Exam PUE Advanced Computational Physics

Jonas Bojanovsky, 12113264

June 18, 2025

Abstract

This paper presents a minimal example of a computational physics report with equations, code, and results. We simulate a basic physical system and analyze its behavior using numerical methods.

1 Units and periodic boundary conditions

We consider a Lennard Jones system with three particles. We have a quadratic box with side length $L = 12\,\text{Å}$

The three particles are at the positions:

$$\vec{r}_1 = \begin{pmatrix}1\\1\\1\end{pmatrix}\text{Å} \qquad \vec{r}_2 = \begin{pmatrix}1\\1\\11\end{pmatrix}\text{Å} \qquad \vec{r}_3 \begin{pmatrix}11\\11\\11\end{pmatrix}\text{Å} \tag{1}$$

Under the minimum image convention, we can implement a simple code to calculate the distances between the particles, implemented by the code

```
for i in range(3):
        if ri[i] - rj[i] > L / 2:
        rj[i] += L
        elif ri[i] - rj[i] < -L / 2:
        rj[i] -= L
        return rj - ri
```

Listing 1: Implementing Minimum image convention

Then, using $\vec{r}_1$ as the reference vector, we can find the center of mass in the minimum image convention, $\vec{X}$, by adding the MIC distances to $\vec{r}_1$ and averaging. This yields

$$\vec{X} = \frac{1}{3}\begin{pmatrix}1\\1\\-1\end{pmatrix}\text{Å} \tag{2}$$

Note that if we need to wrap it back into the original box, naturally this would become $\begin{pmatrix}1/3 & 1/3 & 35/3\end{pmatrix}^{\mathrm{T}}$. With $\sigma = 3.627\,\text{Å}$, the reduced center of mass is $\vec{X}^* = \vec{X}/\sigma$:

$$\vec{X}^* = 0.0919\begin{pmatrix}1\\1\\-1\end{pmatrix}. \tag{3}$$

And in the scaled coordinates, we instead scale by the length: $\vec{X}' = \vec{X}/L$:

$$\vec{X}' = 0.0277\begin{pmatrix}1\\1\\-1\end{pmatrix}. \tag{4}$$

Calculating the potential energy, we use the contributions only from the three particles under the minimum image convention and do not consider the ones from other cells.

$$U_{12} = 4919.11\varepsilon \tag{5}$$
$$U_{13} = 1.67\varepsilon \tag{6}$$
$$U_{23} = 61.30\varepsilon \tag{7}$$

The total sum of these contributions is

$$U = 4982.09\varepsilon.$$

.

2 Parallel tempering MC simulation

2.1 Detailed balance for parallel tempering step

A typical acceptance rule in the Metropolis step to preserve detailed balance is

$$p_{\text{acc}}(x \to y) = \min\left[1, \frac{f(y)}{f(x)}\right], \tag{8}$$

where y (x) is the new (old) sample from the ensemble and f the corresponding probability density function. For our case,

$$f(x) = \frac{1}{Q_{\text{extended}}} \prod_{k=1}^{M} \exp[-\beta_k H(x_k)] \tag{9}$$

In our case, we just worry about the potential, so $H(x) = U(x)$. In the trial step, we want to try to swap the positions from the trajectories from two different temperatures, T_i and T_j. So the swapped probability is:

$$f(y) = \frac{1}{Q_{\text{extended}}} \exp[-\beta_j U(x_i)] \exp[-\beta_i U(x_j)] \prod_{k \neq i,j} \exp[-\beta_k H(x_k)] \tag{10}$$

The ratio simplifies:

$$\frac{f(y)}{f(x)} = \exp\left[(\beta_i - \beta_j)(U(x_i) - U(x_j)\right) = \exp[\Delta\beta\Delta U]] \tag{11}$$

In the exercise, however, the exponent is $-\Delta\beta\Delta U$. Our results agrees with Wikipedia. We believe, however, that this is just an inconsistency in the definitions of $\Delta\beta$ and ΔU, which we will continue to use exactly from the exercise sheet, but using the step with a positive exponent, as in Equation 11.

2.2 Implementing the code

The implementation of the potential can be seen in Appendix A. The implementation of the MC with parallel tempering is in Appendix B. The distributions of the trajectories with and without parallel tempering are in the figures 1 and 2 respectively.

The trajectory of the $= 0.05$ particle at different zoom levels can be seen in Figures 3 – 5.

The simulation was run for 1×10^6 steps. The acceptance ratio can be seen in Table 1.

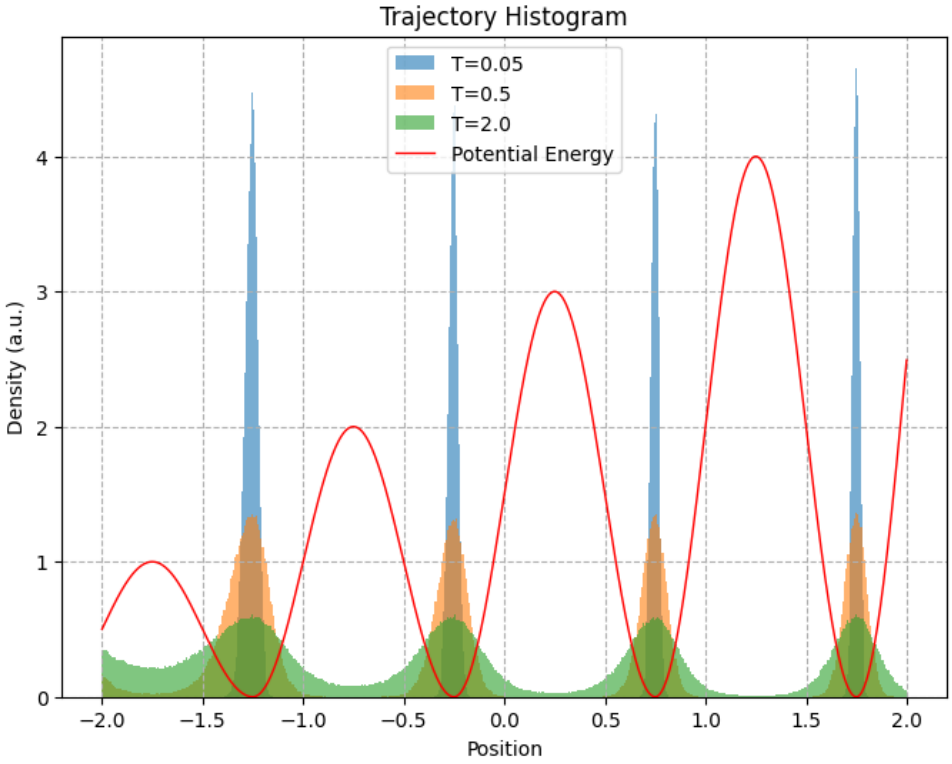

Figure 1: Histogram of the particle position at different temperatures **with parallel tempering**. Potential only for visualization purposes, arbitrary units.

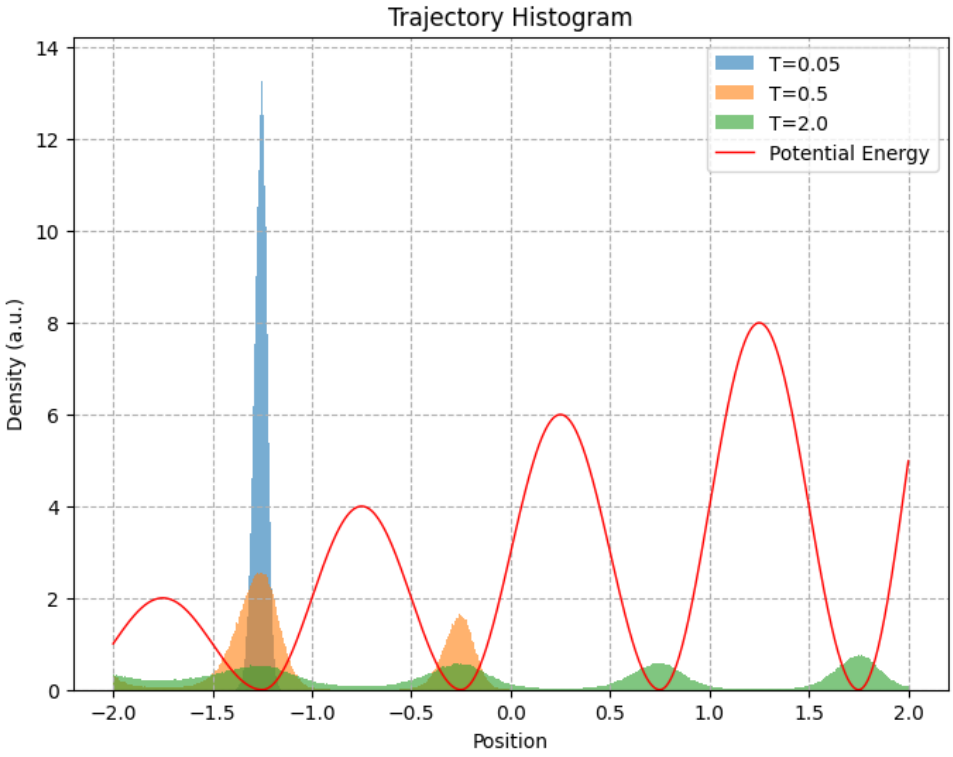

Figure 2: Histogram of the particle position at different temperatures **without parallel tempering**. Potential only for visualization purposes, arbitrary units.

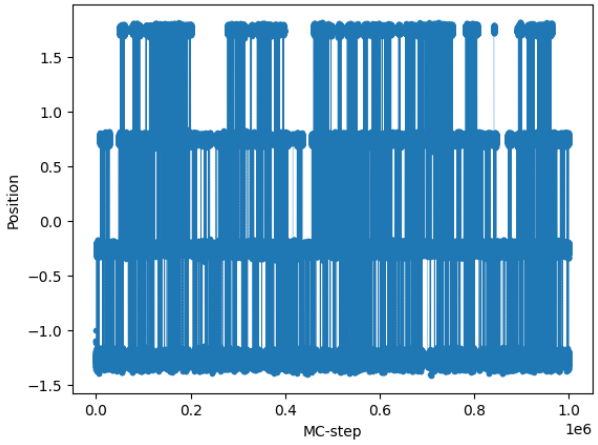

Figure 3: position of $T = 0.05$ particle against time

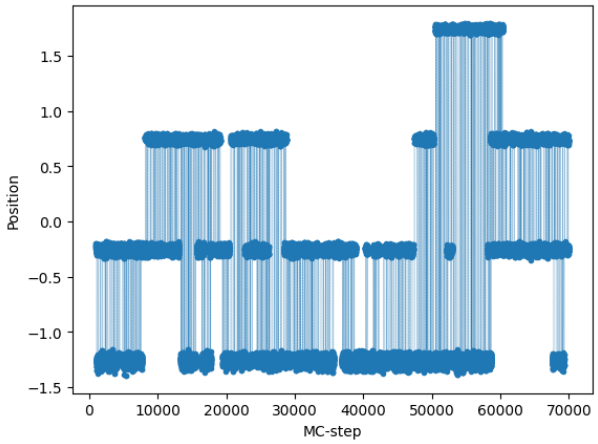

Figure 4: position of $T = 0.05$ particle against time

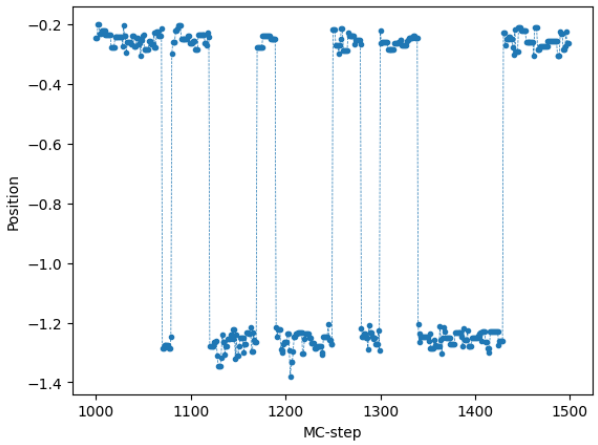

Figure 5: position of $T = 0.05$ particle against time

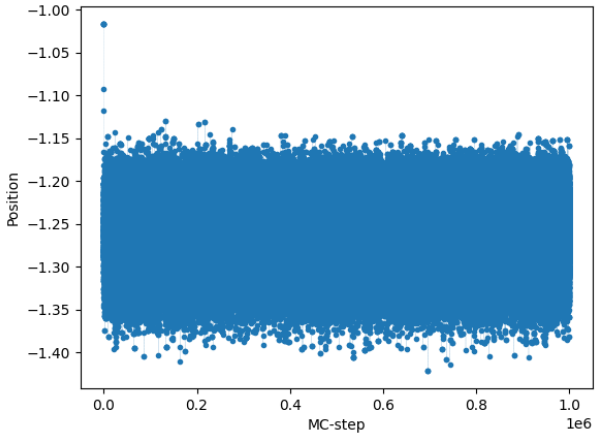

Figure 6: position of $T = 0.05$ particle against time, no annealing

Table 1: Acceptance ratios (bottom two lines are the swapping acceptances for the tempering step)

T	tempering	no temp.
0.05	0.37	0.46
0.5	0.74	0.79
0.2	0.88	0.89
$0.05 \leftrightarrow 0.5$	0.37	–
$0.5 \leftrightarrow 2.0$	0.53	–

2.3 Block averaging

When estimating the standard error in the block averaging method, one tries to correct for the correlation in the samples by dividing the 'trajectory' into blocks of size B and calculating the variance of these averages. Note that there are $M = \lfloor N/B \rfloor$ blocks of this size, where N is the number of simulation steps (in our case 1 Million). Then the estimate for the true variance of the mean is

$$(\Delta\bar{x})^2 = \frac{1}{M(M-1)} \sum_{j=1}^{M} (\bar{X}_j - \langle x \rangle)^2, \tag{12}$$

where $\langle x \rangle$ is the mean of the position (independent of the block size), and $\bar{X}_j$ is the position average of the j^{th} block.

To find the best possible estimate, one should calculate this for different block sizes and see where the computed value converges, while keeping the number of blocks M large enough so that statistics is still meaningful. In this simulation, this was not easy at all because sometimes, the values did not seem to converge at all, see Figure 7.

Nevertheless, the estimated values can be found in Table 2. As a sanity check, we ran the simulation 10 times with tempering and recorded the average position each time. Then, computing the standard deviation of these averages should give a good estimate of the standard error, which is the deviation of the mean. The results here agree well, as we get about $\Delta\bar{x}_{\text{ann.}} = 0.08$ for all temperatures, agreeing well with the estimate from the block-averaging method. The values for the standard error in Table 2 was extracted by simply looking at the graphs and looking at convergence/local maxima and rounding up visually.

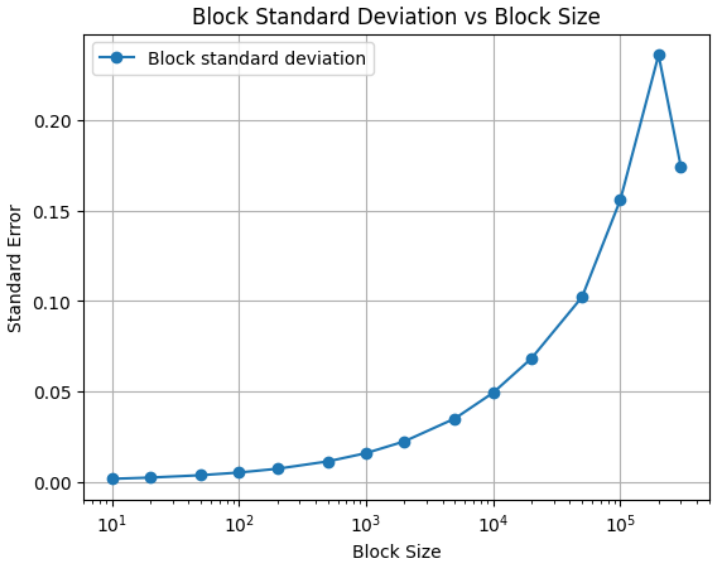

Figure 7: Standard error scaling with block size for N = 1e6 steps, no annealing.

Table 2: Estimates for position averages and Standard errors with annealing (subscript 'ann.') and without.

T	$\langle x \rangle_{\text{temp.}}$	$\Delta\bar{x}_{\text{ann.}}$	$\langle x \rangle$	$\Delta\bar{x}$
0.05	0.07	0.08	-1.25	7×10^{-5}
0.5	0.02	0.07	-0.96	0.25
0.2	-0.09	0.07	-0.16	0.09

2.4 Discussion

As expected from the histograms, the particles spend most of their 'time' in the low-potential regions. Parallel tempering seems to be very effective to let the low-temperature trajectories explore local minima other than the one at -1.25, which is the one the system naturally falls into most likely due to the initial position $x = -1$. Interestingly enough, in the tempering case, the particle spent slightly more time in the right-most minimum than the more left ones (this varied from run to run), probably because once it got there, due to the high maximum at $x = 1.25$, it could not get back so easily.

In the trajectories vs. time figures, one can clearly see how the particle jumps between the minima, showing the tempering process. In Figure 6, where there is no tempering – and especially at low temperature – the particle just stays in the first minimum it falls into until the end of the simulation.

Note that at the beginning of the simulation there was an equilibration time, but after 100 steps the system was already equilibrated, so that this was sometimes just neglected, and sometimes the first 100 steps were thrown away when computing values, especially the averages.

The acceptance ratios of the MC steps get higher as the temperature increases – as expected. Interestingly, for $T = 0.05$, the tempering case is significantly lower than the no-tempering case. This is likely due to the fact that, when swapping with the higher-temperature particle, there are more 'forbidden' regions, and the particle in the no-tempering case justs sits in its local minimum comfortably, which is locally relatively flat.

In the tempering case, the $\langle x \rangle$ move to the left with higher temperature. This is probably due to the minimum at -2, which become more accessible to higher-temperature particles, but the other ones at $U = 0$ potential are due to tempering equally accessible to all particles. On the other hand, the no-tempering averages move to the right with higher temperature, because

only the higher-temperature particles ever overcome the higher and higher barriers to the right.

An outlier in the standard error is the no-tempering, $T = 0.05$ case. Because it very securely sits in the -1.25 minimum, it is extremely well localized there, hence the low $\Delta\bar{x}$.

3 Exponential correlations

From the ACOMP script we know that MSD $\equiv \langle [r(t) - r(0)]^2 \rangle$ is related to the VACF by

$$\text{MSD} = 2 \cdot \int_0^t \mathrm{d}t' \int_0^{t'} \mathrm{d}t'' \left\langle v(0)v(t'') \right\rangle \tag{13}$$

$$= 2 \cdot \int_0^t \mathrm{d}t' \left\langle v^2 \right\rangle \tau \left(1 - e^{-t'/\tau}\right) \tag{14}$$

$$= 2 \left\langle v^2 \right\rangle \tau \left[t - \tau \left(1 - e^{-t/\tau}\right)\right] \tag{15}$$

Similarly, we can get the diffusion coefficient D:

$$D = \frac{1}{3} \int_0^\infty \mathrm{d}t \left\langle v(0)v(t) \right\rangle \tag{16}$$

$$\Rightarrow D = \frac{1}{3} \left\langle v^2 \right\rangle \tau \tag{17}$$

Further, if we expand the MSD, we get

$$2 \left\langle v^2 \right\rangle \left[\tau t - \tau^2 \left(1 - 1 - \frac{t}{\tau} - \frac{t^2}{2\tau^2} - O(t^3)\right)\right] \tag{18}$$

$$= 2 \left\langle v^2 \right\rangle t^2 + O(t^3). \tag{19}$$

So for small t, the system behaves balistically (MSD scales with t^2). On the other hand, for $t \to \infty$ (same as $\tau \to 0$), we get

$$\text{MSD} \to 2 \left\langle v^2 \right\rangle \tau t = 6Dt, \tag{20}$$

which is consistent with theory, and is diffusive motion. So τ is like a relaxation or diffusion time, how long the system 'remembers' it's original velocity, i.e. stays in the ballistic regime before transitioning to the diffusive one.

For water, we assume $D = 31 \times 10^{-5}\,\mathrm{cm}^2/\mathrm{s}$. From the Boltzmann distribution we know that

$$\left\langle v^2 \right\rangle = \frac{3k_\mathrm{B}T}{m}, \tag{21}$$

where $m \approx 18\,\mathrm{amu} = 3 \times 10^{-26}\,\mathrm{kg}$. From that we get $\left\langle v^2 \right\rangle = 4.16 \times 10^9\,\mathrm{cm}^2/\mathrm{s}$. Now, using Equation 17, we get

$$\tau = 22\,\mathrm{fs}.$$

4 Judging results

1. Energy in the NVE ensemble should be (almost) perfectly constant, but here it increases.

2. Sudden step in potential energy hints at a melting process, or at least some breaking of the FCC structure, which is completely ignored. This would be consistent with the parameters, because at $T = 1.5$ and $\rho^* = 0.8$ there should be gas phase for an LJ system, at least definitely not solid.

3. There is no plot of the kinetic energy, why?

4. There is no mention of the time step used, which is critical to examine stability, especially for integrators like Euler-forward.

5. Euler forward is a really bad integrator for energy conserving simulations. A symplectic integrator like velocity-verlet should be implemented instead.

6. 30 particles really is not many for such a system and finite-size effects may play a role, but it is completely ignored.

7. The temperature plot is completely missing from this, although it should not be expected to stay constant.

8. Is there a cutoff radius in the simulation? We use periodic boundary conditions. The cutoff radius may also be difficult.

Based on these issues, especially the ignoring of the melting, missing information about the time step and bad choice of integrator, I would not trust these results or the speaker that they really knew what they were doing.

Appendices

A Defining Potential and quantities

```
def U(x):
        """Potential"""
        U = 1 + np.sin(2 * np.pi * x)

        if x < -2 or x > 2:
        return np.inf
        elif -1.25 <= x <= -0.25:
        U *= 2
        elif -0.25 <= x <= 0.75:
        U *= 3
        elif 0.75 <= x <= 1.75:
        U *= 4
        elif  1.75 <= x <= 2:
        U *= 5
        return U


        Temperatures = np.array([0.05, 0.5, 2.0]) # k_B = 1
        betas = 1/Temperatures

        x_initial = -1.0

        maximum_displacement = 0.1

        parallel_tempering_after = 10
```

Listing 2: Defining the potential and fixed quantities

B Metropolis MC algorithm with parallel tempering

```
def MC_step(x, beta, displacement):
    """Perform a single Monte Carlo step based of precalculated displacement

    Args:
        x (float): Current position
        T (float): Temperature
        displacement (float): Displacement to apply

    Returns:
        tuple: New position and whether the move was accepted
    """
    x_new = x + displacement
    U_x = U(x)
    U_x_new = U(x_new)

    if U_x_new < U_x:
        return x_new, True
    else:
        p_accept = np.exp(-(U_x_new - U_x) * beta)
        if np.random.rand() < p_accept:
            return x_new, True
        else:
            return x, False


```

```python

def Parallel_Tempering_step(x1, x2, beta1, beta2):
    """Perform a parallel tempering step between two temperatures by swapping positions if accepted

    Args:
        x1 (float): Current position at temperature T1
        x2 (float): Current position at temperature T2
        beta1 (float): Inverse temperature 1 (1/T1)
        beta2 (float): Inverse temperature 2 (1/T2)

    Returns:
        whether the swap was accepted
    """
    U_x1 = U(x1)
    U_x2 = U(x2)



    p_swap = np.exp((beta1 - beta2) * (U_x1 - U_x2))
    if np.random.rand() < p_swap:
        return x2, x1, True
    else:
        return x1, x2, False


def precalculate_displacements(maximum_displacement, n_steps):
    """Precalculate displacements for the Monte Carlo steps

    Args:
        maximum_displacement (float): Maximum displacement allowed
        n_steps (int): Number of steps to precalculate

    Returns:
        np.ndarray: Array of displacements
    """
    return np.random.rand(n_steps) * maximum_displacement * 2 - maximum_displacement

def simulation(n_steps, betas = betas, x_initial = x_initial, parallel_tempering_after = parallel_tempering_after, maximum_displacement = maximum_displacement, parallel_tempering = True):
    """Run the Monte Carlo simulation"""

    n_temperatures = len(betas)
    displacements = [precalculate_displacements(maximum_displacement, n_steps) for i in range(n_temperatures)]

    x = np.full(n_temperatures, x_initial)
    energies = np.zeros((n_temperatures, n_steps))
    mc_acceptance_ratio = np.zeros(n_temperatures)
    swap_acceptance_ratio = np.zeros((n_temperatures, n_temperatures))

    single_trajectory = np.empty(n_steps)

    for step in range(n_steps):
        for i in range(n_temperatures):
            x_new, accepted = MC_step(x[i], betas[i], displacements[i][step])
            energies[i, step] = U(x_new)
            if accepted:
                x[i] = x_new
                mc_acceptance_ratio[i] += 1

```

```
        if step % parallel_tempering_after == 0 and step > 0 and
            parallel_tempering:
            # Perform parallel tempering step
            for i in range(n_temperatures - 1):
                x[i], x[i + 1], swap_accepted = Parallel_Tempering_step(x[i], x[
                    i + 1], betas[i], betas[i + 1])
                swap_acceptance_ratio[i, i + 1] += swap_accepted
                swap_acceptance_ratio[i + 1, i] += swap_accepted

        single_trajectory[step] = x[2]  # Store the trajectory of the first
            temperature

    mc_acceptance_ratio /= n_steps
    swap_acceptance_ratio /= (n_steps // parallel_tempering_after)

    return energies, mc_acceptance_ratio, swap_acceptance_ratio,
        single_trajectory


def trajectory_histogram(trajectory, n_bins=100):
    """Plot histogram of the trajectory

    Args:
        trajectory (np.ndarray): Trajectory data
        n_bins (int): Number of bins for the histogram
    """
    plt.figure(figsize=(8, 4))
    plt.hist(trajectory, bins=n_bins, density=True, alpha=0.6, color='g', label=
        'Trajectory Histogram')

    # Plot the potential energy for reference
    x = np.linspace(-2.5, 2.5, 1000)
    U_x = np.vectorize(U)(x)
    plt.plot(x, U_x, 'r-', lw=2, label='Potential Energy')
    plt.title('Trajectory Histogram')
    plt.xlabel('Position')
    plt.ylabel('Density')
    plt.grid()
    plt.legend()
    plt.show()
```