauss-Laguerre quadrature appeared to better approximate the exact result, in terms of accuracy and efficiency. It converged faster to the exact result with less fluctuation, and only needed 30 integration points to converge at the level of third leading digits, while Gauss-Legendre needed 65. The error of two methods followed different form of regression formula, power law with Legendre and exponential with Laguerre. Here we can see the importance of picking the right approximating polynomials for integrands. However, the overall results pointed out the limitation of approximating multi-dimensional integration with orthogonal polynomials, which costs lots of operations with unsatisfactory improvements.

In Monte Carlo method, we also approached the problem in two ways. Brute-force Monte Carlo method showed somewhat disappointing result, for being quite absurd with few integration points and still being incorrect with quite a lot of integration points. On the other hand, improved Monte Carlo method with importance sampling reproduced the exact value within its STD with only few points, without much fluctuations. It also had an advantage of CPU time consumption being about 10^2 times faster than brute-force Monte Carlo to reach the same extent of accuracy. It was also checked if the STD of Monte Carlo method followed the relation $STD \sim N^{-1/2}$.

Lastly, the role of parallelization and optimization was studied. The parallelization approximately halved the CPU time, and again CPU time was reduced to one third of it with optimization option -O3.

References

Hjort-Jensen, M. (2015), Computational physics, accessible at course github repository. https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Lectures

Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS), 8(1), 3-30.

Press, William H., Teukolsky, Saul A., Vetterling, William T., Flannery, Brian P. (2007), Numerical Recipes: The Art of Scientific Computing (3rd ed.), Cambridge University Press, ISBN 9780521880688.