Homework 1

FFR120 - Simulation of Complex Systems

Vincent Udén
udenv@student.chalmers.se

November - 2022

Excercise 1.1

a)

Starting off with Newton's second law

$$F = ma \iff m\frac{\mathrm{d}^2 r(t)}{\mathrm{d}t^2} = -r(t)k, \tag{1}$$

one obtains a second order differential equation where the right hand side the force generated by a harmonic oscillator. This of course assumes that the point of equilibrium sits at $x_0 = 0$. This differential equation is solved by the trigonometric functions sin and cos since

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

and the same goes for sin. If we add a phase shift ϕ and an amplitude A to cosine we can cover the entire solution space with just

$$r(t) = A\cos(\omega t + \phi), \tag{3}$$

since $\cos(t - \pi/2) = \sin(t)$. Finally we'll insert equation (3) into equation (1)

$$\frac{\mathrm{d}^2(A\cos(\omega t + \phi))}{\mathrm{d}t^2} = -\frac{k}{m}r(t) \iff -A\omega^2\cos(\omega t + \phi) = -\frac{k}{m}A\cos(\omega t + \phi), \tag{4}$$

which results in two solutions where $\omega = \pm\sqrt{k/m}$. Since it doesn't matter I'll disregard the negative solution. A and ϕ can be defined using an initial position r_0 and an initial velocity v_0 at $t = 0$. This is done through the system of equations

$$\begin{cases} r(0) = A\cos(\phi) = r_0 \\ \dot{r}(0) = -A\omega\sin(\phi) = v_0. \end{cases} \tag{5}$$

Reorganising these two equations, separating $\cos(\phi), \sin(\phi)$ and A as described in the excercise results in

$$\begin{cases} \cos(\phi) = \frac{r_0}{A} \\ \sin(\phi) = -\frac{v_0}{A\omega} \\ A^2(\cos^2(\phi) + \sin^2(\phi)) = r_0^2 + \frac{v_0^2}{\omega^2} \iff A = \pm\sqrt{r_0^2 + \frac{v_0^2}{\omega^2}}, \end{cases} \tag{6}$$

where the last equation comes from squaring the position and velocity equation before adding them. Note that this is not one unique solution since there are multiple axial symmetries and periodicity in the trigonometric functions.

b), c) & d)

The parameters used were $m = 0.1, k = 5, r_0 = 0.1, v_0 = 0$ resulting in $A = 0.1, \phi = 0, \omega = \sqrt{50}$. The characteristic frequency of this oscillator is $f = \omega/(2\pi) \approx 11.11$ Hz. If we convert the timesteps seen in figure **??** to frequencies ($f = 1/\Delta t$) we obtain $f_1 = 10000, f_2 = 500, f_3 = 50$. Relative to the characteristic frequency these are about 1000, 50 and 5 times bigger. All of them diverge from the analytical solutions, just a different speeds. Therefore my conclusion is that no value of Δt is really appropriate if the oscillator is to be ran forever.

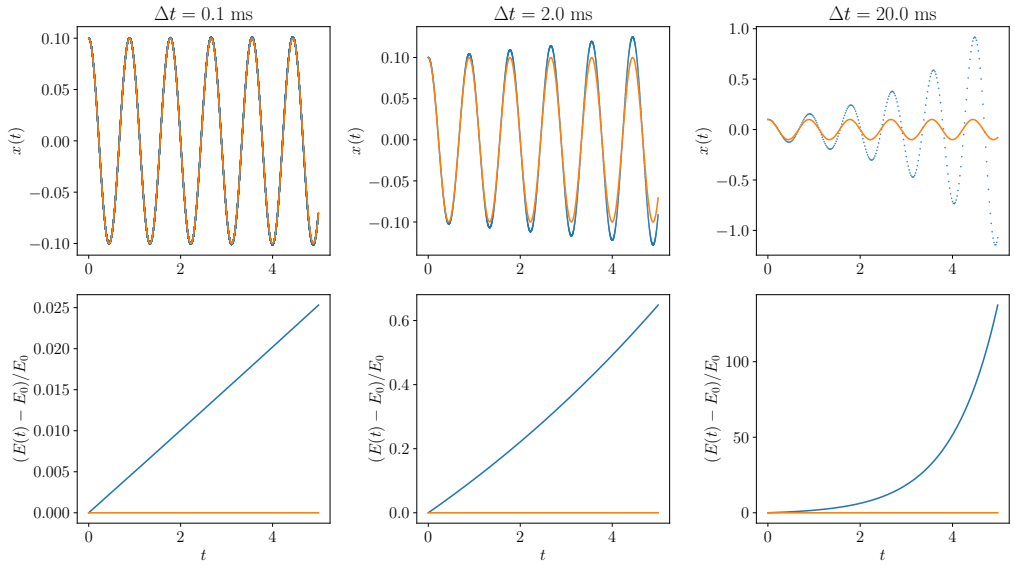

Figure 1: On the first row of plots the analytical path (orange line) and the numerical path (blue dots) for each column using a different timestep Δt. The second row does the same but for the relative deviation from the initial energy.

The energies in figure 1 show the same results as discussed in the previous paragraph. All timesteps results in increasing energy. This is to be expected as the Euler forward integration always lags behind an analytical solution. In this oscillatory case that results in the system turning back towards the

equilibrium too late. If Euler backward integration were to be used instead the opposite problem would occur. The analytical solution doesn't stray at all from the initial energy of the system. If we were too zoom in incredibly far we'd of course spot a tiny floating point error but that isn't an error inherent to the method, it's just a limitation of finite memory.

Excercise 1.3

a)

The program is shown at the end of the report. It's the same program as for excercise 1.1, expect using the other integrator (leapfrog).

b & c)

The Leapfrog integration does a much better job at adhereing to the analytical trajectory. As can be seen in figure 2 the energies deviate as much as 0.5%. The important fact is that they're not drifting further and further away from the analytical solution. Even if restricted the the first period of the oscillator, the error for the same timestep is thousands of times smaller than that of the Euler method.

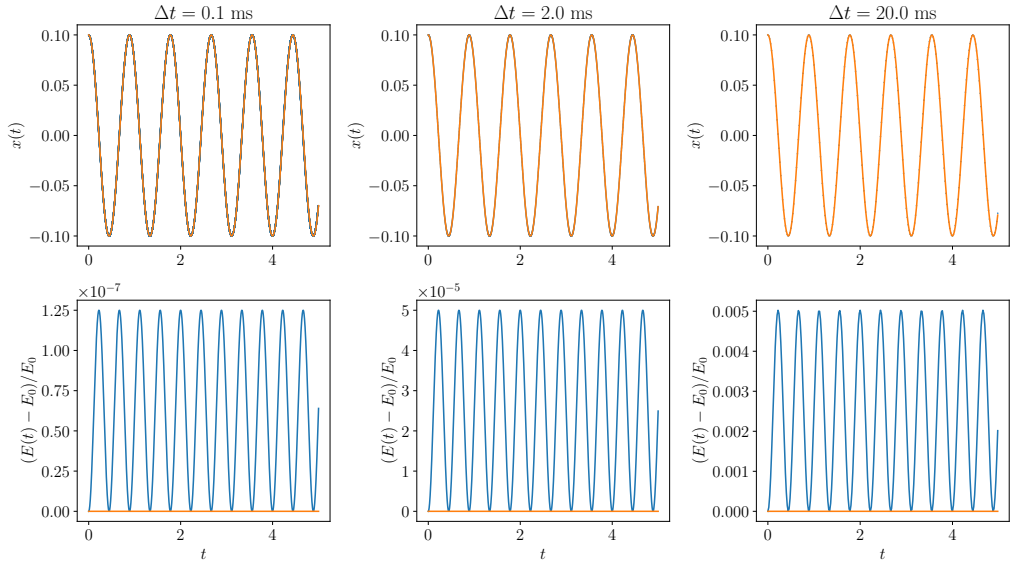

Figure 2: Plots arranged according as in figure 1 except the blue dots being the leapfrog results.

Excercise 1.6

a)

The code for this excercise is shown at the end of the report. It uses the same integrator code as previously, with the force function switched out. Now it uses force derived from a Lennard-Jones potential. Of course this severely impacts the programs performance as the time complexity of simulation n bodies interacting with every other body has the asymptotic time complexity $O(n^2)$. Luckily the interactions obey Newton's 3st law. The force excerted on particle i onto particle j is indentical in size but opposite in direction to the force from j onto i. This means that we can eliminate some duplicate calculations through a triangular iteration, resulting in a complexity of $O(n(n+1)/2)$. Some additional speed up can be achieved when calculating the Lennard-Jones potential. There's no need to take the square root of $\sqrt{(\vec{r}_i - \vec{r}_j)^2} = |\vec{r}_i - \vec{r}_j|$ since it'll be raised to the power of 12 or 6 later. Skipping the root calculation and raising it to the power of 6 and 3 instead saves an expensive calculation which is made for every pair of particles.

When plotting, only a tenth or so of the time steps are saved to be drawn. The need for a small Δt is only vital to ensure conservation of energy, not to view the process with an adequate resolution.

b)

Trying different values of Δt seems to indicate that about $\sigma/(2v_0) \cdot 0.005$ is about the biggest it can be while keeping the baseline level of total energy within 1% of it's initial value.

c)

After running the integration for 100 timesteps the plot shown in figure 3 was obtained. For reference the entire simulation going to $t \approx 250t_0$ runs for 100 000 timesteps.

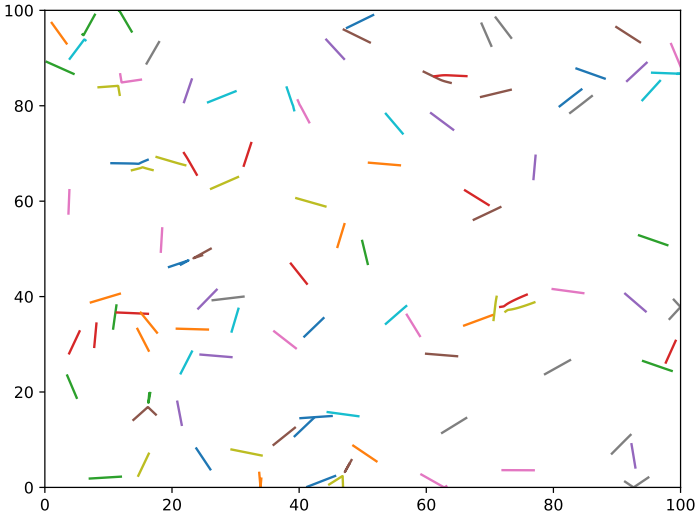

Figure 3: Instantaneous particle trajectories. Note most particles seeming unaffected by each outher, traveling in a straight line. The Lennard-Jones potential leads to interactions only at very small distances.

d)

After running a longer simulation shown in figure 4 the trade off between potenial and kinetic energy becomes quite clear. Each time two particles approach each other for a "collision" the kinetic energy drops as they slow down, repelling each other. At the same time the potential energy rises as they get deeper into each other's potential fields. Afterwards the energy exchange reverses restoring the system to being mostly composed of kinetic energy. Compared to figure 1.7 from the book I seem to have a box which is much bigger compared to the particles since the collisions are more frequent in the book. A final comment on the total energy: The total energy increases slightly over time, in discrete steps most likely corresponding to high energy collisions. This increase in energy is tiny in comparison to even the deviations in energy, let alone the absolute value of the energy, less than 1%.

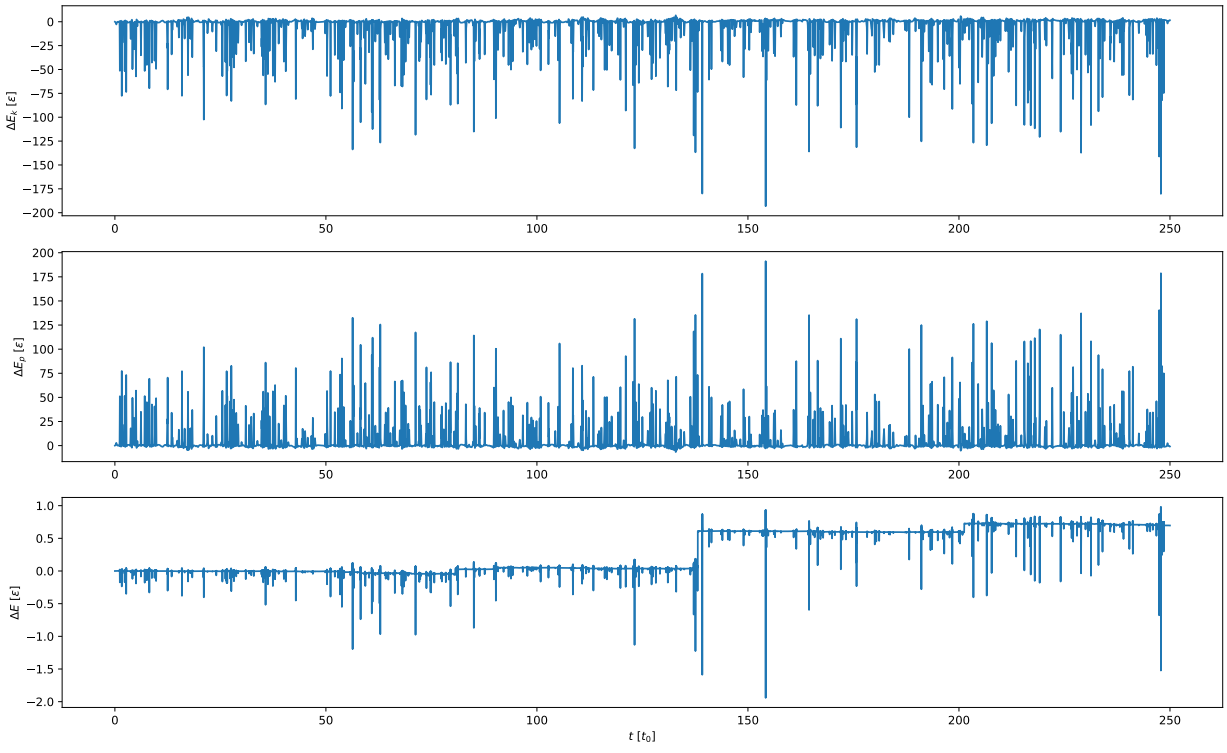

Figure 4: A long runnning simulation up to $t \approx 250t_0$. The two upper subplots show the variation in kinetic and potential energy, measured as the difference between it's current value and the intial value $\Delta E_{p/k}(t) = E_{p/k}(0) - E_{p/k}(t)$

Harmonic-Oscillator/main.py (used for 1.1 and 1.3)

```
#!/usr/bin/python

# System libraries
import numpy as np

from matplotlib import pyplot as plt
from matplotlib.ticker import StrMethodFormatter

# Other project files
from integrate import euler_step, leapfrog

EULER = 0
LEAPFROG = 1

# Assumes equilibrium at x = 0
def spring_force(pos: np.ndarray, k: float):
    return (-k) * pos

def analytical_positions(t: np.ndarray, A: float, omega: float, phi=0.0):
    return A * np.cos(omega*t+phi)

def analytical_velocities(t: np.ndarray, A: float, omega: float, phi=0.0):
    return -omega*A*np.sin(omega*t+phi)

def total_energy(pos: np.ndarray, vel: np.ndarray, k: float, m: float):
    return 0.5 * (k*(pos*pos) + m*(vel*vel))

if __name__ == "__main__":
    m = 0.1
    k = 5.0
    dts = [0.0001, 0.002, 0.02]

    params = {
        "text.usetex": True,
        "font.family": "serif",
        "font.serif": ["Computer Modern Serif"],
        "font.size": 16,
        "figure.figsize": (14, 8)
    }
    plt.rcParams.update(params)

    # Set to EULER or LEAPFROG
    integrator = LEAPFROG

    T = 5 # Simulate for 10 seconds
    for plot_i in range(3):
        dt = dts[plot_i]
        time_steps = int(T / dt)

        position = np.array([0.1])
        velocity = np.array([0.0])
        acceleration = np.array([0.0])

        position_history = np.zeros((time_steps,1))
        energy_history = np.zeros((time_steps,1))

        for t in range(time_steps):
            position_history[t] = position
            energy_history[t] = total_energy(position, velocity, k, m)

            if integrator == EULER:
                acceleration = spring_force(position, k) / m
                (position, velocity) = euler_step(position, velocity, acceleration, dt=dt)
            elif integrator == LEAPFROG:
                (position, velocity) = leapfrog(position, velocity, lambda x: spring_force(x,
                    k)/m, dt=dt)
```

```
        times = np.arange(time_steps) * dt
        analytical_sol = analytical_positions(times, 0.1, np.sqrt(k/m))
        analytical_v = analytical_velocities(times, 0.1, np.sqrt(k/m))
        analytical_energy = total_energy(
                analytical_sol,
                analytical_v,
                k,
                m
        )

        plt.subplot(2, 3, plot_i+1)
        plt.plot(times, position_history[:,0], ".", markersize=1)
        plt.plot(times, analytical_sol)
        plt.title(f"$\Delta t = {dt * 1000:.1f}$ ms")
        plt.ylabel("$x(t)$")

        plt.subplot(2, 3, plot_i+4)
        plt.plot(times, energy_history/analytical_energy[0]-1)
        plt.plot(times, analytical_energy/analytical_energy[0]-1)
        plt.xlabel("$t$")
        plt.ylabel("$(E(t)-E_0)/E_0$")

    plt.tight_layout()
    plt.show()
```

Harmonic-Oscillator/integrate.py (used for 1.1, 1.3 and 1.6)

```
import numpy as np

from typing import Callable, Tuple

def euler_step(pos: np.ndarray, vel: np.ndarray, acc: np.ndarray, dt = 1.0) -> Tuple[np.
    ndarray, np.ndarray]:
    return (pos + vel*dt, vel + acc*dt)

def leapfrog(pos: np.ndarray, vel: np.ndarray, acc_func: Callable[[np.ndarray], np.ndarray],
    dt = 1.0) -> Tuple[np.ndarray, np.ndarray]:
    pos_half = pos + vel * dt/2
    acc_half = acc_func(pos_half)

    vel_next = vel + acc_half * dt
    pos_next = pos_half + vel_next * dt/2
    return (pos_next, vel_next)
```

Two-Dimensional-Gas/main.py (used for 1.6)

```
#!/usr/bin/python

import numpy as np
import os
import sys

sys.path.insert(0, "../Harmonic-Oscillator/")

from datetime import datetime
from matplotlib import pyplot as plt
from integrate import leapfrog
from typing import Tuple
from tqdm import trange

SNAPSHOT = 0
ENERGIES = 1

def init_positions(size: Tuple[int, int], L: float, sigma: float):
    positions = 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 - positions[j,:])**2)) for j in range(i) ]
            too_close = [ r > sigma for r in distances ]

            if all(too_close):
                positions[i,:] = new_pos
                point_placed = True

    return positions

def init_velocities(size: Tuple[int, int], v0: float):
    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_potential(positions: np.ndarray, epsilon: float, sigma: float) -> np.
   ndarray:
    potentials = np.zeros(positions.shape[0])

    for i in range(positions.shape[0]):
        # "Triangular" iteration avoids calculating force on self and duplicate calculations
        for j in range(i+1, positions.shape[0]):
            r = (np.sum((positions[i,:] - positions[j,:])**2))
            magnitude = 4 * epsilon * ( np.power(sigma**2/r,6) - np.power(sigma**2/r,3) )

            potentials[i] += magnitude
            potentials[j] += magnitude
    return potentials

def lennard_jones_force(positions: np.ndarray, epsilon: float, sigma: float) -> np.ndarray:
    forces = np.zeros_like(positions)

    for i in range(positions.shape[0]):
        # "Triangular" iteration avoids calculating force on self and duplicate calculations
        for j in range(i+1, positions.shape[0]):
            r = np.sqrt(np.sum((positions[i,:] - positions[j,:])**2))
            magnitude = 4 * epsilon * ( 12*np.power(sigma,12)*np.power(r, -13) - 6*np.power(
                sigma,6)*np.power(r,-7) )
            # Direction of force exerted on i (towards j)
            direction = (positions[j,:] - positions[i,:]) / r
```

```
            forces[i,:] -= magnitude*direction
            forces[j,:] += magnitude*direction
    return forces

def kinetic_energy(velocities: np.ndarray, m: float, velocity_scale: float) -> float:
    return 0.5 * np.sum((velocities)**2)

def potential_energy(positions: np.ndarray, epsilon: float, sigma: float) -> float:
    # Check the math on this one
    return 0.5 * np.sum(lennard_jones_potential(positions/sigma, epsilon, sigma))

if __name__ == "__main__":
    m       = 0.1
    epsilon = 1.0
    sigma   = 1.0

    velocity_scale = np.sqrt(2*epsilon/m)
    time_scale = sigma*np.sqrt(m/(2*epsilon))

    L = sigma * 100
    N = 10

    positions = init_positions((N,2), L, sigma)
    velocities = init_velocities((N,2), 2*velocity_scale)

    time_steps = 100000
    plot_freq = 1
    dt = sigma/(2*velocity_scale) * 0.005
    position_history = np.empty((time_steps//plot_freq,N,2))

    E_k_history = np.empty(time_steps//plot_freq)
    E_p_history = np.empty(time_steps//plot_freq)

    plotting = ENERGIES
    logging = False

    print(f"---- Configuration ------")
    print(f"  Length scale : {sigma}")
    print(f"    Mass scale : {m}")
    print(f"  Energy scale : {epsilon}")
    print(f"Velocity scale : {velocity_scale:.4f}")
    print(f"    Time scale : {time_scale:.4f}")
    print(f"-------------------------")

    for t in trange(time_steps, desc="Timesteps", ncols=80):
        if logging or plotting == SNAPSHOT:
            position_history[t // plot_freq,:,:] = positions
        if t % plot_freq == 0 and plotting == ENERGIES:
            # Something is wrong with the units. Graphs have the correct shape but are not of
                the same scale
            E_k_history[t//plot_freq] = kinetic_energy(velocities, m, velocity_scale)
            E_p_history[t//plot_freq] = potential_energy(positions, epsilon, sigma)

        (positions, velocities) = leapfrog(positions, velocities, lambda x:
           lennard_jones_force(x, epsilon, sigma), dt=dt)

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

            # Outside right bound
            elif positions[i,0] > L:
                positions[i,0] = 2*L - positions[i,0]
                velocities[i,0] = velocities[i,0] * -1
```

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

            # Outside upper bound
            elif positions[i,1] > L:
                positions[i,1] = 2*L - positions[i,1]
                velocities[i,1] = velocities[i,1] * -1

    if plotting == SNAPSHOT:
        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])
    elif plotting == ENERGIES:
        E_k_history -= E_k_history[0]
        E_p_history -= E_p_history[0]
        plt.subplot(3, 1, 1)
        plt.plot(np.arange(time_steps//plot_freq) * plot_freq * dt / time_scale, E_k_history,
            '', label="Kinetic Energy", markersize=1)
        plt.ylabel("$\Delta E_k$ $[\\varepsilon]$")
        plt.subplot(3, 1, 2)
        plt.plot(np.arange(time_steps//plot_freq) * plot_freq * dt / time_scale, E_p_history,
            '', label="Potential Energy", markersize=1)
        plt.ylabel("$\Delta E_p$ $[\\varepsilon]$")
        plt.subplot(3, 1, 3)
        plt.plot(np.arange(time_steps//plot_freq) * plot_freq * dt / time_scale, (E_k_history
            + E_p_history), '', label="Potential Energy", markersize=1)
        plt.ylabel("$\Delta E$ $[\\varepsilon]$")
        plt.xlabel("$t$ $[t_0]$")

    if logging:
        try:
            os.mkdir("./logs")
        except FileExistsError:
            pass
        np.savetxt("./logs/" + datetime.now().strftime("%d-%m-%Y-%H:%M:%S") + ".txt",
            position_history.reshape(position_history.shape[0],N*2))
    plt.show()
```