H2b: Varitional Monte Carlo

Carl Strandby and Didrik Palmqvist

December 2022

Introduction

The calculation of the ground state energy of a helium atom is a challenging problem that involves the solution of high-dimensional integrals [1]. One approach to this problem is the variational Monte Carlo method, which relies on random number sequences known as Markov Chains. This method is commonly used to approximate the solutions of problems that lack analytical solutions or are computationally expensive to solve using traditional numerical techniques. This paper demonstrates the use of Monte Carlo integration and the variation theorem to obtain the ground state energy of a helium atom. Sections (1) to (2) describe different problems and simulations, along with their theoretical foundations and the analysis of the simulation results. The last section, Section (6), discusses the advantages and disadvantages of using the variational Monte Carlo method for this calculation. The goal of this paper is to provide a theoretical framework for the Monte Carlo technique and to showcase its application to a specific example of a helium atom.

Problem 1

A helium atom is made up of two electrons and a nucleus, that influences each other via electric forces. The influence of motion by this force can be seperated from the nuclear motion, as justified by the Born-Oppenheimer approximation [2]. Then one can apply the variational theorem

$$E[\psi_T] = \frac{\langle\psi_T|\mathcal{H}|\psi_T\rangle}{\langle\psi_T|\psi_T\rangle} \geq E_0 \tag{1}$$

for an arbitrary trial wavefunction ψ_T [3]. This yields an upper bound for the ground state energy E_0. However, this expression can be computed by integrating over the expectation value for other degrees of freedom in a process sometimes referred to as marginalization. Specifically, the energy in equation (1) can be expressed as an integral over the coordinates $\mathcal{R}$ of the electrons;

$$E[\psi_t] = \int d\mathcal{R} E_L(\mathcal{R})\rho(\mathcal{R}) \tag{2}$$

where E_L is the local energy, and $\rho(\mathcal{R})$ it's probability distribution given by the equations

$$E_L = \frac{\mathcal{H}\psi_T(\mathcal{R})}{\psi_T(\mathcal{R})} \tag{3}$$

and

$$\rho(\mathcal{R}) = \frac{|\psi_T(\mathcal{R})|}{\int d\mathcal{R}|\psi_T(\mathcal{R})|^2}, \tag{4}$$

respectively. The specific trial wavefunction parameterized by α used in this report was

$$\psi_T(\mathbf{r}_1, \mathbf{r}_2) = \exp[-2r_1]\exp[-2r_2]\exp\left[\frac{r_{12}}{2(1+\alpha r_{12})}\right], \tag{5}$$

with the corresponding function for the local energy

$$E_L(\mathbf{r}_1,\mathbf{r}_2) = -4\frac{(\hat{\mathbf{r}}_1 - \hat{\mathbf{r}}_2)\cdot(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}(1+\alpha r_{12})^2} - \frac{1}{r_{12}(1+\alpha r_{12})^3} - \frac{1}{4(1+\alpha r_{12})^4} + \frac{1}{r_{12}}, \tag{6}$$

both being from the problem description [3]. To evaluate equation (6) the Metropolis algorithm was implemented and the configuration of the two particles positions where weighted by their probability distribution, obtained by taking the square modulus of the ψ_T. In problem 1, this was done with a parameter value of $\alpha = 0.1$. In the first step of the Metropolis algorithm, the positions of the electrons were chosen by generating a uniform random number $-0.5 \leq u_k \leq 0.5$ for each position coordinate x_k, and multiplying this with an initial displacement d_i as

$$x_{k,\text{initial}} = d_i u_k, \tag{7}$$

and all random numbers used in this project was generated by using time as a seed for the random number generator. For problem 1 the initial displacement was chosen to be $d_i = 1$, since this... and in later problems the system was initialized from a more unlikely configuration by setting $d_i = 50$. Trial positions were in turn generated by adding an offset $d_t u_k$ to the initial positions, where the trial displacement d_t was chosen so that trial positions would be accepted, according to rules described later in this section, around 40 % of the time, which amounted to a displacement factor of $d_t = 1.24$. The trial positions are then given by

$$x_{k,\text{trial}} = x_{k,\text{old}} + du_k. \tag{8}$$

At each step of the algorithm a new trial position was generated and compared to the old positions. The configuration with the highest probability of being occupied was accepted as the configuration carrying over to the next step, or if the ratio between the two probabilities was higher than a random number uniformly distributed between 0 and 1. In pseudocode this can be written as

$$\text{If} \quad \frac{P(\psi_t(\mathbf{r}_{trial}))}{P(\psi_t(\mathbf{r}_{old}))} \geq p, \quad p \in \mathcal{U}(0,1) \implies \mathbf{r}_{new} = \mathbf{r}_, \quad \text{else} \quad \mathbf{r}_{new} = \mathbf{r}_{old}. \tag{9}$$

One may note that the sampled distributions are not normalized, this is however not an issue as it only differs by a constant factor that wouldn't effect the probability of accepting a new position. The Markov-chain generated by performing the algorithm was then compared to the following model for the radial distribution of the electrons

$$\rho(\mathbf{r}) = Z^3 4r^2 e^{-2Zr} \tag{10}$$

with $Z = 2$ for an unscreened nucleus and $Z = 27/6$ for a variationally optimized value. The units used in this project are the Hartree units, in this system the unit of length is the Bohr radius

$$a_0 = \frac{4\pi\epsilon_0\hbar^2}{m_e e^2} \tag{11}$$

and the unit of energy is E_h

$$E_h = \frac{\hbar^2}{m_e a_0^2} \approx 27.211\,\text{eV}. \tag{12}$$

The Metropolis algorithm was performed with $3 \cdot 10^6$ samples, a stepsize $d = 1.24\, a_0$, $\alpha = 0.1$ and the average energy was calculated to $E_0 = -2.877293\, E_h$ and the acceptance ratio along the MCMC-chain was 40.2%. The comparison of the models and the sampled distributions can be seen in figure (1), one may note that the sampled distributions for the different electrons are similar which is what we expect since the electrons are interchangeable identical particle so the model should treat them symmetrically. The distributions also seems to coincide rather well with the radial distribution with the optimized value for Z.

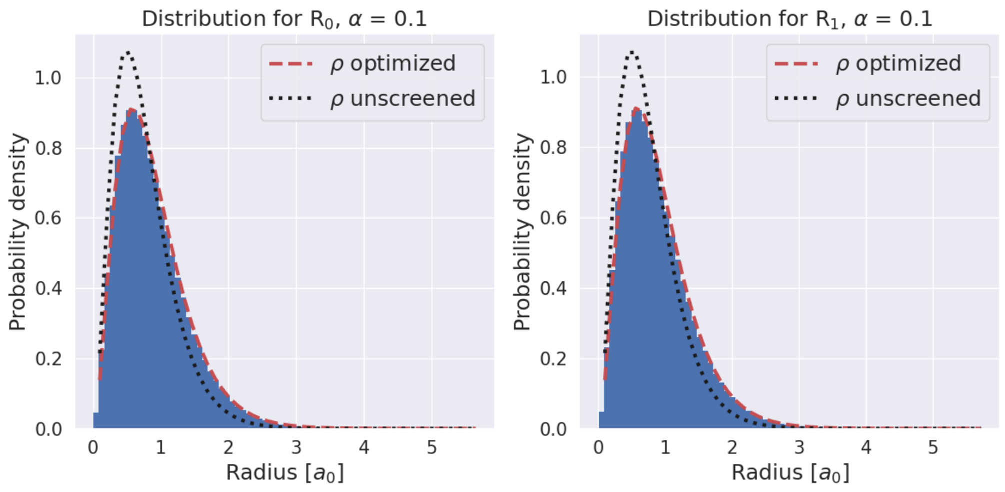


Figure 1: Radial distribution of the particles compared to eq.(10) from simulation with $N = 1e6$ steps and particles initiated with random positions with $x, y, z \in [-0.5, 0.5]\, [a_0]$. The X-axis is in units of Bohr radius a_0.

The angle θ between the $\mathbf{r}_1$ and $\mathbf{r}_2$ can be calculated by

$$\theta = \arccos\left(\frac{\mathbf{r}_1 \cdot \mathbf{r}_2}{|\mathbf{r}_1||\mathbf{r}_2|}\right) \tag{13}$$

and we define

$$x = \cos(\theta). \tag{14}$$

If the positions of the electrons are uncorrelated we would expect the angle between them to be uniformly distributed on the unit sphere with the probability of obtaining an value of x being given by

$$P(x) = \frac{1}{2}, \ -1 < x < 1 \tag{15}$$

and the probability of obtaining a angle θ being given by

$$P(\theta) = \frac{\sin(\theta)}{2}, \quad 0 < \theta < \pi. \tag{16}$$

In figure (2) we see the sampled distributions for θ and x. These differ from the case where the positions of the particles are uncorrelated and the angle θ is uniformly distributed on the unit sphere. This is to be expected since the model used for the trial wavefunction contains an interaction term that depends on the distance between the two electrons r_{12} and thus also the angle between the particles coordinate vectors. This also makes sense physically as the electrons should repell each other with electric forces since they both have negative charge.

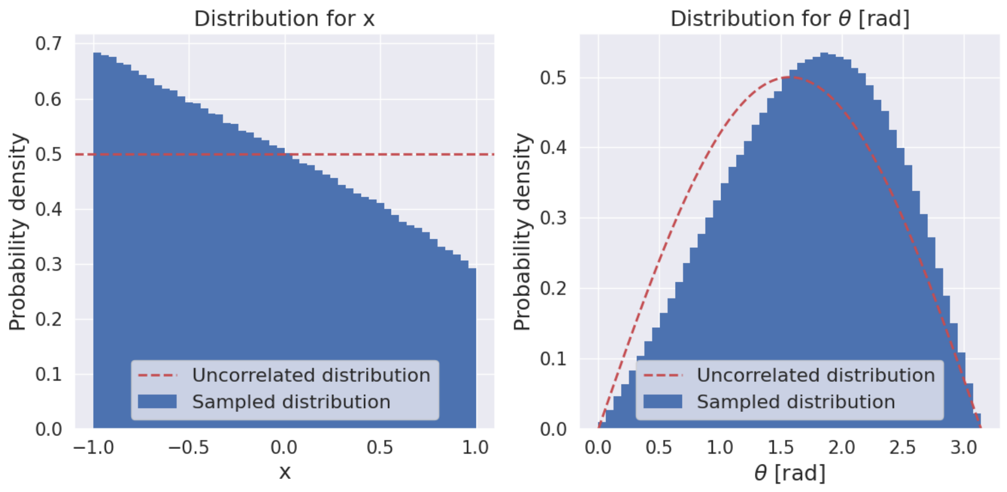


Figure 2: Sampled distribution of x, θ and $\sin(\theta)$ from simulation with $N = 1e6$ steps and particles initiated with random positions with $x, y, z \in [-0.5, 0.5]$. As the distributions deviate from the theoretically uncorrelated distributions, it can be seen that the equations used for the trial wavefunction and local energy takes into account some interaction forces between the two particles.

Problem 2

The next problem deals with equilibration of the MCMC-system and the determination of error bars using statistical inefficiency. The Monte-Carlo method is not self-starting, but depends on measurements made in the previous iterations. Quantities are measured as the average of the sampled distribution. And since it takes a few iterations to move around in the sampled configuration space, an unlikely initial configuration could skew this average. Therefore the system needs to equilibrate for a number of iterations, N_{eq} and the measurements discarded. The number of steps that needs to be discarded will among other things depend on the trial displacement, d_t, and is decided by choosing an initial configuration that is very unlikely, or at least more so than will show up in following initial configurations. Then, N_eq is taken as a value greater than the number of steps before the measurements can be seen to fluctuate according to the final distribution. One such measurement is seen in figure (3). Since the initial displacement d_i was arbitrarily set to 50, and the random number u_k to be $\in [-0.5, 0.5]$, the initial configuration was set to $\mathbf{r}_1 = (25, 25, 25)\,[a_0]$ and $\mathbf{r}_2 = (-25, -25, -25)\,[a_0]$. From this it was decided to use $N_{eq} \geq 1e3$, for following simulations.

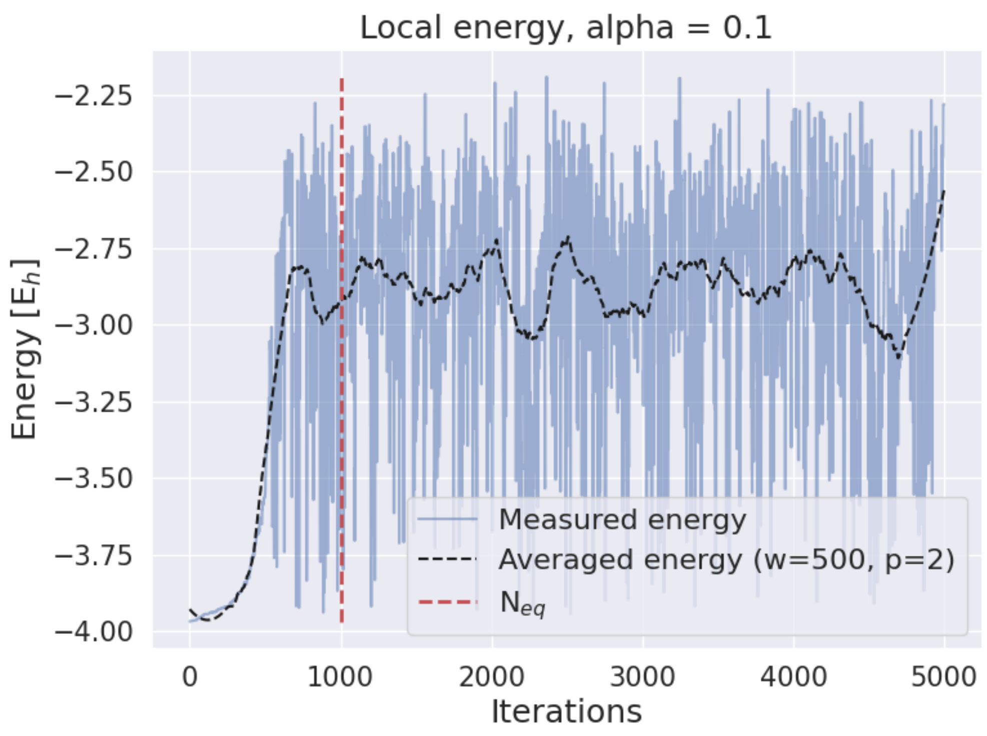


Figure 3: from simulation with $N = 1e6$ steps and particles initiated with random positions with $\mathbf{r}_1 = (25, 25, 25)\,[a_0]$ and $\mathbf{r}_2 = (-25, -25, -25)\,[a_0]$.

Next, it is important to be able to determine error bounds for the result of the MCMC-simulation. One way of doing this is by calculating the variance of a quantity generated from a MCMC-chain, given by

$$\mathrm{Var}[E] = \frac{\mathrm{Var}(\varepsilon)}{N} = \frac{\langle \varepsilon^2 \rangle - \langle \varepsilon \rangle^2}{N} = \frac{\sigma^2}{N}, \tag{17}$$

for N samples where $\mathrm{Var}(\varepsilon)$ denotes the variance of the sampled quantity along the MCMC-chain. However, this method of calculating the variance is only valid for uncorrelated datapoints. And the trial positions depends on the previously sampled distribution. Therefore one needs to introduce a measurement of how many iterations are needed between datapoints to ensure that they are not correlated, referred to as the effective number of samples, N_{eff}, given by

$$N_{\text{eff}} = \frac{N}{n_s}, \tag{18}$$

where n_s is called the statistical inefficiency.

In this paper, two methods to calculate the statistical inefficiency n_s have been used and the data used for this was generated by performing the Metropolis algorithm with $N_{\text{eq}} = 1e4$ equilibration steps, $N = 1e7$ samples and a step size of $d_t = 1.24\,[a_0]$. The first method is to evaluate the correlation function Φ_k defined for a sampled quantity ε as

$$\Phi_k = \frac{\langle \varepsilon_i \varepsilon_{i+k} \rangle - \langle \varepsilon \rangle}{\langle \varepsilon^2 \rangle - \langle \varepsilon \rangle^2}. \tag{19}$$

For a stationary system the correlation function is symmetric $\Phi_k = \Phi_{-k}$ and for large $k > N_c$ the correlation function decays to zero. For long simulations, $N > N_c$, one would then find

$$n_s = \sum_{k=-N_c}^{N_c} \Phi_k = 2\sum_{k=0}^{N_c} \Phi_k - \Phi_0 = 2\sum_{k=0}^{N_c} \Phi_k - 1. \tag{20}$$

By comparing this correlation function to an exponential decay one can define the relaxation time τ_{rel} as $\Phi(t) = \exp(-t/\tau_{\text{rel}})$, which in turn leads to the following expression to calculate the statistical inefficiency

$$\begin{aligned} n_s &= 2\tau_{\text{rel}}, \\ \Phi_{k=\tau_{\text{rel}}} &= \exp(-n_s/\tau_{\text{rel}}) = \exp(-2) \approx 0.1. \end{aligned} \tag{21}$$

The correlation function calculated from the simulation can be seen in figure (4) where the statistical inefficiency was calculated to $n_s = 11.248$ using eq.(20) and the relaxation time $\tau_{\text{rel}} = 5$. The relaxation time would imply $n_s = 10$ but since the sampled correlation function is discrete it doesn't take the value e^{-2} exactly and the value $n_s = 11.248$ was used instead.

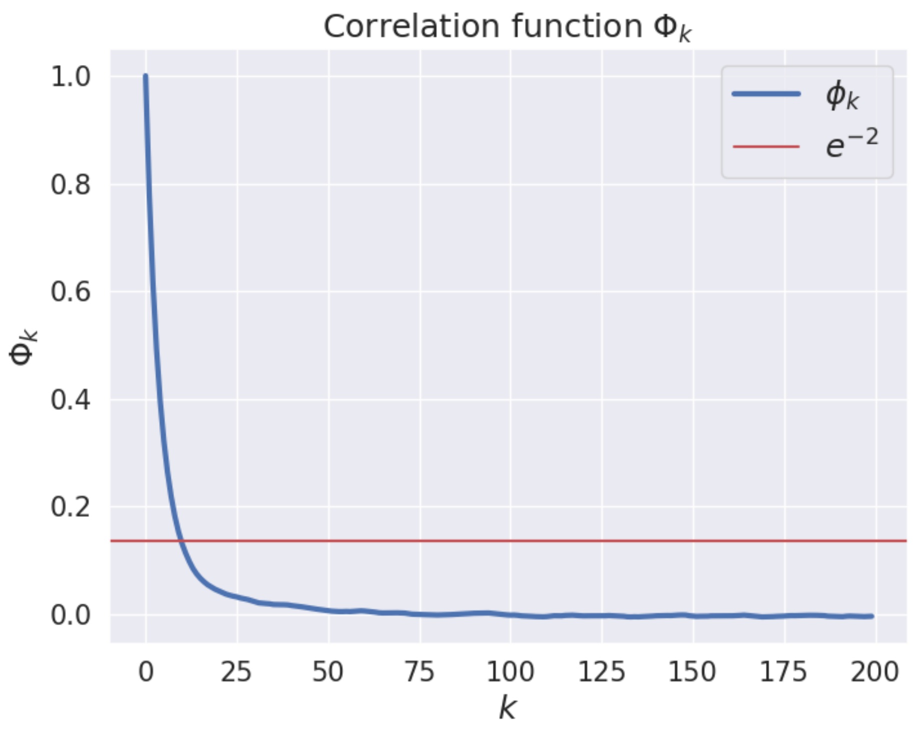


Figure 4: Correlation function Φ_k as a function of the lag.

The second method to calculate the statistical inefficiency is through block averaging. Here the total dataset consisting of N samples is subdivided into to N_B blocks each containing B samples and thus fulfilling $N = BN_B$. The average X_j of ε in each block i can be determined by

$$X_j = \frac{1}{B}\sum_{i=1}^{B} \varepsilon_{i+(j-1)B}, \quad j = 1, 2, ...N_B. \tag{22}$$

If the block size is smaller than n_s the averages are correlated and vice versa. Thus

$$\text{Var}[E] : \begin{cases} \text{Var}[E] = \frac{1}{N_B}\text{Var}[X], & B > n_s \\ \text{Var}[E] > \frac{1}{N_B}\text{Var}[X], & B < n_s \end{cases} \implies n_s = \lim_{B \text{ large}} \frac{B\,\text{Var}[X]}{\text{Var}[E]}.$$

Figure (5) shows a simulation where the statistical inefficiency is presented as a function of the block size and the value of n_s can be seen to converge after the block size 500. The figure also shows a comparison to the value of n_s obtained through evaluation of the correlation function. By calculating the average after the convergence of the block averaging the statistical inefficiency was calculated to $n_s = 11.204$ which is close to the value obtained through the correlation function.

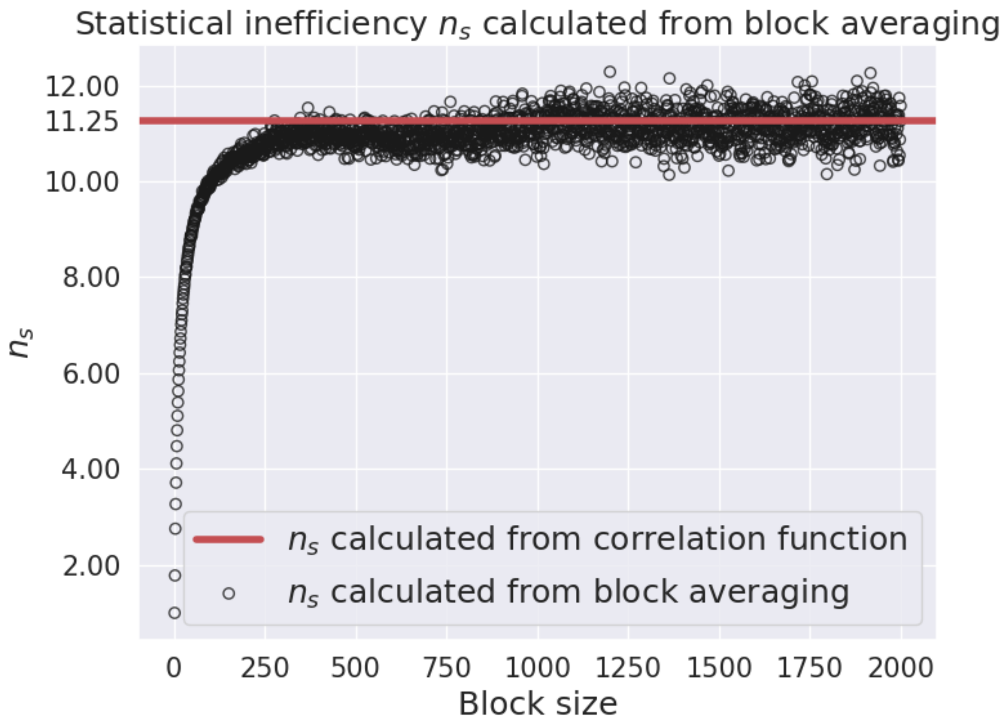


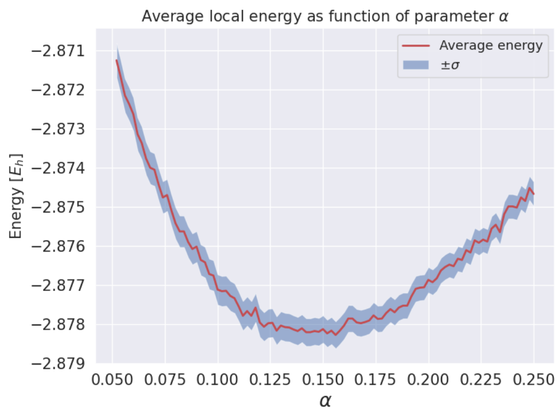
Figure 5: The statistical inefficiency n_s calculated through block averaging as a function of block size compared to the value obtained through the correlation function. The value of n_s obtained from block averaging can be seen to converge after the block size 500. It can be noted that $n_s = 11.248$ calculated from correlation function seems to differ from the value in the figure, 11.25, but the latter has merely been subject to a round off in the plotting process.

Problem 3

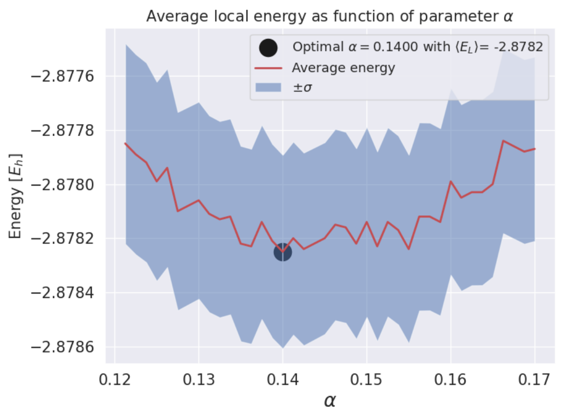
Since we are interested in determining the ground state energy of the helium atom the parameters used for the MCMC-sampling have to be optimized. The trial wavefunction from eq.(5) used for weighting the position coordinates aswell as the expression used to calculate the energy in eq.(6) depend on the parameter α and a value for this parameter can be found that minimizes the simulated energy. One way of determining the optimal α is through scanning the parameter space and determining what the average energy for each α is. To do this two separate scans where performed, an inintial less precise one to locate the area of the minima and a second one to determine it more precisely. The first scan used 100 values for α between 0.05 and 0.25. For each α $N_{\text{run}} = 30$ runs where made with $N_{eq} = 2 \cdot 10^4$ equilibration steps, $N = 10^7$ samples and displacement $d = 1.24$. The second scan was performed with 40 values for α between $[0.12, 1.17]$ with $N_{\text{runs}} = 50$. The standard deviation σ of the energy was calculated through

$$\sigma^2 = \frac{1}{N_{\text{runs}}} \sum_{i=1}^{N_{\text{runs}}} \frac{n_s}{N} \text{Var}[\varepsilon]_i \tag{23}$$

where $\text{Var}[\varepsilon]_i$ denotes the variance of the energy in each run and $n_s = 11.25$ is the statistical inefficiency as calculated in the previous problem. The result of the larger scan can be seen plotted in figure (6a) and the result of the tighter scan can be seen in figure (6b). The tighter scan indicates the optimal value for the parameter to be $\alpha = 0.14$ but since there are still some fluctuations left one cannot with certainty say that this is the true optima, but the optima but it should be close to this value.



(a) Scan of 100 parameter values for α between $[0.05, 0.250]$. The parameters used for this scan where $N = 10^7, N_{\text{run}} = 30, d = 1.24$.



(b) Scan of 40 parameter values for α between $[0.12, 0.17]$. The parameters used for this scan where $N = 10^7, N_{\text{run}} = 50, d = 1.24$.

Figure 6

Problem 4

Another way of the determining the optimal value for a parameter α is through optimization and in this project the variational Monte Carlo method was implemented. For this method the parameter value α was updated in the simulation according to the damped gradient descent method

$$\alpha_{p+1} = \alpha_p - \gamma_p \nabla_\alpha E(\alpha_p) \tag{24}$$

where p iteration of the optimization. The damping γ_p is calculated through

$$\gamma_p = A p^{-\beta} \tag{25}$$

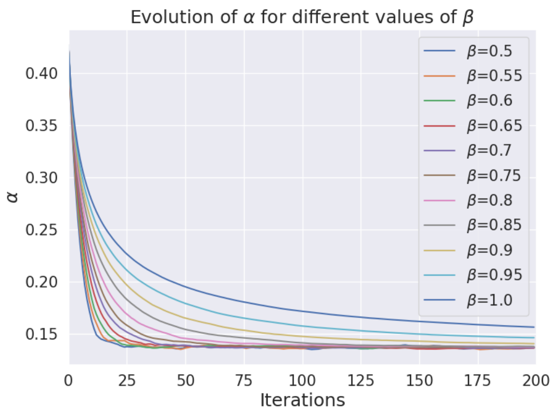
with $A = 1$ and β is a parameter regulating the strength of the dampening. The gradient $\nabla_\alpha E(\alpha_p)$ was calculated as

$$\nabla_\alpha E(\alpha_p) = 2[\langle E_L(\mathcal{R})\nabla_\alpha \ln(\psi_T(\mathcal{R}))\rangle - \langle E_L(\mathcal{R})\rangle\langle\nabla_\alpha \ln(\psi_T(\mathcal{R}))\rangle] \tag{26}$$

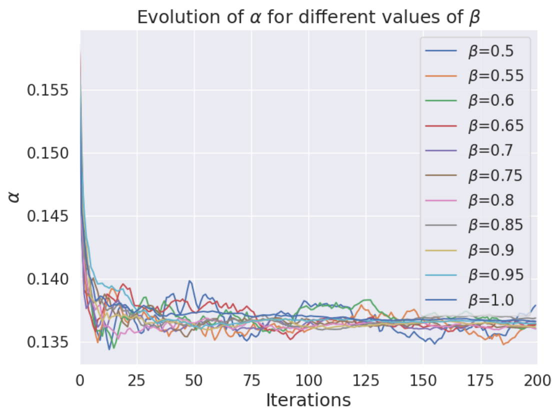
[2] and with the trial wavefunction from eq.(5) we have

$$\nabla_\alpha \ln(\psi_T(\mathcal{R})) = \frac{-r_{12}^2}{2(r_{12}\alpha + 1)^2}. \tag{27}$$

Since the convergence of the optimization depends on the parameter β different values where used. In figure (7b) the optimization was started at $\alpha = 0.5$ far away from the true optima α^* and from this figure it is clear that the optimizations using the larger values for β converges slower as they still have to converge after 200 iterations. Each iteration of optimization used $N_{\text{eq}} = 1e3$ equilibration steps, $N = 1e6$ samples and step size $d_t = 1.24$ When starting at a α closer to the optima all of the optimizers with different values manages to converge, this can be seen in figure (7a) where the optimizers start at $\alpha = 0.2$. Values for resulting α and average energy for the last MCMC-iteration can be seen in table (1).



(a) Evolution of parameter α from initial value $\alpha = 0.5$,



(b) Evolution of parameter α from initial $\alpha = 0.2$

Figure 7: Two measurements of convergence for the damped gradient descent method applied to the parameter α for different initial values, 0.5 and 0.2. The left figure converges slower but is used to showcase the exponential decay of α. In the figure to the right all values of β can be seem to converge and table 1 shows the value of α for the last MCMC-iteration and the corresponding average energy. All iterations were performed with $N = 1e6$ steps, $N_{eq} = 1e3$ discarded steps and an acceptance ratio of around 40 %.

Table 1: Table showing values of α and average energy for the last MCMC-simulation of a damped gradient descent simulation for different values of β.

Meas.	β	Resulting α	$\langle E_L\rangle\,[E_h]$ for resulting α
1	0.50	0.1379	-2.8795
2	0.55	0.1365	-2.8790
3	0.60	0.1364	-2.8787
4	0.65	0.1363	-2.8771
5	0.70	0.1366	-2.8800
6	0.75	0.1361	-2.8786
7	0.80	0.1360	-2.8773
8	0.85	0.1369	-2.8774
9	0.90	0.1362	-2.8773
10	0.95	0.1365	-2.8771
11	1.0	0.1366	-2.8788
Average:		0.1365 ± 0.0005	-2.8782 ± 0.0010

Problem 5

From the result in the previous two sections the optimal α was determined to be $\alpha = 0.1365$ and this value was then used for a more precise calculation of the ground state energy. For this simulation $N_{eq} = 1e6$ equilibration steps where used, along with $N = 10^7$ samples and 100 independent runs with different initial values where performed to obtain the average energy. The average energy of this simulation was $E_0 = -2.8782 \pm 3.59e{-4}\, E_h$. A figure showing the individual averages and variances aswell as results summarized as a boxplot can be seen in figure (8). The boxplots black arms and boxes can be read as dividing the measurements into 25% quantiles, and shows, in addition to the average energy, the highest and the lowest measured values of -2.877, and $-2.88\, E_h$ respectively. The boxplot also shows the measured standard distribution of $3.59e{-4}\, E_h$. The standard deviation was calculated according to equation(17), and it can be noted that the deviation is smaller than the deviation calculated using damped gradient descent, $1e{-3}\, E_h$. The difference could be said to be due to the values being calculated from 100 measurements here and 11 measurements for the damped gradient descent method.

It is interesting that the resulting ground state energy, $-2.8782 \pm 3.59e{-4}\, E_h$, is slightly lower than the Hartree value $E = -2.862\, E_h$. One reason could be that the Hartree value is calculated with the Hartree-Fock method, which uses an independent electron approximation [2] and this model fails to capture the effect of the electrons influencing each other. It is worth noting that by the variational theorem only the true wavefunction of a system will minimize the ground state energy and by neglecting the interactions between the electrons the wavefunction is less more an approximation than the trial function used in this project. This implies that the trial wavefunction in this project is closer to the true ground state of the helium atom rather than the simplified version using the independent electron approximation. The resulting value is also higher than the experimental value of $E_{\text{exp}} = -2.9033\, E_h$, which is also reasonable as the local energy, equation (6), merely provides an upper bound on the local energy, and is not constructed to give an exact value. The simulated result could only be as low as the experimental result for the energy if the trial wavefunction was the true wavefunction and this isn't the case as some approximations have been made to derive it.

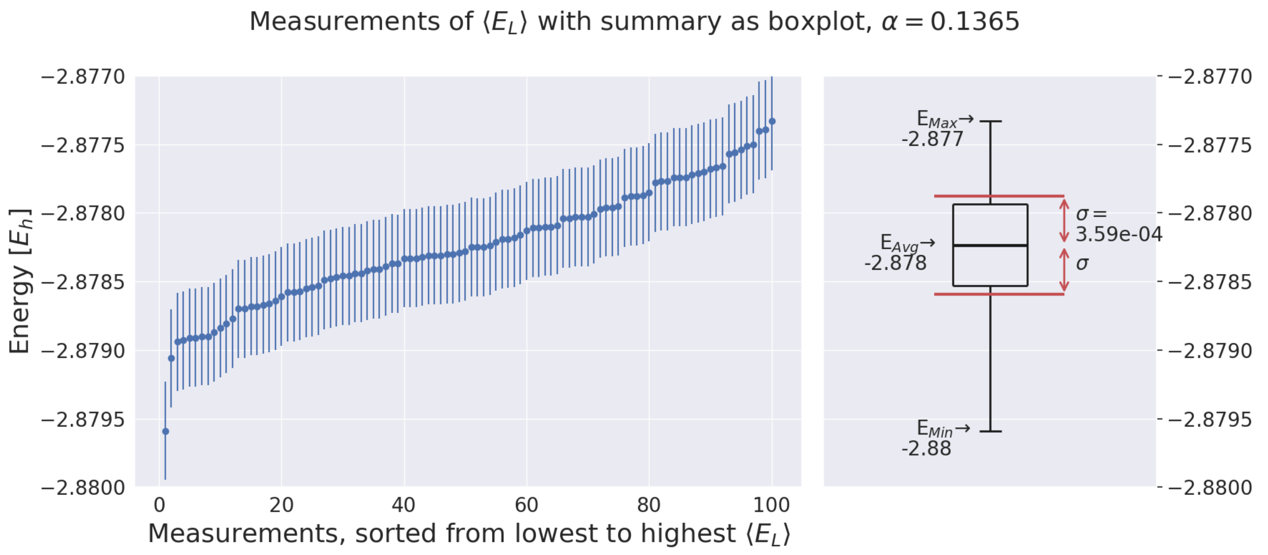


Figure 8: Multiple measurements of ground state energy for helium, with the resulting energy $-2.8782 \pm 3.59e{-4}\, E_h$. In the figure to the left, measurements have been sorted from lowest to highest energy. The figure on the right shows a boxplot where measured values has been divided into 25% quantiles, and shows the maximum, average and lowest measured energies in addition to the standard deviation. Each datapoint corresponds to an MCMC-sampling with $N = 1e7$ steps, $N_{eq} = 1e6$ equilibration steps and the particles were initiated with coordinates with $x, y, z \in [-25, 25]$. Acceptance ratios were consistently around 40 %.

Concluding Discussion

To conclude the Metropolis algorithm and Monte Carlo methods are powerful techniques that can greatly aid in the evaluation of high dimensional integrals that arise in many fields that would otherwise be too computationally costly to perform. Integrals like this often arise in the field of quantum mechanics and as demonstrated in this report the variational Monte Carlo method was able to give reasonable results for the ground state energy of the helium atom. The results for this project found that the ground state energy of the helium atom could be calculated to $-2.8782 \pm 3.59e-4\,E_h$ with an optimal value for the parameter in the trial wavefunction $\alpha = 0.1365 \pm 5e-4$. It was also determined that number of equilibration steps needed for the Metropolis algorithm should be $N_{\text{eq}} > 1e3$ and the statistical inefficiency was calculated to be $n_s = 11.48$. One may note that the quality of the results depend greatly on the trial wavefunction that is used and a more refined trial wavefunction that takes into account more of the physical processes that takes place in the real world system would give a result that more closely corresponds to experimental results. This can be shown from the result that the simulated energy from trial wavefunction used in this project matched the experimental values more closely compared to the Hartree value that used a wavefunction that neglected interactions between the two electrons. This shows us that the method is most powerful when used in combination with a model based in a solid understanding of the true physical processes that simulation tries to emulate.

References

[1] Göran Wahnström. Mc lecture notes, Oct 2021.

[2] Göran Wahnström. Qs lecture notes, Nov 2021.

[3] Göran Wahnström. H2b variational monte carlo, Oct 2021.

A C-code for simulations

A.1 Implementation of MCMC: `run.c`

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <time.h>
#include <unistd.h>
#include <limits.h>
#include <string.h>
#include <stdbool.h>

#include "tools.h"
#include "distribution.h"
#include "MCMC_chain_operations.h"
#include "statistical_ineff.h"
#include "restructured_stat_ineff.h"

#define NDIM 3
#define M_C 1000
#define RELAXATION_TIME 10

void initialize_positions(double **R1, double **R2, double d_displacement);
void MCMC_burn_in(int N_steps, double alpha, double d_displacement, double **R1, double **R2);
double MCMC(int N_steps, double alpha, double d_displacement, double **R1, double **R2, bool is_save);
double MCMC_task3(int N_steps,double alpha, double d_displacement,double **R1, double **R2, double *↩
     E_variance_vec,int index,bool is_save);




int
run(
    int argc,
    char *argv[]
   )
{
    bool is_task1 = false, is_task2 = false, is_task3 = false, is_task4 =  false;
    int task_num = 1;
    if (argc < 2) {
        task_num = 4;

    } else {
        task_num = atol(argv[1]);

        if (task_num > 5) {
            printf("Task should be between 1 and 5\n");
            exit(1);
        }
    }


    // MCMC Parameters
    int N_steps; int N_discarded_steps; double alpha, initial_displacement, d_displacement;
    // alpha Parameters
    int N_alpha_steps, N_beta_steps; double A, beta, E_average;
    bool is_save = true, is_vary_beta = false;


    if(task_num == 1)
    {
        //simulation parameters for task 1
        N_steps = 1e6; N_discarded_steps = 0; alpha = 0.1;
        initial_displacement = 1, d_displacement = 1.24;
        N_alpha_steps = 1; A = 0.; beta = 0.;
        is_task1 = true;
    }
    if(task_num == 2)
    {
        //simulation parameters for task 2
        N_steps = 5e3; N_discarded_steps = 0; alpha = 0.1;
        initial_displacement = 50, d_displacement = 1.24;
        N_alpha_steps = 1; A = 0.; beta = 0.;
        is_task2 = true;
    }
    if(task_num == 3)
    {
        //simulation parameters for task 3
        N_steps = 1e7; N_discarded_steps = 2*1e4; alpha = 0.05;
        initial_displacement = 50, d_displacement = 1.24;
        N_alpha_steps = 1; A = 0.; beta = 0.;
        is_task3 = true;
    }
    if(task_num == 4)
    {
        //simulation parameters for task 4
        N_steps = 1e6; N_discarded_steps = 1e3; alpha = 0.5;
        initial_displacement = 50, d_displacement = 1.24;
        N_alpha_steps = 200; A = 1.; beta = 1.; is_save = false; // beta from 0.5 to 1
        is_vary_beta = true;
        is_task4 = true;
        printf("Hej!");
    }

    //position arrays and variables for storing data
    double **R1 = create_2D_array(N_steps, NDIM), **R2 = create_2D_array(N_steps, NDIM);
    double E_PD_average;
    double *E_local_derivative = malloc(sizeof(double) * N_steps);

    char filename_alpha_results[200], filename_params[200], filename_variance_alpha[200],filename_of_alpha[200];
    char cwd[200], buf[200];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Current working directory %s\n", cwd);
    } else {
        perror("getcwd() error");
        return 1;
    }

    //filenames for saving data
    int written = snprintf(buf, 200, "%s", cwd);
    snprintf(buf + written, 200 - written, "/csv/alpha_results.csv");
    strcpy(filename_alpha_results, buf);
    snprintf(buf + written, 200 - written, "/csv/params.csv");
    strcpy(filename_params, buf);

    snprintf(buf + written, 200 - written, "/csv/variance_alpha.csv");
    strcpy(filename_variance_alpha, buf);
    snprintf(buf + written, 200 - written, "/csv/E_of_alpha.csv");
    strcpy(filename_of_alpha, buf);




    bool open_with_write;

    initialize_positions((double **) R1, (double **) R2, (double) initial_displacement);

    if(is_task2)
    {
        for (int kx = 0; kx < NDIM; kx++)
        {
            R1[0][kx] = initial_displacement * 0.5;
            R2[0][kx] = initial_displacement * (-0.5);
        }
    }

    if(is_task1 || is_task2)
    {
        E_average = 0;
        E_average = MCMC(N_steps, alpha, d_displacement, R1, R2, is_save);

        double param_vector[] = {N_alpha_steps, N_discarded_steps, alpha, A, beta, N_steps, d_displacement, ↩
             is_task1, is_task2, is_task3, is_task4};

        save_vector_to_csv(param_vector, 11, filename_params, true);
        destroy_2D_array(R1, N_steps); destroy_2D_array(R2, N_steps);
        free(E_local_derivative);
    }

    if(is_task3)
    {
        int Number_of_alphas = 40;
        int runs_per_alpha = 50;

        double *temp_E_vec = malloc(sizeof(double)*runs_per_alpha);
        double *temp_variance_vec = malloc(sizeof(double)*runs_per_alpha);
        double *E_average_vec= malloc(sizeof(double)*Number_of_alphas);
        double *E_variance_vec= malloc(sizeof(double)*Number_of_alphas);
        double start_alpha = 0.12, final_alpha= 0.17;
        double alpha_increment = (final_alpha-start_alpha)/((double)Number_of_alphas);
        is_save = false;

        for(int n_alpha=0; n_alpha< Number_of_alphas; ++n_alpha)
        {
            alpha = (double)n_alpha*alpha_increment + start_alpha;
            printf("current alpha = %f\n", alpha);
            for(int run=0; run< runs_per_alpha; ++run)
            {
                //calculating average  energy for separate runs along with variance
                //in each run
                initialize_positions((double **) R1, (double **) R2, (double) d_displacement);
                MCMC_burn_in(N_discarded_steps, alpha, d_displacement, R1, R2);
                temp_E_vec[run] = MCMC_task3(N_steps, alpha, d_displacement, R1, R2,temp_variance_vec,run, ↩
                     is_save);
                printf("average E in run =%f\n",temp_E_vec[run]);

            }

            double average_E =0;
            average_E= average(temp_E_vec, runs_per_alpha);
            double variance_between_runs =0;
            double variance_in_run=0;

            variance_between_runs=variance(temp_E_vec, runs_per_alpha);
            variance_in_run = average(temp_variance_vec, runs_per_alpha);
            E_average_vec[n_alpha] = average_E;
            E_variance_vec[n_alpha]=variance_between_runs+ variance_in_run;
            printf("runs left =%d\n", Number_of_alphas- n_alpha);
        }

        save_vector_to_csv(E_average_vec,Number_of_alphas,filename_of_alpha, true);
        save_vector_to_csv(E_variance_vec, Number_of_alphas,filename_variance_alpha, true);
        free(E_average_vec),free(E_variance_vec), free(temp_E_vec), free(temp_variance_vec);

        return 0;
    }

    if(is_task4)
    {
        E_average = 0;
        if(is_vary_beta)
        {
            N_beta_steps = 11;
        } else {
            N_beta_steps = 1;
        }

        for(int bx = 0; bx < N_beta_steps; bx++)
        {
            if(is_vary_beta)
            {
                //looping over different beta values for optimization
                double beta_array[11] = {0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1};
                beta = beta_array[bx];
                alpha = 0.2;
            }

            for(int ix = 1; ix < N_alpha_steps + 1; ix++)
            {
                //performing optimization
                initialize_positions((double **) R1, (double **) R2, (double) initial_displacement);
                MCMC_burn_in(N_discarded_steps, alpha, d_displacement, R1, R2);


                if (ix == N_alpha_steps && bx == N_beta_steps) {is_save = true; }
                E_average = MCMC(N_steps, alpha, d_displacement, R1, R2, is_save);

                E_PD_average = partialEnergyDerivative(E_local_derivative, alpha, N_steps, R1, R2);

                double gamma = 0;
                if(is_task3 || is_task4)
                {
                    gamma = A * pow(ix, (double) - beta);
                }

                alpha -= gamma * E_PD_average;

                double alpha_result_vector[] = {ix, E_average, alpha, gamma, E_PD_average, beta};
                if(ix == 1 && bx == 0){ open_with_write = true; } else { open_with_write = false; }
                save_vector_to_csv(alpha_result_vector, 6, filename_alpha_results, open_with_write);
                printf("Alpha iteration: %d\n", ix);
```

```c
            }
            printf("Beta iteration: %d\n", bx);
        }


        double param_vector[] = {N_alpha_steps, N_discarded_steps, alpha, A, beta, N_steps, d_displacement, ↩
             is_task1, is_task2, is_task3, is_task4};
        save_vector_to_csv(param_vector, 11, filename_params, true);
        destroy_2D_array(R1, N_steps); destroy_2D_array(R2, N_steps);
        free(E_local_derivative);

        return 0;
    }


}
// Function running the markov-chain
void initialize_positions(double **R1, double **R2, double initial_displacement)
{
    double random_number = 0;
    gsl_rng * r;
    r = init_random_num_generator();

    //printf("Initializing random positions:\n");
    for (int kx = 0; kx < NDIM; kx++)
    {
        // -0.5 to 0.5 because it's length 1 and symmetric around zero
        random_number = gsl_ran_flat(r, -0.5, 0.5);
        R1[0][kx] = initial_displacement * random_number;
        //printf("random number: %f\n", random_number);
        //printf("d_displacement: %f\n", initial_displacement);


        random_number = gsl_ran_flat(r, -0.5, 0.5);
        R2[0][kx] = initial_displacement * random_number;
        //printf("R1[0][%d]: %f, R2[0][%d]: %f\n", kx, R1[0][kx], kx, R2[0][kx]);
    }
    gsl_rng_free(r);
}

//Function for MCMC-burn in
void MCMC_burn_in(int N_steps, double alpha, double d_displacement, double **R1, double **R2)
{
    double R1_test[NDIM], R2_test[NDIM];

    gsl_rng * r;
    r = init_random_num_generator();
    double random_number = 0;

    int accept_count = 0;
    for(int ix = 0; ix < N_steps - 1; ++ix)
    {
        // get proposal positons
        for (int kx = 0; kx < NDIM; ++kx)
        {
            random_number = gsl_ran_flat(r, -0.5,0.5);
            R1_test[kx] = R1[ix][kx] + d_displacement * random_number;
            random_number = gsl_ran_flat(r,-0.5,0.5);
            R2_test[kx] = R2[ix][kx] + d_displacement * random_number;
        }

        // Probability for particle occupying new and old positions
        double prob_test = distribution(R1_test, R2_test, alpha);
        double prob_old = distribution(R1[ix], R2[ix], alpha);

        // If new prob > old, make step OR take exploration step
        if(prob_test / prob_old > gsl_ran_flat(r, 0.0, 1.0))
        {
            // If accepted, save new position in next row.
            for (int kx=0; kx < NDIM; ++kx)
            {
                R1[ix+1][kx] = R1_test[kx];
                R2[ix+1][kx] = R2_test[kx];
            }

            accept_count = accept_count + 1;
        } else {
            // If not accepted save old position in next row
            for (int kx=0; kx < NDIM; ++kx)
            {
                R1[ix+1][kx] = R1[ix][kx];
                R2[ix+1][kx] = R2[ix][kx];
            }
        }
    }
    gsl_rng_free(r);
}

//Function for performing MCMC
double MCMC(int N_steps, double alpha, double d_displacement, double **R1, double **R2, bool is_save)
{
    char filename_R1[200], filename_R2[200], filename_energy[200], filename_xdist[200], filename_theta[200], \
     filename_energy_derivative[200], filename_results[200], filename_phi_k[200], filename_block_avg[200];

    char cwd[200], buf[200];
    if (getcwd(cwd, sizeof(cwd)) == NULL) {
        perror("getcwd() error");
        return 1;
    }

    int written = snprintf(buf, 200, "%s", cwd);
    snprintf(buf + written, 200 - written, "/csv/R1.csv");
    strcpy(filename_R1, buf);
    snprintf(buf + written, 200 - written, "/csv/R2.csv");
    strcpy(filename_R2, buf);
    snprintf(buf + written, 200 - written, "/csv/E_local.csv");
    strcpy(filename_energy, buf);
    snprintf(buf + written, 200 - written, "/csv/x_distribution.csv");
    strcpy(filename_xdist, buf);
    snprintf(buf + written, 200 - written, "/csv/theta.csv");
    strcpy(filename_theta, buf);
    snprintf(buf + written, 200 - written, "/csv/E_local_derivative.csv");
    strcpy(filename_energy_derivative, buf);
    snprintf(buf + written, 200 - written, "/csv/filename_results.csv");
    strcpy(filename_results, buf);
    snprintf(buf + written, 200 - written, "/csv/phi_k.csv");
    strcpy(filename_phi_k, buf);
    snprintf(buf + written, 200 - written, "/csv/block_avg_vec.csv");
    strcpy(filename_block_avg, buf);

    bool open_with_write;

     // Initializing arrays
    int n_phi_rows = 2*M_C+10;
    //int number_of_blocks = 100;
    double *E_local = malloc(sizeof(double) * N_steps);
    double *E_local_derivative = malloc(sizeof(double) * N_steps);
    double *Phi_k_vec = malloc(sizeof(double) *n_phi_rows);
    double *theta_chain = malloc(sizeof(double) * N_steps);
    double *x_chain = malloc(sizeof(double) * N_steps);

    // testing corr func
    int max_lag = 2*1e2;
    double phi_k=0;
    double *phi_k_vec = malloc(sizeof(double)*max_lag);

    // testing block averaging
    int max_block_size = 3000;
    double *block_average_vec = malloc(sizeof(double) *max_block_size);


    double R1_test[NDIM], R2_test[NDIM];

    gsl_rng * r;
    r = init_random_num_generator();
    double random_number = 0;

    int accept_count = 0;
    for(int ix = 0; ix < N_steps - 1; ++ix)
    {
        // get proposal positons
        for (int kx = 0; kx < NDIM; ++kx)
        {
            random_number = gsl_ran_flat(r, -0.5, 0.5);
            R1_test[kx] = R1[ix][kx] + d_displacement * random_number;
            random_number = gsl_ran_flat(r, -0.5, 0.5);
            R2_test[kx] = R2[ix][kx] + d_displacement * random_number;
        }

        // Probability for particle occupying new and old positions
        double prob_test = distribution(R1_test, R2_test, alpha);
        double prob_old = distribution(R1[ix], R2[ix], alpha);

        // If new prob > old, make step OR take exploration step
        if(prob_test / prob_old > gsl_ran_flat(r, 0.0, 1.0))
        {
            // If accepted, save new position in next row.
            for (int kx=0; kx < NDIM; ++kx)
            {
                R1[ix+1][kx] = R1_test[kx];
                R2[ix+1][kx] = R2_test[kx];
            }

            accept_count = accept_count + 1;
        } else {
            // If not accepted save old position in next row
            for (int kx=0; kx < NDIM; ++kx)
            {
                R1[ix+1][kx] = R1[ix][kx];
                R2[ix+1][kx] = R2[ix][kx];
            }
        }

        if(is_save)
        {
            double theta_ix = theta_fun_vec(R1[ix], R2[ix]);
            double x_cos = cos(theta_ix);
            theta_chain[ix] = theta_ix;
        }


    }

    // Calculate energies of all positions in chain
    Energy(E_local, alpha, N_steps, R1, R2);


    double average_E_local = 0;
    for(int ix = 0; ix < N_steps - 1; ++ix)
    {
        average_E_local += E_local[ix]/N_steps;
    }

    if(is_save)
    {
        printf("Accept ratio = %f\n", (double) accept_count/N_steps);

        //testing corrolation function
        for (int lag=0; lag< max_lag; ++lag)
        {
            double phi_inst =phi_lag(E_local, N_steps, lag);
            phi_k_vec[lag] = phi_inst;
            //printf("phi_k=%f\n", phi_inst);
        }

        //testing block averaging
        for (int block_size =1; block_size<max_block_size; ++block_size)
        {
            block_average_vec[block_size] = statistical_ineff_from_BLAV(E_local, N_steps, block_size);
        }


        double E_PD_average = partialEnergyDerivative(E_local_derivative, alpha, N_steps, R1, R2);


        x_distribution(x_chain, N_steps, R1,R2);
        save_matrix_to_csv(R1, N_steps, NDIM, filename_R1);
        save_matrix_to_csv(R2, N_steps, NDIM, filename_R2);

        double result_vec[] = {N_steps, accept_count};
        open_with_write = true;
        save_vector_to_csv(result_vec, 2, filename_results, open_with_write);
        save_transposedvector_to_csv(E_local_derivative, N_steps, filename_energy_derivative, open_with_write);
        save_transposedvector_to_csv(E_local, N_steps, filename_energy, open_with_write);
        save_transposedvector_to_csv(x_chain, N_steps, filename_xdist, open_with_write);
        save_transposedvector_to_csv(theta_chain, N_steps, filename_theta, open_with_write);
        save_transposedvector_to_csv(phi_k_vec, max_lag, filename_phi_k, open_with_write);
        save_transposedvector_to_csv(block_average_vec, max_block_size, filename_block_avg, open_with_write);
    }
    // Destroy and free arrays
    free(E_local), free(E_local_derivative), free(x_chain), free(theta_chain), free(Phi_k_vec), free(↩
        block_average_vec);
    gsl_rng_free(r);

    return average_E_local;
}
```

```


//Function for performing MCMC and saving variance
double MCMC_task3(
    int N_steps,
    double alpha,
    double d_displacement,
    double **R1, double **R2,
    double *E_variance_vec,
    int index,
    bool is_save)
{
    char filename_R1[200], filename_R2[200], filename_energy[200], filename_xdist[200], filename_theta[200], \
     filename_energy_derivative[200], filename_results[200], filename_phi_k[200], filename_block_avg[200];

    char cwd[200], buf[200];
    if (getcwd(cwd, sizeof(cwd)) == NULL) {
        perror("getcwd() error");
        return 1;
    }

    int written = snprintf(buf, 200, "%s", cwd);
    snprintf(buf + written, 200 - written, "/csv/R1.csv");
    strcpy(filename_R1, buf);
    snprintf(buf + written, 200 - written, "/csv/R2.csv");
    strcpy(filename_R2, buf);
    snprintf(buf + written, 200 - written, "/csv/E_local.csv");
    strcpy(filename_energy, buf);
    snprintf(buf + written, 200 - written, "/csv/x_distribution.csv");
    strcpy(filename_xdist, buf);
    snprintf(buf + written, 200 - written, "/csv/theta.csv");
    strcpy(filename_theta, buf);
    snprintf(buf + written, 200 - written, "/csv/E_local_derivative.csv");
    strcpy(filename_energy_derivative, buf);
    snprintf(buf + written, 200 - written, "/csv/filename_results.csv");
    strcpy(filename_results, buf);
    snprintf(buf + written, 200 - written, "/csv/phi_k.csv");
    strcpy(filename_phi_k, buf);
    snprintf(buf + written, 200 - written, "/csv/block_avg_vec.csv");
    strcpy(filename_block_avg, buf);

    bool open_with_write;

     // Initializing arrays
    int n_phi_rows = 2*M_C+10;
    //int number_of_blocks = 100
    double *E_local = malloc(sizeof(double) * N_steps);
    double *E_local_derivative = malloc(sizeof(double) * N_steps);
    double *Phi_k_vec = malloc(sizeof(double) *n_phi_rows);
    double *theta_chain = malloc(sizeof(double) * N_steps);
    double *x_chain = malloc(sizeof(double) * N_steps);

    double R1_test[NDIM], R2_test[NDIM];

    gsl_rng * r;
    r = init_random_num_generator();
    double random_number = 0;

    int accept_count = 0;
    for(int ix = 0; ix < N_steps - 1; ++ix)
    {
        // get proposal positons
        for (int kx = 0; kx < NDIM; ++kx)
        {
            random_number = gsl_ran_flat(r, -0.5, 0.5);
            R1_test[kx] = R1[ix][kx] + d_displacement * random_number;
            random_number = gsl_ran_flat(r, -0.5, 0.5);
            R2_test[kx] = R2[ix][kx] + d_displacement * random_number;
        }

        // Probability for particle occupying new and old positions
        double prob_test = distribution(R1_test, R2_test, alpha);
        double prob_old = distribution(R1[ix], R2[ix], alpha);

        // If new prob > old, make step OR take exploration step
        if(prob_test / prob_old > gsl_ran_flat(r, 0.0, 1.0))
        {
            // If accepted, save new position in next row.
            for (int kx=0; kx < NDIM; ++kx)
            {
                R1[ix+1][kx] = R1_test[kx];
                R2[ix+1][kx] = R2_test[kx];
            }

            accept_count = accept_count + 1;
        } else {
            // If not accepted save old position in next row
            for (int kx=0; kx < NDIM; ++kx)
            {
                R1[ix+1][kx] = R1[ix][kx];
                R2[ix+1][kx] = R2[ix][kx];
            }
        }

        double theta_ix = theta_fun_vec(R1[ix], R2[ix]);
        double x_cos = cos(theta_ix);
        theta_chain[ix] = theta_ix;

    }

    // Calculate energies of all positions in chain
    Energy(E_local, alpha, N_steps, R1, R2);


    double average_E_local = 0;
    for(int ix = 0; ix < N_steps - 1; ++ix)
    {
        average_E_local += E_local[ix]/N_steps;
    }
    double variance_E = variance(E_local,N_steps);
    double statistical_inefficiency = 11;
    E_variance_vec[index] = statistical_inefficiency* variance_E;
    if(is_save)
    {
        printf("Accept ratio = %f\n", (double) accept_count/N_steps);

        x_distribution(x_chain, N_steps, R1,R2);
        save_matrix_to_csv(R1, N_steps, NDIM, filename_R1);
        save_matrix_to_csv(R2, N_steps, NDIM, filename_R2);

        double result_vec[] = {N_steps, accept_count};
        open_with_write = true;
        save_vector_to_csv(result_vec, 2, filename_results, open_with_write);
        save_transposedvector_to_csv(E_local_derivative, N_steps, filename_energy_derivative, open_with_write);
        save_transposedvector_to_csv(E_local, N_steps, filename_energy, open_with_write);
        save_transposedvector_to_csv(x_chain, N_steps, filename_xdist, open_with_write);
        save_transposedvector_to_csv(theta_chain, N_steps, filename_theta, open_with_write);

    }
    // Destroy and free arrays
    free(E_local), free(E_local_derivative), free(x_chain), free(theta_chain), free(Phi_k_vec);//, free(↩
        block_average_vec);
    gsl_rng_free(r);

    return average_E_local;
}
```

A.2 Sampled distribution: distribution.c

```
//
// Created by didri on 2022-11-28.
//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <time.h>

#include "tools.h"


/*
function that calculates the probability of a configuration of electron positions
args:
    double *R1 = position vector for electrion 1 [x,y,z]
    double *R2 = position vector for electrion 2 [x,y,z]
    double alpha = parameter value
returns: psi*psi = probability of positions
```

```c

*/
double distribution(double *R1, double *R2, double alpha){

    double r1 = vector_norm(R1,3);
    double r2 = vector_norm(R2, 3);
    double r12 = distance_between_vectors(R1, R2, 3);
    double psi = exp(-2.0 * r1) * exp(-2.0 * r2) * exp(r12 / (2.0 * (1.0 + alpha * r12)));

    return psi * psi;
}
```

A.3 Code to perform operations on MCMC chains: `MCMC_chain_operations.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <time.h>

#include "tools.h"
//number of spacial dimensions
#define NDIM 3

/*Function that calculates energy at each step of MCMC-chain from two position vectors at each step
    args:
        double *E_local = N_steps long array to save values of energy in
        double alpha = parameter value
        int N_steps = number of steps in mcmc-chain
        double **R1 = 2D-array [3][N_steps] that saves x,y,z for electron 1 in each step of mcmc-chain
        double **R2 = 2D-array [3][N_steps] that saves x,y,z for electron 2 in each step of mcmc-chain
*/

void Energy(double *E_local, double alpha, int N_steps, double **R1, double **R2){

    double r12;
    double *r1_nrm = malloc(sizeof(double) * NDIM), *r2_nrm = malloc(sizeof(double) * NDIM);
    double *diff_vec = malloc(sizeof(double) * NDIM), *diff_nrm = malloc(sizeof(double) * NDIM);
    double prod = 0., div = 0.;

    for (int ix = 0; ix < N_steps; ++ix){
        for (int dim=0; dim<NDIM; ++dim){
            r1_nrm[dim] = R1[ix][dim];
            r2_nrm[dim] = R2[ix][dim];
        }

        normalize_vector(r1_nrm, NDIM);
        normalize_vector(r2_nrm, NDIM);
        r12 = distance_between_vectors(R1[ix], R2[ix], NDIM);

        elementwise_subtraction(diff_vec, R1[ix], R2[ix], NDIM);
        elementwise_subtraction(diff_nrm, r1_nrm, r2_nrm, NDIM);
        prod = dot_product(diff_vec, diff_nrm,NDIM);
        div = (1. + alpha * r12);

        E_local[ix] = - 4.0 \
                    + prod/(r12 * pow(div, 2.0)) \
                    - 1.0/(r12* pow(div, 3.0)) \
                    - 1./(4.0* pow(div,4.0)) \
                    + 1. / r12;

    }
    free(r1_nrm), free(r2_nrm), free(diff_nrm), free(diff_vec);
}

/*Function that calculates gradient with respect to parameter alpha
for performing damped gradient descent.
    args:
        double *E_local_derivative = N_steps long array to save values of derivative
        double alpha = parameter value
        int N_steps = number of steps in mcmc-chain
        double **R1 = 2D-array [3][N_steps] that saves x,y,z for electron 1 in each step of mcmc-chain
        double **R2 = 2D-array [3][N_steps] that saves x,y,z for electron 2 in each step of mcmc-chain
    returns:
       double gradient = Gradient for performing damped gradient descent
*/
```

```


double partialEnergyDerivative(double *E_local_derivative, double alpha, int N_steps, double **R1, double **R2)
{
    double *E_local_chain = malloc(sizeof(double)*N_steps);
    double r12=0, ln_d_psi=0;
    double average_E=0, average_derivative =0, average_mix=0;
    double gradient=0, gradient_step = 0;

    Energy(E_local_chain, alpha, N_steps, R1, R2);

    for(int step=0; step<N_steps; ++step)
    {
        r12 = distance_between_vectors(R1[step], R2[step], NDIM);
        ln_d_psi = - r12*r12/(2*pow((r12*alpha +1),2));
        average_derivative += ln_d_psi;
        average_mix += ln_d_psi*E_local_chain[step];
        average_E += E_local_chain[step];
        gradient_step = E_local_chain[step] * ln_d_psi;
        E_local_derivative[step] = gradient_step;
    }
    average_derivative /= N_steps;
    average_mix /= N_steps;
    average_E /= N_steps;

    gradient = 2*(average_mix - average_E*average_derivative);

    free(E_local_chain);

    return gradient;
}


/*function calculating distribution of x from two position mcmc chains
 * args:
 *      x_chain = array of length N_steps to save distribution in
 *      N_steps = number of steps in mcmc chain
 *      R1_chain = mcmc chain for particle 1, NX3 matrix
 *      R2_chain = mcmc chain for particle 1, NX3 matrix
 */
void x_distribution(double *x_chain, int N_steps, double **R1_chain, double **R2_chain){

    // initializing values used to calculate instance of x
    double length_r1=0, length_r2=0, dot_prod=0;

    // stepping through mcmc chain calculating x at every step and saving in x_chain
    for(int step=0; step<N_steps; ++step)
    {
        length_r1 = vector_norm(R1_chain[step], NDIM);
        length_r2 = vector_norm(R2_chain[step], NDIM);
        dot_prod = dot_product(R1_chain[step], R1_chain[step], NDIM);
        x_chain[step] = dot_prod/(length_r2*length_r1);
    }
}

/*
function that calculates the angle between arrays along entire mcmc-chain
 args:
        double *theta_chain = N_steps long array to save values of angle in
        double alpha = parameter value
        int N_steps = number of steps in mcmc-chain
        double **R1 = 2D-array [3][N_steps] that saves x,y,z for electron 1 in each step of mcmc-chain
        double **R2 = 2D-array [3][N_steps] that saves x,y,z for electron 2 in each step of mcmc-chain
*/
void theta_fun(double *theta_chain, int N_steps, double **R1_chain, double **R2_chain)
{
    // initializing values used to calculate instance of x
    double length_R1=0, length_R2=0, dot_prod=0;
    for(int step = 0; step < N_steps; ++step)
    {
        length_R1 = vector_norm(R1_chain[step], NDIM);
        length_R2 = vector_norm(R2_chain[step], NDIM);
        dot_prod = dot_product(R1_chain[step], R1_chain[step], NDIM);
        theta_chain[step] = acos( dot_prod / (length_R1 * length_R2));
    }
}

/*
function that calculates angle between position vectors for electron
    args:
        double *R1 = position array for electron 1 [x,y,z]
        double *R2 = position array for electron 2 [x,y,z]
    returns:
        double theta = angle between the position vectors of the two electrons
*/
double theta_fun_vec(double *R1, double *R2)
{
    double R1_abs = vector_norm(R1,NDIM);
    double R2_abs = vector_norm(R2,NDIM);
    double R1_R2_dot = dot_product(R1, R2, NDIM);
    double theta = acos( R1_R2_dot / (R1_abs * R2_abs));

    return theta;
}
```

A.4 Functions used to calculate statistical inefficiency: `restructured_stat_ineff.c`

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <time.h>

#include "tools.h"


/*Function that takes in MCMC-chain for energy and calculates correlation function for a lagtime
    args:
        double *E_local_chain = 1D-array with length N_steps, is the mcmc-chain of the energy
        int N_steps = Number for steps in MCMC-chain
        int Lag = lag time for calculating correlation function.

    returns:
        phi_k the evaluated correlation function
*/
double phi_lag(double *E_local_chain, int N_steps, int Lag)
{
    //Initializing variables used in calculations
    double phi_k = 0, average_E_local = 0, average_squared_E_local=0, lagged_average=0;
    int lower_buffer =Lag, buffer_upper = N_steps-Lag;

    //calculating average energy in chain
    for(int step=0; step<N_steps; ++step)
    {
        double E_sample = E_local_chain[step];
        average_E_local += E_sample;
        average_squared_E_local += E_sample*E_sample;
    }

    //normalizing average
    average_E_local /=N_steps; average_squared_E_local /=N_steps;

    //calculating lagged average for a specified lag time
    for(int step=0; step< buffer_upper; ++step)
    {
        lagged_average += E_local_chain[step]*E_local_chain[step+Lag];
    }
    //normalizing lagged average
    lagged_average/=(N_steps-abs(Lag));

    //calculates correlation function phi_k
    phi_k = (lagged_average -average_E_local*average_E_local)/(average_squared_E_local-average_E_local*↩
        average_E_local);

    return phi_k;
}

/* Function that calculates statistical inneficieny from block averaging for a given block size

    args:
        double *E_local_chain = 1D-array with length N_steps, is the mcmc-chain of the energy
        int N_steps = Number for steps in MCMC-chain
        int Block_size = block size for calculating statistical inneficiency.

    returns:
        statistical_inneficiency= statistical inneficiency for a given block size
*/
```

```c
double statistical_ineff_from_BLAV(double *E_local_vec, int N_steps, int Block_size)
{
    //initializing variables used in calculations
    int number_of_blocks = N_steps/Block_size;
    double average_i = 0, variance_block=0, variance_E_local=0, statistical_inneficiency;
    double average_tot =0;
    // array to save averages for each block
    double *block_vec = malloc(sizeof(double)*number_of_blocks);


    //looping through the blocks
    for(int block=0; block<number_of_blocks; ++block)
    {
        //calculate average inside a block
        for(int step =0; step< Block_size; ++step)
        {
            block_vec[block] += E_local_vec[Block_size*(block) +step];

        }
        //normalizes the average inside block
        block_vec[block] /=Block_size;
    }
    //calculates variance of block averages aswell as total variance of mcmc-chain
    variance_block = variance(block_vec,number_of_blocks);
    variance_E_local = variance(E_local_vec,N_steps);

    //calculates statistical inneficiency
    statistical_inneficiency = (variance_block*Block_size)/variance_E_local;
    free(block_vec);
    return statistical_inneficiency;
}
```

A.5 tool functions: `tools.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <time.h>

#include "tools.h"

void
elementwise_addition(
        double *res,
        double *v1,
        double *v2,
        unsigned int len
)
{
    for(int ix=0; ix < len; ix++)
    {
        res[ix] = v1[ix] + v2[ix];
    }
}

void
elementwise_subtraction(
        double *res,
        double *v1,
        double *v2,
        unsigned int len
)
{
    for(int ix=0; ix < len; ix++)
    {
        res[ix] = v1[ix] - v2[ix];
    }
}

void
elementwise_multiplication(
        double *res,
        double *v1,
        double *v2,
        unsigned int len
```

```
)
{
    for(int ix = 0; ix < len; ix++)
    {
        res[ix] = v1[ix] * v2[ix];
    }
}

double
dot_product(
        double *v1,
        double *v2,
        unsigned int len
)
{
    double result = 0;
    for(int ix = 0; ix < len; ix++)
    {
        result += v1[ix] * v2[ix];
    }
    return result;
}

//New from Elsa
double**
create_2D_array(
        unsigned int row_size,
        unsigned int column_size
)
{

    double** array = (double**)calloc(row_size, sizeof(double*));
    for (int i = 0; i < row_size; ++i) {
        array[i] = malloc(column_size * sizeof(double));
    }
    return array;
}

void
destroy_2D_array_pointers(
        double **array
){
    free(*array);
    free(array);
}

void
destroy_2D_array(
        double **array,
        int n_rows
){
    for(int ix = 0; ix < n_rows; ix++){
        free(array[ix]);
    }
    free(array);
}

void
print_2D_array(
        double **array,
        unsigned int row_size,
        unsigned int column_size // Carl: Changed place between row_size and column_size
)
{
    for(int ix = 0; ix < row_size; ix++){
        for(int jx = 0; jx < column_size; jx++){
            if( ix % 1000 == 0){
                printf("%f ", array[ix][jx]);
            }
        }
        printf("\n");
    }
}

void
matrix_multiplication(
        double **result,
        double **v1,
        double **v2,
        unsigned int m,
        unsigned int n,
        unsigned int p
```

```c
)
{
    for(int ix = 0; ix < m; ix++)
    {
        for(int jx = 0; jx < p; jx++)
        {
            for(int kx = 0; kx < n; kx++)
            {
                result[ix][jx] += v1[ix][kx] * v2[kx][jx];
            }
        }
    }
}

double
vector_norm(
        double *v1,
        unsigned int len
)
{
    double result = 0;
    // Euclidian L2 norm
    for(int ix =0; ix < len; ix++)
    {
        result += v1[ix]*v1[ix];
    }
    result = sqrt(result);

    return result;
}


void
normalize_vector(
        double *v1,
        unsigned int len
)
{
    double norm = vector_norm(v1, len);
    for(int ix =0; ix < len; ix++)
    {
        v1[ix] = v1[ix]/norm;
    }
}

double
average(
        double *v1,
        unsigned int len
)
{
    double result = 0;
    for(int i =0; i < len; i++)
    {
        result += v1[i];
    }
    result /= len;

    return result;
}


double
standard_deviation(
        double *v1,
        unsigned int len
)
{
    double ave = average(v1, len);
    double diff[len];

    for(int ix = 0; ix < len; ix++)
    {
        diff[ix] = v1[ix] - ave;
    }

    double std = 0;
    for(int ixx = 0; ixx < len; ixx++)
    {
        std += diff[ixx]*diff[ixx];
    }
    std /= len;
    std = sqrt(std);
    return std;
}

double
distance_between_vectors(
        double *v1,
        double *v2,
        unsigned int len
)
{
    double *distance = calloc(sizeof(double), len);
    elementwise_subtraction(distance, v1, v2, len);
    double result = vector_norm(distance, len);
    free(distance);

    return result;
}

void
print_vector(
        double *vec,  // Vector to print
        unsigned int ndims  // Number of dimensions
)
{
    for (int i = 0; i < ndims; i++) {
        printf("%10.5f ", vec[i]);
    }
}

int save_vector_to_csv(
        double *vec,  // Vector to save
        unsigned int ndims,  // Number of dimensions
        char *filename, // filename
        bool is_empty
)
{
    FILE *fp1;
    if(is_empty == true){
        fp1 = fopen(filename, "w"); // Create a file if is_empty == true
    } else {
        fp1 = fopen(filename, "a"); // Append if file if is_empty == false
    }

    if (fp1 == NULL)
    {
        printf("Error while opening the file.\n");
        return 1;
    }

    for(int i =0; i<ndims; ++i){
        if(i!=ndims-1){
            fprintf(fp1, "%10.5f, ", vec[i]);
        } else {
            fprintf(fp1, "%10.5f \n", vec[i]);
        }
    }


    fclose(fp1);
    return 0;
}

int save_transposedvector_to_csv(
        double *vec,  // Vector to save
        unsigned int ndims,  // Number of dimensions
        char *filename, // filename
        bool is_empty
)
{
    FILE *fp1;
    if(is_empty == true){
        fp1 = fopen(filename, "w"); // Create a file if is_empty == true
    } else {
        fp1 = fopen(filename, "a"); // Append if file if is_empty == false
    }

    if (fp1 == NULL)
    {
        printf("Error while opening the file.\n");
        return 1;
    }

    for(int i =0; i<ndims; ++i){

        fprintf(fp1, "%10.5f, \n", vec[i]);
    }


    fclose(fp1);
    return 0;
}

int save_matrix_to_csv(
        double **matrix,  // Matrix to save
        unsigned int nrows,  // Number of dimensions
        unsigned int ncols,  // Number of dimensions
        char *filename // filename
)
{
    FILE *fp1;
    fp1 = fopen(filename, "w"); // Create a file
    if (fp1 == NULL)
    {
        printf("Error while opening the file.\n");
        return 1;
    }

    for(int ix = 0; ix<nrows; ix++)
    {
        for(int jx = 0; jx<ncols; jx++)
        {
            if(jx == ncols - 1)
            {
                fprintf(fp1, "%10.5f", matrix[ix][jx]);
            } else {
                fprintf(fp1, "%10.5f, ", matrix[ix][jx]);
            }

        }
        fprintf(fp1,"\n");
    }

    fclose(fp1);
    return 0;
}

gsl_rng *
init_random_num_generator()
{
    //int seed = 42;
    //initializing seed through time
    int seed= time(NULL);
    const gsl_rng_type * T;
    gsl_rng * r;
    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc(T);
    gsl_rng_set(r, seed);
    return r;
}

double variance(double *quantity_vec, int number_of_elements)
{
    double average=0, average_square=0, variance=0;

    for(int element=0; element<number_of_elements; ++element)
    {
        average += quantity_vec[element] / number_of_elements;
        average_square += pow(quantity_vec[element],2) / number_of_elements;
    }
    variance = average_square- pow(average,2);
    return variance;
}
```

B Python-code for plotting

B.1 Code used for plotting multiple results: `plot.py`

```python
#import runpy
import alpha_plot
import derivative_energy_plot
import energy_plot
import histogram_plot
import plot_task1_xdist
import plots_task2
import time
import unpack_csv
# TODO: add input
def main(results):
    start_time = time.time()

    alpha_plot.main(results)
    alpha_time = time.time()
    print("Alpha done in: %s seconds" % (alpha_time - start_time))

    derivative_energy_plot.main(results)
    derivative_time = time.time()
    print("Derivative done in: %s seconds" % (derivative_time - alpha_time))

    energy_plot.main(results)
    energy_time = time.time()
    print("Energy done in: %s seconds" % (energy_time - derivative_time))

    histogram_plot.main(results)
    histogram_time = time.time()

    print("Histogram done in: %s seconds" % (histogram_time - energy_time))

    plot_task1_xdist.main(results)
    task1_time = time.time()
    print("Task1 done in: %s seconds" % (task1_time - histogram_time))

    plots_task2.main(results)
    task2_time = time.time()
    print("Task2 done in: %s seconds" % (task2_time - task1_time))
    print("Finished all plots in: %s seconds" % (task2_time - task1_time))

if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.2 Code used for plotting results for task 4: `beta_plot.py`

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import set_plot_style
import unpack_csv
import get_task_str

def main(results):
    sns.set_theme()
    set_plot_style.main()

    # results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace, ↩
        alpha_results, params) = results

    task_str = get_task_str.main()
    alpha_steps = alpha_results.ix.values[:]
    average_energy = alpha_results.E_average.values[:]
    alpha_task4 = alpha_results.alpha.values[:]
    beta_task4 = alpha_results.beta.values[:]
    N_alpha_steps = int(params.N_alpha_steps.values[0])

    values_unique_beta, counts_per_unique_beta = np.unique(beta_task4, return_counts=True)

    n = N_alpha_steps
    sliced_alphas = [alpha_task4[i * n:(i + 1) * n] for i in range((len(alpha_task4) + n - 1) // n )]
    sliced_energy = [average_energy[i * n:(i + 1) * n] for i in range((len(average_energy) + n - 1) // n )]

    slice_linspace = np.linspace(0, n, n, endpoint=False)

    change = np.empty(shape=(len(values_unique_beta), N_alpha_steps-1))
    for idx in range(N_alpha_steps-1):
        for jdx in range(len(values_unique_beta)):
            change[jdx][idx] = sliced_alphas[jdx][idx+1] - sliced_alphas[jdx][idx]


    fig_beta, ax_beta = plt.subplots(1,1)
    fig_change, ax_change = plt.subplots(1,1)
    for idx in range(len(values_unique_beta)):
        #print(sliced_alphas[idx][-1])
        #print("alpha after 200 iterations", sliced_alphas[idx][-1])
        ax_beta.plot(slice_linspace, sliced_alphas[idx][:], label = rf"$\beta$={values_unique_beta[idx]}")
        #print("Average change for last 100 iterations", np.mean(change[idx][-100:]))
        #print(f"{np.mean(change[idx][-100:]):.1e}")
        print(sliced_energy[idx][-1])
        ax_change.plot(slice_linspace[:-1], change[idx][:])

    ax_beta.set_xlabel("Iterations",)
    ax_beta.set_ylabel(r"$\alpha$")
    ax_beta.set_title(r'Evolution of $\alpha$ for different values of $\beta$')
    ax_beta.set_xlim(0,200)
    #ax_beta.set_ylim(0.12,0.14)
    ax_beta.legend(fontsize = 15)
    #ax_beta.set_yscale("log")
    fig_beta.tight_layout()
    fig_beta.savefig(f'plots_python/{task_str}/beta_plot.png')

    ax_change.set_xlabel("Steps",)
    ax_change.set_ylabel("Change in alpha")
    ax_change.set_title(f'Change of  alpha for different values of beta')
    ax_change.set_xlim(150,200)
    ax_change.set_ylim(-0.0005,0.0005)
    #ax_change.set_yscale("log")
    fig_change.tight_layout()
    fig_change.savefig(f'plots_python/{task_str}/change_plot.png')

if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.3 Code used for plotting derivative of energy: `derivative_energy_plot.py`

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter
import seaborn as sns
import set_plot_style
import unpack_csv
import get_task_str

def main(results):
    sns.set_theme()
    set_plot_style.main()

    #results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace, ↩
         alpha_results, params) = results

    task_str = get_task_str.main()

    alpha = alpha_results.alpha.values[0]
    N_steps = params.N_steps.values[0]
    d_displacement = params.d_displacement.values[0]

    # Calculate moving average of energy

    #window_size = N_steps/100; poly_order = 3
    #moving_averages = savgol_filter(E_local_derivative, window_size, poly_order)

    # Plot energy
    fig_energy_deriv, ax_energy_deriv = plt.subplots(1,1)
    ax_energy_deriv.plot(steps_linspace, E_local_derivative, alpha = 0.5, label = "Measured energy derivative")
    #ax_energy_deriv.plot(steps_linspace, moving_averages, 'k--', label = f"Savitzky-Golay with w={window_size}, ↩
         p={poly_order}")
    ax_energy_deriv.set_xlabel("Steps [a.u.]")
    ax_energy_deriv.set_ylabel("Energy [a.u.]")
    ax_energy_deriv.set_title(f'Derivative of local energy, alpha = {alpha}')
    #ax_energy_deriv.set_title(f'Derivative of local energy, alpha = {alpha}')
    ax_energy_deriv.legend(loc="upper right")
    fig_energy_deriv.savefig(f'plots_python/{task_str}/energy_derivative.png')

if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.4 Code used for plotting energy : `energy_plot.py`

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter
import seaborn as sns
import set_plot_style
import unpack_csv
import get_task_str

def main(results):
    sns.set_theme()
    set_plot_style.main()

    #results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace, ↩
        alpha_results, params) = results

    task_str = get_task_str.main()

    alpha = alpha_results.alpha.values[0]
    E_average = alpha_results.E_average.values[0]
    N_steps = params.N_steps.values[0]
    d_displacement = params.d_displacement.values[0]
    std = np.std(E_local)

    # Calculate moving average of energy
    #print(N_steps)
    #print(E_local)
    #window_size = int(N_steps/10); poly_order = 2
    #moving_averages = savgol_filter(E_local, window_size, poly_order)
    print(E_average)
    # Plot energy
    fig_energy, ax_energy = plt.subplots(1,1)
    #ax_energy.scatter(steps_linspace, E_local, alpha = 0.5, label = "Measured energy", facecolor = "none", ↩
        edgecolor="k")
    ax_energy.plot(steps_linspace, E_local, alpha = 0.5, label = "Measured energy")
    #ax_energy.plot(steps_linspace, moving_averages, 'k--', label = f"Averaged energy (w={window_size}, p={↩
        poly_order})")
    ax_energy.hlines(y=E_average, xmin=0, xmax=N_steps, linewidth=2, color='r', linestyles='--', label = f"↩
        E_average = {E_average:.4f}")
    ax_energy.hlines(y=E_average+std, xmin=0, xmax=N_steps, linewidth=2, color='r', linestyles='--', alpha = 0.5,↩
         label = f"E_average = {E_average+std:.4f}")
    ax_energy.hlines(y=E_average-std, xmin=0, xmax=N_steps, linewidth=2, color='r', linestyles='--', alpha = 0.5,↩
         label = f"E_average = {E_average-std:.4f}")


    #ax_energy.vlines(x=1000, ymin=min(E_local), ymax=max(E_local), linewidth=2, color='r', linestyles='--', ↩
        label = f"N$_{{eq}}$")
    ax_energy.set_xlabel("Steps [a.u.]")
    ax_energy.set_ylabel("Energy [a.u.]")
    ax_energy.set_title(f'Local energy, alpha = 0.136')
    ax_energy.legend(loc="lower right", fontsize=16)
    fig_energy.savefig(f'plots_python/{task_str}/energy.png')

if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.5 Code used finding the most recent task: `get_task_str.py`

```
import unpack_csv

def main():
```

```python

    results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace, ↩
         alpha_results, params) = results

    if params.is_task1.values[0]:
        task_str = "task1"
    elif params.is_task2.values[0]:
        task_str = "task2"
    elif params.is_task3.values[0]:
        task_str = "task3"
    elif params.is_task4.values[0]:
        task_str = "task4"
    else:
        task_str = "task5"

    return task_str
```

B.6 Code used to plot histogram: `histogram_plot.py`

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import set_plot_style
import unpack_csv
import get_task_str

def main(results):
    sns.set_theme()
    set_plot_style.main()

    #results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace, ↩
         alpha_results, params) = results

    task_str = get_task_str.main()

    alpha = alpha_results.alpha.values[0]

    def rho(rvec, z):
        rho = z**3 *4 * rvec**2 * np.exp(-2*z*rvec)

        return rho

    ## Plot histogram

    n_bins = 70
    fig_dist, ax_dist = plt.subplots(1,2, figsize=(12,6))

    for idx, R_chain in enumerate([R1, R2]):
        R_norm =  np.square(R_chain)
        R_norm = np.sqrt(np.sum(R_norm, axis=1))

        rvec = np.linspace(0.1, np.max(R_norm))
        counts_r, bins_r = np.histogram(R_norm, bins = n_bins, density = True)

        ax_dist[idx].stairs(counts_r, bins_r, fill=True)
        ax_dist[idx].plot(rvec, rho(rvec, 27/16), color='r', linestyle='--',label= r'$\rho $ optimized', ↩
             linewidth=3)
        ax_dist[idx].plot(rvec, rho(rvec, 2), color='k', linestyle=':',label= r'$\rho $ unscreened', linewidth=3)

        ax_dist[idx].set_title(rf"Distribution for R$_{idx}$, $\alpha$ = 0.1")
        ax_dist[idx].set_xlabel(f"Radius [$a_0$]")
        ax_dist[idx].set_ylabel("Probability density")
        ax_dist[idx].legend()
    plt.tight_layout()
    fig_dist.savefig(f'plots_python/{task_str}/histogram_alpha.png')

if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.7 Code usedfor plotting equilatisation of MCMC-chain: `neq_plot.py`

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter
import seaborn as sns
import set_plot_style
import unpack_csv
import get_task_str

def main(results):
    sns.set_theme()
    set_plot_style.main()

    #results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace, ↩
         alpha_results, params) = results

    task_str = get_task_str.main()

    alpha = alpha_results.alpha.values[0]
    E_average = alpha_results.E_average.values[0]
    N_steps = params.N_steps.values[0]
    d_displacement = params.d_displacement.values[0]

    # Calculate moving average of energy
    #print(N_steps)
    #print(E_local)
    window_size = int(N_steps/10); poly_order = 2
    moving_averages = savgol_filter(E_local, window_size, poly_order)

    # Plot energy
    fig_energy, ax_energy = plt.subplots(1,1)
    ax_energy.plot(steps_linspace, E_local, alpha = 0.5, label = "Measured energy")
    ax_energy.plot(steps_linspace, moving_averages, 'k--', label = f"Averaged energy (w={window_size}, p={↩
         poly_order})")
    #ax_energy.hlines(y=E_average, xmin=0, xmax=N_steps, linewidth=2, color='r', linestyles='--', label = f"↩
         E_average = {E_average}")
    ax_energy.vlines(x=1000, ymin=min(E_local), ymax=max(E_local), linewidth=2, color='r', linestyles='--', label↩
          = f"N$_{{eq}}$")
    ax_energy.set_xlabel("Iterations")
    ax_energy.set_ylabel("Energy [E$_h$]")
    ax_energy.set_title(f'Local energy, alpha = 0.1')
    ax_energy.legend(loc="lower right", fontsize = 16)
    fig_energy.tight_layout()
    fig_energy.savefig(f'plots_python/{task_str}/n_eq.png')

if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.8 Code used for plotting task3: `plot_task_3.py`

```python


import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import set_plot_style
sns.set_theme()
set_plot_style.main()
from scipy.optimize import curve_fit

E_of_alpha = pd.read_csv("../csv/E_of_alpha.csv", engine="pyarrow", names= ["E_of_alpha"])

Variance_vec = pd.read_csv("../csv/variance_alpha.csv", engine="pyarrow", names= ["variance"])

alpha_max = 0.17; alpha_min=0.12

def polynomial(x, a, b, c):
    return a * x**2 + b * x + c


E_of_alpha= E_of_alpha.values.astype(float)
Variance_vec = Variance_vec.values.astype(float)

alpha_step = (alpha_max-alpha_min)/len(E_of_alpha[0])

alpha_vec = np.arange(1,len(E_of_alpha[0])+1)* alpha_step + alpha_min
```

```python
opt_e_ind = np.argmin(E_of_alpha[0])
opt_alpha = alpha_vec[opt_e_ind]
min_E =np.min(E_of_alpha[0])
st =r"Optimal $\alpha =$" +f"{opt_alpha:.4f}" + r" with $\langle E_L\rangle$= "+ f"{min_E:.4f}"



fig_task3, ax_task3 = plt.subplots(1,1)

upper_bound = E_of_alpha[0]+np.sqrt(Variance_vec[0]/(1e7))
lower_bound = E_of_alpha[0]-np.sqrt(Variance_vec[0]/(1e7))
fs = 15

ax_task3.scatter(opt_alpha, min_E, color='k',label = st, s =300, zorder=0, alpha=1)

ax_task3.plot(alpha_vec, E_of_alpha[0], label = "Average energy", color='r', linewidth=2)
ax_task3.fill_between(alpha_vec,lower_bound,upper_bound, label=r"$\pm\sigma$", alpha=0.5)

params, covariance = curve_fit(polynomial, alpha_vec, E_of_alpha[0])

alpha_vals = np.linspace(alpha_min, alpha_max, 50)


ax_task3.legend(fontsize = fs)
ax_task3.set_title(r"Average local energy as function of parameter $\alpha$", fontsize=fs)
ax_task3.set_xlabel(r"$\alpha$", fontsize = fs+4)
ax_task3.set_ylabel(r"Energy [$E_h$]", fontsize = fs)


plt.tight_layout()


ax_task3.legend(fontsize = fs-2)
fig_task3.savefig("plots_python/task3/E_of_alpha.png")

print((Variance_vec[0]))
print(alpha_vec.shape)

opt_e_ind = np.argmin(E_of_alpha[0])

opt_alpha = alpha_vec[opt_e_ind]

print("optimal alpha= ", opt_alpha)
print("with energy", np.min(E_of_alpha[0]))


fig_hist, ax_hist = plt.subplots(1,1)
```

B.9 Code used for plotting task 5: `plot_task_5.py`

```python
# import numpy as np
# import pandas as pd
# import seaborn as sns
# import matplotlib.pyplot as plt
# sns.set_theme()


# E_of_alpha = pd.read_csv("../csv/E_of_alpha.csv", engine="pyarrow", names= ["E_of_alpha"])


import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import gridspec
import set_plot_style
sns.set_theme()
set_plot_style.main()
from scipy.optimize import curve_fit

SMALL_SIZE = 20
MEDIUM_SIZE = 26
BIGGER_SIZE = 26

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=MEDIUM_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=MEDIUM_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)

E_of_alpha = pd.read_csv("../csv/E_of_alpha.csv", engine="pyarrow", names= ["E_of_alpha"])

Variance_vec = pd.read_csv("../csv/variance_alpha.csv", engine="pyarrow", names= ["variance"])


E_of_alpha= E_of_alpha.values.astype(float)
Variance_vec = Variance_vec.values.astype(float)

average_E = np.mean(E_of_alpha)
total_variance = 1/len(Variance_vec[0]) * np.sum(Variance_vec[0]) / (1e7)
std_E = np.sqrt(total_variance)

alpha_vec = np.arange(1,len(E_of_alpha[0])+1)

upperquantile = np.quantile(E_of_alpha, 0.75)
lowerquantile = np.quantile(E_of_alpha, 0.25)

print(upperquantile)
print(lowerquantile)




upper_bound = E_of_alpha[0]+np.sqrt(Variance_vec[0]/(1e7))
lower_bound = E_of_alpha[0]-np.sqrt(Variance_vec[0]/(1e7))

fig_task3, ax_task3 = plt.subplots(1,1)

ax_task3.errorbar(alpha_vec, np.sort(E_of_alpha[0]), yerr = np.sqrt(Variance_vec[0]/(1e7)), label = r"$\langle ↩
    E_L \rangle $", color='b', marker = 'o',linestyle="")
ax_task3.hlines(y=average_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=3, color='r', linestyles='--', label = ↩
    f"E_average = {average_E:.4f}")
ax_task3.hlines(y=average_E+std_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=2, color='r', alpha=0.8, ↩
    linestyles='--', label = rf"$\pm \sigma = \pm {std_E:.5f}$")
ax_task3.hlines(y=average_E-std_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=2, color='r', alpha=0.8, ↩
    linestyles='--')
#ax_task3.fill_between(alpha_vec,lower_bound,upper_bound, label=r"$\pm\sigma$", alpha=0.5)

# ax_task3.plot(alpha_vec, E_of_alpha[0], label = r"$\langle E_L \rangle $", color='b', linewidth=2)
# ax_task3.hlines(y=average_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=3, color='r', linestyles='--', label ↩
    = f"E_average = {average_E:.4f}")
# ax_task3.hlines(y=average_E+std_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=2, color='r', alpha=0.8, ↩
    linestyles='--', label = rf"$+-\sigma = \pm {std_E:.4f}$")
# ax_task3.hlines(y=average_E-std_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=2, color='r', alpha=0.8, ↩
    linestyles='--')
# ax_task3.fill_between(alpha_vec,lower_bound,upper_bound, label=r"$\pm\sigma$", alpha=0.5)

#params, covariance = curve_fit(polynomial, alpha_vec, E_of_alpha[0])

#alpha_vals = np.linspace(alpha_min, alpha_max, 50)


ax_task3.legend()
ax_task3.set_title(rf" $\langle E_L \rangle $ for 100 iterations with alpha = 0.136")
ax_task3.set_xlabel(r"Measurements, sorted from lowest to highest $\langle E_L \rangle $")
ax_task3.set_ylabel(r"Energy [$E_h$]")
opt_e_ind = np.argmin(E_of_alpha[0])
opt_alpha = alpha_vec[opt_e_ind]
min_E =np.min(E_of_alpha[0])

#ax_task3.plot(alpha_vals, polynomial(alpha_vals, *params), linestyle=':',linewidth=4, label=f'Curve fit, \n y = ↩
    {params[0]:.2} x^2 + {params[1]:.2} x + {params[2]:.2}')

plt.tight_layout()


ax_task3.legend()
fig_task3.savefig("plots_python/task5/E_of_alpha.png")

# fake up some dat

fig1, ax1 = plt.subplots(figsize=(3,6))

ax1.hlines(y=np.max(E_of_alpha[0]), xmin=-3, xmax=3, linewidth=2, color='k', label = f"Max = {np.max(E_of_alpha↩
    [0]):.4f}")
ax1.hlines(y=upperquantile, xmin=-10, xmax=10, linewidth=2, color='k', label = f"Upper quantile = {upperquantile↩
    :.4f}")
ax1.hlines(y=average_E, xmin=-10, xmax=10, linewidth=3, color='k', label = f"E_average = {average_E:.4f}")
```

```
ax1.hlines(y=lowerquantile, xmin=-10, xmax=10, linewidth=2, color='k', label = f"Lower_quantile = {lowerquantile:.4f}")
ax1.hlines(y=np.min(E_of_alpha[0]), xmin=-3, xmax=3, linewidth=2, color='k', label = f"Min = {np.min(E_of_alpha[0]):.4f}")

ax1.vlines(x=-10, ymin=lowerquantile, ymax=upperquantile, linewidth=2, color='k')
ax1.vlines(x=10, ymin=lowerquantile, ymax=upperquantile, linewidth=2, color='k')
ax1.vlines(x=0, ymin=upperquantile, ymax=np.max(E_of_alpha[0]), linewidth=2, color='k')
ax1.vlines(x=0, ymin=np.min(E_of_alpha[0]), ymax=lowerquantile, linewidth=2, color='k')

ax1.hlines(y=average_E+std_E, xmin=-15, xmax=20, linewidth=3, color='r', label = f"E_average + sigma = {average_E:.4f}")
ax1.hlines(y=average_E-std_E, xmin=-15, xmax=20, linewidth=3, color='r', label = f"E_average - sigma = {average_E:.4f}")

ax1.annotate(text= "", xy = (20,average_E-std_E), xytext=(20,average_E+std_E), arrowprops=dict(arrowstyle='<->', color="r", linewidth=2))

#(20,average_E)
ax1.set_xlim(-40,40)
ax1.set_yticks([])
ax1.set_xticks([])
ax1.set_title("Boxplot")
fig1.tight_layout()
fig1.savefig("plots_python/task5/Boxplot_alpha.png")


#fig_task3both, ax_task3both = plt.subplots(1,2)
fig_task3both = plt.figure(figsize=(18, 8))
gs = gridspec.GridSpec(1, 2, width_ratios=[2, 1], height_ratios=[1])
ax_task3bothalpha = plt.subplot(gs[0])
ax_task3bothbox = plt.subplot(gs[1])


# ax_task3bothalpha.plot(alpha_vec, E_of_alpha[0], label = r"$\langle E_L \rangle $", color='b', linewidth=2)
# ax_task3bothalpha.fill_between(alpha_vec,lower_bound,upper_bound, label=r"$\pm\sigma$", alpha=0.5)
ax_task3bothalpha.errorbar(alpha_vec, np.sort(E_of_alpha[0]), yerr = np.sqrt(Variance_vec[0]/(1e7)), label = r"$\langle E_L \rangle $", color='b', marker = 'o',linestyle="")
# ax_task3bothalpha.hlines(y=average_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=3, color='r', linestyles='--', label = f"E_average = {average_E:.4f}")
# ax_task3bothalpha.hlines(y=average_E+std_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=2, color='r', alpha=0.8, linestyles='--', label = rf"$\pm \sigma = \pm {std_E:.5f}$")
# ax_task3bothalpha.hlines(y=average_E-std_E, xmin=1, xmax=len(E_of_alpha[0])+1, linewidth=2, color='r', alpha=0.8, linestyles='--')

ax_task3bothbox.hlines(y=np.max(E_of_alpha[0]), xmin=-3, xmax=3, linewidth=2, color='k', label = f"Max = {np.max(E_of_alpha[0]):.4f}")
ax_task3bothbox.hlines(y=upperquantile, xmin=-10, xmax=10, linewidth=2, color='k', label = f"Upper quantile = {upperquantile:.4f}")
ax_task3bothbox.hlines(y=average_E, xmin=-10, xmax=10, linewidth=3, color='k', label = f"E_average = {average_E:.4f}")
ax_task3bothbox.hlines(y=lowerquantile, xmin=-10, xmax=10, linewidth=2, color='k', label = f"Lower_quantile = {lowerquantile:.4f}")
ax_task3bothbox.hlines(y=np.min(E_of_alpha[0]), xmin=-3, xmax=3, linewidth=2, color='k', label = f"Min = {np.min(E_of_alpha[0]):.4f}")

ax_task3bothbox.vlines(x=-10, ymin=lowerquantile, ymax=upperquantile, linewidth=2, color='k')
ax_task3bothbox.vlines(x=10, ymin=lowerquantile, ymax=upperquantile, linewidth=2, color='k')
ax_task3bothbox.vlines(x=0, ymin=upperquantile, ymax=np.max(E_of_alpha[0]), linewidth=2, color='k')
ax_task3bothbox.vlines(x=0, ymin=np.min(E_of_alpha[0]), ymax=lowerquantile, linewidth=2, color='k')

ax_task3bothbox.hlines(y=average_E+std_E, xmin=-15, xmax=20, linewidth=3, color='r', label = f"E_average + sigma = {average_E:.4f}")
ax_task3bothbox.hlines(y=average_E-std_E, xmin=-15, xmax=20, linewidth=3, color='r', label = f"E_average - sigma = {average_E:.4f}")

ax_task3bothbox.annotate(text= "", xy = (20,average_E-std_E), xytext=(20,average_E), arrowprops=dict(arrowstyle='<->', color="r", linewidth=2))
ax_task3bothbox.annotate(text= "", xy = (20,average_E), xytext=(20,average_E+std_E), arrowprops=dict(arrowstyle='<->', color="r", linewidth=2))

ax_task3bothbox.annotate(text= rf"$\sigma =$", xy = (23, average_E), xytext=(23, average_E+std_E/2))
ax_task3bothbox.annotate(text= rf"$\sigma$", xy = (23, average_E), xytext=(23, average_E-std_E/2))
ax_task3bothbox.annotate(text= rf"{std_E:.2e}", xy = (23,average_E), xytext=(23,average_E+std_E/2-0.00015))
ax_task3bothbox.annotate(text= rf"E$_{{Avg}}\to $", xy = (-30,average_E), xytext=(-30,average_E-0.00003))
ax_task3bothbox.annotate(text= rf"{average_E:.4}", xy = (-30,average_E), xytext=(-34,average_E-0.00003-0.00015))

ax_task3bothbox.annotate(text= rf"E$_{{Max}}\to $", xy = (-15,np.max(E_of_alpha[0])), xytext=(-20,np.max(E_of_alpha[0])-0.00003))
ax_task3bothbox.annotate(text= rf"{np.max(E_of_alpha[0]):.4}", xy = (-10,np.max(E_of_alpha[0])), xytext=(-24,np.max(E_of_alpha[0])-0.00003-0.00015))
```

```python
ax_task3bothbox.annotate(text= rf"E$_{{Min}}\to $", xy = (-15,np.min(E_of_alpha[0])), xytext=(-20,np.min(E_of_alpha[0])-0.00003))
ax_task3bothbox.annotate(text= rf"{np.min(E_of_alpha[0]):.4}", xy = (-10,np.min(E_of_alpha[0])), xytext=(-24,np.min(E_of_alpha[0])-0.00003-0.00015))

#ax_task3bothbox.annotate(text= rf"{average_E:.6}", xy = (-28,average_E-0.0001), xytext=(-43,average_E-0.0001))

ax_task3bothbox.set_xlim(-45,45)

ax_task3bothbox.set_ylim([-2.88, -2.877])
ax_task3bothalpha.set_ylim([-2.88, -2.877])
ax_task3bothbox.yaxis.tick_right()
#ax_task3bothbox.set_yticklabels([])
ax_task3bothbox.set_xticks([])
ax_task3bothalpha.set_xlabel("")
ax_task3bothbox.set_ylabel(r"Energy [$E_h$]")

ax_task3bothalpha.set_xlabel(r"Measurements, sorted from lowest to highest $\langle E_L \rangle $")
ax_task3bothalpha.set_ylabel(r"Energy [$E_h$]")
#ax_task3bothbox.legend()
fig_task3both.suptitle(rf" Measurements of $\langle E_L \rangle $ with summary as boxplot, $\alpha = 0.1365$")
# ax_task3bothbox.set_title(rf" Boxplot")
# ax_task3bothalpha.set_title(rf" Measurements of $\langle E_L \rangle $, $\alpha = 0.136$")


fig_task3both.tight_layout()
fig_task3both.savefig("plots_python/task5/Both.png")

#print((Variance_vec[0]))
#print(alpha_vec.shape)

opt_e_ind = np.argmin(E_of_alpha[0])

opt_alpha = alpha_vec[opt_e_ind]

print("optimal alpha= ", opt_alpha)
print("with energy", np.min(E_of_alpha[0]))
```

B.10 Code used for plotting task 1: `plot_task_1_xdist.py`

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import set_plot_style
import unpack_csv
import get_task_str

def main(results):
    sns.set_theme()
    set_plot_style.main()

    #results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace,
        alpha_results, params) = results

    task_str = get_task_str.main()

    arr_sin = np.sin(theta_distribution)
    arr_x = np.cos(theta_distribution)
    x = np.linspace(0, np.pi, 100)
    y = np.sin(x)/2

    n_bins = 50
    fig_xdist, ax_dist = plt.subplots(1,2, figsize =(12,6))

    arr_str = ["x", r"$\theta$ [rad]", r"$\sin(\theta)$"]
    ax_dist[0].axhline(1/2, label='Uncorrelated distribution', color='r', linewidth=2, linestyle='dashed')
    ax_dist[1].plot(x, y, label='Uncorrelated distribution', color='r', linewidth=2, linestyle='dashed')

    for idx, arr_dist in enumerate([arr_x, theta_distribution]):
        counts, bins = np.histogram(arr_dist, bins = n_bins, density = True)
        ax_dist[idx].stairs(counts, bins, fill = True, label='Sampled distribution')
        ax_dist[idx].set_title('Distribution for ' + arr_str[idx])
        ax_dist[idx].set_xlabel(f'{arr_str[idx]}')
        ax_dist[idx].set_ylabel('Probability density')
        ax_dist[idx].legend(fontsize = 16, loc = "lower center")

    plt.tight_layout()
    print(task_str)
    fig_xdist.savefig(f'plots_python/{task_str}/xdist_new.png')

if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.11 Code used for plotting task 2: `plots_task2.py`

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import set_plot_style
import unpack_csv
import get_task_str

def main(results):
    sns.set_theme()
    set_plot_style.main()

    #results = unpack_csv.main()
    (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace, ↩
        alpha_results, params) = results

    task_str = get_task_str.main()

    lag_vec = np.arange(0,len(phi_k))

    block_average = pd.read_csv("../csv/block_avg_vec.csv", engine="pyarrow", names= ["block_average"])
    block_average = block_average.values[:,0].astype(float)

    t_relaxation = np.argmin(np.abs(phi_k-np.exp(-2)))

    #stat_ineff_cor = 2*np.sum(phi_k[:t_relaxation]) #factor 2 from fact that it is symmetric, -1 because we ↩
        count k=0 twice
    stat_ineff_cor = 2*np.sum(phi_k)-1 #factor 2 from fact that it is symmetric, -1 because we count k=0 twice

    stat_ineff_block_av = np.average(block_average[150:])
    print(f"------\nstatistical inefficiency calculated from correlation funcion = {stat_ineff_cor}\n----")
    print(f"------\nstatistical inefficiency calculated from block averaging = {stat_ineff_block_av}\n----")
    print("relaxation time =",t_relaxation)


    fig_phi_k, ax_phi = plt.subplots(1,1)

    ax_phi.plot(lag_vec, phi_k, label = "phi_k")

    # dom h r  r  enhetsl sa
    ax_phi.set_xlabel(r"$k$")
    ax_phi.set_ylabel(r"$\Phi_k$")
    ax_phi.set_title(r'correlation function $\Phi_k$')
    ax_phi.legend()

    fig_phi_k.savefig(f"plots_python/{task_str}/phi_k.png")

    fig_block_avg, ax_blav = plt.subplots(1,1)


    block_size_vec = np.arange(1, len(block_average)+1)

    #ax_blav.plot(block_size_vec, block_average)
    ax_blav.set_xlabel("Block size")
    ax_blav.set_ylabel(r"$n_s$")
    ax_blav.set_title(r"Statistical inefficiency $n_s$ calculated from block averaging")
    ax_blav.axhline(stat_ineff_cor, label =r"$n_s$ calculated from correlation function", color="r", linewidth=4)
    ax_blav.legend()
    ax_blav.scatter(block_size_vec, block_average, facecolor ="none", edgecolor="k", alpha=0.8)

    fig_block_avg.savefig(f"plots_python/{task_str}/block_avg.png")





if(__name__ == "__main__"):
    results = unpack_csv.main()
    main(results)
```

B.12 Code used for unpacking data: `unpack_csv.py`

```python
import numpy as np
import pandas as pd
import time

def main():
    start_time = time.time()

    R1 = pd.read_csv(f"../csv/R1.csv", engine="pyarrow", names = ["R1x", "R1y", "R1z"])
    R2 = pd.read_csv("../csv/R2.csv", engine="pyarrow", names = ["R1x", "R1y", "R1z"])
    E_local = pd.read_csv("../csv/E_local.csv", engine="pyarrow", names= ["E_local"])
    E_local_derivative = pd.read_csv("../csv/E_local_derivative.csv", engine="pyarrow", names= ["↩
        E_local_derivative"])
    x_distribution = pd.read_csv("../csv/x_distribution.csv", engine="pyarrow", names= ["x_distribution"])
    theta_distribution = pd.read_csv("../csv/theta.csv", engine="pyarrow", names= ["theta"])
    phi_k =pd.read_csv("../csv/phi_k.csv", engine="pyarrow", names= ["phi_k"])
    alpha_results = pd.read_csv("../csv/alpha_results.csv", engine="pyarrow", names= ["ix", "E_average", "alpha",↩
         "gamma", "E_PD_average", "beta"])
    params = pd.read_csv("../csv/params.csv", names = ["N_alpha_steps", "N_discarded_steps", "alpha", "A", "beta"↩
        , "N_steps", "d_displacement", "is_task1", "is_task2", "is_task3", "is_task4"])

    steps_linspace = np.linspace(0,int(params.N_steps), int(params.N_steps), endpoint=False)

    E_local=E_local.values[:,0].astype(float)
    E_local_derivative=E_local_derivative.values[:,0].astype(float)
    x_distribution=x_distribution.values[:,0].astype(float)
    theta_distribution = theta_distribution.values[:,0].astype(float)
    phi_k=phi_k.values[:,0].astype(float)

    array_tuple = (R1, R2, E_local, E_local_derivative, x_distribution, theta_distribution, phi_k, steps_linspace↩
        , alpha_results, params)


    return array_tuple
```

B.13 Code used for setting plot parameters: `set_plot_style.py`

```python
import matplotlib as plt

## TODO: If text sies are updated, check also that subplots with custom figsize looks good
def main():
    # set default figure size
    plt.rcParams["figure.figsize"] = [8, 6]

    SMALL_SIZE = 15
    MEDIUM_SIZE = 18
    BIGGER_SIZE = 18

    plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
    plt.rc('axes', titlesize=MEDIUM_SIZE)     # fontsize of the axes title
    plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
    plt.rc('legend', fontsize=MEDIUM_SIZE)    # legend fontsize
    plt.rc('figure', titlesize=BIGGER_SIZE)
```