Homework 3:
Diffusion Monte Carlo

Jonatan Haraldsson
jonhara@chalmers.se

Oscar Stommendal
oscarsto@chalmers.se

Task Nº	Points	Avail. points
Σ		

January 13th 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 Schödinger 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 $a_0 = 0.529\,\text{Å}$ 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)

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

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

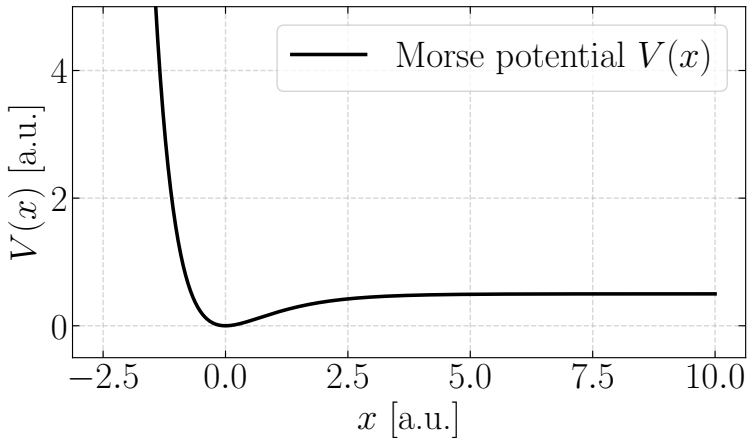


Figure 1: One-dimensional Morse potential given by Equation 1.

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 `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

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

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

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

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

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

where $\lfloor \bullet \rfloor$ 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
               ↪ position
11         }
12         shift++; // Increment the new array index
13     }
14 }
```

Lastly, the instantaneous estimated ground state energy was updated using

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

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,

$$E_{\mathsf{T}}^{(i)} \equiv \langle \mathcal{E} \rangle_i = \frac{1}{i - i_{eq}} \sum_{i'=i_{eq}+1}^{i} \mathcal{E}_{i'}, \tag{6}$$

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

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

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, $E_{\mathsf{T}}^{(i)}$. Afterwards, the ground-state energy was estimated using an average value $\langle E_{\mathsf{T}} \rangle$ 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_{T} are given along with the average $\langle E_{\mathsf{T}} \rangle = 0.377$. 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 $\langle E_{\mathsf{T}} \rangle$ 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 $\langle E_{\mathsf{T}} \rangle \pm \sigma_{\mathcal{E}}$. 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 ($\langle E_{\mathsf{T}} \rangle$).

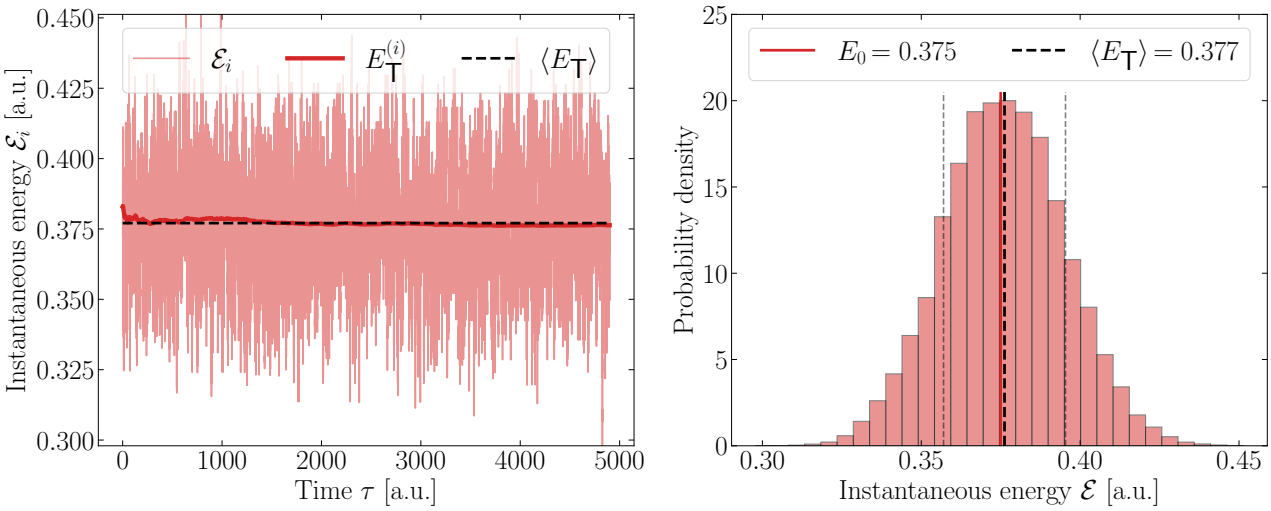


Figure 2: **(a)** Instantaneous ground-state energy $\mathcal{E}_i$ in each iteration, i, of the DMC along with average $\langle E_{\mathsf{T}} \rangle$. The analytic ground-state energy $E_0 = 3/8 = 0.375$ has been plotted for reference. **(b)** Histogram for instantaneous energy, again with $\langle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

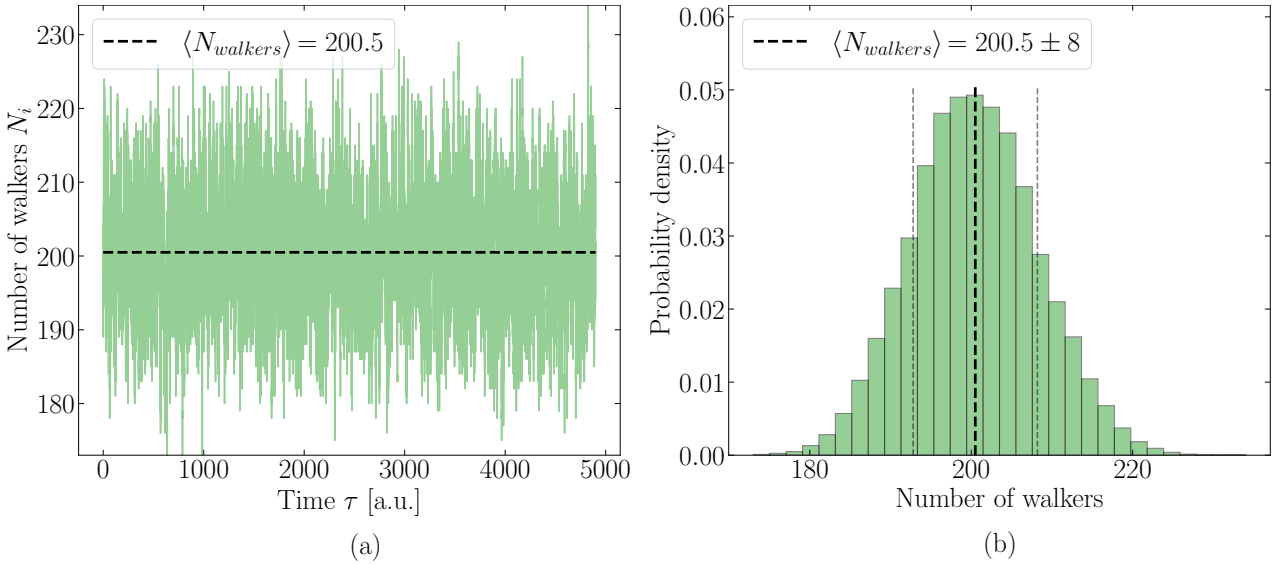


Figure 3: **(a)** Number of walkers N_i in each iteration of the DMC along with the average $\langle N_{walkers} \rangle$. **(b)** Histogram over the number of walkers together with the average $\langle N_{walkers} \rangle$ as well as additional thin lines for $\langle N_{walkers} \rangle \pm \sigma_N$.

Table 1: Estimated ground-state energy using $\langle E_\mathsf{T} \rangle$ 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 σ_N and initial walkers N_0 are also available.

Quantity	**Label**	**Value [a.u.]**
Avg. of acc. average	$\langle \boldsymbol{E_\mathsf{T}} \rangle$	0.3779
Std. of acc. average	$\boldsymbol{\sigma_{E_\mathsf{T}}}$	0.015
Avg. inst. energies	$\langle \boldsymbol{\mathcal{E}} \rangle$	0.3763
Std. of inst. energies	$\boldsymbol{\sigma_\mathcal{E}}$	0.022
Theoretical energy	$\boldsymbol{E_0}$	**0.375**
Avg number of walkers	$\langle \boldsymbol{N_{walkers}} \rangle$	201
Std. number of walkers	$\boldsymbol{\sigma_N}$	7.8
Initial number of walkers	$\boldsymbol{N_0}$	**200**

In contrast to Figure 2 and 3, Figure 4 illustrates the equilibration runs, where Figure 4a contains instantaneous energies $\mathcal{E}$, accumulated average $E_\mathsf{T}^{(i)}$ 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 ($E_\mathsf{T}^{(i)}$) takes slightly more iterations to converge. Since $E_\mathsf{T}^{(i)}$ approaches E_0 from above, the strong initial peak in $E_\mathsf{T}^{(i)}$ 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.

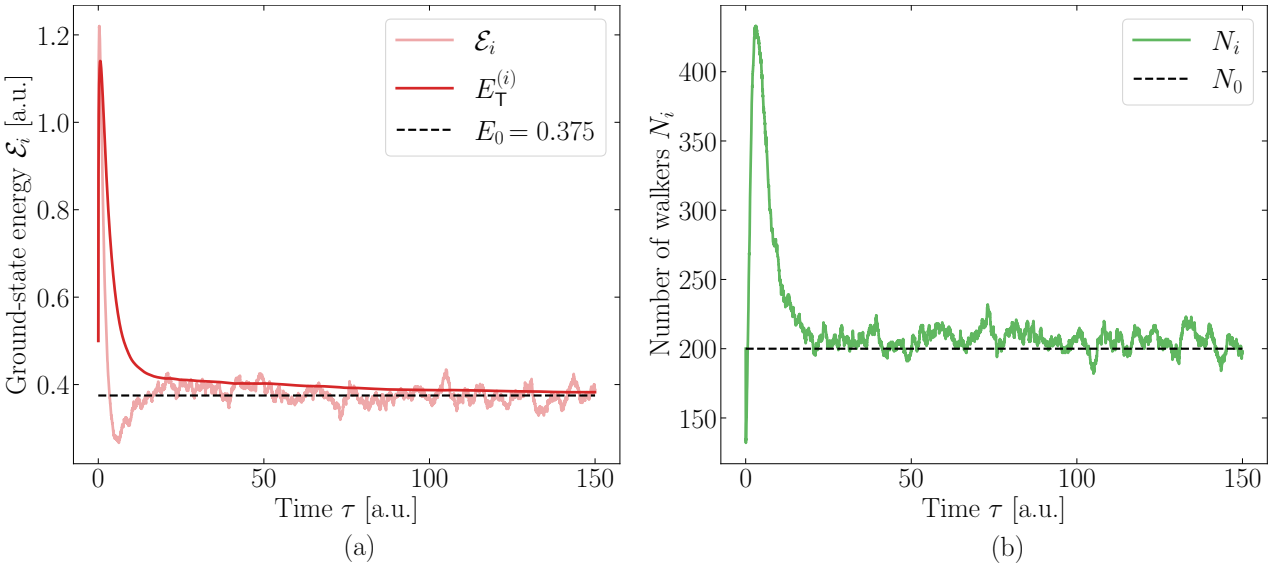


Figure 4: **(a)** Instantaneous energy $\mathcal{E}_i$ and accumulated average $E_{\mathsf{T}}^{(i)}$ during the $i_{eq} = 7\,500$ equilibration runs. The theoretical $E_0 = 3/8$ has been plotted for reference. **(b)** Number of walkers N_i in during the $i_{eq} = 7\,500$ equilibration iterations along 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

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

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

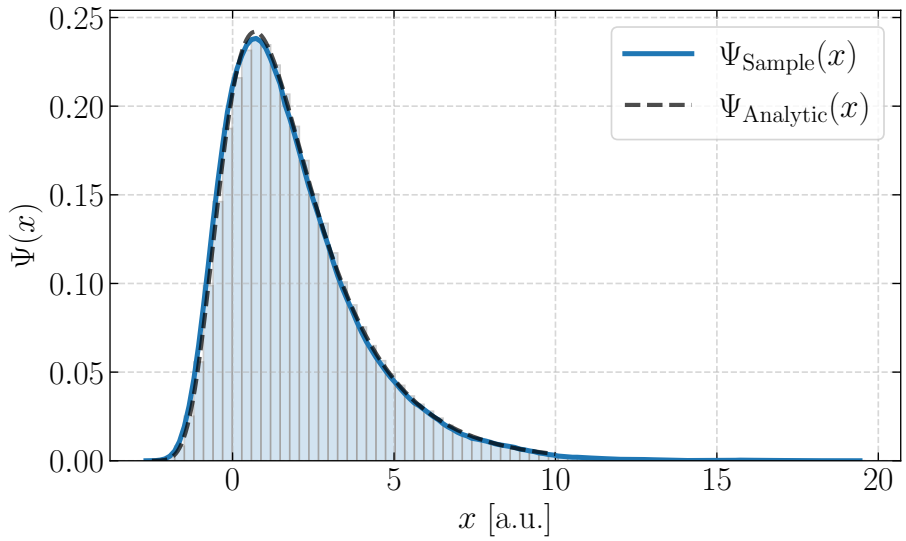


Figure 5: Walker positions x, resembling the sampled wave function Ψ_{Sample} along with the analytically solved wave function $\Psi_{\text{Analytical}}$.

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

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

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 $\boldsymbol{w}$, 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

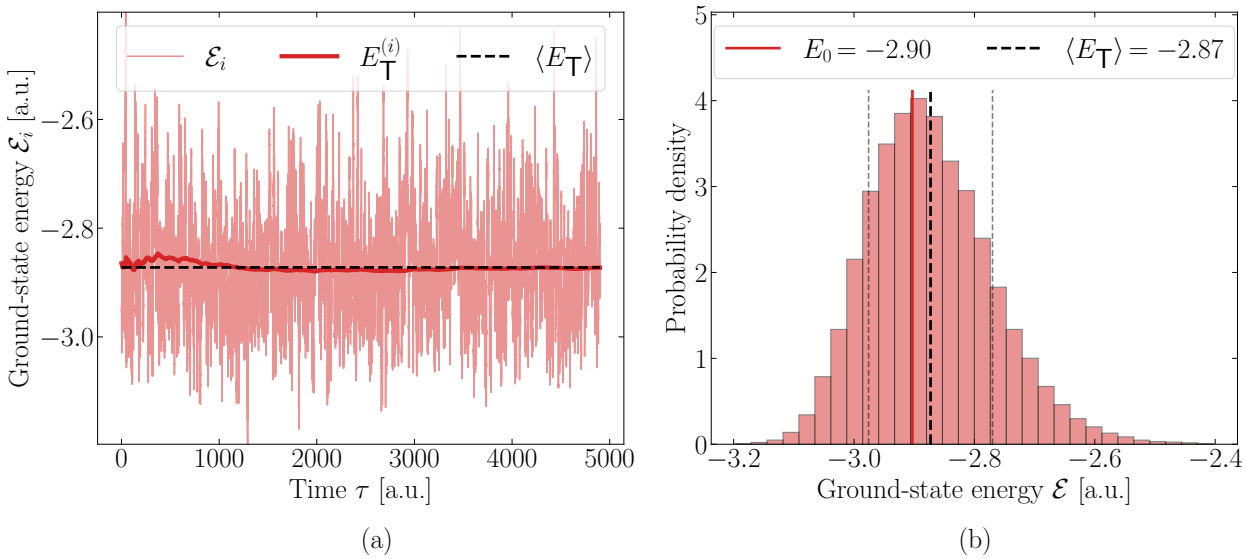
$$\begin{aligned} x_1 &= r_1\,\sin\theta_1\,\cos\varphi_1 & y_1 &= r_1\,\sin\theta_1\,\sin\varphi_1 & z_1 &= r_1\,\cos\theta_1 \\ x_2 &= r_2\,\sin\theta_2\,\cos\varphi_2 & y_2 &= r_2\,\sin\theta_2\,\sin\varphi_2 & z_2 &= r_2\,\cos\theta_2. \end{aligned}$$

Each walker, or row in the walker matrix, was thus given by $\boldsymbol{w}_j = \begin{bmatrix} x_1 & y_1 & z_1 & x_2 & y_2 & z_2 \end{bmatrix}$. 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.

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

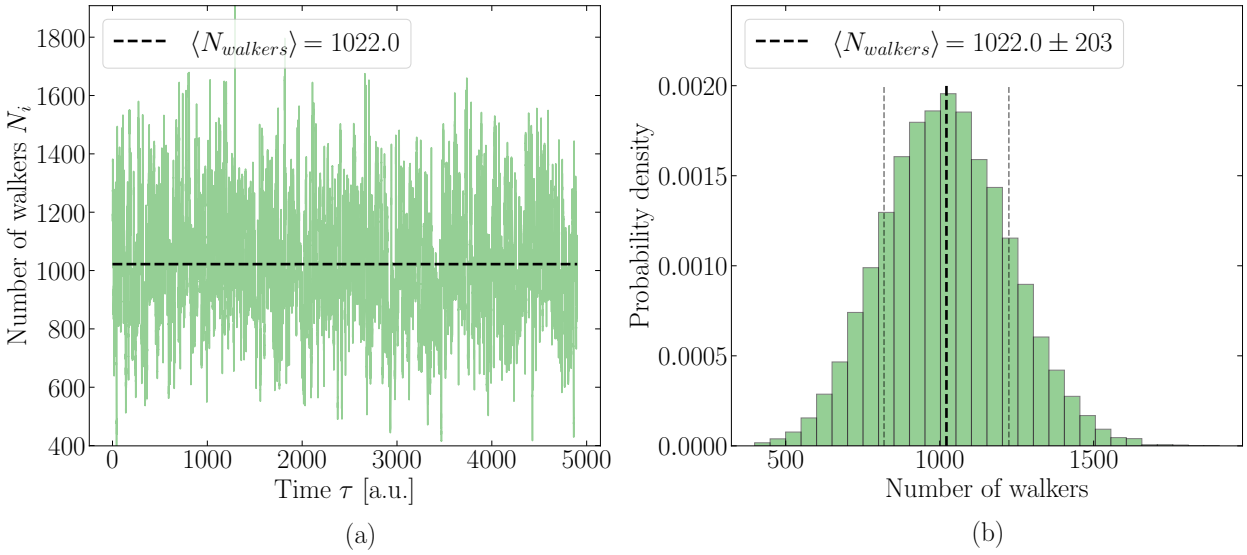
After the surviving walkers were copied, their positions were updated according to Equation 2 with a different random Gaußian number for each coordinate and the energies $\mathcal{E}_i$ and $E_\mathsf{T}^{(i)}$ was calculated according to Equations 5, 6 and 7, again using $\gamma = 0.5$. Moreover, an 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 $E_\mathsf{T}^{(i)}$ and the average $\langle E_\mathsf{T} \rangle = -2.87$. 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 $\langle E_\mathsf{T} \rangle$ 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.



(a) (b)

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



(a) (b)

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 $\langle E_{\mathsf{T}} \rangle$ 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 σ_N and initial walkers N_0 are also available.

Quantity	**Label**	**Value [a.u.]**
Avg. of acc. average	$\boldsymbol{\langle E_{\mathsf{T}\rangle}}$	-2.8725
Std. of acc. average	$\boldsymbol{\sigma_{E_{\mathsf{T}}}}$	0.010
Avg. inst. energies	$\boldsymbol{\langle \mathcal{E} \rangle}$	-2.8727
Std. of inst. energies	$\boldsymbol{\sigma_{\mathcal{E}}}$	0.10
Theoretical energy	$\boldsymbol{E_0}$	$\mathbf{-2.903}$
Avg number of walkers	$\boldsymbol{\langle N_{walkers} \rangle}$	1022
Std. number of walkers	$\boldsymbol{\sigma_N}$	203
Initial walkers	$\boldsymbol{N_0}$	**1000**

Task 3 – DMC with Importance Sampling

To increase the model's efficiency, *importance sampling* (IS) was implemented by introducing Ψ_{T}, a wave function similar to the real wave function of the Helium system. From Ψ_{T}, an expression for the local energy, E_L, is given by

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

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

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

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

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

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 $E_\mathsf{T}^{(0)} = -3.0$. 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 $\langle E_\mathsf{T}\rangle = -2.862$ 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 $r \approx 0.55 = 0.55\,a_0 \approx 0.29$ Å; quite close to $r_{max} = 0.569\,a_0$ [3]. The distribution for the polar angular coordinates, θ, suggests that the electrons have a higher probability of being in the x, y-plane ($\theta = \pi/2$). And, the azimuthal angles ϕ are, conversely, uniformly distributed, meaning that the system is symmetric along the z-axis.

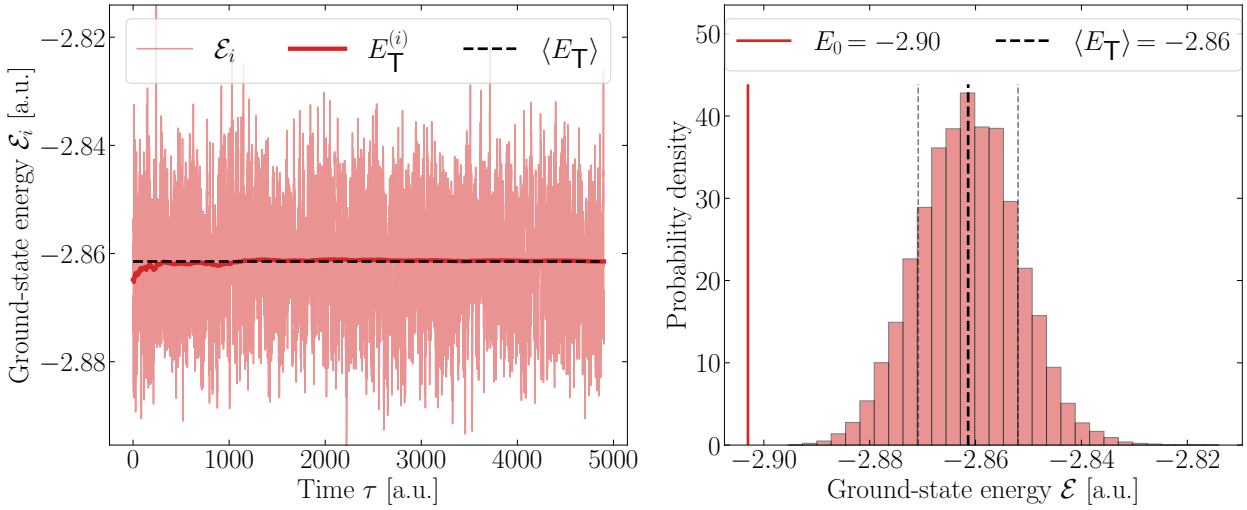


Figure 8: **(a)** Instantaneous ground-state energy $\mathcal{E}_i$ and accumulated average $E_\mathsf{T}^{(i)}$ in each iteration, i, of the DMC simulation of Helium with IS along with average $\langle E_\mathsf{T}\rangle$. Here, the more accurate decomposition, i.e. decomposition 2, of the short-time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

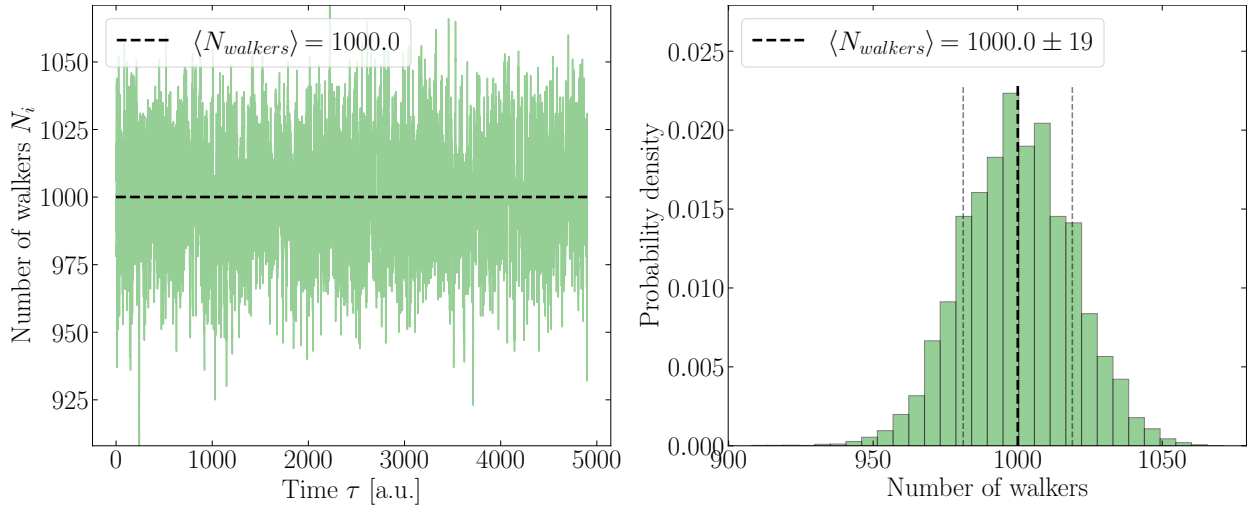


Figure 9: **(a)** Number of walkers N_i in each iteration of the DMC simulation of Helium with IS along with the average $\langle N_{walkers} \rangle$. Here, the more accurate decomposition, i.e. decomposition 2, of the short-time propagator was used. **(b)** Histogram over the number of walkers together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

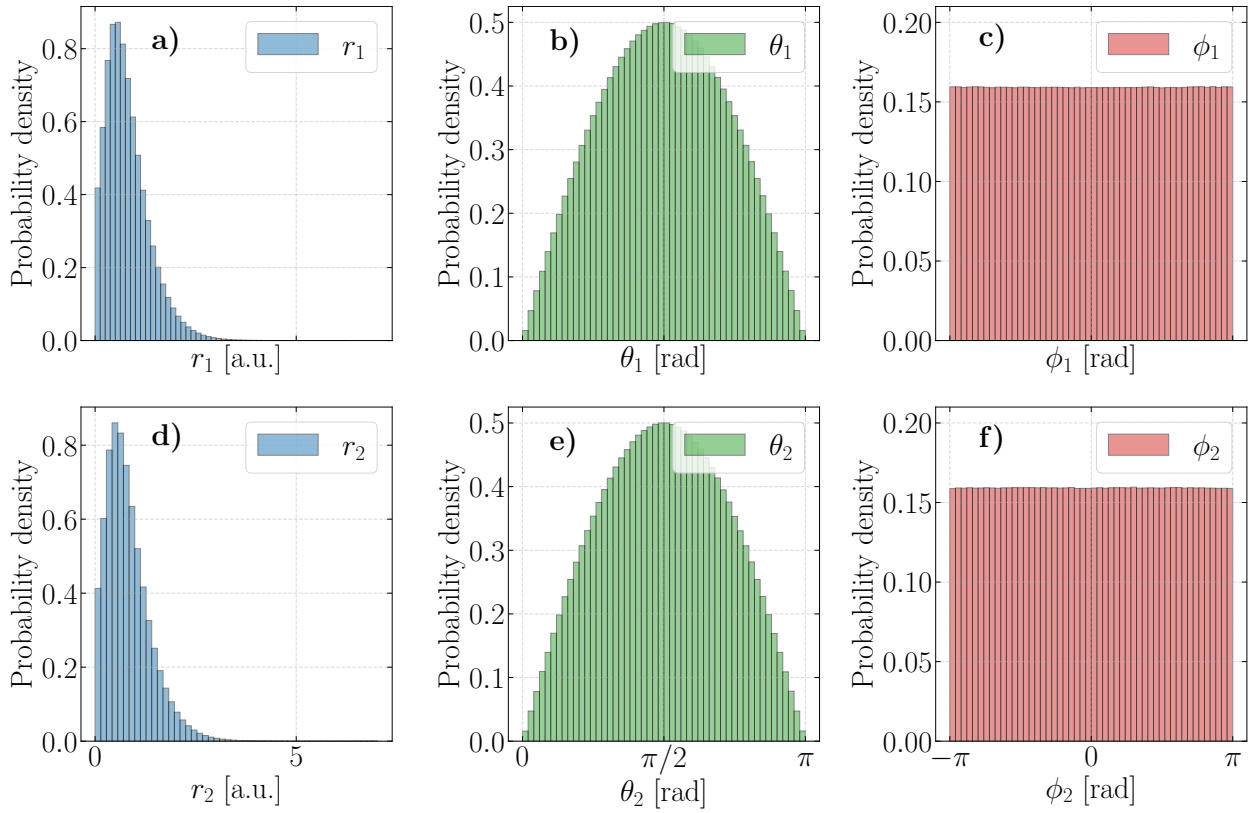


Figure 10: Probability distributions of the spherical coordinate components for the first **(a)** - **(c)**, (r_1, θ_1, ϕ_1), and second **(d)** - **(f)** electron, (r_2, θ_2, ϕ_2) in the Helium atom. Here, decomposition 1 of the short-time propagator was used in the importance sampling.

Second Decomposition

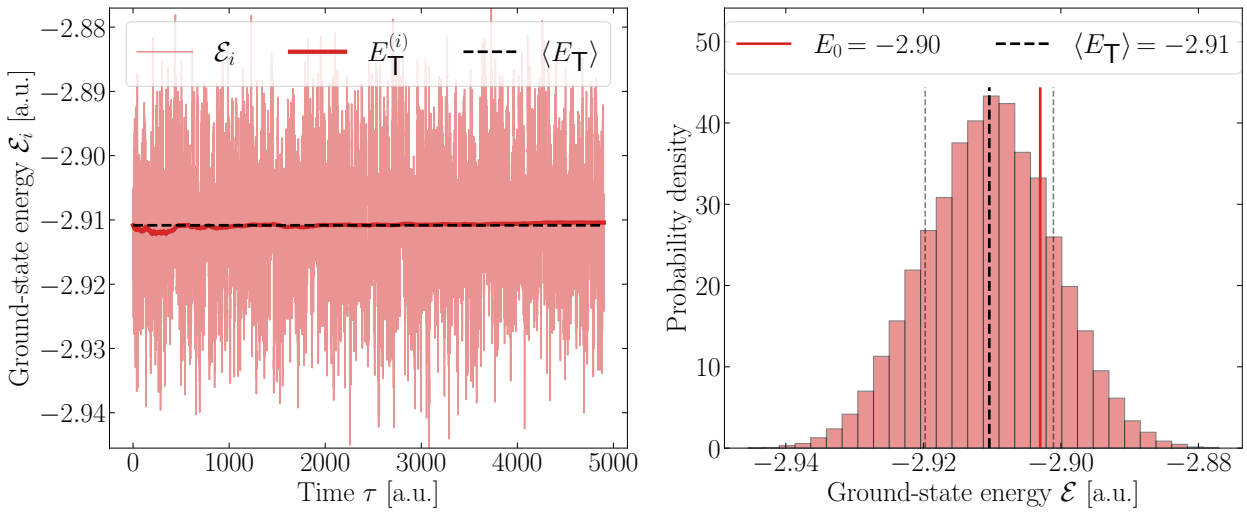
In the second decomposition, positions of surviving walkers were updated in two steps:

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

where $\vec{r}_k^{(int.)}$ is an intermediate half-step and $k = 1, 2$. Again, $N_0 = 1\,000$ walkers were initialized and $\mathcal{E}_0 = -3.0$. The simulation was run for $\tau = 5\,000$ or $\tau/\Delta\tau = 5\,000/0.01 = 500\,000$ iterations with $\tau_{eq} = 100$ or $10\,000$ 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 $\langle E_\mathsf{T} \rangle = -2.911$ 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 $\langle E_\mathsf{T} \rangle \pm \sigma_\mathcal{E}$, 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 $N_0 = 1\,000$, 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 $r_{max} \approx 0.55\,[\text{a.u}] = 0.55\,a_0 \approx 2.9\,\text{Å}$. The angular coordinates (θ and ϕ) 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.



(a) (b)

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 $\langle E_\mathsf{T} \rangle$. 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 $\langle E_\mathsf{T} \rangle$ and E_0 for reference and additional thin lines for $\langle E_\mathsf{T} \rangle \pm \sigma_\mathcal{E}$.

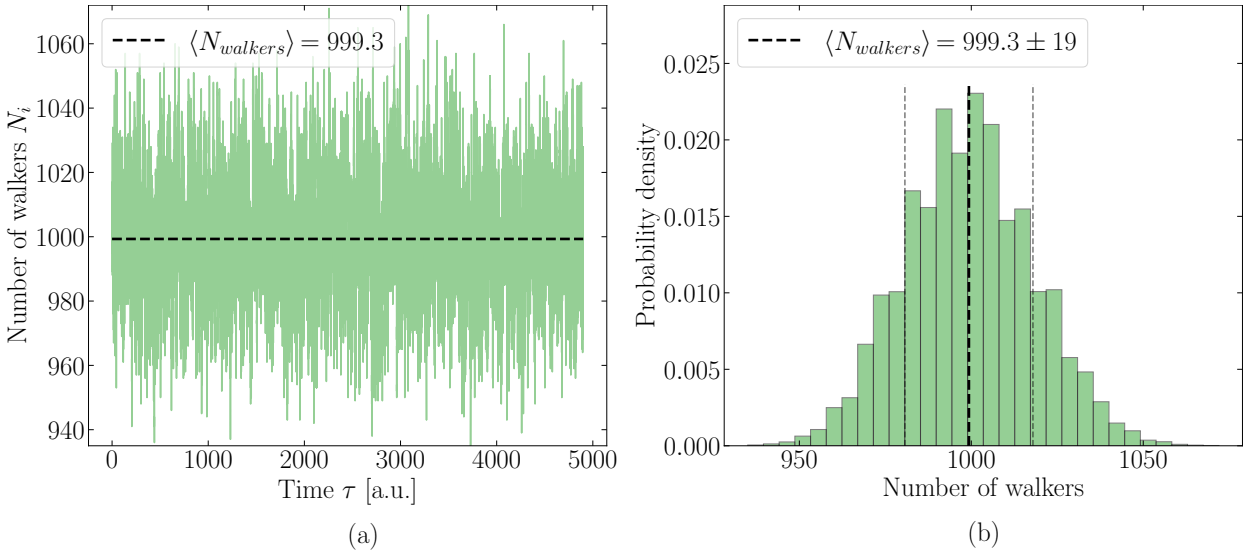


Figure 12: **(a)** Number of walkers N_i in each iteration of the DMC simulation of Helium with IS along with the average $\langle N_{walkers} \rangle$. Here, the second decomposition of the short-time propagator was used. **(b)** Histogram over the number of walkers together with the average $\langle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

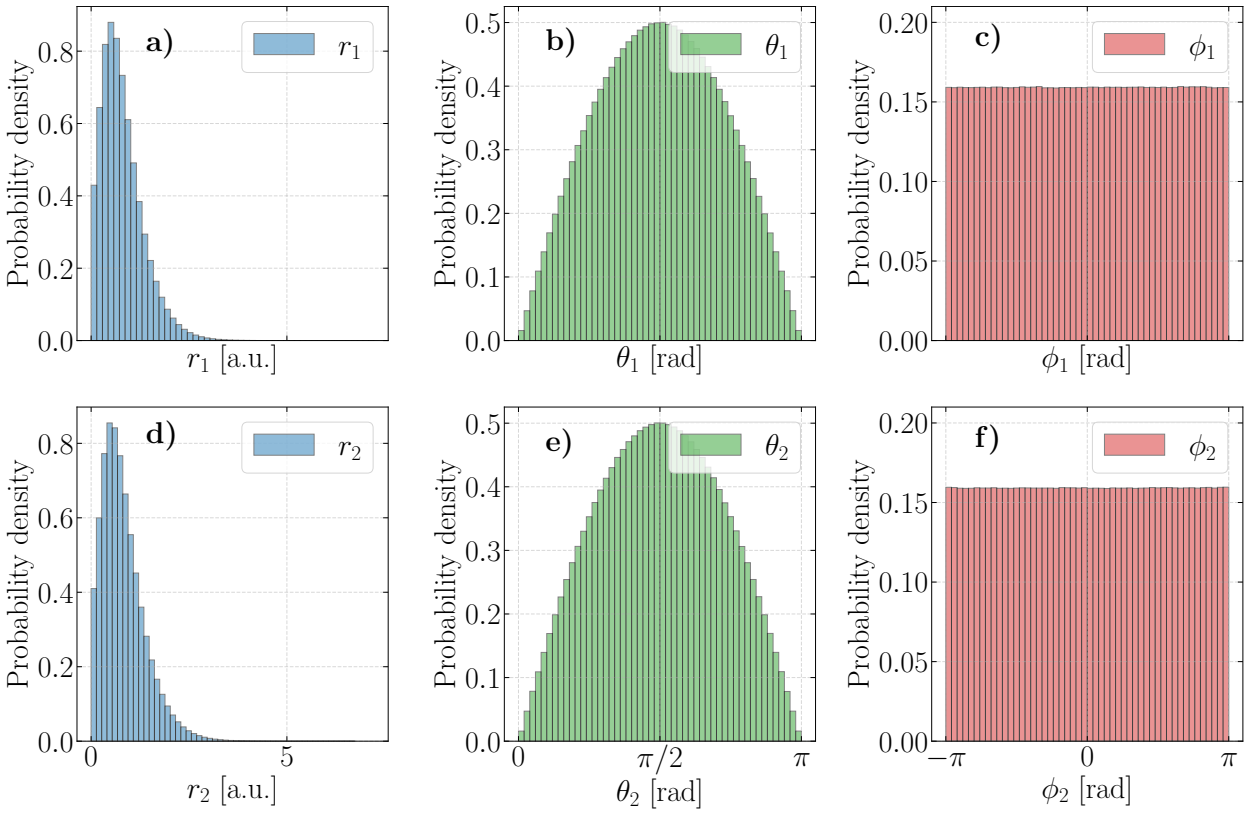


Figure 13: Probability distributions of the spherical coordinate components for the first **(a)** – **(c)**, (r_1, θ_1, ϕ_1), and second electron **(d)** – **(f)**, (r_2, θ_2, ϕ_2) in the Helium atom. Here, the more accurate decomposition, i.e. decomposition 2, of the short-time propagator was used in the importance sampling.

Comparison and Additional Discussion

To conclude, averages for energy estimations and number of walkers along with there 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-be-side comparison of estimated ground-state energy using $\langle E_{\mathsf{T}} \rangle$ 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 σ_N and initial walkers N_0 are also available.

Quantity	**Label**	**1st Decomp. [a.u.]**	**2nd Decomp. [a.u.]**
Avg. of acc. average	$\boldsymbol{\langle E_{\mathsf{T}} \rangle}$	-2.8616	-2.9109
Std. of acc. average	$\boldsymbol{\sigma_{E_{\mathsf{T}}}}$	0.0010	0.00051
Avg. inst. energies	$\boldsymbol{\langle \mathcal{E} \rangle}$	-2.8615	-2.9104
Std. of inst. energies	$\boldsymbol{\sigma_{\mathcal{E}}}$	0.0095	0.0093
Theoretical energy	$\boldsymbol{E_0}$	$\mathbf{-2.903}$	$\mathbf{-2.903}$
Avg number of walkers	$\boldsymbol{\langle N_{walkers} \rangle}$	999.9	999.3
Std. number of walkers	$\boldsymbol{\sigma_N}$	18.9	18.6
Initial walkers	$\boldsymbol{N_0}$	**1000**	**1000**

Additionally, when considering the accumulated mean $E_{\mathsf{T}}^{(i)}$ in Figure 11a, there are small fluctuations for $\tau < 1\,000$, which seems to slightly affect the energy estimation $\langle E_{\mathsf{T}} \rangle$ 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 $E_{\mathsf{T}}^{(i)}$ does not converge to E_0 in these cases.

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

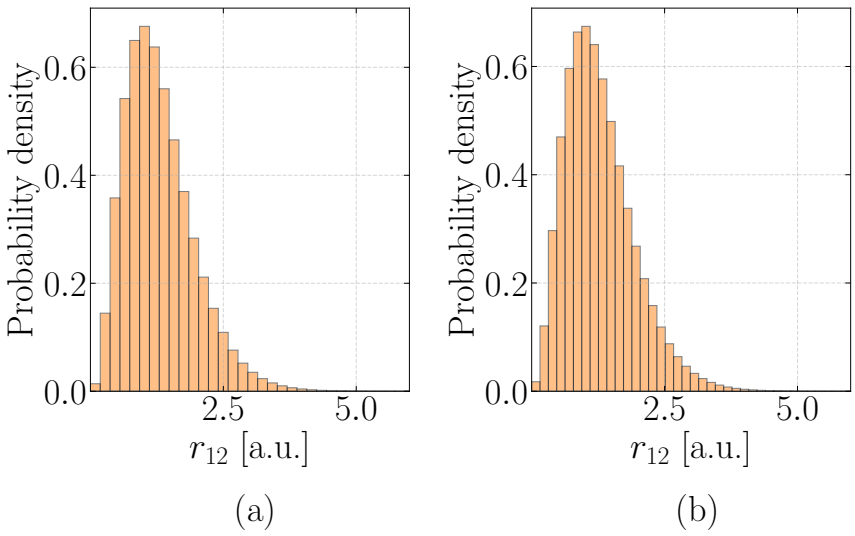


Figure 14: Probability distributions of the distance r_{12} between the two electrons in Helium during the DMC simulations with IS using decomposition 1 **(a)** and 2 **(b)**.

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, $\langle E_\mathsf{T}\rangle(\Delta\tau)$, 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 $\langle E_\mathsf{T}\rangle$ and $\Delta\tau$, thus, a linear fit $E_\mathsf{T} = 0.656\Delta\tau - 2.926$ was made. From the linear fit, an optimal $E_\mathsf{T}^* = -2.926$ was estimated by $\Delta\tau \to 0$. 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 ($E_\mathsf{T} = -0.815\Delta\tau^2 + 0.295\Delta\tau - 2.933$) as $\langle E_\mathsf{T}\rangle$ depended quadratically on $\Delta\tau$. Again, letting $\Delta\tau \to 0$ gives $E_\mathsf{T}^* = -2.933$

Another backwards-engineering approach is to start with the theoretical $E_0 = -2.903$, and the 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.

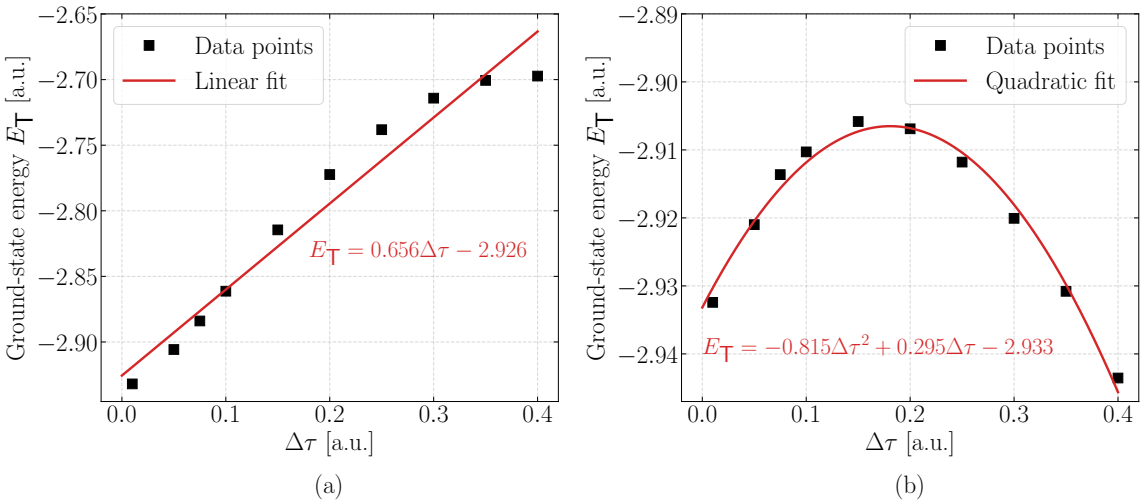


Figure 15: The total average of the ground state energy of Helium, E_T, for different time-steps $\Delta\tau$ using decomposition 1 **(a)** and 2 **(b)** along with a curve fit for both cases.

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 ($\langle E_\mathsf{T}\rangle = -2.911$ 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/~wetzel/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.

[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

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

int
run(
    int argc,
    char *argv[]
   )
{
    // ------------------------------- Constants ------------------------------- //
    gsl_rng *U = init_gsl_rng(19); // Random number generator
    double gamma = 0.5; // Energy damping factor
    DMC_results results; // Results struct
    char filename[100]; // Filename

    // ------------------------------- Task 1a ------------------------------- //
    int N_0 = 200; // Initial number of walkers
    double dtau = 0.02; // Time step
    double tau = 5000; // Total time
    int N_its = tau / dtau; // Number of iterations
    double E_T_start = 0.5; // Initial energy
    int its_eq = 20 / dtau; // Number of equilibration steps

    // Initialize the walkers
    double **walkmen_pos = create_2D_array(N_0 * 100, 1);
    init_walkmen_1D(walkmen_pos, N_0);

    // Equilibrate the walkers
    results = DMC(its_eq, N_0, N_0+1, dtau, E_T_start, U, gamma, NULL, NULL, 1, false,
    ↪  1);
    E_T_start = results.E_T;
    int N_sprinters = results.N_sprinters;

    // Run the simulation
    FILE* fp_ET = fopen("data/task_1/1D/ET_Nwalk_non_eq.csv", "w");
    fprintf(fp_ET, "E_T_avg, N_sprinters, N_survived/N_sprinters (%%), E_T\n");
    FILE* fp_w = fopen("data/task_1/1D/I_was_walkin_in_morse.csv", "w");

    DMC(N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma, fp_ET, fp_w, 1, false, 1);

    // Close the files
    fclose(fp_ET);
    fclose(fp_w);
```

```
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,
        ↪  U, gamma, NULL, NULL, 6, false, 1);
65      double E_T_start = results.E_T;
66      int N_sprinters = results.N_sprinters;
67
68      // Run the simulation
69      sprintf(filename, "data/task_1/6D/ET_Nwalk_non_eq.csv");
70      FILE* fp_ET = fopen(filename, "w");
71      fprintf(fp_ET, "E_T_avg, N_sprinters, N_survived/N_sprinters (%%), E_T\n");
72      FILE* fp_w = fopen("data/task_1/6D/I_was_walkin_in_morse.csv", "w");
73
74      results = DMC(walkmen_pos_init, N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma,
        ↪  fp_ET, NULL, 6, false, 1);
75
76      // Close the files
77      fclose(fp_ET);
78      fclose(fp_w);
79
80      // ---------------------------------- Task 2a ---------------------------------- //
81      dtau = 0.1; // Time step
82      its_eq = 1000 / dtau; // Number of equilibration steps
83      int decomp = 2; // Decomposition
84      N_its = 50000; // Number of iterations
85
86      // Equilibrate the walkers
87      results = DMC(its_eq, N_0, N_sprinters_init, dtau, E_T_start_init, U, gamma, NULL,
        ↪  NULL, 6, true, decomp);
88      double E_T_start = results.E_T;
89      int N_sprinters = results.N_sprinters;
90
91      // Run the simulation
92      sprintf(filename, "data/task_2/ET_Nwalk_non_eq_decomp_%i_dtau_%f.csv", decomp, dtau);
93      FILE *fp_ET_IS = fopen(filename, "w");
94      fprintf(fp_ET_IS, "E_T_avg, N_sprinters, N_survived/N_sprinters (%%), E_T\n");
95
96      sprintf(filename, "data/task_2/I_was_walkin_in_morse_decomp_%i_dtau_%f.csv", decomp,
        ↪  dtau);
97      FILE* fp_w_IS = fopen(filename, "w");
98      results = DMC(N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma, fp_ET_IS, fp_w_IS,
        ↪  6, true, decomp);
99
100     // Close the files
```

```
    fclose(fp_ET_IS);
    fclose(fp_w_IS);

    // ---------------------------------- Task 2b ---------------------------------- //
    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
    decomp = 1; // Decomposition

    // File to store the results
    sprintf(filename, "data/task_2/ET_Nwalks_%i.csv", decomp);
    FILE *fp_ET_IS = fopen(filename, "w");
    fprintf(fp_ET_IS, "dtau, E_T_avg\n");

    // Run the simulation
    for (int i = 0; i < 10; i++)
    {
        dtau = dtaus[i]; // Time step
        printf("dtau: %f\n", dtau);

        // Equilibrate the walkers
        results = DMC(its_eq, N_0, N_sprinters_init, dtau, E_T_start_init, U, gamma, NULL, NULL, 6, true, decomp);
        double E_T_start = results.E_T;
        int N_sprinters = results.N_sprinters;

        // Run the simulation
        results = DMC(N_its, N_0, N_sprinters, dtau, E_T_start, U, gamma, NULL, NULL, 6, true, decomp);
        fprintf(fp_ET_IS, "%f, %f\n", dtau, results.E_T_avg);
        printf("E_T_avg: %f\n", results.E_T_avg);
    }

    // Close the file
    fclose(fp_ET_IS);

    return 0;
}

DMC_results
DMC(
    int N_its,
    int N_0,
    int N_sprinters,
    double dtau,
    double E_T_start,
    gsl_rng *U,
    double gamma,
    FILE *fp_ET,
    FILE *fp_w,
    int ndim,
    bool IS,
    int decomp
   )
{
    double E_T = E_T_start; // Initial energy
    double E_T_avg = E_T_start; // Initial average energy
    DMC_results results; // Results struct
```

```
    int N_survivers = N_sprinters; // Number of surviving walkers
    double **walkmen_pos = create_2D_array(N_sprinters, ndim); // Walkers positions
    // Initialize the walkers
    if (ndim == 1)
    {
        init_walkmen_1D(walkmen_pos, N_sprinters);
    }
    else
    {
        init_walkmen_6D(walkmen_pos, N_sprinters, U);
    }
    for (int i = 0; i < N_its ; i++)
    {
        // Generate new positions
        update_positions(walkmen_pos, N_survivers, ndim, dtau, fp_w, U, IS, decomp);
        int *num_walkers = (int *)calloc(N_sprinters, sizeof(int)); // Array to store the
        ↪  offsprings of each walker
        // Calculate the offspring of each walker
        for(int j = 0; j < N_sprinters; j++)
        {
            if (ndim == 1)
            {
                int m = (int)(weight(dtau, E_T, morse(walkmen_pos[j][0])) +
                ↪  gsl_rng_uniform(U) * 1.); // Number of offsprings
                num_walkers[j] = m; // Store the number of offsprings
            }
            else if (ndim == 6)
            {
                if (IS)
                {
                    double *R = walkmen_pos[j]; // Walker's position
                    double E_L = local_energy(R, 0.15); // Local energy
                    int m = (int)(weight(dtau, E_T, E_L) + gsl_rng_uniform(U) * 1.); //
                    ↪  Number of offsprings
                    num_walkers[j] = m; // Store the number of offsprings
                }
                else
                {
                    // Convert to spherical coordinates
                    double r_1 = sqrt(pow(walkmen_pos[j][0], 2) + pow(walkmen_pos[j][1],
                    ↪  2) + pow(walkmen_pos[j][2], 2));
                    double r_2 = sqrt(pow(walkmen_pos[j][3], 2) + pow(walkmen_pos[j][4],
                    ↪  2) + pow(walkmen_pos[j][5], 2));
                    double r_12 = sqrt(pow(walkmen_pos[j][3] - walkmen_pos[j][0], 2) +
                    ↪  pow(walkmen_pos[j][4] - walkmen_pos[j][1], 2) +
                    ↪  pow(walkmen_pos[j][5] - walkmen_pos[j][2], 2));
                    double V_tot = - 2 / sqrt(pow(r_1, 2) + 1e-4) - 2 / sqrt(pow(r_2, 2)
                    ↪  + 1e-4) + 1 / sqrt(pow(r_12, 2) + 1e-4); // Potential energy
                    int m = (int)(weight(dtau, E_T, V_tot) + gsl_rng_uniform(U) * 1.); //
                    ↪  Number of offsprings
                    num_walkers[j] = m; // Store the number of offsprings
                }
            }
        }
        N_survivers = int_sum(num_walkers, N_sprinters); // Number of surviving walkers
        double **walkmen_pos_new = create_2D_array(N_survivers, ndim); // New array to
        ↪  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
                ↪  (num_walkers[j] > 0)
208             {
209                 for(int n = 0; n < ndim; n++)
210                 {
211                     walkmen_pos_new[M][n] = walkmen_pos[k][n]; // Copy walker's position
212                 }
213                 M++; // Increment the new array index
214             }
215         }
216         destroy_2D_array(walkmen_pos); // Free the memory of the old array
217         walkmen_pos = create_2D_array(N_survivers, ndim); // Create a new array to store
            ↪  the walkers
218         // Update the walker's position
219         for (int k = 0; k < N_survivers; k++)
220         {
221             for (int n = 0; n < ndim; n++)
222             {
223                 walkmen_pos[k][n] = walkmen_pos_new[k][n];
224             }
225         }
226
227         // Updating ET
228         double sprinter_ratio = (double)N_survivers / (double)N_0; // Ratio of surviving
            ↪  walkers
229         E_T = E_T_avg - gamma * log(sprinter_ratio); // E_T at i+1
230         E_T_avg = E_T / (i+1) + E_T_avg * i / (i+1); // E_T_avg at i+1
231
232         // Print the results to the file
233         if (fp_ET != NULL)
234         {
235             fprintf(fp_ET, "%lf, %i, %f, %f\n", E_T_avg, N_survivers, (100. *
                ↪  (double)N_survivers) / (double)N_sprinters, E_T);
236         }
237         N_sprinters = N_survivers; // Update the number of sprinters
238
239         // Free the memory
240         destroy_2D_array(walkmen_pos_new);
241         free(num_walkers);
242     }
243     // Store the results in the struct
244     results.E_T = E_T;
245     results.E_T_avg = E_T_avg;
246     results.N_sprinters = N_sprinters;
247     results.R = walkmen_pos;
248
249     return results;
250 }
251
252 void
253 init_walkmen_1D(
254             double **walkers,
```

```
             int N_walkers
            )
{
    int j = 0; // Index
    // Initialize the walkers
    for (double i = -5; i < 5; i+=10./N_walkers)
    {
        walkers[j][0] = i;
        j+=1;
    }
}

void
init_walkmen_6D(
                double **walkers,
                int N_walkers,
                gsl_rng *U
               )
{
    // Initialize the walkers
    for(int i = 0; i < N_walkers; i++)
    {
        double r_1 = 0.7 + gsl_rng_uniform(U);
        double theta_1 = acos(2 * gsl_rng_uniform(U) - 1);
        double phi_1 = 2 * M_PI * gsl_rng_uniform(U);
        double r_2 = 0.7 + gsl_rng_uniform(U);
        double theta_2 = acos(2 * gsl_rng_uniform(U) - 1);
        double phi_2 = 2 * M_PI * gsl_rng_uniform(U);

        walkers[i][0] = r_1 * sin(theta_1) * cos(phi_1); // x_1
        walkers[i][1] = r_1 * sin(theta_1) * sin(phi_1); // y_1
        walkers[i][2] = r_1 * cos(theta_1); // z_1
        walkers[i][3] = r_2 * sin(theta_2) * cos(phi_2); // x_2
        walkers[i][4] = r_2 * sin(theta_2) * sin(phi_2); // y_2
        walkers[i][5] = r_2 * cos(theta_2); // z_2
    }
}

double
morse(
      double x
     )
{
    double V = 0.5 * pow((1  - exp(-x)),2); // Morse potential

    return V;
}

double
weight(
       double dtau,
       double E_T,
       double V
      )
{
    double W = exp((E_T - V) * dtau); // Weight of the walker

```

```
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_survivers,
```

```
                 int ndim,
                 double dtau,
                 FILE *fp_w,
                 gsl_rng *U,
                 bool IS,
                 int decomp
                )
{
    // Update the positions of the walkers
    for (int l = 0; l < N_survivers; l++)
    {
        for (int n = 0; n < ndim; n++)
        {
            // Diffusive part
            walkmen_pos[l][n] = walkmen_pos[l][n] + gsl_ran_gaussian(U, 1.) * sqrt(dtau);
        }
        if (IS)
        {
            // Drift velocity
            drift_velocity(walkmen_pos[l], 0.15, dtau, decomp, U);
        }
        // Write the walker's position to the file
        if (fp_w != NULL)
        {
            for (int n = 0; n < ndim; n++)
            {
                if (ndim == 1)
                {
                    fprintf(fp_w, "%lf\n", walkmen_pos[l][0]);
                }
                else
                {
                    if (n < ndim - 1)
                    {
                        fprintf(fp_w, "%lf, ", walkmen_pos[l][n]);
                    }
                    else
                    {
                        fprintf(fp_w, "%lf\n", walkmen_pos[l][n]);
                    }
                }
            }
        }
    }
}

void
drift_velocity(
                double *R,
                double alpha,
                double dtau,
                int decomp,
                gsl_rng *U
              )
{
    double x_1 = R[0];
    double y_1 = R[1];
```

```
    double z_1 = R[2];
    double x_2 = R[3];
    double y_2 = R[4];
    double z_2 = R[5];

    double r_1 = sqrt(pow(x_1, 2) + pow(y_1, 2) + pow(z_1, 2)); // Magnitude of the first electron
    double r_2 = sqrt(pow(x_2, 2) + pow(y_2, 2) + pow(z_2, 2)); // Magnitude of the second electron

    double r12_vec[3] = {x_2 - x_1, y_2 - y_1, z_2 - z_1}; // Vector between the electrons
    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

    double r1_hat[3] = {x_1 / r_1, y_1 / r_1, z_1 / r_1}; // Unit vector of the first electron
    double r2_hat[3] = {x_2 / r_2, y_2 / r_2, z_2 / r_2}; // Unit vector of the second electron
    double r12_hat[3] = {r12_vec[0] / r12, r12_vec[1] / r12, r12_vec[2] / r12}; // Unit vector between the electrons

    // Compute drift velocity for the electrons
    double v_F1[3];
    double v_F2[3];
    for (int i = 0; i < 3; i++)
    {
        v_F1[i] = - 2 * r1_hat[i] - (1.0 / (2. * pow(1 + alpha * r12, 2))) * r12_hat[i];
        v_F2[i] = - 2 * r2_hat[i] + (1.0 / (2. * pow(1 + alpha * r12, 2))) * r12_hat[i];
    }

    // Second decomposition
    if (decomp == 2)
    {
        // First half step
        x_1 += v_F1[0] * dtau / 2;
        y_1 += v_F1[1] * dtau / 2;
        z_1 += v_F1[2] * dtau / 2;
        x_2 += v_F2[0] * dtau / 2;
        y_2 += v_F2[1] * dtau / 2;
        z_2 += v_F2[2] * dtau / 2;
        double r_1 = sqrt(pow(x_1, 2) + pow(y_1, 2) + pow(z_1, 2)); // Magnitude of the first electron
        double r_2 = sqrt(pow(x_2, 2) + pow(y_2, 2) + pow(z_2, 2)); // Magnitude of the second electron

        double r12_vec[3] = {x_2 - x_1, y_2 - y_1, z_2 - z_1}; // Vector between the electrons
        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

        double r1_hat[3] = {x_1 / r_1, y_1 / r_1, z_1 / r_1}; // Unit vector of the first electron
        double r2_hat[3] = {x_2 / r_2, y_2 / r_2, z_2 / r_2}; // Unit vector of the second electron
        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
```

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

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

int
int_sum(
        int *v,
        int len
       )
{
    int sum = 0;
    for(int i = 0;i < len; i++)
    {
        sum += v[i];
    }
    return sum;
}

void
addition_with_constant(
                       double *res,
                       double *v,
                       double constant,
                       unsigned int len)
{
    for (int i = 0; i < len; i++) {
        res[i] = v[i] + constant;
    }
}

void
multiplication_with_constant(
                             double *res,
                             double *v,
                             double constant,
                             unsigned int len)
```

```
{
    for (int i = 0; i < len; i++) {
        res[i] = v[i] * constant;
    }
}

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

double **
create_2D_array(
                unsigned int row_size,
                unsigned int column_size
               )
{
    // Allocate memory for row pointers
    double **array = malloc(row_size * sizeof(double *));
    if (array == NULL) {
        fprintf(stderr, "Memory allocation failed for row pointers.\n");
        return NULL;
    }

    // Allocate a single contiguous block for all elements, i.e. all matrix elements
    // get stored in a single block of memory, array[0] is the start of the block
    // i.e. here we have array[0][index] as the way to access the elements
    array[0] = malloc(row_size * column_size * sizeof(double));
    if (array[0] == NULL) {
        fprintf(stderr, "Memory allocation failed for data block.\n");
        free(array);
        return NULL;
    }

    // Set the row pointers to the appropriate positions in the block, in order to
    // allow for the array[i][j] syntax. Now array[i] points to the start of the
    // i-th row.
    for (int i = 1; i < row_size; i++) {
        array[i] = array[0] + i * column_size;
    }

    return array;
}

void
destroy_2D_array(
                 double **array
                )
```

```
{
    if (array != NULL) {
        free(array[0]);  // Free the contiguous block
        free(array);      // Free the row pointers
    }
}

void
matrix_vector_multiplication(
                             double *result,
                             double **A,
                             double *b,
                             unsigned int n,
                             unsigned int m
                            )
{
    for (int i = 0; i < n; i++) {
        result[i] = 0.;
        for (int j = 0; j < m; j++) {
            result[i] += A[i][j] * b[j];
        }
    }
}

void
matrix_matrix_multiplication(
                             double **result,
                             double **A,
                             double **B,
                             unsigned int n,
                             unsigned int m,
                             unsigned int k
                            )
{
    for (int i = 0; i < n; i++) {
        for (int kappa = 0; kappa < k; kappa++) {
            result[i][kappa] = 0.;
            for (int j = 0; j < m; j++) {
                result[i][kappa] += A[i][j] * B[j][kappa];
            }
        }
    }
}

double
vector_norm(
            double *v1,
            unsigned int len
           )
{
    double res = 0.;
    for (int i = 0; i < len; i++) {
        res += v1[i] * v1[i];
    }
    return sqrt(res);
}

```

```
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                )
```

```
{
    double corr = 0.;
    double avg = average(data, data_len);
    double var = variance(data, data_len);

    for (int i = 0; i < data_len - time_lag_ind; i++)
    {
        corr += (data[i]*data[i+time_lag_ind] - avg*avg) / var;
    }

    return corr / (data_len - time_lag_ind);
}

double
block_average(
              double *data,
              int data_len,
              int block_size
             )
{
    int num_blocks = data_len / block_size;
    double *block_averages = (double *)calloc(num_blocks, sizeof(double));
    for (int i = 0; i < num_blocks; i++) {
        double sum = 0.0;
        for (int j = 0; j < block_size; j++) {
            sum += data[i * block_size + j];
        }
        block_averages[i] = sum / block_size;
    }
    double block_avg = block_size * variance(block_averages, num_blocks) / variance(data,
    ↪  data_len);
    free(block_averages);

    return block_avg;
}

double
distance_between_vectors(
                         double *v1,
                         double *v2,
                         unsigned int len
                        )
{
    double res = 0.;
    // With previous defined functions
    double *diff = malloc(len * sizeof(double));
    multiplication_with_constant(v1, v1, -1., len);
    elementwise_addition(diff, v1, v2, len);
    res = vector_norm(diff, len);
    free(diff);
    return res;
}

void
cumulative_integration(
                       double *res,
                       double *v,
```

```
                        double dx,
                        unsigned int v_len
                       )
{
    double sum = 0.;
    res[0] = 0.;
    for (int i = 1; i < v_len; i++) {
        sum = 0.5 * (v[i - 1] + v[i]) * dx;
        res[i] = res[i - 1] + sum;
    }
}

void
write_xyz(
          FILE *fp,
          char *symbol,
          double **positions,
          double **velocities,
          double alat,
          int natoms
         )
{
    fprintf(fp, "%i\nLattice=\"%f 0.0 0.0 0.0 %f 0.0 0.0 0.0 %f\" ", natoms, alat, alat,
    ↪  alat);
    fprintf(fp, "Properties=species:S:1:pos:R:3:vel:R:3 pbc=\"T T T\"\n");
    for(int i = 0; i < natoms; ++i){
        fprintf(fp, "%s %f %f %f %f %f %f\n",
                symbol, positions[i][0], positions[i][1], positions[i][2],
                velocities[i][0], velocities[i][1], velocities[i][2]);
    }
}

void fft_freq(
              double *res,
              int n,
              double timestep
             )
{
    for (int i = 0; i < n; i++) {
        if (i < n / 2) {
            res[i] = 2 * M_PI * i / (n * timestep);
        }
        else {
            res[i] = 2 * M_PI * (i - n) / (n * timestep);
        }
    }
}

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

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

B Source code in Python

```python
# Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Latex style
plt.style.use('default')
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rc('font', size=25.75)
plt.rcParams['text.latex.preamble'] = r'\usepackage{amsmath}'
plt.rcParams['text.latex.preamble'] = r'\usepackage{bm}'
# Set ticks on both sides
plt.rcParams['xtick.direction'] = 'in'
plt.rcParams['ytick.direction'] = 'in'
plt.rcParams['xtick.major.size'] = 5
plt.rcParams['ytick.major.size'] = 5
plt.rcParams['xtick.top'] = True
plt.rcParams['ytick.right'] = True

# Functions
def read_data(task, ndim, decomp=1, dtau=0.01, walkmen=False, eq=False):
    '''
    Read data from csv files.

    Args:
        task (int): Task number.
        ndim (int): Number of dimensions.
        decomp (int): Decomposition number.
        dtau (float): Time step.
        walkmen (bool): If True, read walkmen data.
        eq (bool): If True, read equilibrium data.

    Returns:
        if task == 1 and eq:
            data_eq (np.ndarray): Equilibrium data.
        elif task == 1 and not eq:
            data_ET (np.ndarray): Non-equilibrium data.
            data_walkmen (np.ndarray): Walkmen data.
        elif task == 2 and walkmen:
            data_ET (np.ndarray): Non-equilibrium data.
            data_extrapol (np.ndarray): Extrapolated data.
            R (list): List of arrays with the R values.
            r12 (np.ndarray): Distance between walkers.
        elif task == 2 and not walkmen:
            data_ET (np.ndarray): Non-equilibrium data.
            data_extrapol (np.ndarray): Extrapolated data.
    '''
    if task == 1:
        if eq:
            path_eq = f'data/task_{task}/{ndim}D/ET_Nwalk_eq_run.csv'
            data_eq = pd.read_csv(path_eq, delimiter=',', on_bad_lines='skip')
            data_eq = data_eq.apply(pd.to_numeric, errors='coerce')
            data_eq = data_eq.fillna(np.nan)
            data_eq = np.asarray(data_eq)
```

```
            return data_eq
        else:
            path_ET = f'data/task_{task}/{ndim}D/ET_Nwalk_non_eq.csv'
            data_ET = pd.read_csv(path_ET, delimiter=',', on_bad_lines='skip')
            data_ET = data_ET.apply(pd.to_numeric, errors='coerce')
            data_ET = data_ET.fillna(np.nan)
            data_ET = np.asarray(data_ET)

            path_walkmen = f'data/task_{task}/{ndim}D/I_was_walkin_in_morse.csv'
            try:
                data_walkmen = pd.read_csv(path_walkmen, delimiter=',',
                ↪  on_bad_lines='skip')
                data_walkmen = data_walkmen.apply(pd.to_numeric, errors='coerce')
                data_walkmen = data_walkmen.fillna(np.nan)
                data_walkmen = data_walkmen[~np.isnan(data_walkmen)]
            except (FileNotFoundError, pd.errors.EmptyDataError, pd.errors.ParserError)
            ↪  as e:
                print(f"Error reading {path_walkmen}: {e}")
                data_walkmen = None

            return data_ET, data_walkmen

    elif task == 2:
        path_ET = f'data/task_{task}/ET_Nwalk_non_eq_decomp_{decomp}_dtau_{dtau:.6f}.csv'
        data_ET = np.asarray(pd.read_csv(path_ET, delimiter=','))

        path_extrapol = f'data/task_{task}/ET_Nwalks_{decomp}.csv'
        data_extrapol = np.asarray(pd.read_csv(path_extrapol, delimiter=','))

        if walkmen:
            path_walkmen =
            ↪  f'data/task_{task}/I_was_walkin_in_morse_decomp_{decomp}_dtau_{dtau:.6f}.csv'
            data_walkmen = np.asarray(pd.read_csv(path_walkmen, delimiter=',',
            ↪  on_bad_lines='skip'))

            # Convert to spherical coordinates
            r_1 = np.sqrt(data_walkmen[:, 0]**2 + data_walkmen[:, 1]**2 + data_walkmen[:,
            ↪  2]**2)
            theta_1 = np.arccos(data_walkmen[:, 2] / r_1)
            phi_1 = np.arctan2(data_walkmen[:, 1], data_walkmen[:, 0])
            r_2 = np.sqrt(data_walkmen[:, 3]**2 + data_walkmen[:, 4]**2 + data_walkmen[:,
            ↪  5]**2)
            theta_2 = np.arccos(data_walkmen[:, 5] / r_2)
            phi_2 = np.arctan2(data_walkmen[:, 4], data_walkmen[:, 3])

            # Distance between walkers
            r12 = np.sqrt((data_walkmen[:, 0] - data_walkmen[:, 3])**2 + (data_walkmen[:,
            ↪  1] - data_walkmen[:, 4])**2 + (data_walkmen[:, 2] - data_walkmen[:,
            ↪  5])**2)

            # Total data
            R = [r_1, theta_1, phi_1, r_2, theta_2, phi_2]

            return data_ET, data_extrapol, R, r12
        else:
            return data_ET, data_extrapol
```

```python

def plot_GSE(ET, tau, dtau, E_0, ndim, task, save=False):
    '''
    Plot the ground-state energy as a function of time.

    Args:
        ET (np.ndarray): Ground-state energy.
        tau (np.ndarray): Time.
        dtau (float): Time step.
        E_0 (float): Theoretical ground-state energy.
        ndim (int): Number of dimensions.
        task (int): Task number.
        save (bool): If True, save the figure.
    '''
    fig, ax = plt.subplots(1, 2, figsize=(17, 8))

    # Plot the energy as a function of time
    ax[0].plot(tau, ET, color='tab:red', alpha=0.5)
    ax[0].hlines(E_0, 0, len(ET)*dtau, color='tab:red', label="$E_0$"+f"$\\,={E_0:.3f}$",lw=2.5)
    ax[0].hlines(np.mean(ET), 0, len(ET)*dtau, colors='k', label='$\\langle \\mathcal{E}\\rangle$'+f'$\\, = {np.mean(ET):.3f}$',lw=2.5,ls = 'dashed')
    ax[0].legend()
    ax[0].set_xlabel('Time $\\tau$ [a.u.]')
    ax[0].set_ylabel('Ground-state energy $\\mathcal{E}_i$ [a.u.]')
    # ax[0].set_ylim(-3.2, -2.4)
    ax[0].legend(loc='upper center', ncol=2)

    # Plot the histogram of the ground-state energy
    ax[1].hist(ET, bins=30, density=True, color='tab:red', alpha=0.5, edgecolor='k')
    std = np.std(ET)
    ax[1].vlines(x=np.mean(ET)-std, ymin=0, ymax=44, color='k', linestyle='dashed', lw=1.5, alpha=0.5)
    ax[1].vlines(x=np.mean(ET)+std, ymin=0, ymax=44, color='k', linestyle='dashed', lw=1.5, alpha=0.5)
    ax[1].vlines(E_0, 0, 44, color='tab:red', label="$E_0$"+f"$\\,={E_0:.3f}$",lw=2.5)
    ax[1].vlines(np.mean(ET), 0, 44, colors='k', label='$\\langle \\mathcal{E}\\rangle$'+f'$\\, = {np.mean(ET):.3f}$',lw=2.5,ls = 'dashed')
    ax[1].set_xlabel('Ground-state energy $\\mathcal{E}$ [a.u.]')
    ax[1].set_ylabel('Probability density')
    ax[1].set_ylim(0, 51)
    ax[1].legend(loc='upper center', ncol=2)

    # ax[1].set_xlim(-3.2, -2.5)
    # ax[0].text(2400, -2.965, '(a)')
    # ax[1].text(-2.912, -11, '(b)')
    plt.tight_layout()

    # Save figure
    if save:
        save_fig(plt, 'E_T', task, ndim)

    plt.show()

def plot_walkmen(tau, walk, ndim=1, task=1, save=False):
    '''
    Plot the number of walkers as a function of time.
```

```
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='$\\langle
    ↪  N_{walkers}\\rangle$'+f'$\\,= {np.mean(walk):.0f}$',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',
    ↪  lw=1.5, alpha=0.5)
181     ax[1].vlines(x=np.mean(walk)+std, ymin=0, ymax=0.0235, color='k', linestyle='dashed',
    ↪  lw=1.5, alpha=0.5)
182     ax[1].vlines(np.mean(walk), 0, 0.0235, colors='k', linestyle='dashed',
    ↪  label='$\\langle N_{walkers}\\rangle$'+f'$\\,= {np.mean(walk):.0f}$',lw=2.5)
183     ax[1].set_ylim(0, 0.027)
184     ax[1].set_xlim(925, 1080)
185     ax[1].legend(loc='upper left')
186
187     ax[0].text(2400, 886, '(a)')
188     ax[1].text(997, -0.006, '(b)')
189     plt.tight_layout()
190
191     # Save figure
192     if save:
193         save_fig(plt, 'N_walkers', task, ndim)
194
195     plt.show()
196
197 def plot_Rhist(R, decomp, save=False):
198     '''
199     Plot the histogram of the walkers' positions.
200
201     Args:
202         R (list): List of arrays with the R values.
203         decomp (int): Decomposition number.
204         save (bool): If True, save the figure.
205     '''
206     fig, axs = plt.subplots(2, 3, figsize=(18, 12), sharex='col')
207     # Plot the histogram of the walkers' positions for the first electron
208     axs[0][0].hist(R[0], bins=50, density=True, color='tab:blue', alpha=0.5,
    ↪  edgecolor='k', label='$r_1$')
```

```
    axs[0][1].hist(R[1], bins=50, density=True, color='tab:green', alpha=0.5, edgecolor='k', label='$\\theta_1$')
    axs[0][2].hist(R[2], bins=50, density=True, color='tab:red', alpha=0.5, edgecolor='k', label='$\\phi_1$')

    # Plot the histogram of the walkers' positions for the second electron
    axs[1][0].hist(R[3], bins=50, density=True, color='tab:blue', alpha=0.5, edgecolor='k', label='$r_2$')
    axs[1][1].hist(R[4], bins=50, density=True, color='tab:green', alpha=0.5, edgecolor='k', label='$\\theta_2$')
    axs[1][2].hist(R[5], bins=50, density=True, color='tab:red', alpha=0.5, edgecolor='k', label='$\\phi_2$')

    # Set labels and ticks
    for i in range(2):
        axs[i][0].set_xlabel(f'$r_{i+1}$ [a.u.]')
        axs[i][0].text(1.4, 0.8, '\\textbf{{a)}}' if i == 0 else '\\textbf{{d)}}')
        axs[i][1].set_xlabel(f'$\\theta_{i+1}$ [rad]')
        axs[i][1].set_xticks([0, np.pi/2, np.pi])
        axs[i][1].set_xticklabels(['0', '$\\pi / 2$', '$\\pi$'])
        axs[i][1].set_yticks([0, 0.1, 0.2, 0.3, 0.4, 0.5])
        axs[i][1].text(0.3, 0.457, '\\textbf{{b)}}' if i == 0 else '\\textbf{{e)}}')
        axs[i][2].set_xlabel(f'$\\phi_{i+1}$ [rad]')
        axs[i][2].set_xticks([-np.pi, 0, np.pi])
        axs[i][2].set_xticklabels(['$-\\pi$', '0', '$\\pi$'])
        axs[i][2].set_ylim(0, 0.21)
        axs[i][2].text(-2.5, 0.185, '\\textbf{{c)}}' if i == 0 else '\\textbf{{f)}}')

    for ax in axs.flatten():
        ax.grid(alpha=0.5, linestyle='--', lw=1)
        ax.legend()
        ax.set_ylabel('Probability density')

    plt.tight_layout()

    # Save figure
    if save:
        save_fig(plt, f'R_hist_{decomp}', 2, 6)

    plt.show()

def save_fig(fig, name, task, ndim):
    '''
    Save figure as pdf.

    Args:
        fig (plt.figure): Figure to save.
        name (str): Name of the figure.
        task (int): Task number.
        ndim (int): Number of dimensions.
    '''
    fig.savefig(f'figs/task_{task}/{ndim}D/{name}.pdf', bbox_inches='tight')

######################################## Task 1 ########################################
# Plot the Morse potential and the wavefunction
x = np.linspace(-2.5, 10, 10000)
psi = np.exp(-np.exp(-x) - x/2) / np.sqrt(np.pi)
```

```
Morse = 0.5 * (1 - np.exp(-x)) ** 2

# Morse potential
plt.figure(figsize=(8, 5))
plt.plot(x, Morse, label='Morse potential $V(x)$', color='k', lw=2.5)
plt.xlabel('$x$ [a.u.]')
plt.ylabel('$V(x)$ [a.u.]')
plt.grid(alpha=0.5, linestyle='--', lw=1)
plt.ylim(-.5, 5)
plt.legend()
plt.tight_layout()
# save_fig(plt, 'Morse', 1, 1)

plt.figure(figsize=(8, 5))

# Read walker data
_, walker_pos = read_data(1, 1)
bin_size = 150
counts, bins = np.histogram(walker_pos, bins=bin_size)

# Normalize manually
bin_width = bins[1] - bins[0]  # Width of each bin
normalized_counts = counts / (np.sum(counts) * bin_width)

# Plot the manually normalized histogram
plt.hist(walker_pos, bins=int(bin_size/2), density=True, color='tab:blue', alpha=0.2,
↪  edgecolor='black', lw=1)
plt.plot(bins[:-1], normalized_counts, color='tab:blue', lw=3,
↪  label='$\\Psi_\\mathrm{Sample}(x)$')
plt.plot(x, psi, label='$\\Psi_\\mathrm{Analytic}(x)$', color = 'k', lw=2.5, ls =
↪  'dashed',alpha = 0.7)
plt.xlabel('$x$ [a.u.]')
plt.ylabel('$\\Psi(x)$')
plt.grid(alpha=0.5, linestyle='--', lw=1)

plt.legend()
plt.tight_layout()
# save_fig(plt, 'Wavefunc', 1, 1)

plt.show()

# Read data
ET_data, _ = read_data(1, 1)
ET_avg, walk, mort_rate, ET = ET_data.T
tau = np.linspace(0, len(ET) * 0.02, len(ET))
save = False

# Plot the ground-state energy and the number of walkers
plot_GSE(ET, tau, 0.02, 3/8, 1, 1, save)

plot_walkmen(tau, walk, save)

# plt.figure()
# #mortality_rate = read_data('ET_Nwalk_non_eq.csv')[:,2]
# plt.plot(tau, mort_rate)
# plt.hlines(100,0,np.max(tau),'b', linestyle = 'dashed')
# plt.hlines(np.mean(mort_rate), 0, np.max(tau),colors='k', label='Mortality rate'+f' =
↪  {np.mean(mort_rate):.1f}\\%')
```

```
# plt.legend()

# Read data
ET_avg_eq, walk_eq, mort_rate_eq, ET_eq = read_data(1, 1, eq=True)[:, :].T
ET_avg_eq = np.insert(ET_avg_eq, 0, 0.5)
walk_eq = np.insert(walk_eq, 0, 200)
ET_eq = np.insert(ET_eq, 0, 0.5)

# Plot the ground-state energy and the number of walkers in equilibrium
fig, ax = plt.subplots(1, 2, figsize=(17, 8))
ax[0].plot(tau[:len(ET_eq)], ET_eq,color='tab:red', alpha=0.4, lw=2.5,
↪  label='$\\mathcal{E}_i$')
ax[0].plot(tau[:len(ET_avg_eq)], ET_avg_eq, color='tab:red', alpha=1, lw=2.5,
↪  label='$E_\\textsf{T}^{(i)}$')
ax[1].plot(tau[:len(walk_eq)], walk_eq, color='tab:green', alpha=0.75, lw=2.5,
↪  label='$N_i$')
ax[0].set_xlabel('Time $\\tau$ [a.u.]')
ax[0].set_ylabel('Ground-state energy $\\mathcal{E}_i$ [a.u.]')
ax[0].hlines(3/8, 0, len(ET_eq) * 0.02, color='k', label="$E_0$"+f"$\\,={3/8:.3f}$",
↪  lw=2, ls = 'dashed')
ax[1].hlines(200, 0, len(walk_eq) * 0.02, color='k', label="$N_0$", lw=2, ls='dashed')

ax[1].set_xlabel('Time $\\tau$ [a.u.]')
ax[1].set_ylabel('Number of walkers $N_i$')

ax[0].text(74, 0.01, '(a)')
ax[1].text(74, 50, '(b)')
ax[0].legend(loc='upper right')
ax[1].legend(loc='upper right')
plt.tight_layout()
# save_fig(plt, 'ET_Nwalk_eq', 1)

plt.show()

######################################## Task 2 ########################################
# Read data
data_ET, _ = read_data(1, 6)
ET_avg, walk, mort_rate, ET = data_ET.T
dtau = 0.01
tau = np.linspace(0, len(ET) * dtau, len(ET))
E_0_He = -2.903
save = False

# Plot the ground-state energy and the number of walkers
plot_GSE(ET, tau, dtau, E_0_He, ndim=6, task=1, save=save)
plot_walkmen(tau, walk, ndim=6, task=1, save=save)

######################################## Task 3 ########################################
dtau = 0.1
E_0_He = -2.903

# Read data for the first decomposition
data_ET_1, data_extrapol_1 = read_data(2, 1, decomp=1, dtau=dtau)
ET_avg_1, walk_1, mort_rate_1, ET_1 = data_ET_1.T

# Read data for the second decomposition
data_ET_2, data_extrapol_2 = read_data(2, 1, decomp=2, dtau=dtau)
```

```python
ET_avg_2, walk_2, mort_rate_2, ET_2 = data_ET_2.T

# Plot the ground-state energy and the number of walkers for the first decomposition
tau = np.linspace(0, len(ET_1) * dtau, len(ET_1))
plot_GSE(ET_1, tau, dtau, E_0_He, ndim=6, task=2, save=False)
plot_walkmen(tau, walk_1, ndim=6, task=2, save=False)

# Plot the ground-state energy and the number of walkers for the second decomposition
plot_GSE(ET_2, tau, dtau, E_0_He, ndim=6, task=2, save=False)
plot_walkmen(tau, walk_2, ndim=6, task=2, save=False)

# Read the walker data for the first and second decomposition
_, _, R_1, r12_1 = read_data(2, 6, decomp=1, dtau=dtau, walkmen=True)
_, _, R_2, r12_2 = read_data(2, 6, decomp=2, dtau=dtau, walkmen=True)

# Plot the histogram of the distance between walkers for the first and second
↪  decomposition
fig, axs = plt.subplots(1, 2, figsize=(12, 7))
axs[0].hist(r12_1, bins=50, density=True, color='tab:orange', alpha=0.5, edgecolor='k')
axs[1].hist(r12_2, bins=50, density=True, color='tab:orange', alpha=0.5, edgecolor='k')

# Set labels and ticks
for ax in axs:
    ax.set_xticks([2.5, 5])
    ax.set_xlim(0, 6)
    ax.set_xlabel('$r_{12}$ [a.u.]')
    ax.set_ylabel('Probability density')
    ax.grid(alpha=0.5, linestyle='--', lw=1)

plt.tight_layout()
axs[0].text(2.7, -0.175, '(a)')
axs[1].text(2.7, -0.175, '(b)')
# save_fig(plt, 'r12', 2, 6)

# Plot the histogram of the walkers' positions
plot_Rhist(R_1, 1, save=False)
plot_Rhist(R_2, 2, save=False)


######################################### Task 4 ########################################
# Fit the extrapolated data
fit_1 = np.polyfit(data_extrapol_1[:,0], data_extrapol_1[:,1], 1)
fit_2 = np.polyfit(data_extrapol_2[:,0], data_extrapol_2[:,1], 2)
dtaus = np.linspace(0, 0.4, 1000)

print(f"Best estimation of E_0 for decomp 1: {fit_1[1]:.3f}")
print(f"Best estimation of E_0 for decomp 2: {fit_2[2]:.3f}")

# Plot the extrapolated data
fig, axs = plt.subplots(1, 2, figsize=(17, 8))
axs[0].plot(data_extrapol_1[:,0], data_extrapol_1[:,1], 's', label='Data points',
↪  color='k', markersize=10)
axs[0].plot(dtaus, np.polyval(fit_1, dtaus), label='Linear fit', color='tab:red', lw=2.5)

axs[1].plot(data_extrapol_2[:,0], data_extrapol_2[:,1], 's', label='Data points',
↪  color='k', markersize=10)
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'$E_\\textsf{{T}} = {fit_1[0]:.3f}\\Delta \\tau
    ↪  {fit_1[1]:.3f}$', color='tab:red', fontsize=24)
436 axs[1].text(0., -2.94, f'$E_\\textsf{{T}} = {fit_2[0]:.3f}\\Delta \\tau^2 +
    ↪  {fit_2[1]:.3f}\\Delta \\tau {fit_2[2]:.3f}$', color='tab:red', fontsize=24)
437
438 plt.tight_layout()
439 # save_fig(plt, 'extrapolation', 2, 6)
440 plt.show()
```