Homework 1

Simulation of complex systems FFR120

CHALMERS
UNIVERSITY OF TECHNOLOGY

Nicole Adamah

November 14, 2022

Contents

1 Exercise 1.1 **1**

2 Exercise 1.3 **6**

3 Exercise 1.6 **10**

4 Exercise 1.8 **13**

1 Exercise 1.1

Harmonic oscillator simulated using the Euler algorithm. *Let us consider a harmonic oscillator: a point-like particle of mass m attached to a perfect spring with an elastic constant k. The particle is placed on a horizontal, frictionless surface. Initially, the particle is at position r0 with velocity v0.*

a. Analytically calculate the trajectory of the particle as a function of time.
One thing to have in mind is that this is not an unique solution since in trigonometry there are multiple axial symmetries and periodicity's.

We can write the force F acting on the particle with Newton's second law:

$$F = ma$$

By including the particle's position and velocity, this can be rewritten as:

$$F = m\frac{\mathrm{d}^2 r(t)}{\mathrm{d}t^2},$$

Now considering the the whole force acting on the system:

$$F = m\frac{\mathrm{d}^2 r(t)}{\mathrm{d}t^2} = -r(t)k,$$

We can set the point of equilibrium $\longrightarrow \mathrm{x}_0 = 0$ for convenience to find a general solution. We see that the desired function remains unchanged up to a constant factor when taking two derivatives so therefore we can use trigonometry for solving the differential. Starting with the cos function and then adding a phase shift and an amplitude to meet all solutions for all possible initial conditions:

$$r(t) = cos(\omega t) \longrightarrow \frac{\mathrm{d}^2 \cos(\omega t)}{\mathrm{d}t^2} = -\omega^2 \cos(\omega t)$$

$$r(t) = A\sin(\omega t + \phi) \longrightarrow \frac{\mathrm{d}^2 A\sin(\omega t + \phi)}{\mathrm{d}t^2} = -\omega^2 A\sin(\omega t + \phi)$$

We can now use the correlation, $\sin(t) = \cos(t - \pi/2)$:

$$\frac{\mathrm{d}^2(A\cos(\omega t + \phi))}{\mathrm{d}t^2} = -\frac{k}{m}r(t)$$
$$-A\omega^2\cos(\omega t + \phi) = -\frac{k}{m}A\cos(\omega t + \phi),$$

we can then obtiain $\omega = \pm\sqrt{k/m}$, but we will only focus on the positive root which meets the requirements.

By using an initial position r_0 and an initial velocity v_0 at $t = 0$ we can obtain A and ϕ:.

$$\begin{cases} r(0) = A\cos(\phi) = r_0 \\ \dot{r}(0) = -A\omega\sin(\phi) = v_0 \end{cases}$$

We can now solve for $\cos(\phi) and \sin(\phi)$ as presented in the exercise description which results in these equations:

$$\begin{cases} \cos(\phi) = \frac{r_0}{A} \\ \sin(\phi) = -\frac{v_0}{A\omega} \end{cases}$$

A can be obtained by regular Pythagoras distance calculation with the position and velocity equations:

$$A^2\left(\cos^2(\phi) + \sin^2(\phi)\right) = r_0^2 + \frac{v_0^2}{\omega^2} \Longleftrightarrow A = \pm\sqrt{r_0^2 + \frac{v_0^2}{\omega^2}}$$

b. Write a program that propagates the trajectory using the Euler algorithm. What is an appropriate value for Δt?

To determine an appropriate value for Δt I compared the trajectory for the Euler algorithm with the characteristic frequency of the motion. The parameters for the characteristic frequency trajectory were: $k = 2, m = 1, r_0 = 0, v_0 = 1 \longrightarrow$ these parameters were used to obtain: $A, \omega, f and \phi$ with the equations from section 1.1a. I tried many different values on Δt which can be seen in figure figure 1, 2 and 3. Figure 1 shows the trajectory with $\Delta t = 0.01$, figure 2 shows the trajectory with $\Delta t = 0.1$ and figure 3 shows the trajectory with $\Delta t = 0.2$. By looking at the three plots we can see that a smaller value on Δt is more suitable. But one thing I noticed was that the trajectory obtained using the Euler algorithm eventually deviates from the analytical solution when increasing the time simulating the motion(Argun et al., 2021). This can be seen in figure 4 where $\Delta t = 0.01$ but t is increased from 1000 to 10000 compared to the plot in figure 1.

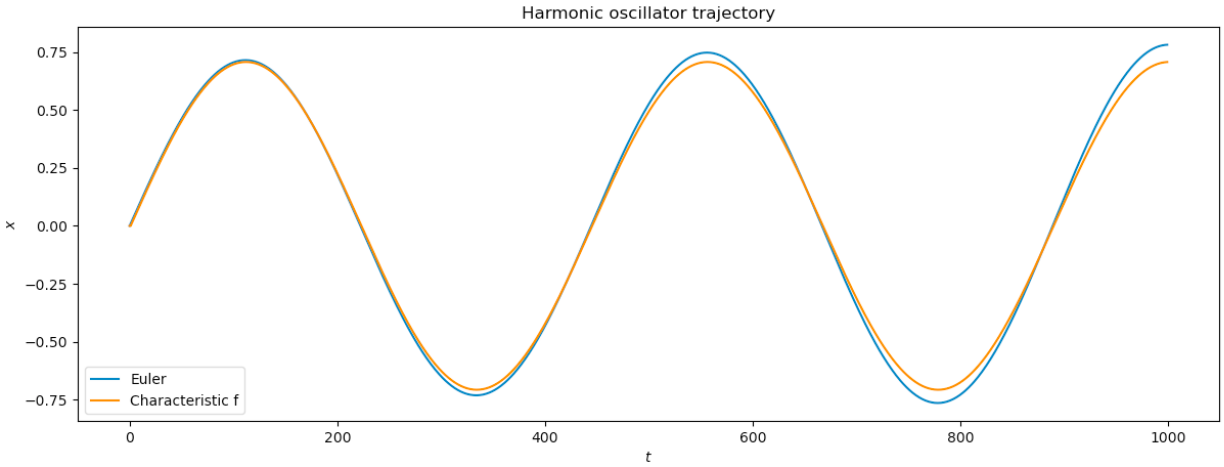

Figure 1: *Trajectory with $\Delta t = 0.01$*

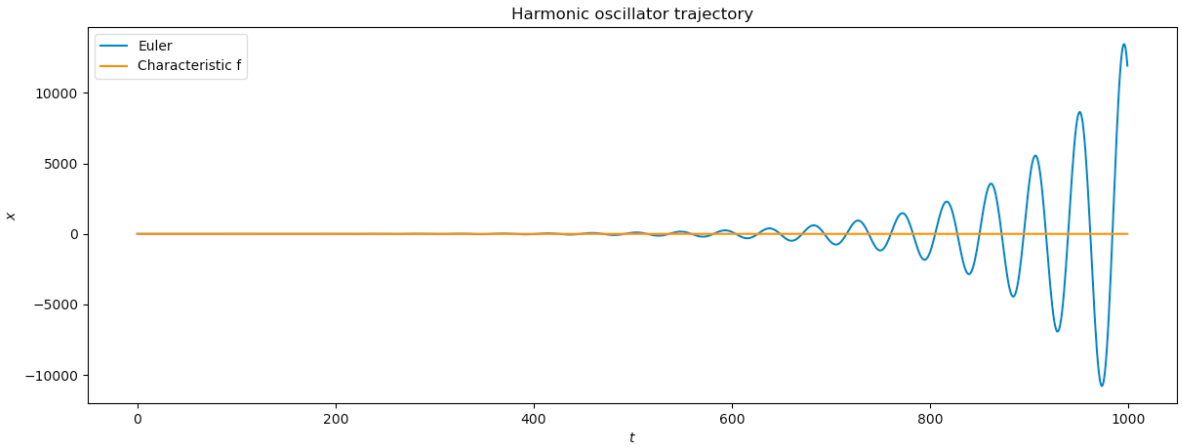

Figure 2: *Trajectory with* $\Delta t = 0.1$

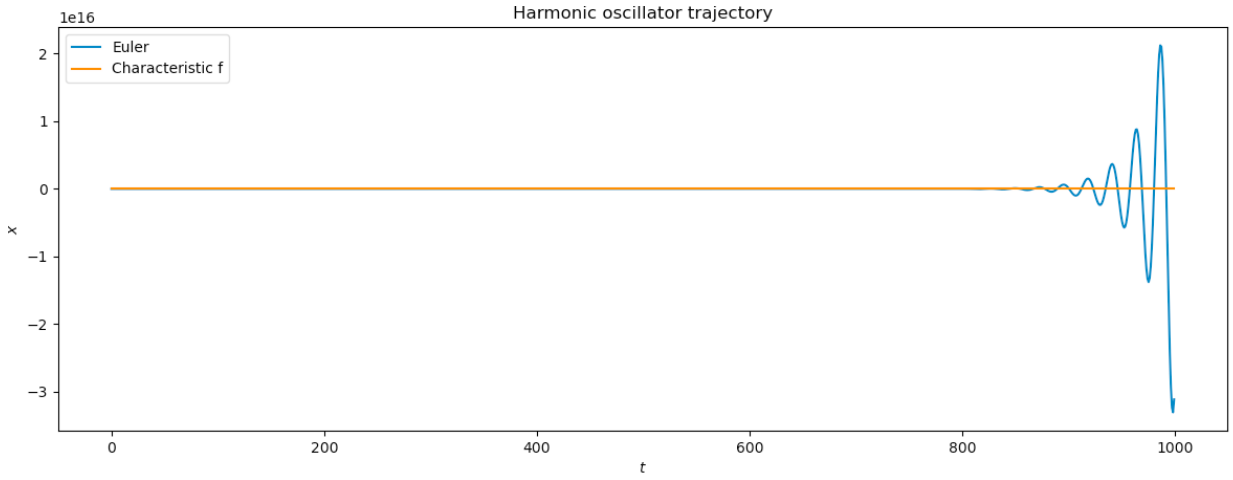

Figure 3: *Trajectory with* $\Delta t = 0.2$

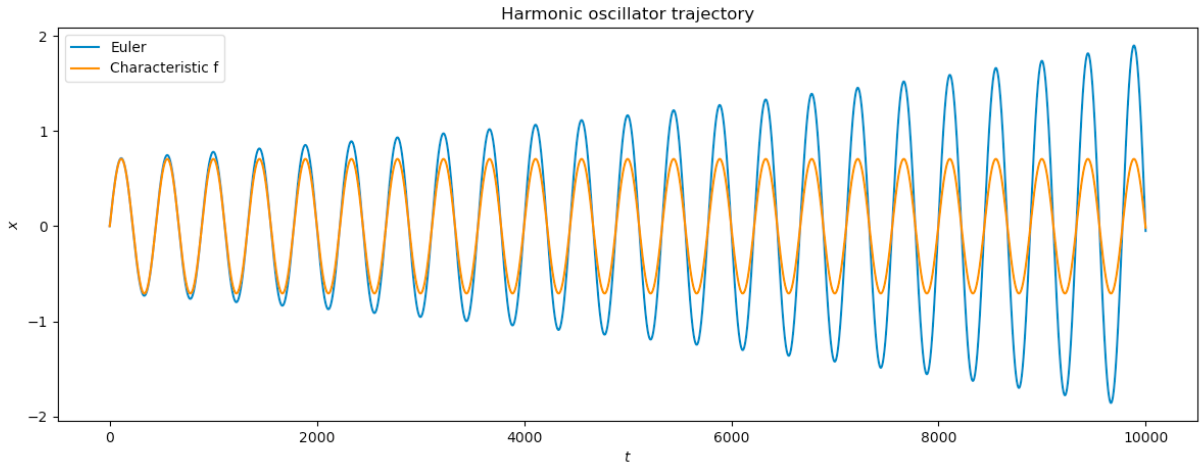

Figure 4: *Trajectory with $\Delta t = 0.01$ but with t=10000*

c. Calculate the total mechanical energy of the particle as a function of time, analytically and by simulation. Compare the results

We can see by comparing figure 5, 6 and 7 that for the three values on Δt the mechanical energy is increasing compared to the analytical solution. So even for small values on Δt the error eventually causes divergence(Argun et al., 2021), as seen in figure 5.

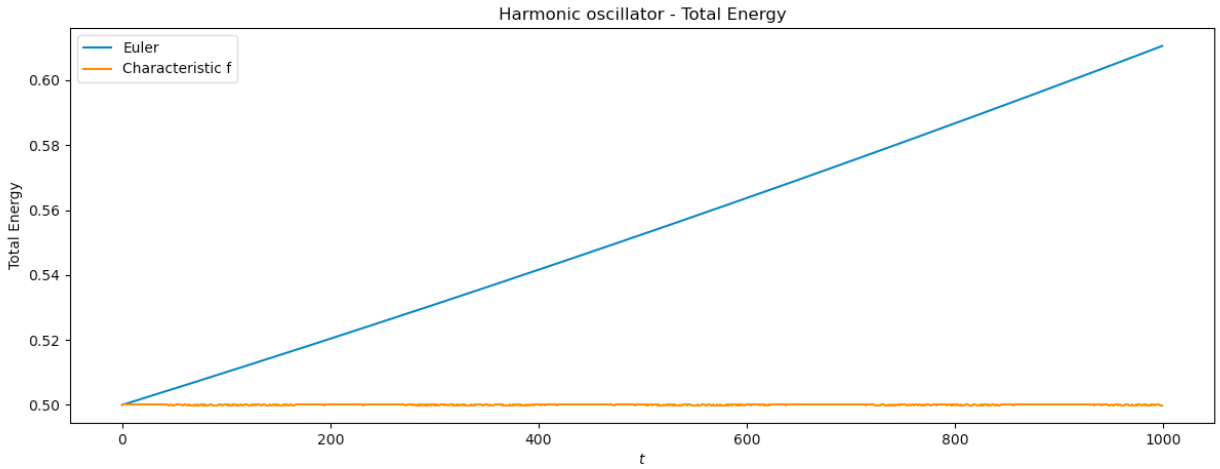

Figure 5: *Total mechanical energy using the Euler algorithm with $\Delta t = 0.01$*

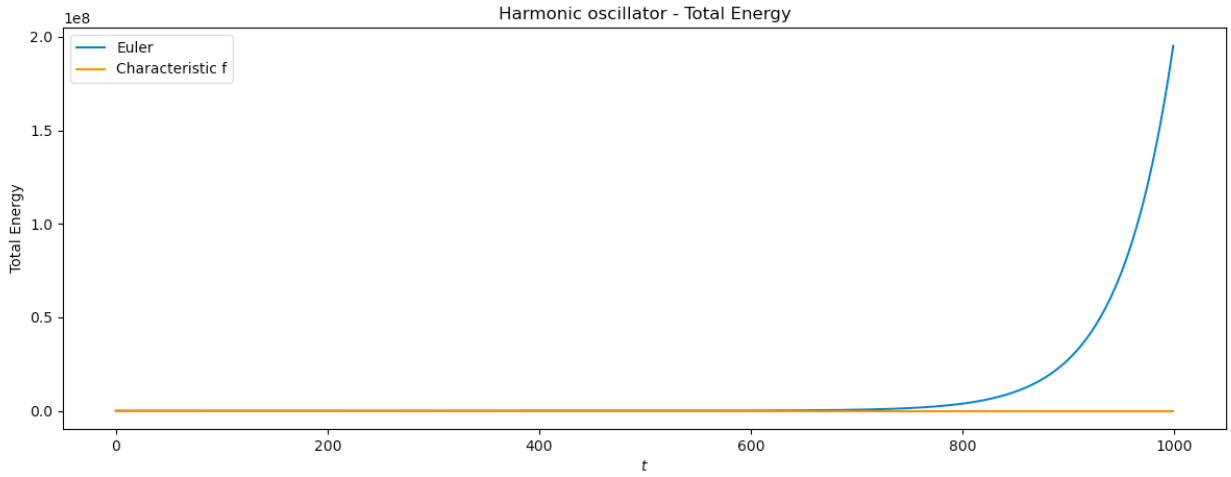

Figure 6: *Total mechanical energy using the Euler algorithm with* $\Delta t = 0.1$

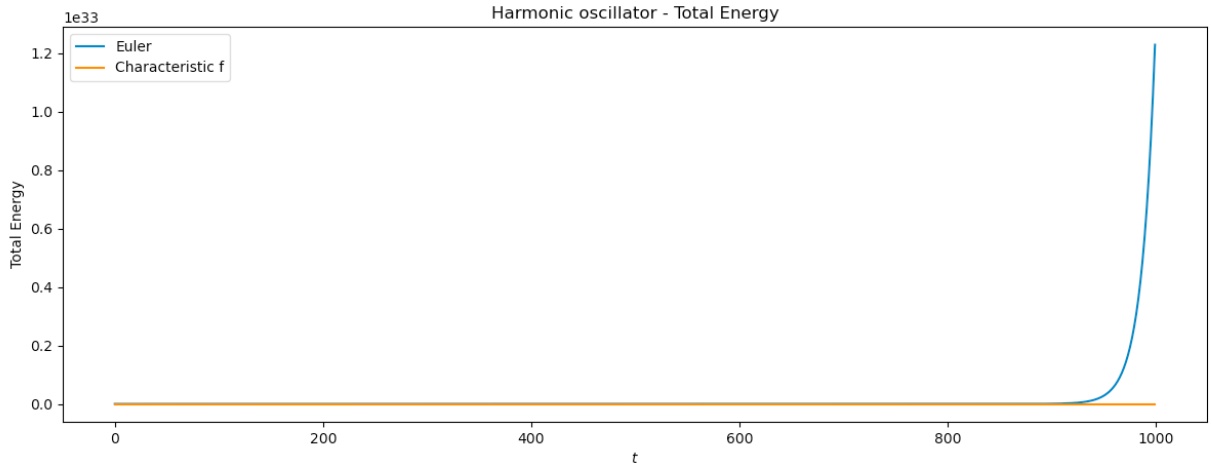

Figure 7: *Total mechanical energy using the Euler algorithm with* $\Delta t = 0.2$

d. Is the total mechanical energy conserved by the analytical solution? And by the simulation?

We can see by looking at the plots from previous section that the mechanical energy is conserved by the analytical solution but not by the simulations with the different values on Δt. So in short, the energy is constant and conserved in the simple harmonic motion obtained by the analytical solution.

2 Exercise 1.3

Harmonic oscillator simulated using the leapfrog algorithm. *Repeat exercise 1.1 using the leapfrog algorithm instead of the Euler algorithm.*
a. *Write a program that simulates the trajectory using the leapfrog algorithm.*
b. *Compare the simulated trajectory with the analytical trajectory and with the trajectory simulated using the Euler method. What do you observe?*
By comparing figure 8, 9, 10 and 11 one can clearly see that the trajectory using the Leapfrog algorithm is more agrees more with the analytical solution than Euler's. This conclusion can be drawn for all three Δt and even figure 11 shows the accuracy and that the Leapfrog solution doesn't diverge.

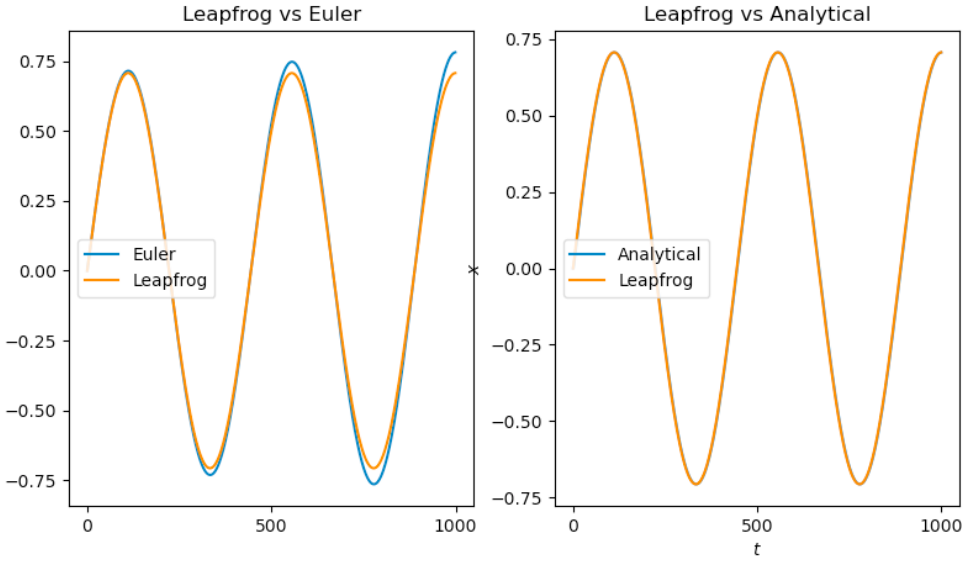

Figure 8: *Trajectory with* $\Delta t = 0.01$

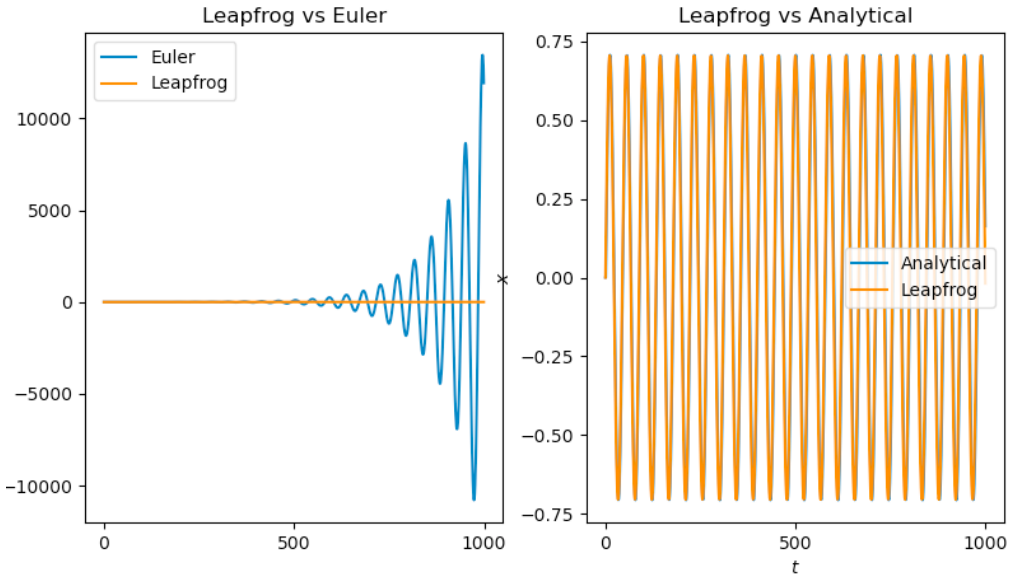

Figure 9: *Trajectory with* $\Delta t = 0.1$

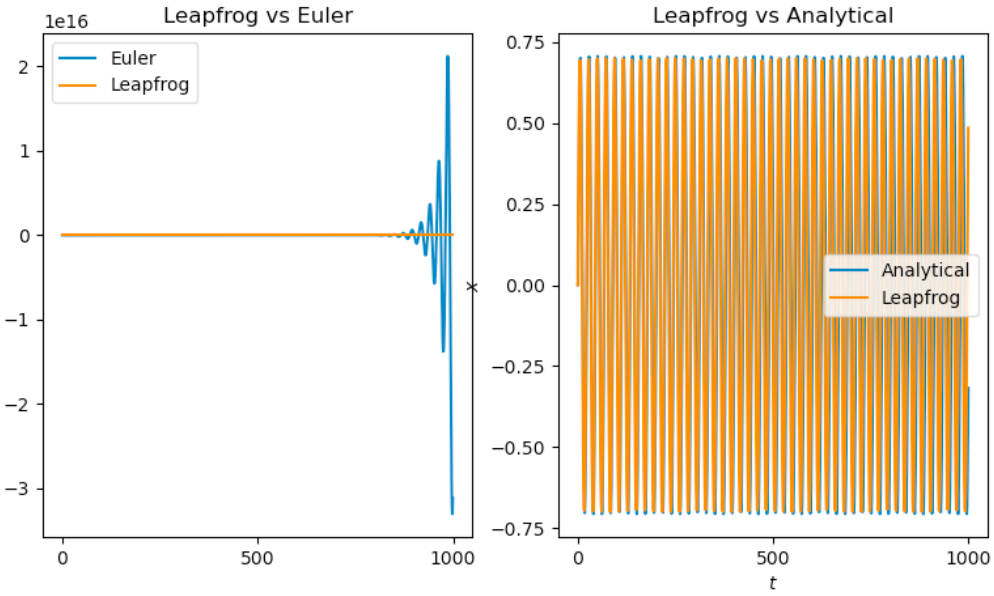

Figure 10: *Trajectory with* $\Delta t = 0.2$

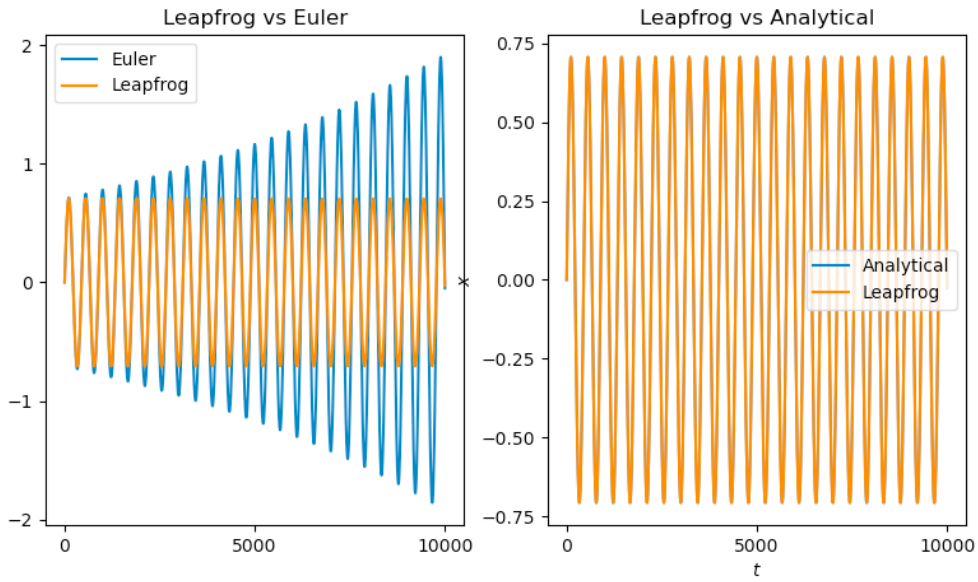

Figure 11: *Trajectory with $\Delta t = 0.01$ but with t=10000*

c. *Compare the total energies of the analytical and numerical solutions. Which method appears to perform better?*
As similarly stated in the previous section we can clearly conclude from figures 12,13 and 14 that the Leapfrog algorithm appears to perform better. For the Leapfrog solution and the analytical solution the mechanical energy is conserved. Thus the error is significantly smaller for the Leapfrog algorithm compared to Euler's.

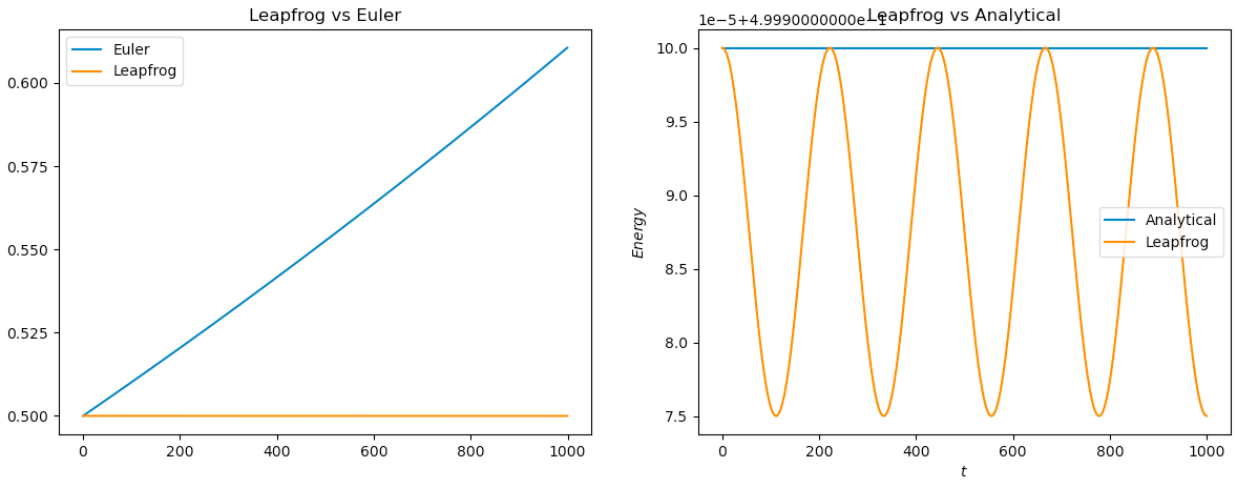

Figure 12: *Total mechanical energy the with $\Delta t = 0.01$*

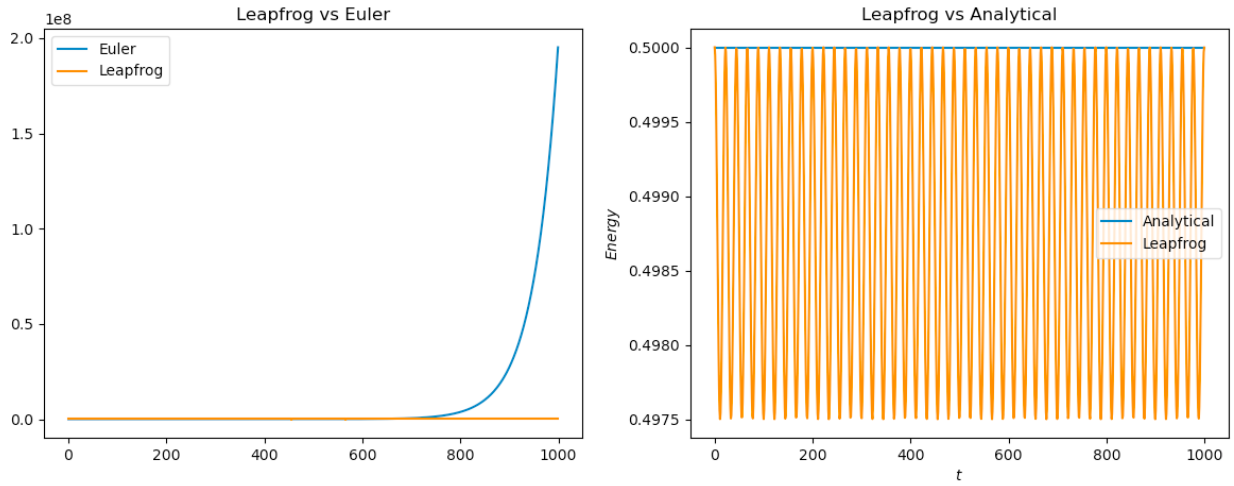

Figure 13: *Total mechanical energy with* $\Delta t = 0.1$

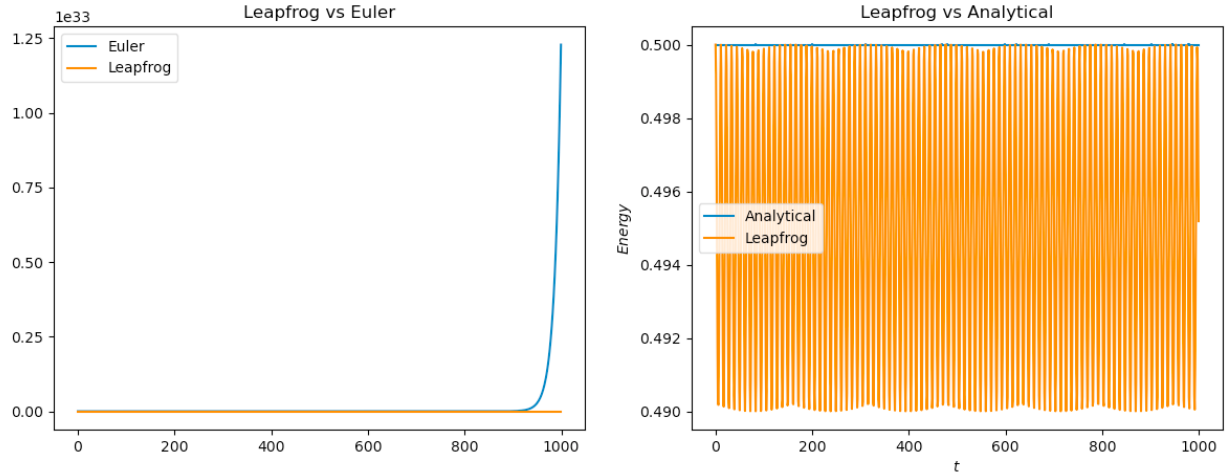

Figure 14: *Total mechanical energy with* $\Delta t = 0.2$

3 Exercise 1.6

Two-dimensional gas in a box. ***a.*** *Simulate and visualize the evolution of this gas of particles (start with $N = 100$) implementing equation (1.10) using the leapfrog algorithm. Randomize the initial positions of the particles, and orient the velocities randomly*

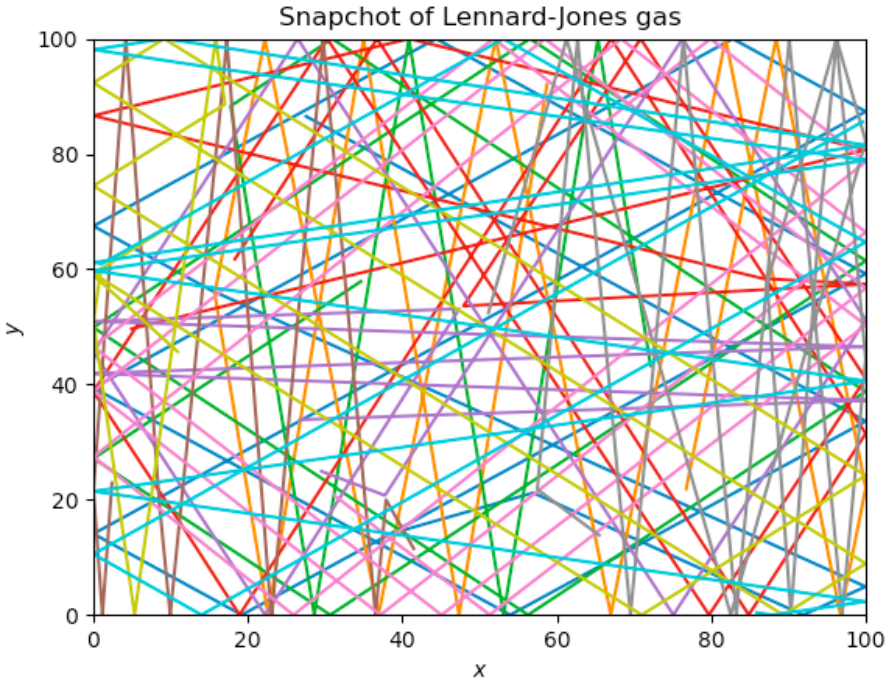

Figure 15: *Visualization of the evolution of Lennard-Jones gas*

b. Determine an appropriate time step, Δt
I tried the simulation with different values on Δt and noticed that a smaller time step was more appropriate regarding for example conserving the mechanical energy. An appropriate value on Δt can be based on the equation presented in the book $\sigma / (2v_0) \cdot 0.02$ which is the upper limit for the time step Argun et al., 2021.

c. Check your code by plotting the instantaneous positions and velocities at each iteration for a small number of time steps. Compare your results with figure

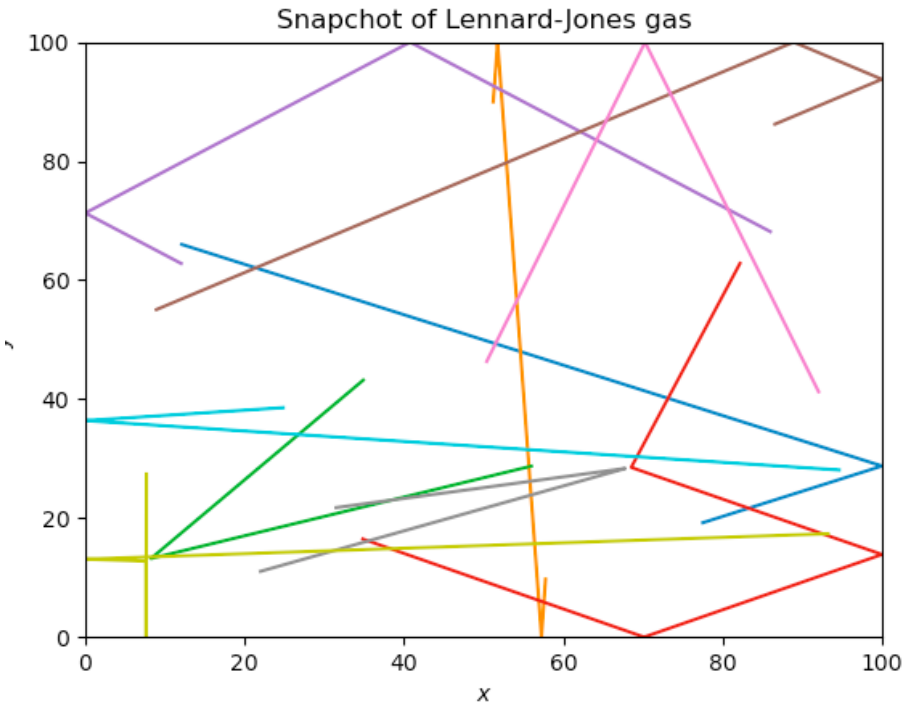

Figure 16: *Visualization of the evolution of Lennard-Jones gas for a small number of time steps*

d. Plot the kinetic, potential, and total energies of the system as a function of time. Compare your results with figure.

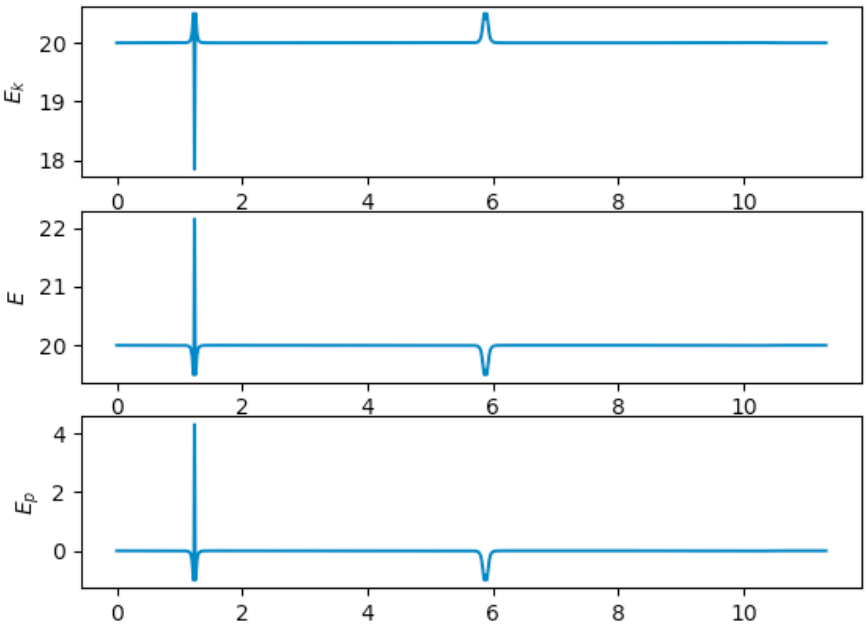

Figure 17: *Potential, kinetic and total energy Lennard-Jones gas*

4 Exercise 1.8

Brownian motion.

References

Argun, A., Callegari, A., & Volpe, G. (2021). *Simulation of complex systems* (1st ed.). IOP Publishing.

Code for Exercise 1.1 & 1.3

```
import numpy as np
from matplotlib import pyplot as plt
import argparse
def characteristic_freq(N, k, m,dt):
    w = np.sqrt(k / m)
    A = (1/w)
    phi = np.arcsin(-1/(w*A))
    x2 = np.zeros(N)
    v2 = np.ones(N)
    for t in range(N-1):
        x2[t+1] = A * np.cos(w * dt*t + phi)
        v2[t+1] = -w * A * np.sin(w * dt*t + phi)
    return x2, v2

def euler_algorithm(N,dt,k,m):
    omega = np.sqrt(k / m)
    A = (1/omega)
    x = np.zeros(N)
    v = np.ones(N)

    for t in range(N - 1):
        x[t + 1] = x[t] + v[t] * dt
        v[t + 1] = v[t] - k * x[t] / m * dt
    return x, v

def leapfrog_alogithm(N,dt,k,m):
    x1  = np.zeros(N)
    xh = np.zeros(N)
    v1  = np.ones(N)

    for t in range(N - 1):
        xh[t] = x1[t] + 0.5 * v1[t] * dt
        v1[t + 1] = v1[t] - k * xh[t] / m * dt
        x1[t + 1] = xh[t] + 0.5 * v1[t + 1] * dt
    return x1,v1

def energy_euler(N,m,k,x,v):

    Ek = np.zeros(N)
    Ep = np.zeros(N)
    for t in range(N):
        Ek[t] = 0.5 * m * v[t] * v[t]
        Ep[t] = 0.5 * k * x[t] * x[t]
    return Ek+Ep

def energy_leapfrog(N,m,k,x1,v1):
    Ek = np.zeros(N)
```

```
    Ep = np.zeros(N)
    for t in range(N):
        Ek[t] = 0.5 * m * v1[t] * v1[t]
        Ep[t] = 0.5 * k * x1[t] * x1[t]
    return Ek+Ep


def plot_trajectories(x,x2,legend1,legend2,ylabel,title):
    plt.figure(figsize=(15,5))
    plt.plot(x)
    plt.plot(x2)
    plt.legend([legend1,legend2])
    plt.ylabel(ylabel)
    plt.xlabel('$t$')
    plt.title(title)
    plt.rcParams.update({'font.size': 18})
    plt.show()

def compare_plot(x,x2,x3, legend1,legend2,legend3, ylabel):
    plt.figure(figsize=(15,5))
    plt.subplot(1, 2, 1)
    plt.plot(x)
    plt.plot(x3)
    plt.legend([legend1,legend3])
    plt.title('Leapfrog vs Euler')
    plt.subplot(1, 2,2)
    plt.plot(x2)
    plt.plot(x3)
    plt.legend([legend2, legend3])
    plt.ylabel(ylabel)
    plt.xlabel('$t$')
    plt.title('Leapfrog vs Analytical')
    plt.rcParams.update({'font.size': 20})
    plt.show()


def main(args):
    #mute the plots that you dont want to run to make the code more
efficient
    #exersice 1.1a
    N = 1000
    dt = 0.01 #change value to compare
    k = 2
    m = 1
    x1,v1 = euler_algorithm(N,dt,k,m)
    x2,v2 = characteristic_freq(N, k, m,dt)
    plot_trajectories(x1,x2,'Euler', 'Characteristic f',
'$x$','Harmonic oscillator trajectory')
```

```
    #1.1c
    E_tot = energy_euler(N, m, k, x1, v1)
    E_tot2 = energy_euler(N, m, k, x2, v2)
    plot_trajectories(E_tot, E_tot2, 'Euler', 'Characteristic
f','Total Energy', 'Harmonic oscillator - Total Energy')

    #exercise 1.3
    x3,v3 = leapfrog_alogithm(N, dt, k, m)
    compare_plot(x1, x2, x3, 'Euler','Analytical','Leapfrog', '$x$')
    E_tot3 = energy_leapfrog(N, m, k, x3, v3)
    compare_plot(E_tot, E_tot2, E_tot3,
'Euler','Analytical','Leapfrog', '$Energy$')

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simulate molecular
dynamics")
    parser.add_argument("--outdir", "-o", type=str, default=".",
help="Out directory")
    args = main(parser.parse_args())
```

Code for Exercise 1.6

```
import numpy as np
from matplotlib import pyplot as plt
from tqdm import trange
from typing import Callable, Tuple
import argparse

SNAPSHOT = 0
ENERGIES = 1

def leapfrog(x0: np.ndarray, v0: np.ndarray, xh_f:
Callable[[np.ndarray], np.ndarray], dt = 1.0) -> Tuple[np.ndarray,
np.ndarray]:
    x0 = x0 + v0 * dt/2
    xh = xh_f(x0)
    v = v0 + xh * dt
    x = x0 + v * dt/2
    return (x, v)

def initialize_pos(size: Tuple[int, int], L, sigma):
    pos = np.zeros(size)

    for i in range(size[0]):
        point_placed = False

        while not point_placed:
            new_pos = np.random.rand(1, 2) * L
            distances = [ np.sqrt(np.sum((new_pos - pos[j,:])**2))
for j in range(i) ]
            instabilities = [ r > sigma for r in distances ]
            if all(instabilities):
                pos[i,:] = new_pos
                point_placed = True
    return pos

def initialize_v(size: Tuple[int, int], v0):
    angles = (np.random.rand(size[0]) * 2 *
np.pi).reshape(size[0],1)
    directions = np.column_stack((np.cos(angles), np.sin(angles)))
    return directions * v0

def lennard_jones_f(positions, e, s):
    f = np.zeros_like(positions)
    for i in range(positions.shape[0]):
        for j in range(i+1, positions.shape[0]):
            r = np.sqrt(np.sum((positions[i,:] -
positions[j,:])**2))
```

```
            magnitude = 4 * e * (12 * np.power(s, 12) * np.power(r,
-13) - 6 * np.power(s, 6) * np.power(r, -7))
            direction = (positions[j,:] - positions[i,:]) / r
            f[i,:] -= magnitude*direction
            f[j,:] += magnitude*direction
    return f

def lennard_jones_p(positions:np.ndarray, e, s):
    p = np.zeros(positions.shape[0])
    for i in range(positions.shape[0]):
        for j in range(i+1, positions.shape[0]):
            r = (np.sum((positions[i,:] - positions[j,:])**2))
            magnitude = 4 * e * (np.power(s ** 2 / r, 6) -
np.power(s ** 2 / r, 3))
            p[i] += magnitude
            p[j] += magnitude
    return p

def potential_energy(x, e, s):
    return 0.5 * np.sum(lennard_jones_p(x, e, s))

def kinetic_energy(v, m, v_scale):
    return 0.5 * m * np.sum((v/v_scale)**2)

def plot_snap(N, position_history):
    for i in range(N):
        plt.plot(position_history[:, i, 0].squeeze(),
position_history[:, i, 1].squeeze())
    plt.gca().set_xlim([0, L])
    plt.gca().set_ylim([0, L])
    plt.ylabel('$y$')
    plt.xlabel('$x$')
    plt.title('Snapchot of Lennard-Jones gas')
    plt.show()

def plot_energies(e,steps,freq,t_scale,Ek,Ep,dt):
    plt.subplot(3, 1, 1)
    plt.plot(np.arange(steps // freq) * dt / t_scale, Ek / e, '',
label="Kinetic Energy",markersize=1)
    plt.ylabel("$E_k$")
    plt.subplot(3, 1, 3)
    plt.plot(np.arange(steps // freq) * dt / t_scale, Ep / e, '',
label="Total Energy", markersize=1)
    plt.ylabel("$E_p$")
    plt.subplot(3, 1, 2)
    plt.plot(np.arange(steps // freq) * dt / t_scale, (Ek + Ep /
e), '',label="Potential Energy", markersize=1)
    plt.ylabel("$E$")
```

```
    plt.show()

def main(args):
    e = 1.0 #epsilon
    s = 1.0 #sigma
    m = 1 #mass
    v_scale = np.sqrt(2 * e / m)
    t_scale = s * np.sqrt(m / (2 * e))
    L = s * 100
    N = 10
    steps = 40000
    freq = 5
    dt = 0.001
    #dt = s / (2 * v_scale) * 0.02
    x = initialize_pos((N, 2), L, s) #positions
    v = initialize_v((N, 2), 2 * v_scale) #velocities
    x_history = np.empty((steps // freq, N, 2))
    Ek = np.empty(steps // freq)
    Ep = np.empty(steps // freq)

    plotting = ENERGIES
    for t in trange(steps):
        if plotting == SNAPSHOT:
            x_history[t // freq, :, :] = x
        if t % freq == 0 and plotting == ENERGIES:

            Ek[t // freq] = kinetic_energy(v, m, v_scale)
            Ep[t // freq] = potential_energy(x, e, s)

        (x, v) = leapfrog(x, v, lambda x: lennard_jones_f(x, e, s),
dt=dt)

        for i in range(N):
            # Outside left bound
            if x[i, 0] < 0:
                x[i, 0] = x[i, 0] * -1
                v[i, 0] = v[i, 0] * -1

            # Outside right bound
            elif x[i, 0] > L:
                x[i, 0] = 2 * L - x[i, 0]
                v[i, 0] = v[i, 0] * -1

            # Outside lower bound
            elif x[i, 1] < 0:
                x[i, 1] = x[i, 1] * -1
                v[i, 1] = v[i, 1] * -1
```

```
            # Outside upper bound
            elif x[i, 1] > L:
                x[i, 1] = 2 * L - x[i, 1]
                v[i, 1] = v[i, 1] * -1

    if plotting == SNAPSHOT:
        plot_snap(N, x_history)
    elif plotting == ENERGIES:
        plot_energies(e, steps, freq, t_scale, Ek, Ep, dt)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simulate molecular
dynamics")
    parser.add_argument("--outdir", "-o", type=str, default=".",
help="Out directory")
    args = main(parser.parse_args())
```