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()
```