

# Homework 3:

## Diffusion Monte Carlo

Jonatan Haraldsson  
[jonhara@chalmers.se](mailto:jonhara@chalmers.se)

Oscar Stommendal  
[oscarsto@chalmers.se](mailto:oscarsto@chalmers.se)

Task N <sup>o</sup>	Points	Avail. points
$\Sigma$		

January 13<sup>th</sup> 2025  
Course: *Computational Physics*, 7.5 hp  
Course Code: *FKA 122*

*Physics, MSc.*  
CHALMERS UNIVERSITY OF TECHNOLOGY



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Introduction

With the last decades' increase in computational power, simulating physical processes using computers has become an important tool in all aspects of modern physics. Although this, computers still have their limitations and certain systems are notoriously difficult to solve. One class of such problems are Quantum Many-body Problems, where the time-dependent Schrödinger equation has to be solved for all particles in the system simultaneously. However, there is still hope; methods such as the *Diffusion Monte Carlo* (DMC) can provide an exact solution to the ground state energies and wave functions [1].

The basic idea of DMC is to first introduce an imaginary time  $\tau = it$  and then shifting the energy scale by the so-called trial energy  $\mathcal{E}$ . In essence, the Hamiltonian of a system is first split into two parts, one part being the kinetic energy and the second part the potential energy subtracted with  $\mathcal{E}$  (the energy shift). The Green's function is then used to define the short-time propagator, which describe the time-evolution of the wave-function. This can be decomposed for sufficiently small time-steps  $\Delta\tau$  into a *diffusive* and a *reactive* part, which account for the kinetic and potential energy's influence on the spreading of the wave function, respectively. By iterating over many time-steps, the algorithm adjusts  $\mathcal{E}$  dynamically, ensuring it reflects the ground-state energy  $E_0$ .

In this report, DMC was implemented twice – first in its most basic form and then using importance sampling, which will be described later. In both cases, the Helium atom was studied with the goal to determine the ground state energy. However, as an introduction, a single particle in a Morse potential was studied. Atomic units (a.u.) were used during all simulations, i.e.  $\hbar = e = m_e = 4\pi\epsilon_0 = 1$ . Using this unit system, the length unit was the Bohr radius  $a_0 = 0.529 \text{ \AA}$  and the energy-ditto the Hartree energy  $E_H = 27.2 \text{ eV}$ .

## Task 1 – Basic DMC 1D Implementation

In the first task, the dynamics of a single particle moving in the one-dimensional Morse potential (note the use of atomic units from here on, see the [Introduction](#))

$$V(x) = \frac{1}{2}(1 - e^{-x})^2 \quad (1)$$

was realized in C using DMC. To illustrate, the Morse potential is given in Figure 1.

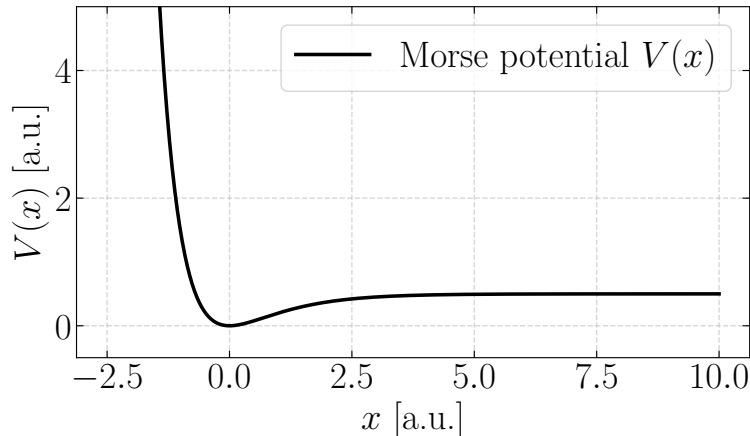


Figure 1: One-dimensional Morse potential given by Equation 1.

The first step to obtaining the ground state energy and wave function, was to initialize  $N_0 = 200$  random walkers uniformly on the interval  $w \in [-5, 5]$ . In  $\mathbb{C}$ , a  $(N_0 \times 1)$  vector  $\vec{w}$  such that  $w_j = x_j \in [-5, 5]$  stored the positions of every walker  $j$ . Subsequently, the walkers' positions  $x_j$  were updated accordingly

$$x_j^{(new)} = x_j^{(old)} + \sqrt{\Delta\tau} \mathcal{G}, \quad (2)$$

where  $x_j^{(new)}$  is the updated position and  $\mathcal{G}$  is a Gaussian random number with mean 0 and variance 1. Next, each walker was associated with a weight given by

$$W(x_j) \equiv e^{(\mathcal{E}_i - V(x_j))\Delta\tau}, \quad (3)$$

where  $\mathcal{E}_i$  is the instantaneous estimated ground state energy at iteration  $i$  and  $\Delta\tau = 0.02$  is the time-step. In each iteration, the walkers' destinies were determined through a *death-and-birth* algorithm, and the number of "offsprings" for each walker was given by

$$m_j = \lfloor W(x_j) + \mathcal{U} \rfloor, \quad (4)$$

where  $\lfloor \cdot \rfloor$  denotes the floor function and  $\mathcal{U}$  is a random uniformly distributed number on the interval  $[0, 1]$ . Put simply, the total number of walkers in each iteration  $N_i$  was updated by making  $m_j$  copies of walker  $j$  according to the provided code snippet. Note on line 6 that no copies are made if  $m_j = 0$ . The number of new walkers  $N_{i+1}$  is, thus, given by  $\sum_j^{N_i} m_j$ .

#### Code Snippet Task 1, Copying machine

```

1  int shift = 0;
2  // Populate the new array with the walkers
3  for (int j = 0; j < N_walkers; j++)
4  {
5      // If the walker survives (m_walker[j] > 0)
6      for (int m = 0; m < m_walker[j]; m++)
7      {
8          for(int n = 0; n < ndim; n++)
9          {
10             walkmen_pos_new[shift][n] = walkmen_pos[j][n]; // Copy walker's
11             ↪ position
12         }
13         shift++; // Increment the new array index
14     }

```

Lastly, the instantaneous estimated ground state energy was updated using

$$\mathcal{E}_{i+1} = \mathcal{E}_i - \gamma \ln \frac{N_i}{N_0}, \quad (5)$$

where  $\gamma = 0.5$  is a damping parameter. However, instead of using the instantaneous value of the ground state energy,  $\mathcal{E}_i$ , the accumulated mean,

$$E_{\top}^{(i)} \equiv \langle \mathcal{E} \rangle_i = \frac{1}{i - i_{eq}} \sum_{i'=i_{eq}+1}^i \mathcal{E}_{i'}, \quad (6)$$

was used. Here,  $i_{eq}$  is the number of *equilibration* iterations. As an initial value for the equilibration,  $\mathcal{E}_0$  was set to 0.5. The accumulated average was updated accordingly

$$E_{\top}^{(i+1)} = \frac{1}{i - i_{eq} + 1} \mathcal{E}_{i+1} + \frac{i - i_{eq}}{i - i_{eq} + 1} E_{\top}^{(i)}. \quad (7)$$

Then, the new energy  $\mathcal{E}_{i+1}$  and the new positions  $x^{(new)}$  were used to calculate new weights with Equation 3, and new walkers was born/killed using the condition in Equation 4. This procedure was repeated for  $\tau = 5\,000$  or  $5\,000/0.02 = 250\,000$  iterations, 7 500 of which were equilibration iterations, and therefore not counted in the accumulated average,  $E_{\top}^{(i)}$ . Afterwards, the ground-state energy was estimated using an average value  $\langle E_{\top} \rangle$  over all iterations, and the wave function was obtained by sorting the walker positions in a histogram.

In Figure 2a, instantaneous ground-state energies  $\mathcal{E}$  and accumulate average  $E_{\top}$  are given along with the average  $\langle E_{\top} \rangle = 0.377$ . Here,  $i > i_{eq}$ ; in other words, after the equilibration, which clarifies why the accumulated average is quite flat. Moreover, Figure 2b illustrates the distribution of instantaneous samples ( $\mathcal{E}_i$ ), the theoretical  $E_0 = 3/8 = 0.375$  and  $\langle E_{\top} \rangle$  in a histogram with the option `density=True` in Python, which gave a probability density on the  $y$ -axis instead of counts. The thinner dashed lines in the histogram are  $\langle E_{\top} \rangle \pm \sigma_{\mathcal{E}}$ . In addition, the number of walkers in each iteration is given in Figure 3a along with the average  $\langle N_{walkers} \rangle = 200$ . The fact that  $\langle N_{walkers} \rangle = N_0 = 200$  indicates that the model has stabilized. Figure 3b displays a histogram of the number of walkers, and thinner lines for  $\langle N_{walkers} \rangle \pm \sigma_N$ , where  $\sigma_N = 7.8$ , again, with `density=True` in Python. To summarize, some of the averages along with their corresponding standard deviations are available in Table 1. On the whole, the results suggests that DMC accurately can determine ground-state energy using both the instantaneous average ( $\langle \mathcal{E} \rangle$ ) and the average of the accumulated mean ( $\langle E_{\top} \rangle$ ).

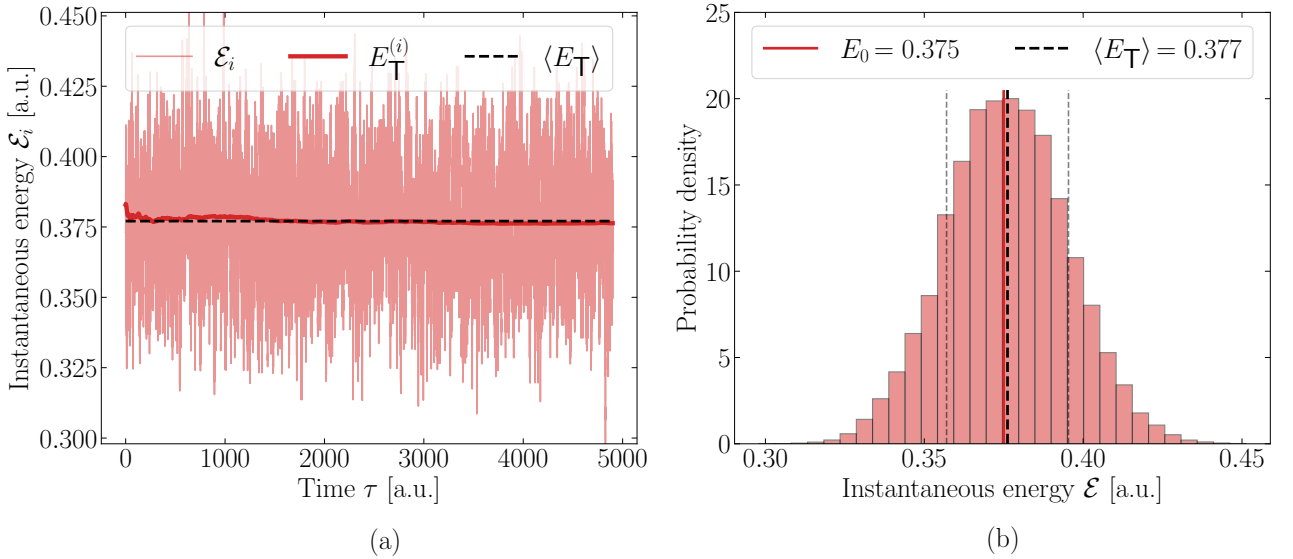


Figure 2: (a) Instantaneous ground-state energy  $\mathcal{E}_i$  in each iteration,  $i$ , of the DMC along with average  $\langle E_{\top} \rangle$ . The analytic ground-state energy  $E_0 = 3/8 = 0.375$  has been plotted for reference. (b) Histogram for instantaneous energy, again with  $\langle E_{\top} \rangle$  and  $E_0$  for reference and additional thin lines for  $\langle E_{\top} \rangle \pm \sigma_{\mathcal{E}}$ .

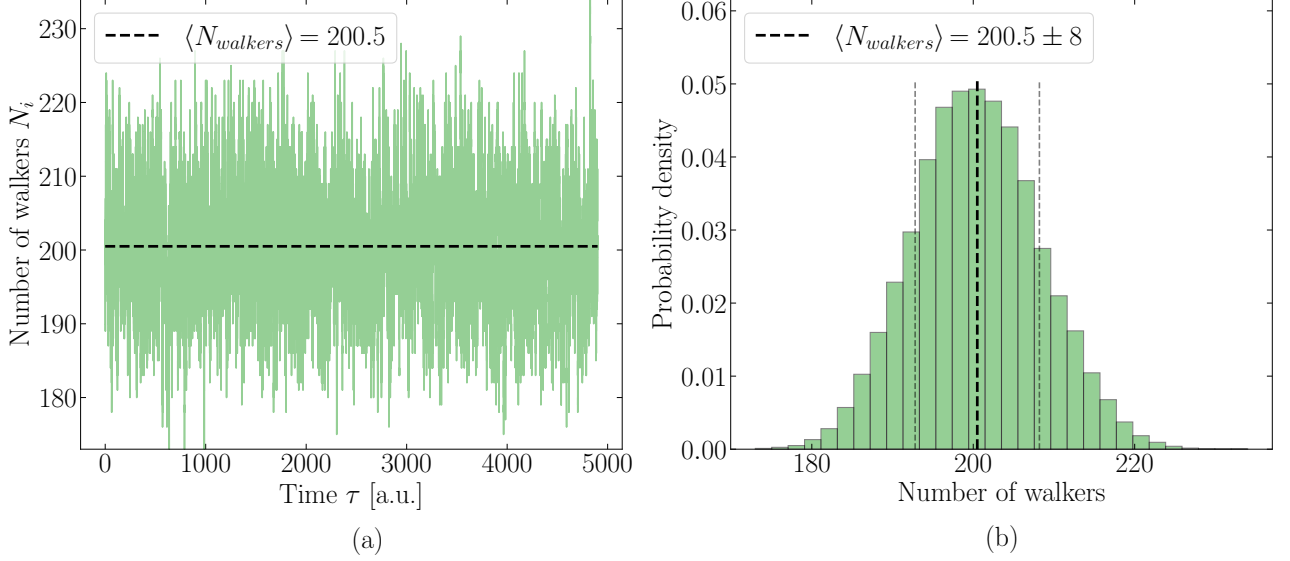


Figure 3: **(a)** Number of walkers  $N_i$  in each iteration of the DMC along with the average  $\langle N_{walkers} \rangle$ . **(b)** Histogram over the number of walkers together with the average  $\langle N_{walkers} \rangle$  as well as additional thin lines for  $\langle N_{walkers} \rangle \pm \sigma_N$ .

Table 1: Estimated ground-state energy using  $\langle E_T \rangle$  and  $\langle \mathcal{E} \rangle$ , along with their standard deviations. The theoretical ground-state energy  $E_0$  was also included. In addition, the average number of walkers  $\langle N_{walkers} \rangle$ , walker standard deviation  $\sigma_N$  and initial walkers  $N_0$  are also available.

Quantity	Label	Value [a.u.]
<i>Avg. of acc. average</i>	$\langle E_T \rangle$	0.3779
<i>Std. of acc. average</i>	$\sigma_{E_T}$	0.015
<i>Avg. inst. energies</i>	$\langle \mathcal{E} \rangle$	0.3763
<i>Std. of inst. energies</i>	$\sigma_{\mathcal{E}}$	0.022
<b><i>Theoretical energy</i></b>	<b><math>E_0</math></b>	<b>0.375</b>
<i>Avg number of walkers</i>	$\langle N_{walkers} \rangle$	201
<i>Std. number of walkers</i>	$\sigma_N$	7.8
<b><i>Initial number of walkers</i></b>	<b><math>N_0</math></b>	<b>200</b>

In contrast to Figure 2 and 3, Figure 4 illustrates the equilibration runs, where Figure 4a contains instantaneous energies  $\mathcal{E}$ , accumulated average  $E_T^{(i)}$  while Figure 4b contains  $N_i$  for  $i < i_{eq}$ . Although the results eventually stabilize around  $E_0$  or  $N_0$  in both cases, the accumulated average energy ( $E_T^{(i)}$ ) takes slightly more iterations to converge. Since  $E_T^{(i)}$  approaches  $E_0$  from above, the strong initial peak in  $E_T^{(i)}$  may cause the slow convergence towards  $E_0$ . Moreover, a similar peak is seen in  $N_i$  (see Figure 4b), which is a direct consequence of the initial walkers positions  $x \in [-5, 5]$ . Considering Equation 1 and 4,  $x < -2$  will give  $W(x) \approx 0$  and thus  $m_j = 0$ . Furthermore, this gives  $N_1 = \sum_j^{N_0} m_j < N_0$ , which according to Equation 5 gives an increased  $\mathcal{E}$ . After the first couple of iterations, the increased  $\mathcal{E}$  will, however, stabilize as all walker positions  $x < -2$  are killed. Finally, the wave function in Figure 5 also resembles this as  $\Psi \equiv 0$  for  $x < -2$ . In conclusion, the model eliminates the non-physical walker positions quite well, and therefore  $\tau_{eq} = 100$  was used in the following tasks.

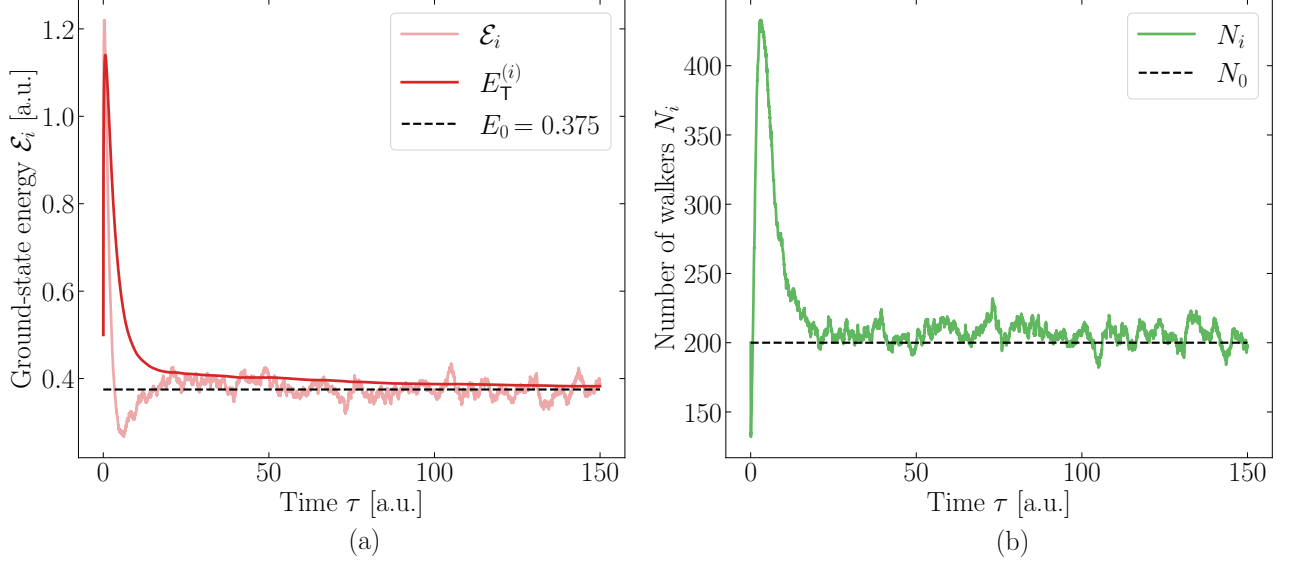


Figure 4: **(a)** Instantaneous energy  $\mathcal{E}_i$  and accumulated average  $E_T^{(i)}$  during the  $i_{eq} = 7500$  equilibration runs. The theoretical  $E_0 = 3/8$  has been plotted for reference. **(b)** Number of walkers  $N_i$  in during the  $i_{eq} = 7500$  equilibration iterations along with the initial number of walkers  $N_0 = 200$ .

Lastly, the sampled wave function, obtained by plotting a histogram of the walker positions  $x_j$ , is given in Figure 5. Similarly, the option `density=True` was used to obtain proper normalization. For reference, the analytically solved wave function, given by

$$\Psi_{\text{Analytic}}(x) = \frac{1}{\sqrt{\pi}} e^{e^{-x} - x/2}, \quad (8)$$

was also included. The pre-factor  $1/\sqrt{\pi}$  in Equation 8 was obtained through the normalization condition, i.e.  $\int_{-\infty}^{\infty} \Psi^* \Psi dx = 1$ . In conclusion, the DMC algorithm yields an accurate estimation of both the energy and the wave function of the ground state.

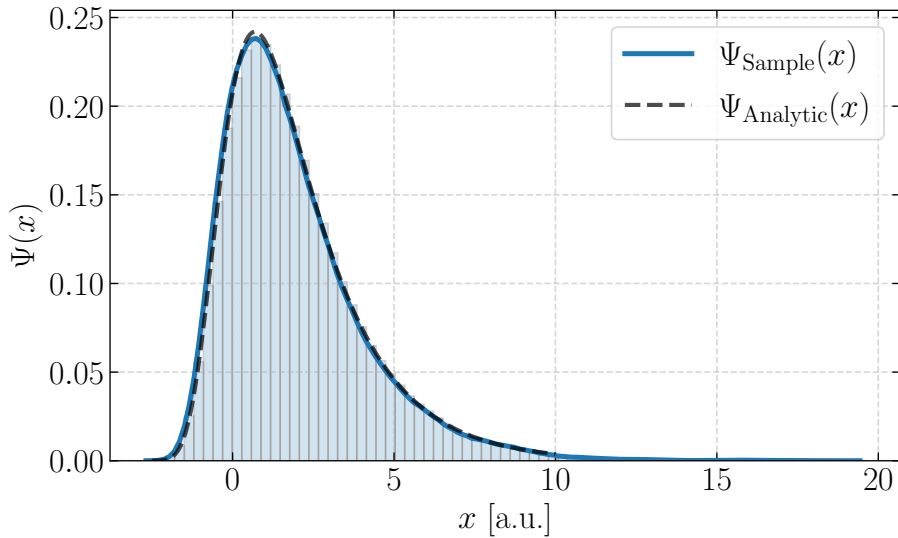


Figure 5: Walker positions  $x$ , resembling the sampled wave function  $\Psi_{\text{Sample}}$  along with the analytically solved wave function  $\Psi_{\text{Analytic}}$ .

## Task 2 – Basic DMC Applied to the Helium Atom

After the basic DMC algorithm was implemented, the same procedure was used to study the Helium atom, with the Hamiltonian

$$\mathcal{H} = \underbrace{-\frac{1}{2}(\nabla_1^2 + \nabla_2^2)}_{\mathcal{H}_1} \underbrace{-\frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}}_{\mathcal{H}_0}, \quad (9)$$

where  $r_1$  and  $r_2$  are the position of the two electrons and  $r_{12} = |\vec{r}_1 - \vec{r}_2|$  the distance between these. The Hamiltonian is conveniently divided into two parts,  $\mathcal{H}_0$  and  $\mathcal{H}_1$ , since only  $\mathcal{H}_0$  is used in the subsequent *death-birth* algorithm. To properly simulate the electrons, the system was generalized in 6 dimensions – 3 for each electron, which gave a  $(N_0 \times 6)$  walker matrix  $\mathbf{w}$ , where  $N_0 = 1000$ . The initial positions were set to

$$\begin{aligned} r_1 &= 0.7 + \mathcal{U}_1 & \theta_1 &= \arccos(2\mathcal{U}_2 - 1) & \varphi_1 &= 2\pi\mathcal{U}_3 \\ r_2 &= 0.7 + \mathcal{U}_4 & \theta_2 &= \arccos(2\mathcal{U}_5 - 1) & \varphi_2 &= 2\pi\mathcal{U}_6, \end{aligned}$$

where  $\mathcal{U}_i$  are independent, uniformly distributed random numbers on  $[0, 1]$ . The conversion to Cartesian coordinates gave

$$\begin{aligned} x_1 &= r_1 \sin \theta_1 \cos \varphi_1 & y_1 &= r_1 \sin \theta_1 \sin \varphi_1 & z_1 &= r_1 \cos \theta_1 \\ x_2 &= r_2 \sin \theta_2 \cos \varphi_2 & y_2 &= r_2 \sin \theta_2 \sin \varphi_2 & z_2 &= r_2 \cos \theta_2. \end{aligned}$$

Each walker, or row in the walker matrix, was thus given by  $\mathbf{w}_j = [x_1 \ y_1 \ z_1 \ x_2 \ y_2 \ z_2]$ . Following the previous procedure, the same condition in the *death-birth* algorithm was used in order to decide which walkers to keep and copy, i.e.

$$m_j = \lfloor e^{(\mathcal{E}_i - \mathcal{H}_0)\Delta\tau} + \mathcal{U} \rfloor. \quad (10)$$

After the surviving walkers were copied, their positions were updated according to Equation 2 with a different random Gaußian number for each coordinate and the energies  $\mathcal{E}_i$  and  $E_T^{(i)}$  was calculated according to Equations 5, 6 and 7, again using  $\gamma = 0.5$ . Moreover, an initial guess of  $\mathcal{E}_0 = -3$  and the time-step  $\Delta\tau = 0.01$  was used. The lower time-step was needed here since the algorithm was rather unstable without importance sampling. DMC was then performed for  $\tau = 5\,000$  or  $5\,000/0.01 = 500\,000$  iterations following 10\,000 equilibration iterations ( $\tau_{eq} = 100$ ); 2\,500 more than in Task 1 due to the shorter  $\Delta\tau$ .

Similar to Task 1, Figure 6a contains instantaneous energies  $\mathcal{E}$ , accumulated average  $E_T^{(i)}$  and the average  $\langle E_T \rangle = -2.87$ . In addition, Figure 6b contains a histogram of the instantaneous energies. Table 2 displays averages and standard deviations for the energies and walkers. Compared to Task 1, the instantaneous energies have a higher standard deviation, while the average  $\langle E_T \rangle$  is still somewhat close to the experimentally found  $E_0 = -2.903$  [2]. This is reasonable since the lack of importance sampling yields less effective control of the fluctuations in the sampled energies, resulting in more variability. We also notice that the average number of walkers here has a considerable larger deviation than in the previous task. This is expected for the same reason as above, since DMC without importance sampling of the Helium atom is unstable, e.g. due to the singularity when the electrons move very close to each other. As we will see in the next task, one can stabilize the algorithm by introducing a function that mimics the real wave function, i.e. importance sampling.

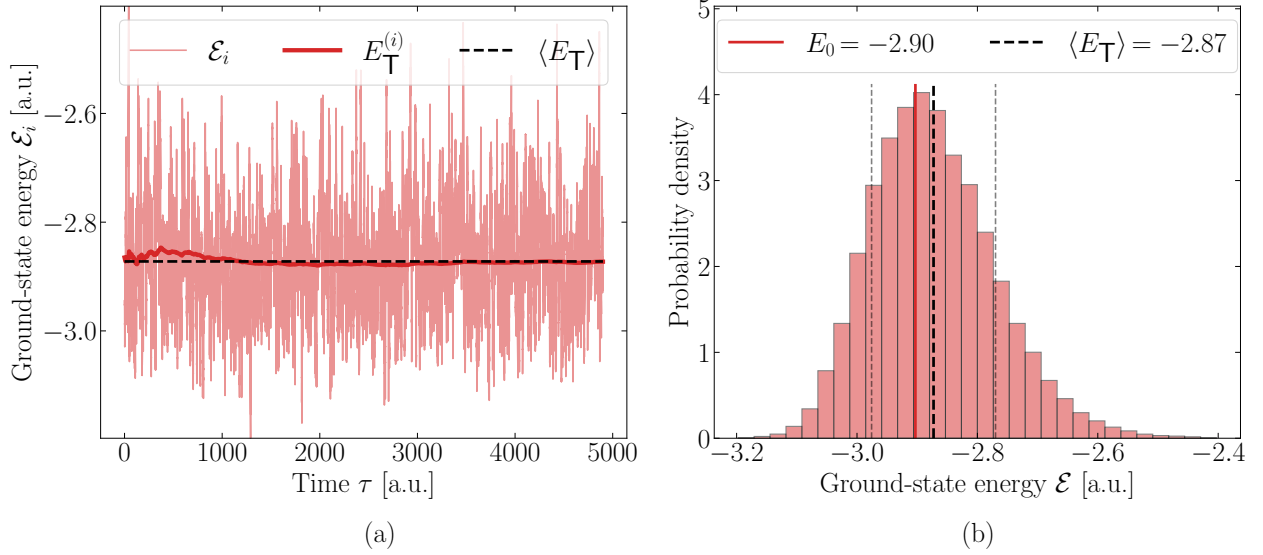


Figure 6: **(a)** Instantaneous ground-state energy  $\mathcal{E}_i$  and accumulated average  $E_T^{(i)}$  in each iteration,  $i$ , of the DMC simulation of Helium without importance sampling along with average  $\langle E_T \rangle$ . The analytic ground-state energy  $E_0 = -2.903$  has been plotted for reference. **(b)** Histogram for instantaneous energy, again with  $\langle E_T \rangle$  and  $E_0$  for reference and additional thin lines for  $\langle E_T \rangle \pm \sigma_{\mathcal{E}}$ .

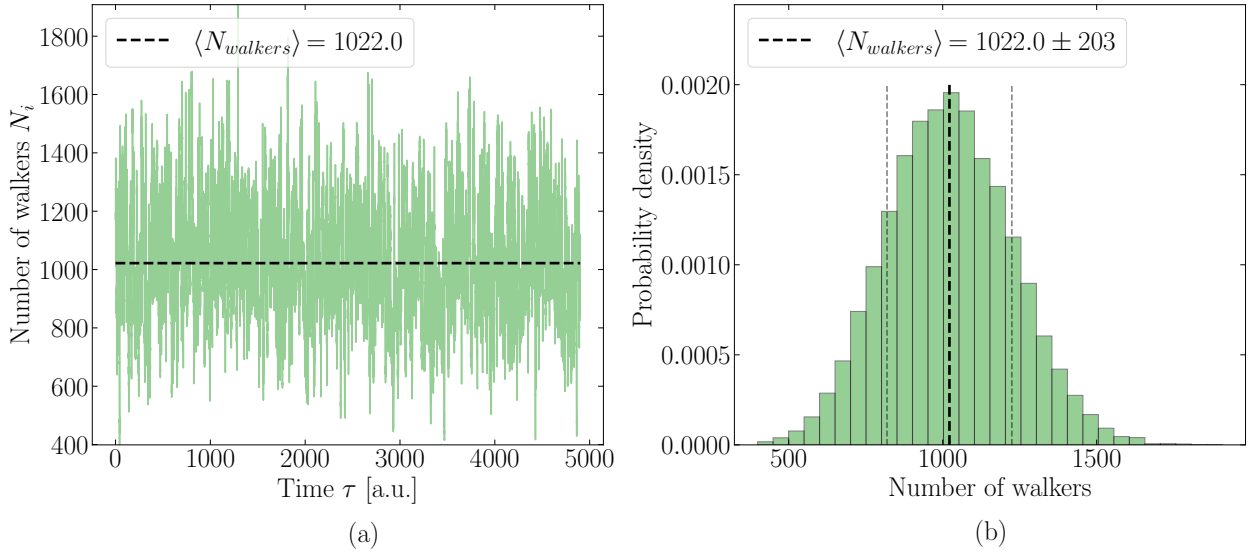


Figure 7: **(a)** Number of walkers  $N_i$  in each iteration of the DMC simulation of Helium without importance sampling along with the average  $\langle N_{walkers} \rangle$ . **(b)** Histogram over the number of walkers together with the average  $\langle N_{walkers} \rangle$  as well as additional thin lines for  $\langle N_{walkers} \rangle \pm \sigma_N$ .



Table 2: Estimated ground-state energy using  $\langle E_T \rangle$  and  $\langle \mathcal{E} \rangle$ , along with their standard deviations. The theoretical ground-state energy  $E_0$  is also included. In addition, the average number of walkers  $\langle N_{walkers} \rangle$ , walker standard deviation  $\sigma_N$  and initial walkers  $N_0$  are also available.

Quantity	Label	Value [a.u.]
<i>Avg. of acc. average</i>	$\langle E_T \rangle$	-2.8725
<i>Std. of acc. average</i>	$\sigma_{E_T}$	0.010
<i>Avg. inst. energies</i>	$\langle \mathcal{E} \rangle$	-2.8727
<i>Std. of inst. energies</i>	$\sigma_{\mathcal{E}}$	0.10
<b><i>Theoretical energy</i></b>	<b><math>E_0</math></b>	<b>-2.903</b>
<i>Avg number of walkers</i>	$\langle N_{walkers} \rangle$	1022
<i>Std. number of walkers</i>	$\sigma_N$	203
<b><i>Initial walkers</i></b>	<b><math>N_0</math></b>	<b>1000</b>

### Task 3 – DMC with Importance Sampling

To increase the model’s efficiency, *importance sampling* (IS) was implemented by introducing  $\Psi_T$ , a wave function similar to the real wave function of the Helium system. From  $\Psi_T$ , an expression for the local energy,  $E_L$ , is given by

$$E_L = -4 + \frac{(\hat{r}_1 - \hat{r}_2) \cdot (\vec{r}_1 - \vec{r}_2)}{r_{12}(1 + \alpha r_{12})^2} - \frac{1}{r_{12}(1 + \alpha r_{12})^3} - \frac{1}{4r_{12}(1 + \alpha r_{12})^4} + \frac{1}{r_{12}}, \quad (11)$$

where  $\hat{r}_k = \vec{r}_k/r_k$  is a unit vector for  $k = 1, 2$  and  $\alpha = 0.15$  is a parameter found through Variational Monte Carlo.  $E_L$  was used instead of the potential, giving the new criterion

$$m_j = \lfloor e^{(\mathcal{E}_i - E_L)\Delta\tau} + \mathcal{U} \rfloor, \quad (12)$$

to determine if walker  $j$  was killed/copied. In addition to introducing  $E_L$ , drift velocities

$$\begin{aligned} \vec{v}_{F1}(\vec{r}_1) &= -2\hat{r}_1 - \frac{\hat{r}_{12}}{2(1 + \alpha r_{12})^2} \quad \text{for } \vec{r}_1 = \begin{bmatrix} x_1 & y_1 & z_1 \end{bmatrix}, \\ \vec{v}_{F2}(\vec{r}_2) &= -2\hat{r}_2 + \frac{\hat{r}_{12}}{2(1 + \alpha r_{12})^2} \quad \text{for } \vec{r}_2 = \begin{bmatrix} x_2 & y_2 & z_2 \end{bmatrix} \end{aligned}$$

were introduced, and used to update the positions of the surviving walkers. These velocities are essentially used to “push” the walkers to more important regions of the energy landscape and arises from a third part of the short-time propagator introduced due to the local energy (the others being the reactive and diffusive parts). This third part can then be simplified using two different decompositions.

## First Decomposition

In the first decomposition, positions of surviving walkers were updated using

$$\vec{r}_k^{(new)} = \vec{v}_{Fk}(\vec{r}_k^{(old)})\Delta\tau + \vec{r}_k^{(old)},$$

where  $k = 1, 2$ . Similar to Task 2,  $N_0 = 1\,000$  walkers were initialized and  $E_{\top}^{(0)} = -3.0$ . The simulation was run for  $\tau = 5\,000$ , which gave  $\tau/\Delta\tau = 5\,000/0.01 = 500\,000$  iterations. Again, an equilibration was run for  $\tau = 100$  or  $10\,000$  iterations.

The resulting samples, e.g. instantaneous energies, accumulated average energies and the average of the accumulated average, are available in Figure 8a, while Figure 8b contains a histogram of the instantaneous energies along with the average of the accumulated average  $\langle E_{\top} \rangle = -2.862$  and the theoretical value  $E_0 = -2.903$ . Although the instantaneous energies  $\mathcal{E}$  (Figure 8b) are rather narrow distributed with a standard deviation of  $\sigma_{\mathcal{E}} = 0.0094$ , the samples are slightly (0.04) off compared to the theoretical  $E_0$ . Finally, the number of walkers in each iteration along with a histogram are given in Figure 9. Here, the average  $\langle N_{walkers} \rangle$  is stable at  $N_0 = 1\,000$  and a standard deviation of  $\sigma_N = 19$ .

In this case, each walkers contained two electron positions or six coordinates, and thus six probability distributions are given in Figure 10. Comparing the two electrons, there is visually no difference in their distributions, which is an expected result since both electrons are occupying the  $1s^2$  state. For the  $r$ -coordinate, there is a maximum at  $r \approx 0.55 = 0.55 a_0 \approx 0.29 \text{ \AA}$ ; quite close to  $r_{max} = 0.569 a_0$  [3]. The distribution for the polar angular coordinates,  $\theta$ , suggests that the electrons have a higher probability of being in the  $x, y$ -plane ( $\theta = \pi/2$ ). And, the azimuthal angles  $\phi$  are, conversely, uniformly distributed, meaning that the system is symmetric along the  $z$ -axis.

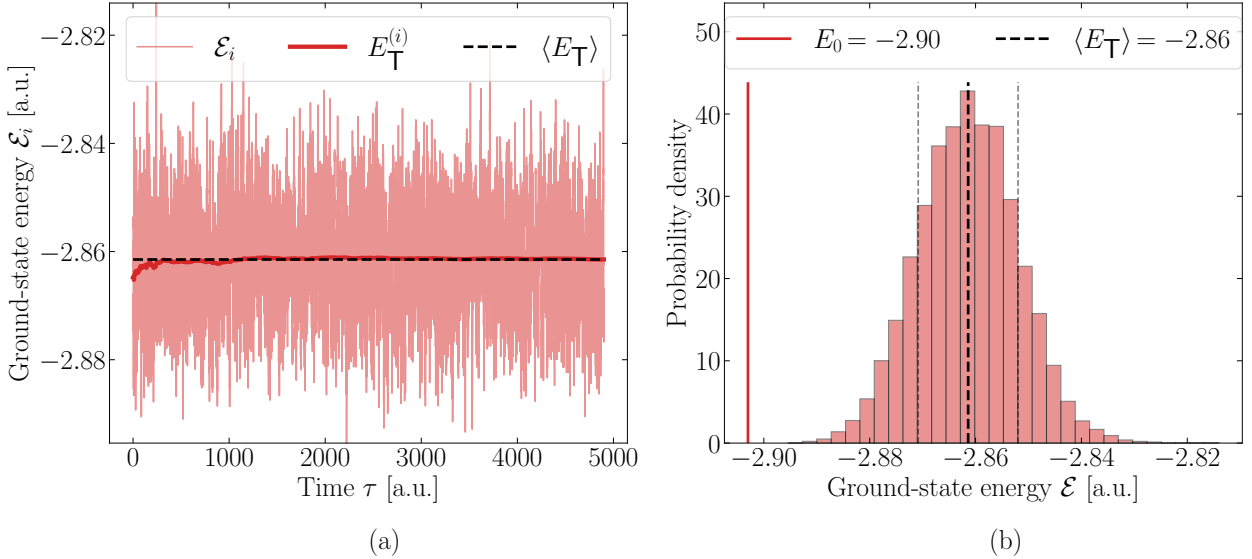


Figure 8: **(a)** Instantaneous ground-state energy  $\mathcal{E}_i$  and accumulated average  $E_{\top}^{(i)}$  in each iteration,  $i$ , of the DMC simulation of Helium with IS along with average  $\langle E_{\top} \rangle$ . Here, the more accurate decomposition, i.e. decomposition 2, of the short-time propagator was used. The analytic ground-state energy  $E_0 = -2.903$  has been plotted for reference. **(b)** Histogram for instantaneous energy, again with  $\langle E_{\top} \rangle$  and  $E_0$  for reference and additional thin lines for  $\langle E_{\top} \rangle \pm \sigma_{\mathcal{E}}$ .

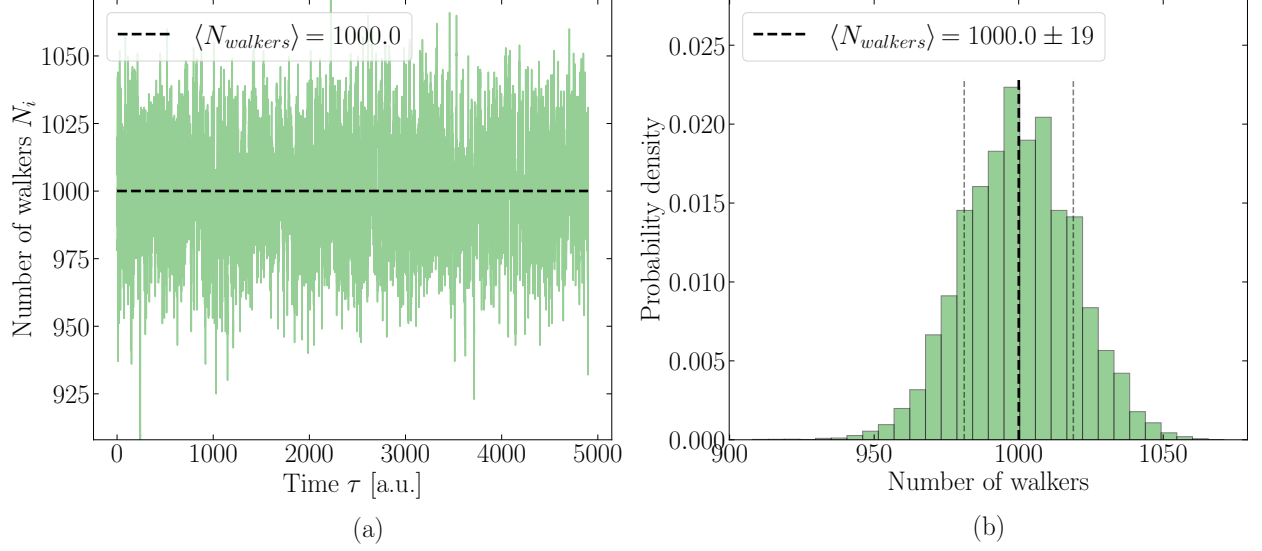


Figure 9: **(a)** Number of walkers  $N_i$  in each iteration of the DMC simulation of Helium with IS along with the average  $\langle N_{walkers} \rangle$ . Here, the more accurate decomposition, i.e. decomposition 2, of the short-time propagator was used. **(b)** Histogram over the number of walkers together with the average  $\langle N_{walkers} \rangle$  as well as additional thin lines for  $\langle N_{walkers} \rangle \pm \sigma_N$ .

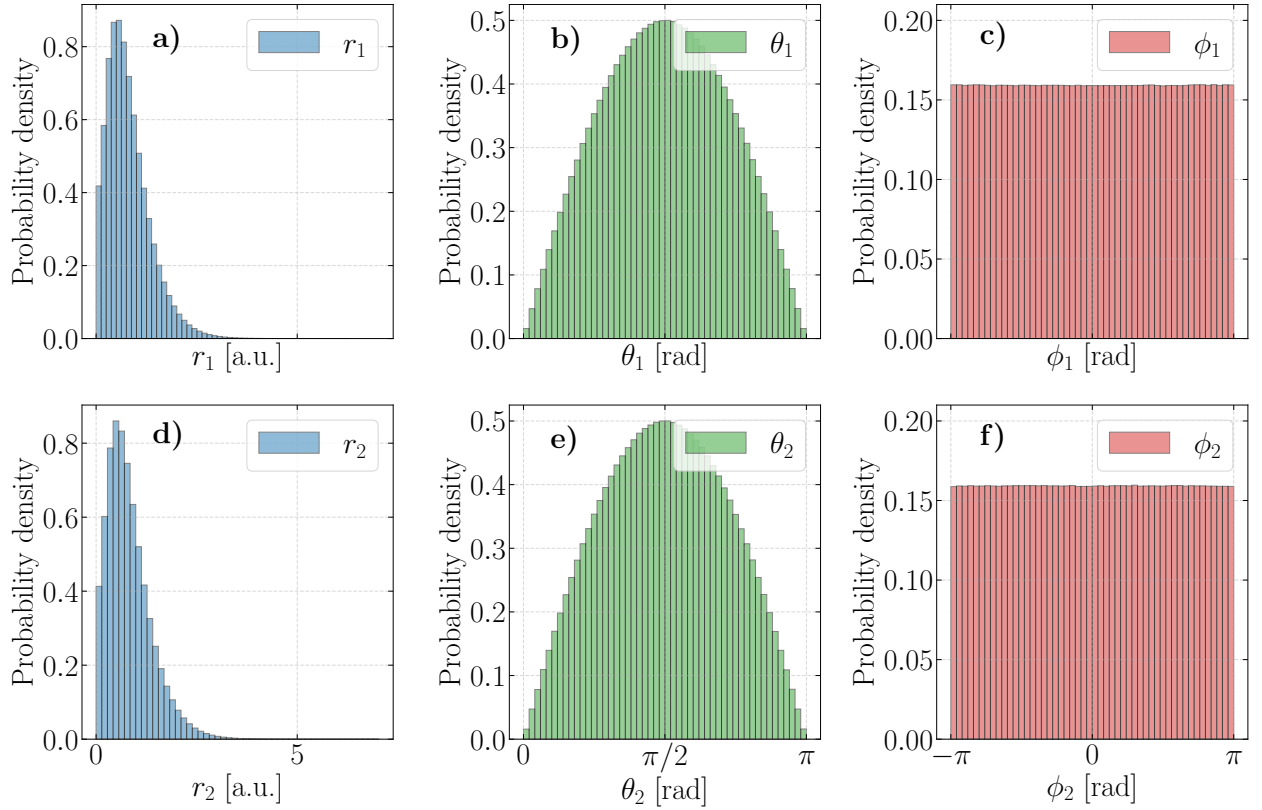


Figure 10: Probability distributions of the spherical coordinate components for the first **(a)** - **(c)**,  $(r_1, \theta_1, \phi_1)$ , and second **(d)** - **(f)** electron,  $(r_2, \theta_2, \phi_2)$  in the Helium atom. Here, decomposition 1 of the short-time propagator was used in the importance sampling.

## Second Decomposition

In the second decomposition, positions of surviving walkers were updated in two steps:

$$\begin{aligned} (i) \quad \vec{r}_k^{(int.)} &= \vec{v}_{Fk}(\vec{r}_k^{(old)})\Delta\tau/2 + \vec{r}_k^{(old)} \\ (ii) \quad \vec{r}_k^{(new)} &= \vec{v}_{Fk}(\vec{r}_k^{(int.)})\Delta\tau + \vec{r}_k^{(old)} \end{aligned}$$

where  $\vec{r}_k^{(int.)}$  is an intermediate half-step and  $k = 1, 2$ . Again,  $N_0 = 1000$  walkers were initialized and  $\mathcal{E}_0 = -3.0$ . The simulation was run for  $\tau = 5000$  or  $\tau/\Delta\tau = 5000/0.01 = 500000$  iterations with  $\tau_{eq} = 100$  or 10000 equilibrium iterations.

The resulting samples for the second decomposition, e.g. instantaneous energies, accumulated average energies and the average of the accumulated average, are available in Figure 11a, while Figure 11b contains a histogram of the instantaneous energies along with the average of the accumulated average  $\langle E_T \rangle = -2.911$  and the theoretical value  $E_0 = -2.903$ . In comparison to Figure 8b, the energy distribution Figure 11b are still shifted compared to  $E_0$ , however, by  $< 0.01$ , which suggests that the second decomposition is an improvement over the first. Besides,  $E_0$  falls in the range  $\langle E_T \rangle \pm \sigma_{\mathcal{E}}$ , where  $\sigma_{\mathcal{E}} = 0.0093$ . Finally, the number of walkers in each iteration along with a histogram is given in Figure 12a and Figure 12b. Here,  $N$  had a standard deviation of  $\sigma_N = 19$ , which is a large improvement from the case without IS. The average  $\langle N_{walkers} \rangle = 999$  falls one short of the initial  $N_0 = 1000$ , which is reasonable.

Similar to the first decomposition, the six probability distributions connected to each walker are given in Figure 13. Comparing the two decompositions, there is little-to-no difference in their distributions visually. The second decompositions also produces a  $r_{max} \approx 0.55$  [a.u.]  $= 0.55 a_0 \approx 2.9 \text{ \AA}$ . The angular coordinates ( $\theta$  and  $\phi$ ) also have an almost identical shape compared to what is found in Figure 10. While some improvements were seen in the energy estimations, there is no significant differences in the walker distributions (wave functions) were obtained by changing to the second decomposition.

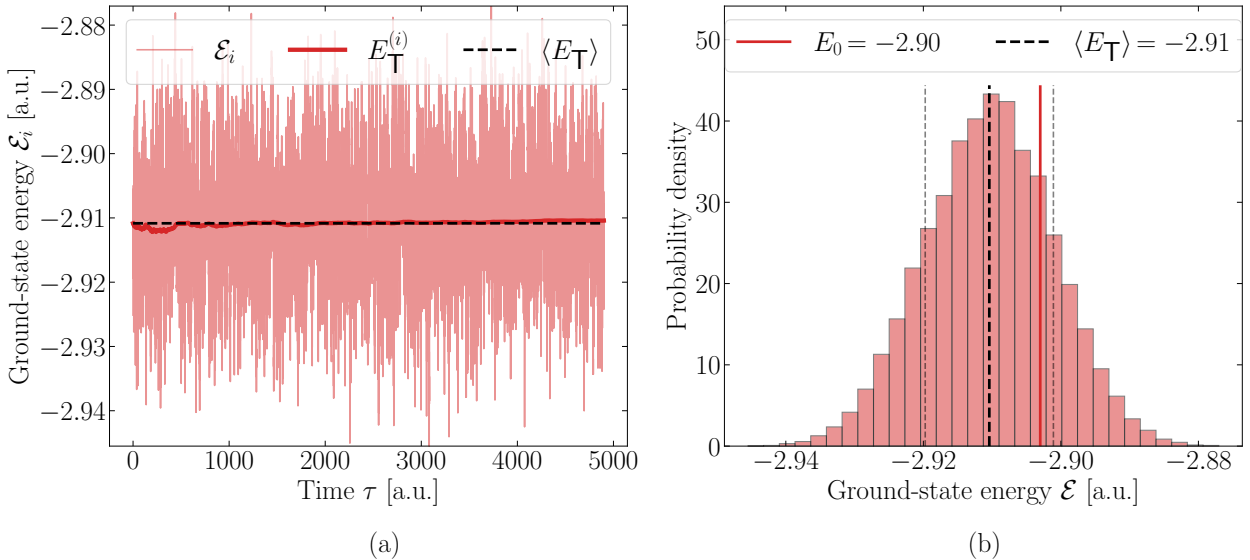


Figure 11: **(a)** Instantaneous ground-state energy  $\mathcal{E}_i$  in each iteration,  $i$ , of the DMC simulation of Helium with IS along with average  $\langle E_T \rangle$ . Here, the second decomposition of the short-time propagator was used. The analytic ground-state energy  $E_0 = -2.903$  has been plotted for reference. **(b)** Histogram for instantaneous energy, again with  $\langle E_T \rangle$  and  $E_0$  for reference and additional thin lines for  $\langle E_T \rangle \pm \sigma_{\mathcal{E}}$ .

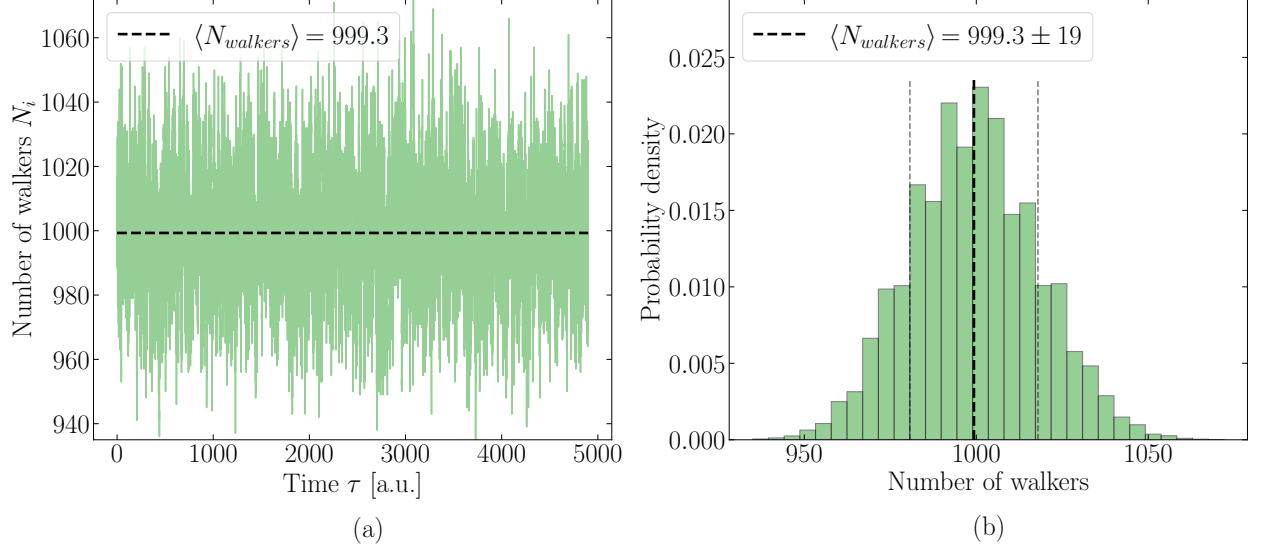


Figure 12: **(a)** Number of walkers  $N_i$  in each iteration of the DMC simulation of Helium with IS along with the average  $\langle N_{walkers} \rangle$ . Here, the second decomposition of the short-time propagator was used. **(b)** Histogram over the number of walkers together with the average  $\langle N_{walkers} \rangle$  as well as additional thin lines for  $\langle N_{walkers} \rangle \pm \sigma_N$ .

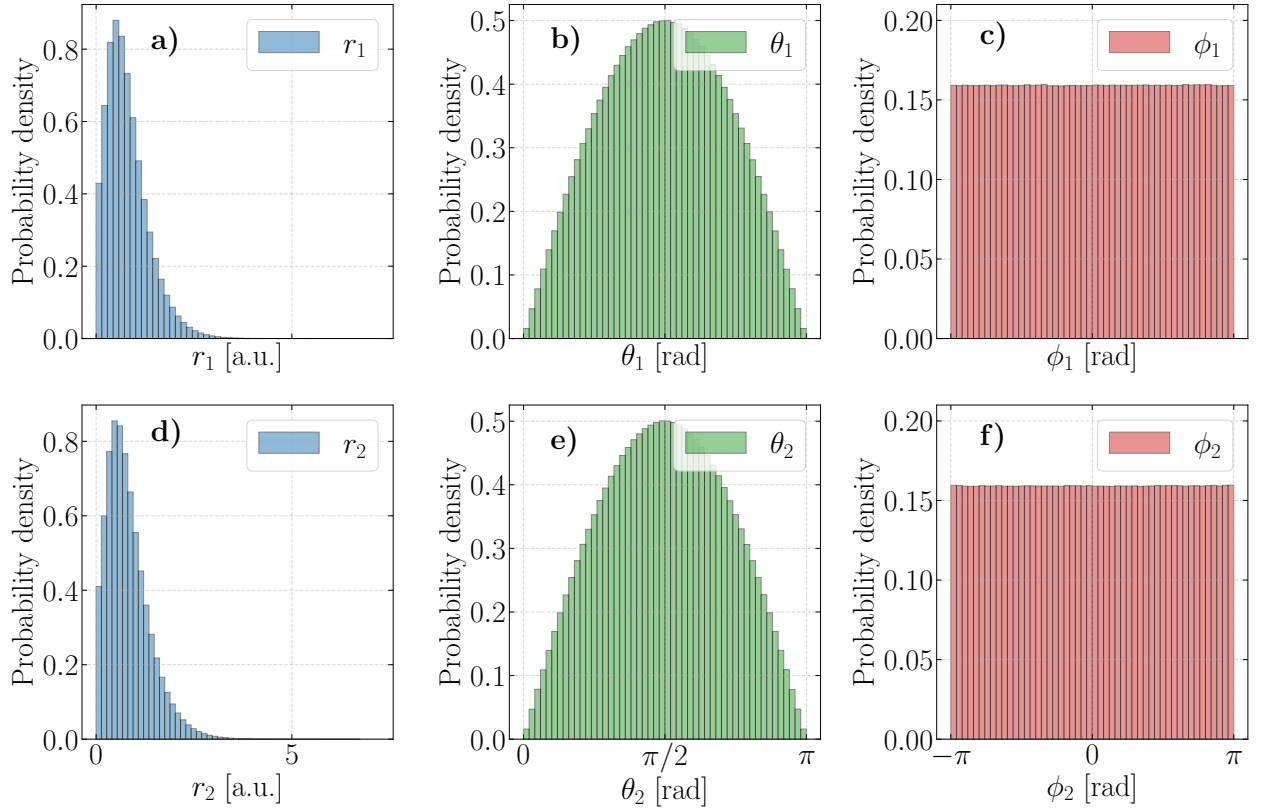


Figure 13: Probability distributions of the spherical coordinate components for the first **(a)** – **(c)**,  $(r_1, \theta_1, \phi_1)$ , and second electron **(d)** – **(f)**,  $(r_2, \theta_2, \phi_2)$  in the Helium atom. Here, the more accurate decomposition, i.e. decomposition 2, of the short-time propagator was used in the importance sampling.

## Comparison and Additional Discussion

To conclude, averages for energy estimations and number of walkers along with their corresponding standard deviations are found in Table 3. As previously mentioned, there is a slight improvement in energy estimation using the second decomposition.

Table 3: Side-by-side comparison of estimated ground-state energy using  $\langle E_T \rangle$  and  $\langle \mathcal{E} \rangle$ , along with their standard deviations for both decompositions. The theoretical ground-state energy  $E_0$  is also included. In addition, the average number of walkers  $\langle N_{walkers} \rangle$ , walker standard deviation  $\sigma_N$  and initial walkers  $N_0$  are also available.

Quantity	Label	1 <sup>st</sup> Decomp. [a.u.]	2 <sup>nd</sup> Decomp. [a.u.]
<i>Avg. of acc. average</i>	$\langle E_T \rangle$	−2.8616	−2.9109
<i>Std. of acc. average</i>	$\sigma_{E_T}$	0.0010	0.00051
<i>Avg. inst. energies</i>	$\langle \mathcal{E} \rangle$	−2.8615	−2.9104
<i>Std. of inst. energies</i>	$\sigma_{\mathcal{E}}$	0.0095	0.0093
<b><i>Theoretical energy</i></b>	<b><math>E_0</math></b>	<b>−2.903</b>	<b>−2.903</b>
<i>Avg number of walkers</i>	$\langle N_{walkers} \rangle$	999.9	999.3
<i>Std. number of walkers</i>	$\sigma_N$	18.9	18.6
<b><i>Initial walkers</i></b>	<b><math>N_0</math></b>	<b>1000</b>	<b>1000</b>

Additionally, when considering the accumulated mean  $E_T^{(i)}$  in Figure 11a, there are small fluctuations for  $\tau < 1000$ , which seems to slightly affect the energy estimation  $\langle E_T \rangle$  in the negative direction. Looking forward, an even longer equilibration time, for instance  $\tau_{eq} = 1000$ , should resolve this issue and perhaps give a small improvement in the energy estimation. Although similar fluctuations are seen in both Figure 6 and 8, running longer equilibration will probably not improve the energy estimation, since  $E_T^{(i)}$  does not converge to  $E_0$  in these cases.

A final comparison is given in Figure 14, where histograms for  $r_{12}$  using the first (a) and second decomposition (b). The similarities emphasize again that the decompositions produce almost identical distributions of walkers.

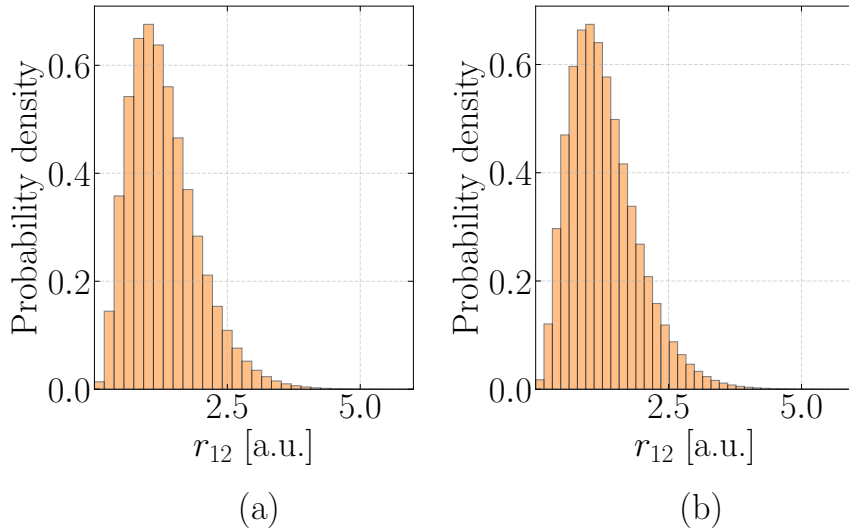


Figure 14: Probability distributions of the distance  $r_{12}$  between the two electrons in Helium during the DMC simulations with IS using decomposition 1 (a) and 2 (b).

## Task 4 – Extrapolation in $\Delta\tau$

In the last task, the ground-state energy was calculated by running the DMC with different time-steps  $\Delta\tau \in \{0.01, 0.05, 0.075, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4\}$  using the two decompositions from Task 3. The corresponding plots,  $\langle E_T \rangle(\Delta\tau)$ , are given in Figure 15a for the first decomposition and Figure 15b for the second. Examining Figure 15a, the first decomposition has a positive correlation between  $\langle E_T \rangle$  and  $\Delta\tau$ , thus, a linear fit  $E_T = 0.656\Delta\tau - 2.926$  was made. From the linear fit, an optimal  $E_T^* = -2.926$  was estimated by  $\Delta\tau \rightarrow 0$ . A similar approach was used to determine an optimal value for  $E_T$  with the second decomposition (Figure 15b), however, in this case using a quadratic fit ( $E_T = -0.815\Delta\tau^2 + 0.295\Delta\tau - 2.933$ ) as  $\langle E_T \rangle$  depended quadratically on  $\Delta\tau$ . Again, letting  $\Delta\tau \rightarrow 0$  gives  $E_T^* = -2.933$

Another backwards-engineering approach is to start with the theoretical  $E_0 = -2.903$ , and then determining the optimal time-step. In the first decomposition, a time-step of  $\Delta\tau \approx 0.035$  should give energy close to  $E_0$ . On the contrary, the second decomposition's extrapolated curve does not even reach the theoretical value  $E_0 = -2.903$ , however, a time-step of  $\Delta\tau \approx 0.18$  should give the best estimate. Interestingly, a short time-step — more iterations — does not produce a more accurate energy estimate, especially in the second decomposition.

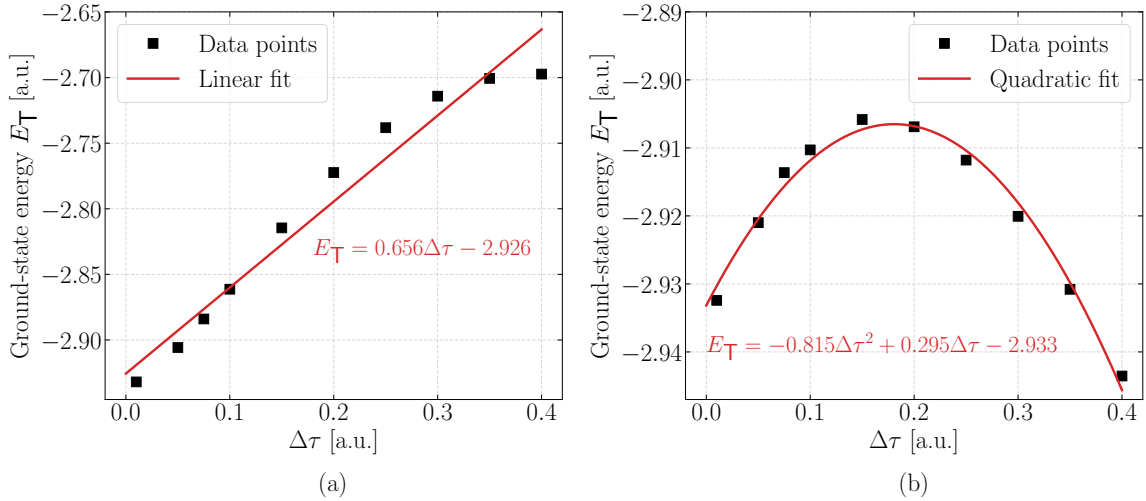


Figure 15: The total average of the ground state energy of Helium,  $E_T$ , for different time-steps  $\Delta\tau$  using decomposition 1 (a) and 2 (b) along with a curve fit for both cases.

## Conclusion & Comparison

In summary, the ground-state energy and wave function of the electrons in a He atom have been obtained using DMC with and without importance sampling. Furthermore, two decompositions were used in to update the positions of individual walkers. The second decomposition, which had an intermediate step, gave a slightly more accurate ground-state energy, while the wave functions were almost identical. In the future, we suggest running even more equilibration runs, and perhaps even more iterations in total before taking the average to estimate the ground-state energy. This would definitely eliminate any fluctuations in the accumulated average. Lastly, taken into account the lack of computational power (and storage) in our personal laptops, we are satisfied with the results, which at best gave less than 0.01 away from the experimentally found value ( $\langle E_T \rangle = -2.911$  compared to  $E_0 = -2.903$ ).

*Finally, we want to thank all the fallen walkers for their service. RIP.*

## References

- [1] B. F. Ioan Kosztin and K. Schulten, *Introduction to the Diffusion Monte Carlo Method*, 1110 West Green Street, Urbana, Illinois 61801. [Online]. Available: <https://www.thphys.uni-heidelberg.de/~wetzels/qmc2006/KOSZ96.pdf> (visited on 12/29/2024).
- [2] A. Kramida, Yu. Ralchenko, J. Reader, and NIST ASD Team, NIST Atomic Spectra Database (ver. 5.12), Gaithersburg, MD., 2024. DOI: <https://doi.org/10.18434/T4W30F>. [Online]. Available: [https://physics.nist.gov/cgi-bin/ASD/ie.pl?spectra=He&submit=Retrieve+Data&units=3&format=0&order=0&at\\_num\\_out=on&sp\\_name\\_out=on&ion\\_charge\\_out=on&el\\_name\\_out=on&seq\\_out=on&shells\\_out=on&level\\_out=on&ion\\_conf\\_out=on&e\\_out=1&unc\\_out=on&biblio=on](https://physics.nist.gov/cgi-bin/ASD/ie.pl?spectra=He&submit=Retrieve+Data&units=3&format=0&order=0&at_num_out=on&sp_name_out=on&ion_charge_out=on&el_name_out=on&seq_out=on&shells_out=on&level_out=on&ion_conf_out=on&e_out=1&unc_out=on&biblio=on).
- [3] J. B. Mann, *Atomic Structure Calculations II. Hartree-Fock Wavefunctions and Radial Expectation Values: Hydrogen to Lawrencium*. Los Alamos, New Mexico: Los Alamos Scientific Laboratory of the University of California, 1968. [Online]. Available: <https://www.osti.gov/servlets/purl/4553157/> (visited on 01/10/2025).



# Appendix

## A Source code in C

### A.1 run.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5  #include <gsl/gsl_rng.h>
6  #include <gsl/gsl_randist.h>
7  #include "tools.h"
8  #include "run.h"
9
10 int
11 run(
12     int argc,
13     char *argv[]
14 )
15 {
16     // ----- Constants ----- //
17     gsl_rng *U = init_gsl_rng(19); // Random number generator
18     double gamma = 0.5; // Energy damping factor
19     DMC_results results; // Results struct
20     char filename[100]; // Filename
21
22     // ----- Task 1a ----- //
23     int N_0 = 200; // Initial number of walkers
24     double dtau = 0.02; // Time step
25     double tau = 5000; // Total time
26     int N_its = tau / dtau; // Number of iterations
27     double E_T_start = 0.5; // Initial energy
28     int its_eq = 20 / dtau; // Number of equilibration steps
29
30     // Initialize the walkers
31     double **walkmen_pos = create_2D_array(N_0 * 100, 1);
32     init_walkmen_1D(walkmen_pos, N_0);
33
34     // Equilibrate the walkers
35     results = DMC(its_eq, N_0, N_0+1, dtau, E_T_start, U, gamma, NULL, NULL, 1, false,
36     ↪ 1);
37     E_T_start = results.E_T;
38     int N_sprinters = results.N_sprinters;
39
40     // Run the simulation
41     FILE* fp_ET = fopen("data/task_1/1D/ET_Nwalk_non_eq.csv", "w");
42     fprintf(fp_ET, "E_T_avg, N_sprinters, N_survived/N_sprinters (%%), E_T\n");
43     FILE* fp_w = fopen("data/task_1/1D/I_was_walkin_in_morse.csv", "w");
44
45     DMC(N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma, fp_ET, fp_w, 1, false, 1);
46
47     // Close the files
48     fclose(fp_ET);
49     fclose(fp_w);
50 }
```

```

49
50 // ----- Task 1b ----- //
51 int N_0 = 1000; // Initial number of walkers
52 int N_sprinters_init = N_0; // Initial number of sprinters
53 double dtau = 0.01; // Time step
54 double tau = 5000; // Total time
55 int N_its = tau / dtau; // Number of iterations
56 double E_T_start_init = -3; // Initial energy
57 int its_eq = 100 / dtau; // Number of equilibration steps
58
59 // Initialize the walkers
60 double **walkmen_pos_init = create_2D_array(N_0 * 10, 6);
61 init_walkmen_6D(walkmen_pos_init, N_0, U);
62
63 // Equilibrate the walkers
64 results = DMC(walkmen_pos_init, its_eq, N_0, N_sprinters_init, dtau, E_T_start_init,
65 ↪ U, gamma, NULL, NULL, 6, false, 1);
66 double E_T_start = results.E_T;
67 int N_sprinters = results.N_sprinters;
68
69 // Run the simulation
70 sprintf(filename, "data/task_1/6D/ET_Nwalk_non_eq.csv");
71 FILE* fp_ET = fopen(filename, "w");
72 fprintf(fp_ET, "E_T_avg, N_sprinters, N_survived/N_sprinters (%%), E_T\n");
73 FILE* fp_w = fopen("data/task_1/6D/I_was_walkin_in_morse.csv", "w");
74
75 results = DMC(walkmen_pos_init, N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma,
76 ↪ fp_ET, NULL, 6, false, 1);
77
78 // Close the files
79 fclose(fp_ET);
80 fclose(fp_w);
81
82 // ----- Task 2a ----- //
83 dtau = 0.1; // Time step
84 its_eq = 1000 / dtau; // Number of equilibration steps
85 int decomp = 2; // Decomposition
86 N_its = 50000; // Number of iterations
87
88 // Equilibrate the walkers
89 results = DMC(its_eq, N_0, N_sprinters_init, dtau, E_T_start_init, U, gamma, NULL,
90 ↪ NULL, 6, true, decomp);
91 double E_T_start = results.E_T;
92 int N_sprinters = results.N_sprinters;
93
94 // Run the simulation
95 sprintf(filename, "data/task_2/ET_Nwalk_non_eq_decomp_%i_dtau_%f.csv", decomp, dtau);
96 FILE *fp_ET_IS = fopen(filename, "w");
97 fprintf(fp_ET_IS, "E_T_avg, N_sprinters, N_survived/N_sprinters (%%), E_T\n");
98
99 sprintf(filename, "data/task_2/I_was_walkin_in_morse_decomp_%i_dtau_%f.csv", decomp,
100 ↪ dtau);
101 FILE* fp_w_IS = fopen(filename, "w");
102 results = DMC(N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma, fp_ET_IS, fp_w_IS,
103 ↪ 6, true, decomp);
104
105 // Close the files

```

```

101 fclose(fp_ET_IS);
102 fclose(fp_w_IS);
103
104 // ----- Task 2b ----- //
105 double dtaus[10] = {0.01, 0.05, 0.075, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4}; // Time
    ↪ steps
106 decomp = 1; // Decomposition
107
108 // File to store the results
109 sprintf(filename, "data/task_2/ET_Nwalks_%i.csv", decomp);
110 FILE *fp_ET_IS = fopen(filename, "w");
111 fprintf(fp_ET_IS, "dtau, E_T_avg\n");
112
113 // Run the simulation
114 for (int i = 0; i < 10; i++)
115 {
116     dtau = dtaus[i]; // Time step
117     printf("dtau: %f\n", dtau);
118
119     // Equilibrate the walkers
120     results = DMC(its_eq, N_0, N_sprinters_init, dtau, E_T_start_init, U, gamma,
    ↪ NULL, NULL, 6, true, decomp);
121     double E_T_start = results.E_T;
122     int N_sprinters = results.N_sprinters;
123
124     // Run the simulation
125     results = DMC(N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma, NULL, NULL, 6,
    ↪ true, decomp);
126     fprintf(fp_ET_IS, "%f, %f\n", dtau, results.E_T_avg);
127     printf("E_T_avg: %f\n", results.E_T_avg);
128 }
129
130 // Close the file
131 fclose(fp_ET_IS);
132
133 return 0;
134 }
135
136 DMC_results
137 DMC(
138     int N_its,
139     int N_0,
140     int N_sprinters,
141     double dtau,
142     double E_T_start,
143     gsl_rng *U,
144     double gamma,
145     FILE *fp_ET,
146     FILE *fp_w,
147     int ndim,
148     bool IS,
149     int decomp
150 )
151 {
152     double E_T = E_T_start; // Initial energy
153     double E_T_avg = E_T_start; // Initial average energy
154     DMC_results results; // Results struct

```

```

155     int N_survivors = N_sprinters; // Number of surviving walkers
156     double **walkmen_pos = create_2D_array(N_sprinters, ndim); // Walkers positions
157     // Initialize the walkers
158     if (ndim == 1)
159     {
160         init_walkmen_1D(walkmen_pos, N_sprinters);
161     }
162     else
163     {
164         init_walkmen_6D(walkmen_pos, N_sprinters, U);
165     }
166     for (int i = 0; i < N_its ; i++)
167     {
168         // Generate new positions
169         update_positions(walkmen_pos, N_survivors, ndim, dtau, fp_w, U, IS, decomp);
170         int *num_walkers = (int *)calloc(N_sprinters, sizeof(int)); // Array to store the
171         // ↳ offsprings of each walker
172         // Calculate the offspring of each walker
173         for(int j = 0; j < N_sprinters; j++)
174         {
175             if (ndim == 1)
176             {
177                 int m = (int)(weight(dtau, E_T, morse(walkmen_pos[j][0])) +
178                 // ↳ gsl_rng_uniform(U) * 1.); // Number of offsprings
179                 num_walkers[j] = m; // Store the number of offsprings
180             }
181             else if (ndim == 6)
182             {
183                 if (IS)
184                 {
185                     double *R = walkmen_pos[j]; // Walker's position
186                     double E_L = local_energy(R, 0.15); // Local energy
187                     int m = (int)(weight(dtau, E_T, E_L) + gsl_rng_uniform(U) * 1.); //
188                     // ↳ Number of offsprings
189                     num_walkers[j] = m; // Store the number of offsprings
190                 }
191                 else
192                 {
193                     // Convert to spherical coordinates
194                     double r_1 = sqrt(pow(walkmen_pos[j][0], 2) + pow(walkmen_pos[j][1],
195                     // ↳ 2) + pow(walkmen_pos[j][2], 2));
196                     double r_2 = sqrt(pow(walkmen_pos[j][3], 2) + pow(walkmen_pos[j][4],
197                     // ↳ 2) + pow(walkmen_pos[j][5], 2));
198                     double r_12 = sqrt(pow(walkmen_pos[j][3] - walkmen_pos[j][0], 2) +
199                     // ↳ pow(walkmen_pos[j][4] - walkmen_pos[j][1], 2) +
200                     // ↳ pow(walkmen_pos[j][5] - walkmen_pos[j][2], 2));
201                     double V_tot = - 2 / sqrt(pow(r_1, 2) + 1e-4) - 2 / sqrt(pow(r_2, 2)
202                     // ↳ + 1e-4) + 1 / sqrt(pow(r_12, 2) + 1e-4); // Potential energy
203                     int m = (int)(weight(dtau, E_T, V_tot) + gsl_rng_uniform(U) * 1.); //
204                     // ↳ Number of offsprings
205                     num_walkers[j] = m; // Store the number of offsprings
206                 }
207             }
208         }
209     }
210     N_survivors = int_sum(num_walkers, N_sprinters); // Number of surviving walkers
211     double **walkmen_pos_new = create_2D_array(N_survivors, ndim); // New array to
212     // ↳ store the surviving walkers

```

```

202
203     int M = 0; // Index for the new array
204     // Populate the new array with the walkers
205     for (int k = 0; k < N_sprinters; k++)
206     {
207         for (int m = 0; m < num_walkers[k]; m++) // If the walker survives
208             ↪ (num_walkers[j] > 0)
209         {
210             for(int n = 0; n < ndim; n++)
211             {
212                 walkmen_pos_new[M][n] = walkmen_pos[k][n]; // Copy walker's position
213             }
214             M++; // Increment the new array index
215         }
216     }
217     destroy_2D_array(walkmen_pos); // Free the memory of the old array
218     walkmen_pos = create_2D_array(N_survivors, ndim); // Create a new array to store
219     ↪ the walkers
220     // Update the walker's position
221     for (int k = 0; k < N_survivors; k++)
222     {
223         for (int n = 0; n < ndim; n++)
224         {
225             walkmen_pos[k][n] = walkmen_pos_new[k][n];
226         }
227     }
228
229     // Updating ET
230     double sprinter_ratio = (double)N_survivors / (double)N_0; // Ratio of surviving
231     ↪ walkers
232     E_T = E_T_avg - gamma * log(sprinter_ratio); // E_T at i+1
233     E_T_avg = E_T / (i+1) + E_T_avg * i / (i+1); // E_T_avg at i+1
234
235     // Print the results to the file
236     if (fp_ET != NULL)
237     {
238         fprintf(fp_ET, "%lf, %i, %f, %f\n", E_T_avg, N_survivors, (100. *
239             ↪ (double)N_survivors) / (double)N_sprinters, E_T);
240     }
241     N_sprinters = N_survivors; // Update the number of sprinters
242
243     // Free the memory
244     destroy_2D_array(walkmen_pos_new);
245     free(num_walkers);
246 }
247
248 // Store the results in the struct
249 results.E_T = E_T;
250 results.E_T_avg = E_T_avg;
251 results.N_sprinters = N_sprinters;
252 results.R = walkmen_pos;
253
254 return results;
255 }
256
257 void
258 init_walkmen_1D(
259     double **walkers,

```

```

255         int N_walkers
256     )
257 {
258     int j = 0; // Index
259     // Initialize the walkers
260     for (double i = -5; i < 5; i+=10./N_walkers)
261     {
262         walkers[j][0] = i;
263         j+=1;
264     }
265 }
266
267 void
268 init_walkmen_6D(
269     double **walkers,
270     int N_walkers,
271     gsl_rng *U
272 )
273 {
274     // Initialize the walkers
275     for(int i = 0; i < N_walkers; i++)
276     {
277         double r_1 = 0.7 + gsl_rng_uniform(U);
278         double theta_1 = acos(2 * gsl_rng_uniform(U) - 1);
279         double phi_1 = 2 * M_PI * gsl_rng_uniform(U);
280         double r_2 = 0.7 + gsl_rng_uniform(U);
281         double theta_2 = acos(2 * gsl_rng_uniform(U) - 1);
282         double phi_2 = 2 * M_PI * gsl_rng_uniform(U);
283
284         walkers[i][0] = r_1 * sin(theta_1) * cos(phi_1); // x_1
285         walkers[i][1] = r_1 * sin(theta_1) * sin(phi_1); // y_1
286         walkers[i][2] = r_1 * cos(theta_1); // z_1
287         walkers[i][3] = r_2 * sin(theta_2) * cos(phi_2); // x_2
288         walkers[i][4] = r_2 * sin(theta_2) * sin(phi_2); // y_2
289         walkers[i][5] = r_2 * cos(theta_2); // z_2
290     }
291 }
292
293 double
294 morse(
295     double x
296 )
297 {
298     double V = 0.5 * pow((1 - exp(-x)),2); // Morse potential
299
300     return V;
301 }
302
303 double
304 weight(
305     double dtau,
306     double E_T,
307     double V
308 )
309 {
310     double W = exp((E_T - V) * dtau); // Weight of the walker
311 }

```

```

312     return W;
313 }
314
315 double local_energy(
316     double *R,
317     double alpha
318 )
319 {
320     double x_1 = R[0];
321     double y_1 = R[1];
322     double z_1 = R[2];
323     double x_2 = R[3];
324     double y_2 = R[4];
325     double z_2 = R[5];
326
327     double r_12 = sqrt(pow(x_2 - x_1, 2) + pow(y_2 - y_1, 2) + pow(z_2 - z_1, 2)); //
    ↪ Distance between the electrons
328
329     double r1_mag = sqrt(pow(x_1, 2) + pow(y_1, 2) + pow(z_1, 2)); // Magnitude of the
    ↪ first electron
330     double r2_mag = sqrt(pow(x_2, 2) + pow(y_2, 2) + pow(z_2, 2)); // Magnitude of the
    ↪ second electron
331
332     // Unit vector of the first electron
333     double r1_hat_x = x_1 / r1_mag;
334     double r1_hat_y = y_1 / r1_mag;
335     double r1_hat_z = z_1 / r1_mag;
336
337     // Unit vector of the second electron
338     double r2_hat_x = x_2 / r2_mag;
339     double r2_hat_y = y_2 / r2_mag;
340     double r2_hat_z = z_2 / r2_mag;
341
342     // Unit vector between the electrons
343     double delta_hat_x = r2_hat_x - r1_hat_x;
344     double delta_hat_y = r2_hat_y - r1_hat_y;
345     double delta_hat_z = r2_hat_z - r1_hat_z;
346
347     // Delta vector between the electrons
348     double delta_x = x_2 - x_1;
349     double delta_y = y_2 - y_1;
350     double delta_z = z_2 - z_1;
351
352     double energy_dot = delta_hat_x * delta_x + delta_hat_y * delta_y + delta_hat_z *
    ↪ delta_z;
353
354     // Local energy
355     double E_L = - 4 + energy_dot / (r_12 * pow(1 + alpha * r_12, 2)) - 1 / (r_12 * pow(1
    ↪ + alpha * r_12, 3)) - 1 / (4 * pow(1 + alpha * r1_mag, 4)) + 1 / r_12;
356
357     return E_L;
358 }
359
360 void
361 update_positions(
362     double **walkmen_pos,
363     int N_survivors,

```

```

364         int ndim,
365         double dtau,
366         FILE *fp_w,
367         gsl_rng *U,
368         bool IS,
369         int decomp
370     )
371 {
372     // Update the positions of the walkers
373     for (int l = 0; l < N_survivors; l++)
374     {
375         for (int n = 0; n < ndim; n++)
376         {
377             // Diffusive part
378             walkmen_pos[l][n] = walkmen_pos[l][n] + gsl_rng_gaussian(U, 1.) * sqrt(dtau);
379         }
380         if (IS)
381         {
382             // Drift velocity
383             drift_velocity(walkmen_pos[l], 0.15, dtau, decomp, U);
384         }
385         // Write the walker's position to the file
386         if (fp_w != NULL)
387         {
388             for (int n = 0; n < ndim; n++)
389             {
390                 if (ndim == 1)
391                 {
392                     fprintf(fp_w, "%lf\n", walkmen_pos[l][0]);
393                 }
394                 else
395                 {
396                     if (n < ndim - 1)
397                     {
398                         fprintf(fp_w, "%lf, ", walkmen_pos[l][n]);
399                     }
400                     else
401                     {
402                         fprintf(fp_w, "%lf\n", walkmen_pos[l][n]);
403                     }
404                 }
405             }
406         }
407     }
408 }
409
410 void
411 drift_velocity(
412     double *R,
413     double alpha,
414     double dtau,
415     int decomp,
416     gsl_rng *U
417 )
418 {
419     double x_1 = R[0];
420     double y_1 = R[1];

```



```

421 double z_1 = R[2];
422 double x_2 = R[3];
423 double y_2 = R[4];
424 double z_2 = R[5];
425
426 double r_1 = sqrt(pow(x_1, 2) + pow(y_1, 2) + pow(z_1, 2)); // Magnitude of the first
↳ electron
427 double r_2 = sqrt(pow(x_2, 2) + pow(y_2, 2) + pow(z_2, 2)); // Magnitude of the
↳ second electron
428
429 double r12_vec[3] = {x_2 - x_1, y_2 - y_1, z_2 - z_1}; // Vector between the
↳ electrons
430 double r12 = sqrt(pow(x_2 - x_1, 2) + pow(y_2 - y_1, 2) + pow(z_2 - z_1, 2)); //
↳ Distance between the electrons
431
432 double r1_hat[3] = {x_1 / r_1, y_1 / r_1, z_1 / r_1}; // Unit vector of the first
↳ electron
433 double r2_hat[3] = {x_2 / r_2, y_2 / r_2, z_2 / r_2}; // Unit vector of the second
↳ electron
434 double r12_hat[3] = {r12_vec[0] / r12, r12_vec[1] / r12, r12_vec[2] / r12}; // Unit
↳ vector between the electrons
435
436 // Compute drift velocity for the electrons
437 double v_F1[3];
438 double v_F2[3];
439 for (int i = 0; i < 3; i++)
440 {
441     v_F1[i] = - 2 * r1_hat[i] - (1.0 / (2. * pow(1 + alpha * r12, 2))) * r12_hat[i];
442     v_F2[i] = - 2 * r2_hat[i] + (1.0 / (2. * pow(1 + alpha * r12, 2))) * r12_hat[i];
443 }
444
445 // Second decomposition
446 if (decomp == 2)
447 {
448     // First half step
449     x_1 += v_F1[0] * dtau / 2;
450     y_1 += v_F1[1] * dtau / 2;
451     z_1 += v_F1[2] * dtau / 2;
452     x_2 += v_F2[0] * dtau / 2;
453     y_2 += v_F2[1] * dtau / 2;
454     z_2 += v_F2[2] * dtau / 2;
455     double r_1 = sqrt(pow(x_1, 2) + pow(y_1, 2) + pow(z_1, 2)); // Magnitude of the
↳ first electron
456     double r_2 = sqrt(pow(x_2, 2) + pow(y_2, 2) + pow(z_2, 2)); // Magnitude of the
↳ second electron
457
458     double r12_vec[3] = {x_2 - x_1, y_2 - y_1, z_2 - z_1}; // Vector between the
↳ electrons
459     double r12 = sqrt(pow(x_2 - x_1, 2) + pow(y_2 - y_1, 2) + pow(z_2 - z_1, 2)); //
↳ Distance between the electrons
460
461     double r1_hat[3] = {x_1 / r_1, y_1 / r_1, z_1 / r_1}; // Unit vector of the first
↳ electron
462     double r2_hat[3] = {x_2 / r_2, y_2 / r_2, z_2 / r_2}; // Unit vector of the
↳ second electron
463     double r12_hat[3] = {r12_vec[0] / r12, r12_vec[1] / r12, r12_vec[2] / r12}; //
↳ Unit vector between the electrons

```

```

464
465 // Compute drift velocity for the electrons
466 for (int i = 0; i < 3; i++)
467 {
468     v_F1[i] = -2 * r1_hat[i] - (1.0 / (2. * pow(1 + alpha * r12, 2))) *
        ↪ r12_hat[i];
469     v_F2[i] = -2 * r2_hat[i] + (1.0 / (2. * pow(1 + alpha * r12, 2))) *
        ↪ r12_hat[i];
470 }
471 }
472
473 // Update the positions
474 R[0] += v_F1[0] * dtau;
475 R[1] += v_F1[1] * dtau;
476 R[2] += v_F1[2] * dtau;
477 R[3] += v_F2[0] * dtau;
478 R[4] += v_F2[1] * dtau;
479 R[5] += v_F2[2] * dtau;
480 }
481
482 gsl_rng *
483 init_gsl_rng(
484     int seed
485 )
486 {
487     const gsl_rng_type * T;
488     gsl_rng * r;
489     gsl_rng_env_setup();
490     T = gsl_rng_default; // Default random number generator
491     r = gsl_rng_alloc(T); // Allocate memory for the random number generator
492
493     if (!r) {
494         fprintf(stderr, "Error: Could not allocate memory for RNG.\n");
495         exit(EXIT_FAILURE); // Exit if allocation fails
496     }
497
498     // Set the seed
499     gsl_rng_set(r, seed);
500
501     return r;
502 }

```

## A.2 tools.c

```

1     #include <stdio.h>
2     #include <stdlib.h>
3     #include <math.h>
4     #include <gsl/gsl_fft_real.h>
5     #include <gsl/gsl_fft_halfcomplex.h>
6     #include <complex.h>
7
8     #include "tools.h"
9
10    void

```

```

11 elementwise_addition(
12     double *res,
13     double *v1,
14     double *v2,
15     unsigned int len
16 )
17 {
18     for (int i = 0; i < len; i++) {
19         res[i] = v1[i] + v2[i];
20     }
21 }
22
23 void
24 elementwise_multiplication(
25     double *res,
26     double *v1,
27     double *v2,
28     unsigned int len
29 )
30 {
31     for (int i = 0; i < len; i++) {
32         res[i] = v1[i] * v2[i];
33     }
34 }
35
36 int
37 int_sum(
38     int *v,
39     int len
40 )
41 {
42     int sum = 0;
43     for(int i = 0; i < len; i++)
44     {
45         sum += v[i];
46     }
47     return sum;
48 }
49
50 void
51 addition_with_constant(
52     double *res,
53     double *v,
54     double constant,
55     unsigned int len)
56 {
57     for (int i = 0; i < len; i++) {
58         res[i] = v[i] + constant;
59     }
60 }
61
62 void
63 multiplication_with_constant(
64     double *res,
65     double *v,
66     double constant,
67     unsigned int len)

```

```

68 {
69     for (int i = 0; i < len; i++) {
70         res[i] = v[i] * constant;
71     }
72 }
73
74 double
75 dot_product(
76     double *v1,
77     double *v2,
78     unsigned int len
79 )
80 {
81     double res = 0.;
82     for (int i = 0; i < len; i++) {
83         res += v1[i] * v2[i];
84     }
85     return res;
86 }
87
88 double **
89 create_2D_array(
90     unsigned int row_size,
91     unsigned int column_size
92 )
93 {
94     // Allocate memory for row pointers
95     double **array = malloc(row_size * sizeof(double *));
96     if (array == NULL) {
97         fprintf(stderr, "Memory allocation failed for row pointers.\n");
98         return NULL;
99     }
100
101     // Allocate a single contiguous block for all elements, i.e. all matrix elements
102     // get stored in a single block of memory, array[0] is the start of the block
103     // i.e. here we have array[0][index] as the way to access the elements
104     array[0] = malloc(row_size * column_size * sizeof(double));
105     if (array[0] == NULL) {
106         fprintf(stderr, "Memory allocation failed for data block.\n");
107         free(array);
108         return NULL;
109     }
110
111     // Set the row pointers to the appropriate positions in the block, in order to
112     // allow for the array[i][j] syntax. Now array[i] points to the start of the
113     // i-th row.
114     for (int i = 1; i < row_size; i++) {
115         array[i] = array[0] + i * column_size;
116     }
117
118     return array;
119 }
120
121 void
122 destroy_2D_array(
123     double **array
124 )

```

```

125 {
126     if (array != NULL) {
127         free(array[0]); // Free the contiguous block
128         free(array);    // Free the row pointers
129     }
130 }
131
132 void
133 matrix_vector_multiplication(
134     double *result,
135     double **A,
136     double *b,
137     unsigned int n,
138     unsigned int m
139 )
140 {
141     for (int i = 0; i < n; i++) {
142         result[i] = 0.;
143         for (int j = 0; j < m; j++) {
144             result[i] += A[i][j] * b[j];
145         }
146     }
147 }
148
149 void
150 matrix_matrix_multiplication(
151     double **result,
152     double **A,
153     double **B,
154     unsigned int n,
155     unsigned int m,
156     unsigned int k
157 )
158 {
159     for (int i = 0; i < n; i++) {
160         for (int kappa = 0; kappa < k; kappa++) {
161             result[i][kappa] = 0.;
162             for (int j = 0; j < m; j++) {
163                 result[i][kappa] += A[i][j] * B[j][kappa];
164             }
165         }
166     }
167 }
168
169 double
170 vector_norm(
171     double *v1,
172     unsigned int len
173 )
174 {
175     double res = 0.;
176     for (int i = 0; i < len; i++) {
177         res += v1[i] * v1[i];
178     }
179     return sqrt(res);
180 }
181

```

```

182
183 void
184 normalize_vector(
185     double *v,
186     unsigned int len
187 )
188 {
189     double norm = vector_norm(v, len);
190     multiplication_with_constant(v, v, 1. / norm, len);
191 }
192
193 double
194 average(
195     double *v,
196     unsigned int len
197 )
198 {
199     double res = 0.;
200     for (int i = 0; i < len; i++) {
201         res += v[i];
202     }
203     return res / len;
204 }
205
206
207 double
208 standard_deviation(
209     double *v,
210     unsigned int len
211 )
212 {
213     double avg = average(v, len);
214     double res = 0.;
215     for (int i = 0; i < len; i++) {
216         res += (v[i] - avg) * (v[i] - avg);
217     }
218     return sqrt(res / len);
219 }
220
221 double
222 variance(
223     double *v,
224     unsigned int len
225 )
226 {
227     double var;
228     var = pow(standard_deviation(v, len), 2);
229
230     return var;
231 }
232
233 double
234 autocorrelation(
235     double *data,
236     int data_len,
237     int time_lag_ind
238 )

```

```

239 {
240     double corr = 0.;
241     double avg = average(data, data_len);
242     double var = variance(data, data_len);
243
244     for (int i = 0; i < data_len - time_lag_ind; i++)
245     {
246         corr += (data[i]*data[i+time_lag_ind] - avg*avg) / var;
247     }
248
249     return corr / (data_len - time_lag_ind);
250 }
251
252 double
253 block_average(
254     double *data,
255     int data_len,
256     int block_size
257 )
258 {
259     int num_blocks = data_len / block_size;
260     double *block_averages = (double *)calloc(num_blocks, sizeof(double));
261     for (int i = 0; i < num_blocks; i++) {
262         double sum = 0.0;
263         for (int j = 0; j < block_size; j++) {
264             sum += data[i * block_size + j];
265         }
266         block_averages[i] = sum / block_size;
267     }
268     double block_avg = block_size * variance(block_averages, num_blocks) / variance(data,
269 ↪ data_len);
270     free(block_averages);
271
272     return block_avg;
273 }
274
275 double
276 distance_between_vectors(
277     double *v1,
278     double *v2,
279     unsigned int len
280 )
281 {
282     double res = 0.;
283     // With previous defined functions
284     double *diff = malloc(len * sizeof(double));
285     multiplication_with_constant(v1, v1, -1., len);
286     elementwise_addition(diff, v1, v2, len);
287     res = vector_norm(diff, len);
288     free(diff);
289     return res;
290 }
291
292 void
293 cumulative_integration(
294     double *res,
295     double *v,

```

```

295         double dx,
296         unsigned int v_len
297     )
298 {
299     double sum = 0.;
300     res[0] = 0.;
301     for (int i = 1; i < v_len; i++) {
302         sum = 0.5 * (v[i - 1] + v[i]) * dx;
303         res[i] = res[i - 1] + sum;
304     }
305 }
306
307 void
308 write_xyz(
309     FILE *fp,
310     char *symbol,
311     double **positions,
312     double **velocities,
313     double alat,
314     int natoms
315 )
316 {
317     fprintf(fp, "%i\nLattice=\"%f 0.0 0.0 0.0 %f 0.0 0.0 0.0 %f\" ", natoms, alat, alat,
318 ↪ alat);
319     fprintf(fp, "Properties=species:S:1:pos:R:3:vel:R:3 pbc=\"T T T\"\n");
320     for (int i = 0; i < natoms; ++i){
321         fprintf(fp, "%s %f %f %f %f %f %f\n",
322             symbol, positions[i][0], positions[i][1], positions[i][2],
323             velocities[i][0], velocities[i][1], velocities[i][2]);
324     }
325 }
326
327 void fft_freq(
328     double *res,
329     int n,
330     double timestep
331 )
332 {
333     for (int i = 0; i < n; i++) {
334         if (i < n / 2) {
335             res[i] = 2 * M_PI * i / (n * timestep);
336         }
337         else {
338             res[i] = 2 * M_PI * (i - n) / (n * timestep);
339         }
340     }
341 }
342
343 /* Freely given functions */
344 void
345 skip_line(
346     FILE *fp
347 )
348 {
349     int c;
350     while (c = fgetc(fp), c != '\n' && c != EOF);

```



```

351
352 void
353 read_xyz(
354     FILE *fp,
355     char *symbol,
356     double **positions,
357     double **velocities,
358     double *alat
359 )
360 {
361     int natoms;
362     if(fscanf(fp, "%i\nLattice=\"%lf 0.0 0.0 0.0 %lf 0.0 0.0 0.0 %lf\" ", &natoms, alat,
363 ↪ alat, alat) == 0){
364         perror("Error");
365     }
366     skip_line(fp);
367     for(int i = 0; i < natoms; ++i){
368         fscanf(fp, "%s %lf %lf %lf ",
369             symbol, &positions[i][0], &positions[i][1], &positions[i][2]);
370         fscanf(fp, "%lf %lf %lf\n",
371             &velocities[i][0], &velocities[i][1], &velocities[i][2]);
372     }
373 }
374
375 void powerspectrum(
376     double *res,
377     double *signal,
378     int n,
379     double timestep
380 )
381 {
382     /* Declaration of variables */
383     double *complex_coefficient = malloc(sizeof(double) * 2*n); // array for the complex
384     ↪ fft data
385     double *data_cp = malloc(sizeof(double) * n);
386
387     /*make copy of data to avoid messing with data in the transform*/
388     for (int i = 0; i < n; i++) {
389         data_cp[i] = signal[i];
390     }
391
392     /* Declare wavetable and workspace for fft */
393     gsl_fft_real_wavetable *real;
394     gsl_fft_real_workspace *work;
395
396     /* Allocate space for wavetable and workspace for fft */
397     work = gsl_fft_real_workspace_alloc(n);
398     real = gsl_fft_real_wavetable_alloc(n);
399
400     /* Do the fft*/
401     gsl_fft_real_transform(data_cp, 1, n, real, work);
402
403     /* Unpack the output into array with alternating real and imaginary part */
404     gsl_fft_halfcomplex_unpack(data_cp, complex_coefficient, 1, n);
405
406     /*fill the output powspec_data with the powerspectrum */
407     for (int i = 0; i < n; i++) {

```

```

406     res[i] =
407         ↪ (complex_coefficient[2*i]*complex_coefficient[2*i]+complex_coefficient[2*i+1]*complex_coefficient[2*i+1])
408     res[i] *= timestep / n;
409 }
410
411 /* Free memory of wavetable and workspace */
412 gsl_fft_real_wavetable_free(real);
413 gsl_fft_real_workspace_free(work);
414 free(complex_coefficient);
415 free(data_cp);
416 }

```

## B Source code in Python

```
1  # Importing libraries
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import pandas as pd
5
6  # Latex style
7  plt.style.use('default')
8  plt.rc('text', usetex=True)
9  plt.rc('font', family='serif')
10 plt.rc('font', size=25.75)
11 plt.rcParams['text.latex.preamble'] = r'\usepackage{amsmath}'
12 plt.rcParams['text.latex.preamble'] = r'\usepackage{bm}'
13 # Set ticks on both sides
14 plt.rcParams['xtick.direction'] = 'in'
15 plt.rcParams['ytick.direction'] = 'in'
16 plt.rcParams['xtick.major.size'] = 5
17 plt.rcParams['ytick.major.size'] = 5
18 plt.rcParams['xtick.top'] = True
19 plt.rcParams['ytick.right'] = True
20
21 # Functions
22 def read_data(task, ndim, decomp=1, dtau=0.01, walkmen=False, eq=False):
23     '''
24     Read data from csv files.
25
26     Args:
27         task (int): Task number.
28         ndim (int): Number of dimensions.
29         decomp (int): Decomposition number.
30         dtau (float): Time step.
31         walkmen (bool): If True, read walkmen data.
32         eq (bool): If True, read equilibrium data.
33
34     Returns:
35         if task == 1 and eq:
36             data_eq (np.ndarray): Equilibrium data.
37         elif task == 1 and not eq:
38             data_ET (np.ndarray): Non-equilibrium data.
39             data_walkmen (np.ndarray): Walkmen data.
40         elif task == 2 and walkmen:
41             data_ET (np.ndarray): Non-equilibrium data.
42             data_extrapol (np.ndarray): Extrapolated data.
43             R (list): List of arrays with the R values.
44             r12 (np.ndarray): Distance between walkers.
45         elif task == 2 and not walkmen:
46             data_ET (np.ndarray): Non-equilibrium data.
47             data_extrapol (np.ndarray): Extrapolated data.
48     '''
49     if task == 1:
50         if eq:
51             path_eq = f'data/task_{task}/{ndim}D/ET_Nwalk_eq_run.csv'
52             data_eq = pd.read_csv(path_eq, delimiter=',', on_bad_lines='skip')
53             data_eq = data_eq.apply(pd.to_numeric, errors='coerce')
54             data_eq = data_eq.fillna(np.nan)
55             data_eq = np.asarray(data_eq)
```

```

56         return data_eq
57     else:
58         path_ET = f'data/task_{task}/{ndim}D/ET_Nwalk_non_eq.csv'
59         data_ET = pd.read_csv(path_ET, delimiter=',', on_bad_lines='skip')
60         data_ET = data_ET.apply(pd.to_numeric, errors='coerce')
61         data_ET = data_ET.fillna(np.nan)
62         data_ET = np.asarray(data_ET)
63
64         path_walkmen = f'data/task_{task}/{ndim}D/I_was_walkin_in_morse.csv'
65         try:
66             data_walkmen = pd.read_csv(path_walkmen, delimiter=',',
67                                     ↪ on_bad_lines='skip')
68             data_walkmen = data_walkmen.apply(pd.to_numeric, errors='coerce')
69             data_walkmen = data_walkmen.fillna(np.nan)
70             data_walkmen = data_walkmen[~np.isnan(data_walkmen)]
71         except (FileNotFoundError, pd.errors.EmptyDataError, pd.errors.ParserError)
72             ↪ as e:
73             print(f"Error reading {path_walkmen}: {e}")
74             data_walkmen = None
75
76         return data_ET, data_walkmen
77
78 elif task == 2:
79     path_ET = f'data/task_{task}/ET_Nwalk_non_eq_decomp_{decomp}_dtau_{dtau:.6f}.csv'
80     data_ET = np.asarray(pd.read_csv(path_ET, delimiter=','))
81
82     path_extrapol = f'data/task_{task}/ET_Nwalks_{decomp}.csv'
83     data_extrapol = np.asarray(pd.read_csv(path_extrapol, delimiter=','))
84
85     if walkmen:
86         path_walkmen =
87             ↪ f'data/task_{task}/I_was_walkin_in_morse_decomp_{decomp}_dtau_{dtau:.6f}.csv'
88         data_walkmen = np.asarray(pd.read_csv(path_walkmen, delimiter=',',
89             ↪ on_bad_lines='skip'))
90
91         # Convert to spherical coordinates
92         r_1 = np.sqrt(data_walkmen[:, 0]**2 + data_walkmen[:, 1]**2 + data_walkmen[:,
93             ↪ 2]**2)
94         theta_1 = np.arccos(data_walkmen[:, 2] / r_1)
95         phi_1 = np.arctan2(data_walkmen[:, 1], data_walkmen[:, 0])
96         r_2 = np.sqrt(data_walkmen[:, 3]**2 + data_walkmen[:, 4]**2 + data_walkmen[:,
97             ↪ 5]**2)
98         theta_2 = np.arccos(data_walkmen[:, 5] / r_2)
99         phi_2 = np.arctan2(data_walkmen[:, 4], data_walkmen[:, 3])
100
101         # Distance between walkers
102         r12 = np.sqrt((data_walkmen[:, 0] - data_walkmen[:, 3])**2 + (data_walkmen[:,
103             ↪ 1] - data_walkmen[:, 4])**2 + (data_walkmen[:, 2] - data_walkmen[:,
104             ↪ 5])**2)
105
106         # Total data
107         R = [r_1, theta_1, phi_1, r_2, theta_2, phi_2]
108
109     return data_ET, data_extrapol, R, r12
110 else:
111     return data_ET, data_extrapol

```

```

105
106 def plot_GSE(ET, tau, dtau, E_0, ndim, task, save=False):
107     '''
108     Plot the ground-state energy as a function of time.
109
110     Args:
111         ET (np.ndarray): Ground-state energy.
112         tau (np.ndarray): Time.
113         dtau (float): Time step.
114         E_0 (float): Theoretical ground-state energy.
115         ndim (int): Number of dimensions.
116         task (int): Task number.
117         save (bool): If True, save the figure.
118     '''
119     fig, ax = plt.subplots(1, 2, figsize=(17, 8))
120
121     # Plot the energy as a function of time
122     ax[0].plot(tau, ET, color='tab:red', alpha=0.5)
123     ax[0].hlines(E_0, 0, len(ET)*dtau, color='tab:red',
124         ↪ label="$E_0$" + f"$\\,={E_0:.3f}$", lw=2.5)
125     ax[0].hlines(np.mean(ET), 0, len(ET)*dtau, colors='k', label='$\\langle$
126         ↪ 

```

```

157
158 Args:
159     tau (np.ndarray): Time.
160     walk (np.ndarray): Number of walkers.
161     ndim (int): Number of dimensions.
162     task (int): Task number.
163     save (bool): If True, save the figure.
164     '''
165 fig, ax = plt.subplots(1, 2, figsize=(17, 8))
166
167 # Plot the number of walkers as a function of time
168 ax[0].plot(tau, walk, color='tab:green', alpha=0.5)
169 ax[0].hlines(np.mean(walk), 0, np.max(tau), colors='k', label='$\\langle N_{walkers} \\rangle$', linestyle='dashed', lw=2.5)
170 ax[0].set_xlabel('Time $\\tau$ [a.u.]')
171 ax[0].set_ylabel('Number of walkers $N_i$')
172 ax[0].set_ylim(925, 1100)
173 ax[0].legend(loc='upper left')
174
175 # Plot the histogram of the number of walkers
176 ax[1].hist(walk, bins=30, density=True, color='tab:green', alpha=0.5, edgecolor='k')
177 ax[1].set_xlabel('Number of walkers')
178 ax[1].set_ylabel('Probability density')
179 std = np.std(walk)
180 ax[1].vlines(x=np.mean(walk)-std, ymin=0, ymax=0.0235, color='k', linestyle='dashed',
181             lw=1.5, alpha=0.5)
182 ax[1].vlines(x=np.mean(walk)+std, ymin=0, ymax=0.0235, color='k', linestyle='dashed',
183             lw=1.5, alpha=0.5)
184 ax[1].vlines(np.mean(walk), 0, 0.0235, colors='k', linestyle='dashed',
185             label='$\\langle N_{walkers} \\rangle$', lw=2.5)
186 ax[1].set_ylim(0, 0.027)
187 ax[1].set_xlim(925, 1080)
188 ax[1].legend(loc='upper left')
189
190 ax[0].text(2400, 886, '(a)')
191 ax[1].text(997, -0.006, '(b)')
192 plt.tight_layout()
193
194 # Save figure
195 if save:
196     save_fig(plt, 'N_walkers', task, ndim)
197
198 plt.show()
199
200 def plot_Rhist(R, decomp, save=False):
201     '''
202     Plot the histogram of the walkers' positions.
203     '''
204     Args:
205     R (list): List of arrays with the R values.
206     decomp (int): Decomposition number.
207     save (bool): If True, save the figure.
208     '''
209     fig, axs = plt.subplots(2, 3, figsize=(18, 12), sharex='col')
210     # Plot the histogram of the walkers' positions for the first electron
211     axs[0][0].hist(R[0], bins=50, density=True, color='tab:blue', alpha=0.5,
212                   edgecolor='k', label='$r_1$')

```

```

209     axs[0][1].hist(R[1], bins=50, density=True, color='tab:green', alpha=0.5,
    ↪     edgecolor='k', label='$\\theta_1$')
210     axs[0][2].hist(R[2], bins=50, density=True, color='tab:red', alpha=0.5,
    ↪     edgecolor='k', label='$\\phi_1$')
211
212     # Plot the histogram of the walkers' positions for the second electron
213     axs[1][0].hist(R[3], bins=50, density=True, color='tab:blue', alpha=0.5,
    ↪     edgecolor='k', label='$r_2$')
214     axs[1][1].hist(R[4], bins=50, density=True, color='tab:green', alpha=0.5,
    ↪     edgecolor='k', label='$\\theta_2$')
215     axs[1][2].hist(R[5], bins=50, density=True, color='tab:red', alpha=0.5,
    ↪     edgecolor='k', label='$\\phi_2$')
216
217     # Set labels and ticks
218     for i in range(2):
219         axs[i][0].set_xlabel(f'$r_{i+1}$ [a.u.]')
220         axs[i][0].text(1.4, 0.8, '\\textbf{a}') if i == 0 else '\\textbf{d}')
221         axs[i][1].set_xlabel(f'$\\theta_{i+1}$ [rad]')
222         axs[i][1].set_xticks([0, np.pi/2, np.pi])
223         axs[i][1].set_xticklabels(['0', '$\\pi / 2$', '$\\pi$'])
224         axs[i][1].set_yticks([0, 0.1, 0.2, 0.3, 0.4, 0.5])
225         axs[i][1].text(0.3, 0.457, '\\textbf{b}') if i == 0 else '\\textbf{e}')
226         axs[i][2].set_xlabel(f'$\\phi_{i+1}$ [rad]')
227         axs[i][2].set_xticks([-np.pi, 0, np.pi])
228         axs[i][2].set_xticklabels(['-$\\pi$', '0', '$\\pi$'])
229         axs[i][2].set_ylim(0, 0.21)
230         axs[i][2].text(-2.5, 0.185, '\\textbf{c}') if i == 0 else '\\textbf{f}')
231
232     for ax in axs.flatten():
233         ax.grid(alpha=0.5, linestyle='--', lw=1)
234         ax.legend()
235         ax.set_ylabel('Probability density')
236
237     plt.tight_layout()
238
239     # Save figure
240     if save:
241         save_fig(plt, f'R_hist_{decomp}', 2, 6)
242
243     plt.show()
244
245 def save_fig(fig, name, task, ndim):
246     '''
247     Save figure as pdf.
248
249     Args:
250         fig (plt.figure): Figure to save.
251         name (str): Name of the figure.
252         task (int): Task number.
253         ndim (int): Number of dimensions.
254     '''
255     fig.savefig(f'figs/task_{task}/{ndim}D/{name}.pdf', bbox_inches='tight')
256
257     ##### Task 1 #####
258     # Plot the Morse potential and the wavefunction
259     x = np.linspace(-2.5, 10, 10000)
260     psi = np.exp(-np.exp(-x) - x/2) / np.sqrt(np.pi)

```

```

261 Morse = 0.5 * (1 - np.exp(-x)) ** 2
262
263 # Morse potential
264 plt.figure(figsize=(8, 5))
265 plt.plot(x, Morse, label='Morse potential  $V(x)$ ', color='k', lw=2.5)
266 plt.xlabel('$x$ [a.u.]')
267 plt.ylabel('$V(x)$ [a.u.]')
268 plt.grid(alpha=0.5, linestyle='--', lw=1)
269 plt.ylim(-.5, 5)
270 plt.legend()
271 plt.tight_layout()
272 # save_fig(plt, 'Morse', 1, 1)
273
274 plt.figure(figsize=(8, 5))
275
276 # Read walker data
277 _, walker_pos = read_data(1, 1)
278 bin_size = 150
279 counts, bins = np.histogram(walker_pos, bins=bin_size)
280
281 # Normalize manually
282 bin_width = bins[1] - bins[0] # Width of each bin
283 normalized_counts = counts / (np.sum(counts) * bin_width)
284
285 # Plot the manually normalized histogram
286 plt.hist(walker_pos, bins=int(bin_size/2), density=True, color='tab:blue', alpha=0.2,
287 ↪ edgecolor='black', lw=1)
288 plt.plot(bins[:-1], normalized_counts, color='tab:blue', lw=3,
289 ↪ label='$\\Psi_{\\mathrm{Sample}}(x)$')
290 plt.plot(x, psi, label='$\\Psi_{\\mathrm{Analytic}}(x)$', color = 'k', lw=2.5, ls =
291 ↪ 'dashed', alpha = 0.7)
292 plt.xlabel('$x$ [a.u.]')
293 plt.ylabel('$\\Psi(x)$')
294 plt.grid(alpha=0.5, linestyle='--', lw=1)
295
296 plt.legend()
297 plt.tight_layout()
298 # save_fig(plt, 'Wavefunc', 1, 1)
299
300 plt.show()
301
302 # Read data
303 ET_data, _ = read_data(1, 1)
304 ET_avg, walk, mort_rate, ET = ET_data.T
305 tau = np.linspace(0, len(ET) * 0.02, len(ET))
306 save = False
307
308 # Plot the ground-state energy and the number of walkers
309 plot_GSE(ET, tau, 0.02, 3/8, 1, 1, save)
310
311 plot_walkmen(tau, walk, save)
312
313 # plt.figure()
314 # #mortality_rate = read_data('ET_Nwalk_non_eq.csv')[:,2]
315 # plt.plot(tau, mort_rate)
316 # plt.hlines(100,0,np.max(tau),'b', linestyle = 'dashed')
317 # plt.hlines(np.mean(mort_rate), 0, np.max(tau),colors='k', label='Mortality rate'+f' =
318 ↪ {np.mean(mort_rate):.1f}\\%')

```



```

315 # plt.legend()
316
317 # Read data
318 ET_avg_eq, walk_eq, mort_rate_eq, ET_eq = read_data(1, 1, eq=True)[: , :].T
319 ET_avg_eq = np.insert(ET_avg_eq, 0, 0.5)
320 walk_eq = np.insert(walk_eq, 0, 200)
321 ET_eq = np.insert(ET_eq, 0, 0.5)
322
323 # Plot the ground-state energy and the number of walkers in equilibrium
324 fig, ax = plt.subplots(1, 2, figsize=(17, 8))
325 ax[0].plot(tau[:len(ET_eq)], ET_eq, color='tab:red', alpha=0.4, lw=2.5,
326           ↪ label='$\\mathcal{E}_i$')
327 ax[0].plot(tau[:len(ET_avg_eq)], ET_avg_eq, color='tab:red', alpha=1, lw=2.5,
328           ↪ label='$E_{\\textsf{T}^{(i)}}$')
329 ax[1].plot(tau[:len(walk_eq)], walk_eq, color='tab:green', alpha=0.75, lw=2.5,
330           ↪ label='$N_i$')
331 ax[0].set_xlabel('Time $\\tau$ [a.u.]')
332 ax[0].set_ylabel('Ground-state energy $\\mathcal{E}_i$ [a.u.]')
333 ax[0].hlines(3/8, 0, len(ET_eq) * 0.02, color='k', label="$E_0"+f"$\\,={3/8:.3f}$",
334           ↪ lw=2, ls='dashed')
335 ax[1].hlines(200, 0, len(walk_eq) * 0.02, color='k', label="$N_0$", lw=2, ls='dashed')
336
337 ax[1].set_xlabel('Time $\\tau$ [a.u.]')
338 ax[1].set_ylabel('Number of walkers $N_i$')
339
340 ax[0].text(74, 0.01, '(a)')
341 ax[1].text(74, 50, '(b)')
342 ax[0].legend(loc='upper right')
343 ax[1].legend(loc='upper right')
344 plt.tight_layout()
345 # save_fig(plt, 'ET_Nwalk_eq', 1)
346
347 plt.show()
348
349 ##### Task 2 #####
350 # Read data
351 data_ET, _ = read_data(1, 6)
352 ET_avg, walk, mort_rate, ET = data_ET.T
353 dtau = 0.01
354 tau = np.linspace(0, len(ET) * dtau, len(ET))
355 E_0_He = -2.903
356 save = False
357
358 # Plot the ground-state energy and the number of walkers
359 plot_GSE(ET, tau, dtau, E_0_He, ndim=6, task=1, save=save)
360 plot_walkmen(tau, walk, ndim=6, task=1, save=save)
361
362 ##### Task 3 #####
363 dtau = 0.1
364 E_0_He = -2.903
365
366 # Read data for the first decomposition
367 data_ET_1, data_extrapol_1 = read_data(2, 1, decomp=1, dtau=dtau)
368 ET_avg_1, walk_1, mort_rate_1, ET_1 = data_ET_1.T
369
370 # Read data for the second decomposition
371 data_ET_2, data_extrapol_2 = read_data(2, 1, decomp=2, dtau=dtau)

```

```

368 ET_avg_2, walk_2, mort_rate_2, ET_2 = data_ET_2.T
369
370 # Plot the ground-state energy and the number of walkers for the first decomposition
371 tau = np.linspace(0, len(ET_1) * dtau, len(ET_1))
372 plot_GSE(ET_1, tau, dtau, E_0_He, ndim=6, task=2, save=False)
373 plot_walkmen(tau, walk_1, ndim=6, task=2, save=False)
374
375 # Plot the ground-state energy and the number of walkers for the second decomposition
376 plot_GSE(ET_2, tau, dtau, E_0_He, ndim=6, task=2, save=False)
377 plot_walkmen(tau, walk_2, ndim=6, task=2, save=False)
378
379 # Read the walker data for the first and second decomposition
380 _, _, R_1, r12_1 = read_data(2, 6, decomp=1, dtau=dtau, walkmen=True)
381 _, _, R_2, r12_2 = read_data(2, 6, decomp=2, dtau=dtau, walkmen=True)
382
383 # Plot the histogram of the distance between walkers for the first and second
    ↪ decomposition
384 fig, axs = plt.subplots(1, 2, figsize=(12, 7))
385 axs[0].hist(r12_1, bins=50, density=True, color='tab:orange', alpha=0.5, edgecolor='k')
386 axs[1].hist(r12_2, bins=50, density=True, color='tab:orange', alpha=0.5, edgecolor='k')
387
388 # Set labels and ticks
389 for ax in axs:
390     ax.set_xticks([2.5, 5])
391     ax.set_xlim(0, 6)
392     ax.set_xlabel('$r_{12}$ [a.u.]')
393     ax.set_ylabel('Probability density')
394     ax.grid(alpha=0.5, linestyle='--', lw=1)
395
396 plt.tight_layout()
397 axs[0].text(2.7, -0.175, '(a)')
398 axs[1].text(2.7, -0.175, '(b)')
399 # save_fig(plt, 'r12', 2, 6)
400
401 # Plot the histogram of the walkers' positions
402 plot_Rhist(R_1, 1, save=False)
403 plot_Rhist(R_2, 2, save=False)
404
405
406 ##### Task 4 #####
407 # Fit the extrapolated data
408 fit_1 = np.polyfit(data_extrapol_1[:,0], data_extrapol_1[:,1], 1)
409 fit_2 = np.polyfit(data_extrapol_2[:,0], data_extrapol_2[:,1], 2)
410 dtaus = np.linspace(0, 0.4, 1000)
411
412 print(f"Best estimation of E_0 for decomp 1: {fit_1[1]:.3f}")
413 print(f"Best estimation of E_0 for decomp 2: {fit_2[2]:.3f}")
414
415 # Plot the extrapolated data
416 fig, axs = plt.subplots(1, 2, figsize=(17, 8))
417 axs[0].plot(data_extrapol_1[:,0], data_extrapol_1[:,1], 's', label='Data points',
    ↪ color='k', markersize=10)
418 axs[0].plot(dtaus, np.polyval(fit_1, dtaus), label='Linear fit', color='tab:red', lw=2.5)
419
420 axs[1].plot(data_extrapol_2[:,0], data_extrapol_2[:,1], 's', label='Data points',
    ↪ color='k', markersize=10)
421 axs[1].plot(dtaus, np.polyval(fit_2, dtaus), label='Quadratic fit', color='tab:red',
    ↪ lw=2.5)

```

```

422
423 # Set labels and ticks
424 for ax in axs:
425     ax.set_xlabel('$\\Delta\\tau$ [a.u.]')
426     ax.set_ylabel('Ground-state energy $E_\\textsf{T}$ [a.u.]')
427     ax.legend()
428     ax.grid(alpha=0.5, linestyle='--', lw=1)
429 axs[1].legend(loc='upper right')
430 axs[1].set_ylim(-2.947, -2.89)
431
432 axs[0].text(0.185, -3.015, '(a)')
433 axs[1].text(0.185, -2.9605, '(b)')
434
435 axs[0].text(0.18, -2.835, f'$E_\\textsf{T} = \\textsf{{fit_1[0]:.3f}}\\Delta \\tau$'
436 ↪ f'$\\textsf{{fit_1[1]:.3f}}$', color='tab:red', fontsize=24)
437
438 axs[1].text(0., -2.94, f'$E_\\textsf{T} = \\textsf{{fit_2[0]:.3f}}\\Delta \\tau^2 +$'
439 ↪ f'$\\textsf{{fit_2[1]:.3f}}\\Delta \\tau \\textsf{{fit_2[2]:.3f}}$', color='tab:red', fontsize=24)
440
441 plt.tight_layout()
442 # save_fig(plt, 'extrapolation', 2, 6)
443 plt.show()

```