Computational Physics - Exercise 5: Molecular dynamics simulations

Torben Steegmann Chih-Chun Kuo

Introduction

Some physical phenomena cannot be solved analytically. One well-known example is the Three-body Problem, where the trajectory of three interacting bodies must be calculated numerically due to the complex interactions involved. In this exercise, we demonstrate several numerical methods using a harmonic oscillator as our system of study. The harmonic oscillator is a fundamental model in physics, representing systems that exhibit simple harmonic motion, such as springs and pendulums.

We chose the harmonic oscillator because it has an analytical solution, which allows us to compare our numerical results and verify their accuracy. By exploring different numerical methods, we aim to understand their strengths and weaknesses, as well as their applicability to more complex systems where analytical solutions are not available.

Simulation model and method

The exercise is split into four different sub-tasks, where the first three implement different time-integration methods to solve the motion of a harmonic oscillator. The analytical solution of such an oscillator is well-known. The force acting on the body of a spring is described as follows:

$$F(t) = m * \frac{d^2x}{dt^2} = ma = -kx,$$

therefore:

$$\frac{d^2x}{dt^2} = \frac{-k}{m}x$$

The body in our system has a constant mass of $m = 1$ and the spring constant is also set to $k = 1$. This reduces the formula for the force to $F(t) = -x$.

With this, the trajectory can be rewritten as:

$$x(t) = x_0 \cos(t) + v_0 \sin(t).$$

With our initial starting condition set to $x_0 = 0, v_0 = 1$ this reduces to:

$$x(t) = \sin(t).$$

Instead of solving the differential equations, we can approximate their solution with one of several algorithms. These algorithms update the velocity and positions of bodies iteratively.

Euler algorithm

The first one is the Euler algorithm. It iteratively updates the velocity and position as follows:

$$x(t + \Delta t) = x(t) + v(t) * \Delta t$$
$$v(t + \Delta t) = v(t) + a(t) * \Delta t$$

This is generally perceived to be the simplest approximation method, making it easy to implement and fast, but it is also known for having many drawbacks. As it only approximates the position and velocity, its calculations result in an local error of $O\Big((\Delta t)^2\Big)$ and global error of $O(\Delta t)$. The local error refers to the error introduced by one time step, while the global error refers to the cumulative error caused by many iterations. The error is large, because all we do is calculate the next position and velocity using the old position, velocity, and acceleration. In reality, these do not stay constant between two time steps, but we treat them as though they do. It is also neither stable nor symplectic.

Euler-Cromer algorithm

Another algorithm to approximate the analytical result is the Euler-Cromer algorithm. For this exercise, we were presented with two different implementations of it:

$$\text{(a) } v(t+\Delta t) = v(t) - V'(x(t)) \cdot \Delta t$$
$$\text{(a) } x(t+\Delta t) = x(t) + v(t+\Delta t) \cdot \Delta t$$

$$\text{(b) } x(t+\Delta t) = x(t) + v(t) \cdot \Delta t$$
$$\text{(b) } v(t+\Delta t) = v(t) - V'(x(t+\Delta t)) \cdot \Delta t$$

The difference between (a) and (b) lies in the order of the updates. In (a), we update the velocity first and then update the position using the new velocity. In (b), we first update the position and then update the velocity using the new position. As all we do is change the order of updates, the errors are both $O\Big((\Delta t)^2\Big)$ locally and $O(\Delta t)$ globally. In contrast to the Euler algorithm, the Euler-Cromer algorithm is symplectic and stable.

Velocity Verlet algorithm

The last approximation algorithm we implement is the Velocity Verlet algorithm. Its update steps are as follows:

$$x(t+\Delta t) = x(t) + v(t) \cdot \Delta t + 0.5 \cdot a(t) \cdot (\Delta t)^2$$
$$v(t+\Delta t) = v(t) + 0.5 \cdot (a(t) + a(t+\Delta t)) \cdot \Delta t$$

Here, the velocity interpolates the current and next acceleration, and we also use the acceleration in the positional update. Therefore, the local error reduces to $O\Big((\Delta t)^3\Big)$ while the global error reduces to $O\Big((\Delta t)^2\Big)$. This reduction in error is important, as this means that the cumulative error increases slower than the number of time steps. It is also symplectic, stable, and even time-reversible, as velocities and positions are calculated simultaneously. Because we do not need to solve any linear equations, it is still simple and fast to implement, making it the most widely used approximation algorithm.

Harmonic oscillators are foundational models in physics, with applications ranging from mechanical systems like springs and pendulums to electrical circuits and molecular dynamics. The ability to approximate solutions using numerical methods is crucial, especially for more complex systems that cannot be solved analytically. Understanding these methods and their limitations is essential for practical applications in engineering and science.

Coupled Oscillators

The last part of the exercise uses the Velocity Verlet algorithm to solve the equation of motion of many coupled oscillators. The motion of N coupled oscillators is described by as follows:

$$H = K + V = \frac{1}{2}\sum_{n=1}^{N} v_n^2 + \frac{1}{2}\sum_{n=1}^{N-1} \left(x_n - x_{n+1}\right)^2,$$

with

$$\frac{\partial V}{\partial x_k} = 2x_k - x_{k-1} - x_{k+1}, 1 < k < N$$

$$\frac{\partial V}{\partial x_1} = x_1 - x_2$$

$$\frac{\partial V}{\partial x_k} = x_N - x_{N-1}$$

As the Velocity Verlet is an iterative approach, to calculate the positions and velocities of all oscillators at the next time step, we need to know their positions and velocities of the current time step and their acceleration a. As we know $a = \frac{F}{m}$ and since $m = 1$ it follows $a = F$. The force is the negative gradient of the oscillator so $F = -\frac{\partial V}{\partial x}$ so that $a = -\frac{\partial V}{\partial x}$.

For all the Exercises we were given different configurations. These can either change the time step size Δt, the number of time steps, the starting conditions x_0, v_0 or the amount of oscillators N.

Implementation

The implementation follows the described methods closely. We begin by defining the Euler Method, which takes the time step size as an argument. This allows us to define the length of the simulation with $\frac{10000}{\text{step}}$, keeping the total simulation time constant, even with different time step sizes. After initializing v_0 and x_0, we perform the integration according to the formula and plot the results. As the other integration methods work similarly in theory, their implementations are almost identical. Note that the two different Euler-Cromer methods were implemented as 'Euler_Cromer_a' and 'Euler_Cromer_b'.

Because we had already implemented the Velocity Verlet Algorithm, solving the system of coupled oscillators is also straightforward. We simply update the acting force for the acceleration. One thing to be careful about is the order of updates. As the oscillators are coupled, the force acting on a body depends on the others in the system. Therefore, we cannot perform all time steps for one oscillator and then move on to the next. Instead, we must update all simultaneously.

Simulation results

Euler Method

Figure 1 - Figure 3 show the results of Euler method with $\Delta t = 0.1, 0.01, 0.001$. We can see from the subplots over the entire time range that the results increase rapidly, eventually leading to values that diverge significantly. Therefore, we created two additional subplots for each time step: one showing the beginning of the time and the other showing the end. This makes it easier to understand the behavior.

When $\Delta t = 0.1$, we see at around t=10, the position starts to deviate from the analytical value. This deviation decreases as we reduce Δt. For example, with $\Delta t = 0.001$, we observe minimal deviation at the beginning of the simulation.

However, reducing the time step alone is not sufficient to completely solve the problem of divergence, although it does mitigate it to some extent. The final values at the end of the simulation demonstrate this: for $\Delta t = 0.1$ is about 10^{216}, while with $\Delta t = 0.01$ it reaches 10^{21} and with $\Delta t = 0.001$ it reaches 10^{2}. The global error is supposed to be $O(\Delta t)$ theoretically. However, due to the instability of the algorithm, the error grows exponentially with larger time steps. Nonetheless, we can still observe that a smaller time step leads to a smaller error.

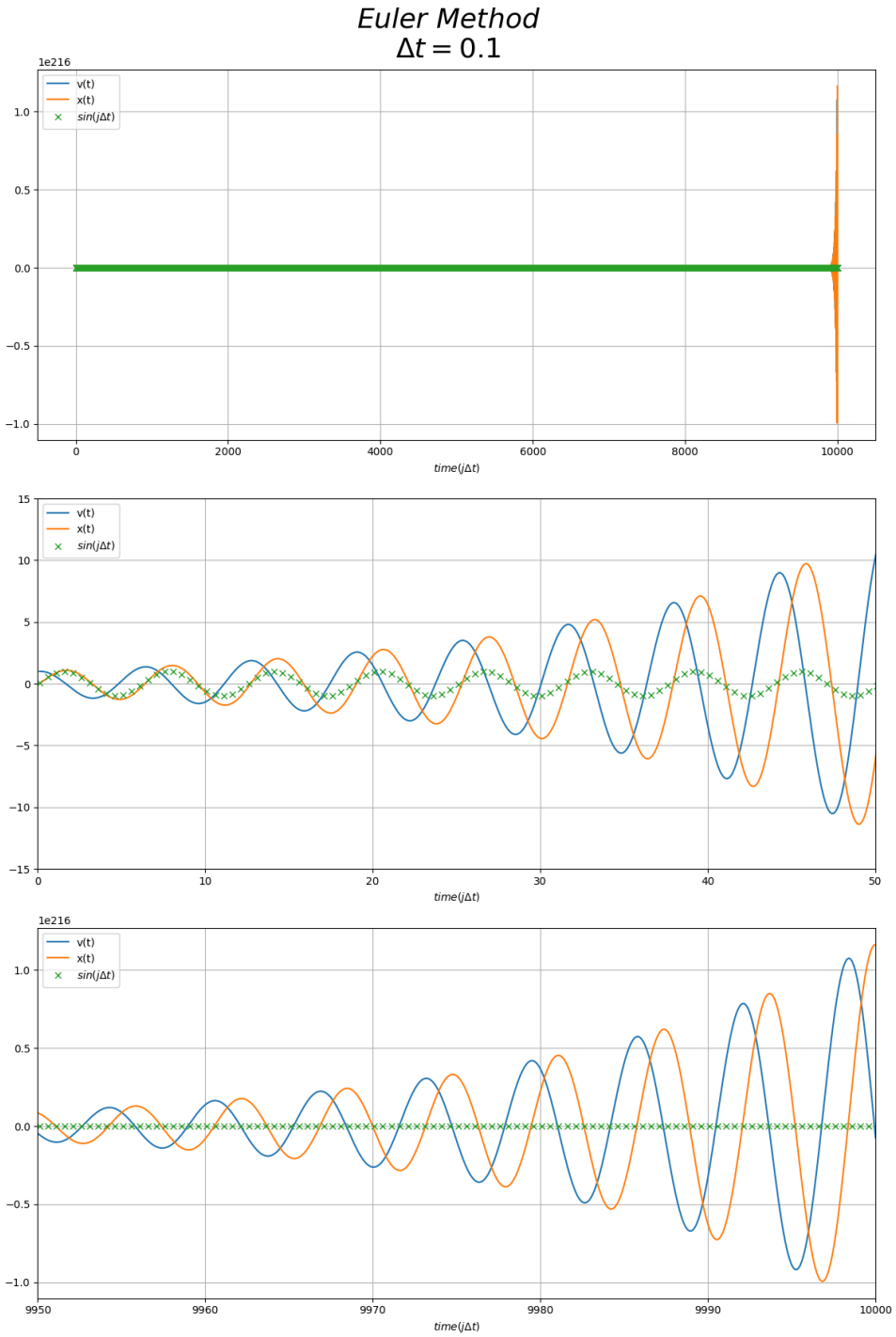

Figure 1: Single harmonic oscillator's position and velocity simulated using the Euler algorithm with time step $\Delta t = 0.1$.

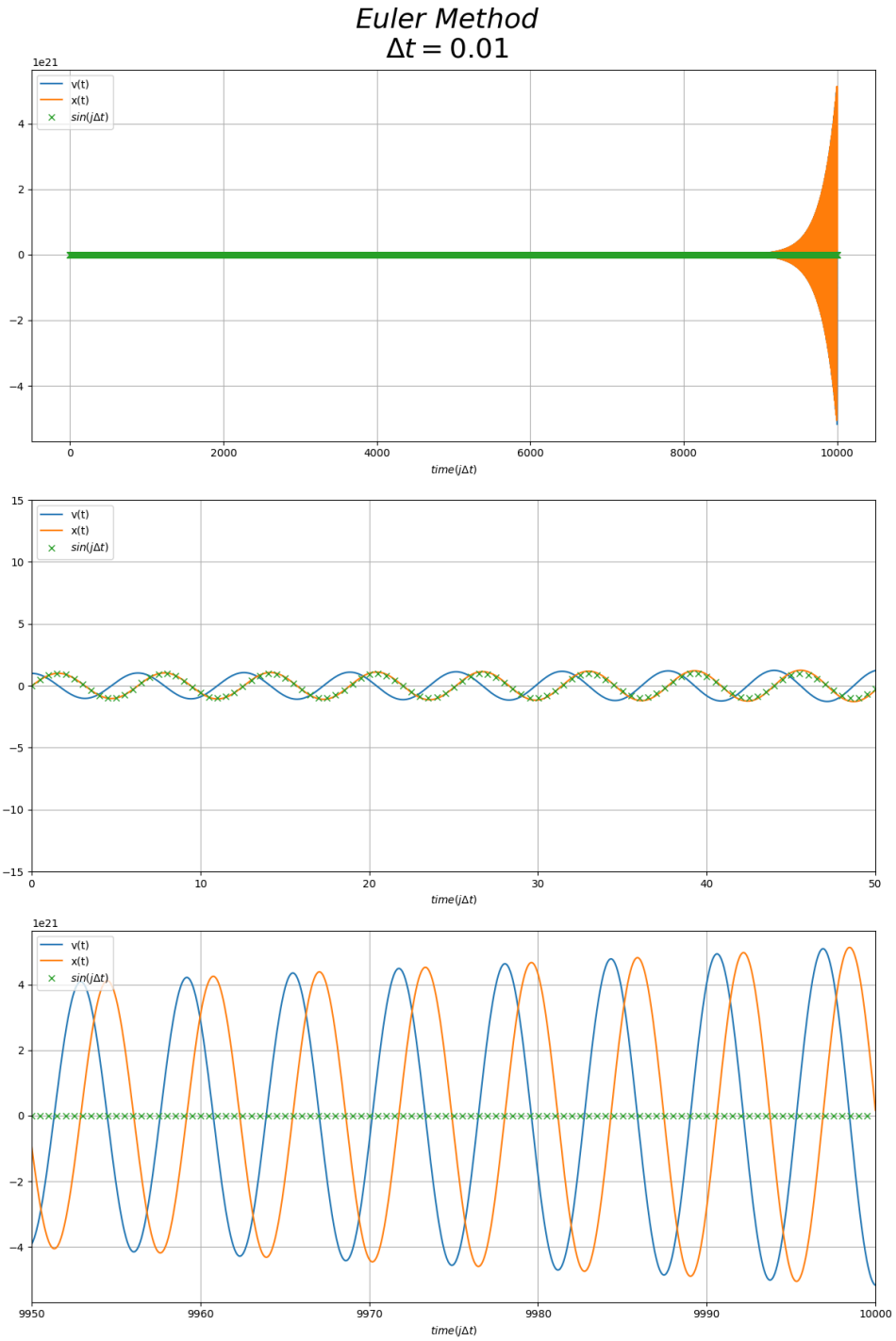

Figure 2: Single harmonic oscillator's position and velocity simulated using the Euler algorithm with time step $\Delta t = 0.01$.

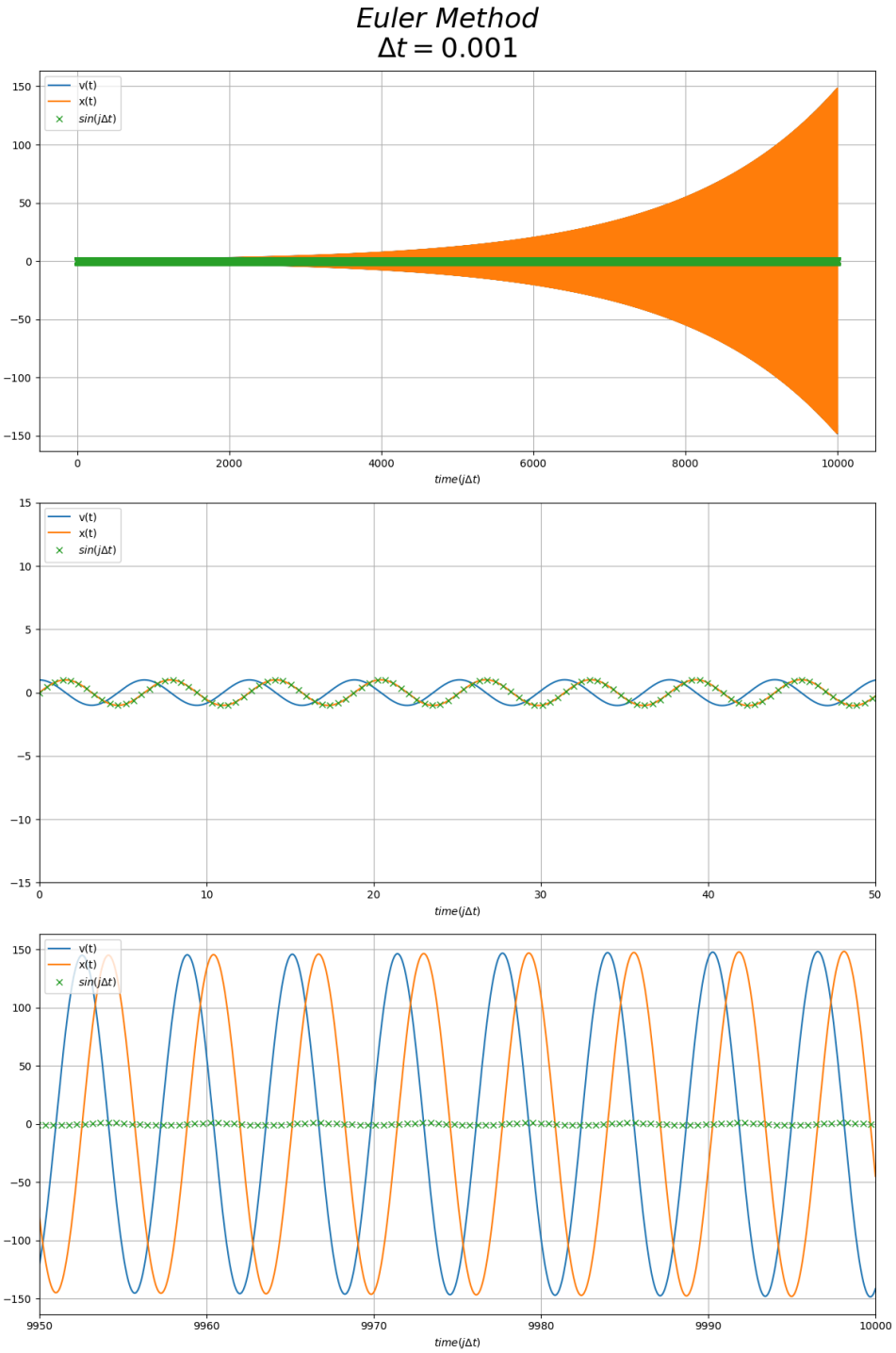

Figure 3: Single harmonic oscillator's position and velocity simulated using the Euler algorithm with time step $\Delta t = 0.001$.

Euler-Cromer Method

Figure 4 to Figure 6 show the results of variant (a) of the Euler-Cromer method with $\Delta t = 0.1, 0.01, 0.001$, while Figure 7 to Figure 9 show those of variant (b). These two variants produce identical results except

the phase shift in energy fluctuation. The energies generally remain around 0.5 with minor fluctuations.

Compared to the Euler method, we do not observe significant deviations at the beginning. However, there are time shifts in position and velocity towards the end of the simulations. The amplitude remains at one, indicating that this algorithm is stable.

We also observe that as the time step decreases, the deviation in position and velocity at the final time, as well as the fluctuation in energy, become smaller. The amplitude of energy fluctuates from 0.03 with $\Delta t = 0.1$ to 0.0003 with $\Delta t = 0.001$. This behavior supports the theoretical global error of $O(\Delta t)$. Moreover, the consistency of the energy fluctuation from the beginning to the end also proves that this fluctuation arises from algorithmic error.

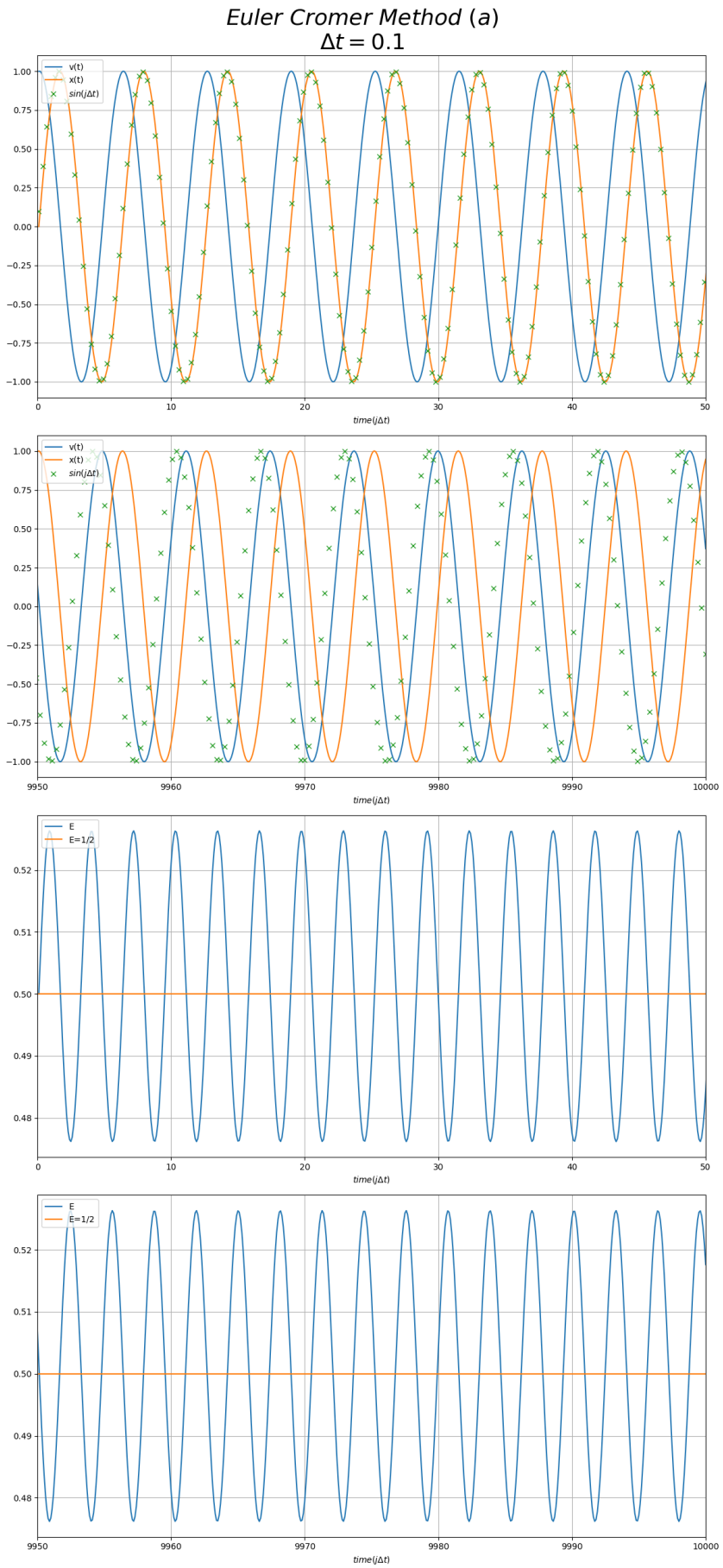

Figure 4: Single harmonic oscillator's position and velocity simulated using variation (a) of the Euler-Cromer algorithm with $\Delta t = 0.1$.

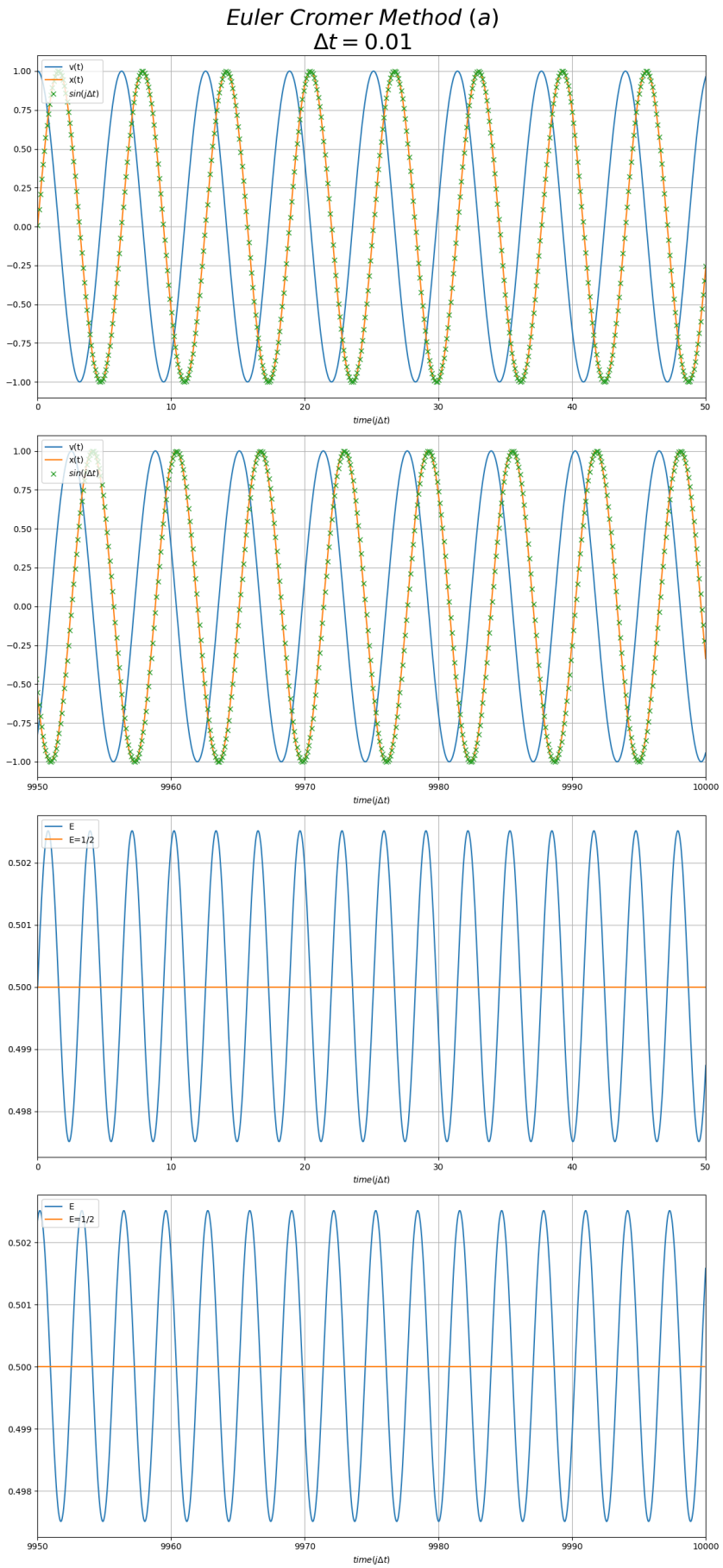

Figure 5: Single harmonic oscillator's position and velocity simulated using variation (a) of the Euler-Cromer algorithm with $\Delta t = 0.01$.

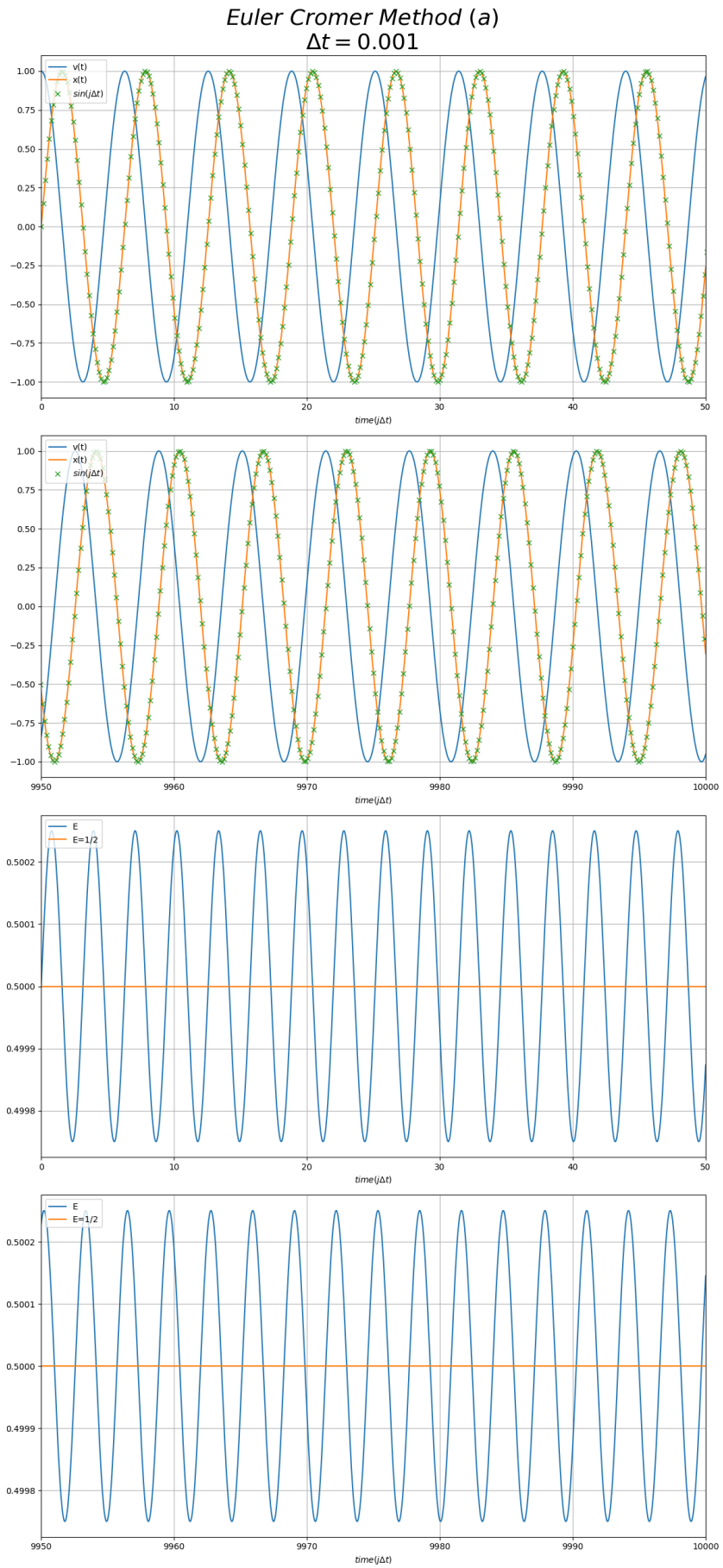

Figure 6: Single harmonic oscillator's position and velocity simulated using variation (a) of the Euler-Cromer algorithm with $\Delta t = 0.001$.

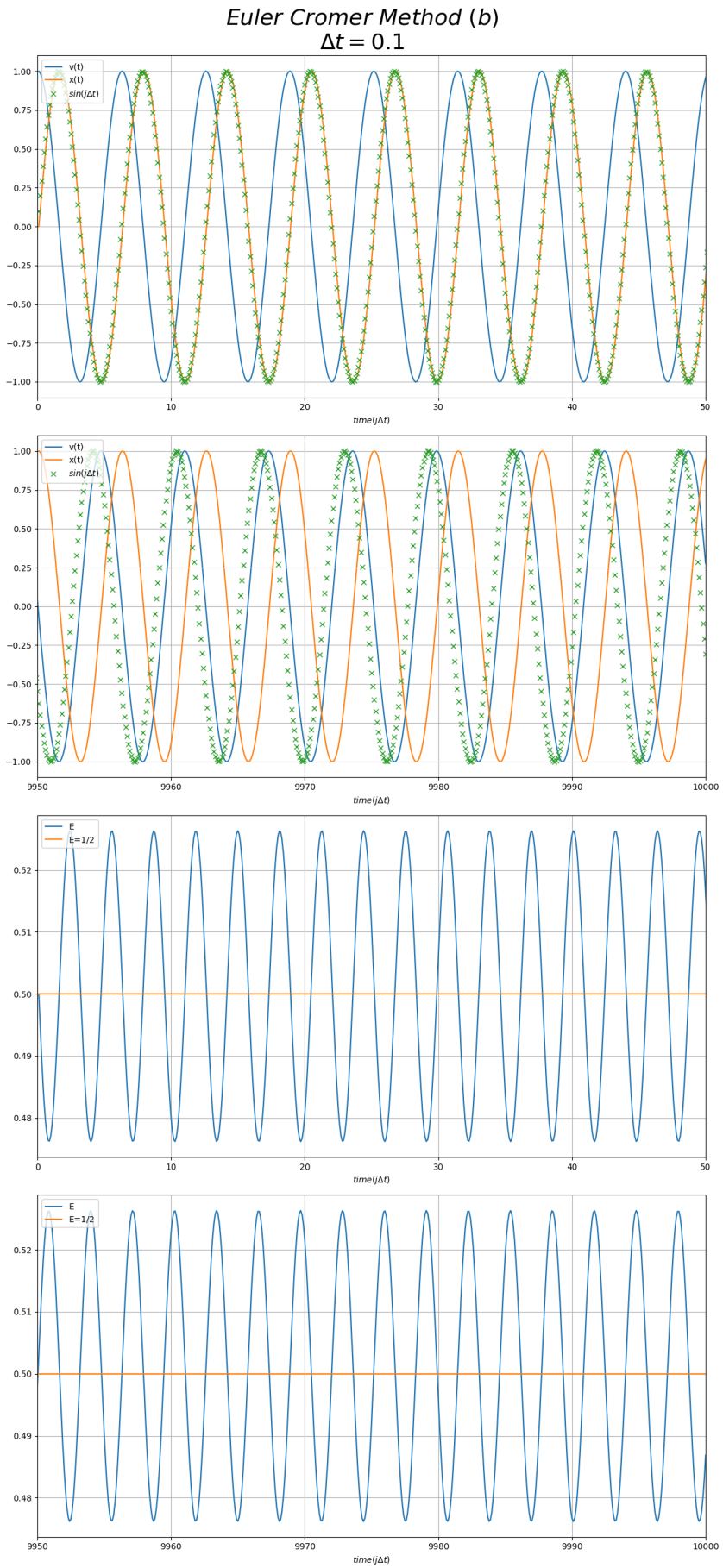

Figure 7: Single harmonic oscillator's position and velocity simulated using variation (b) of the Euler-Cromer algorithm with $\Delta t = 0.1$.

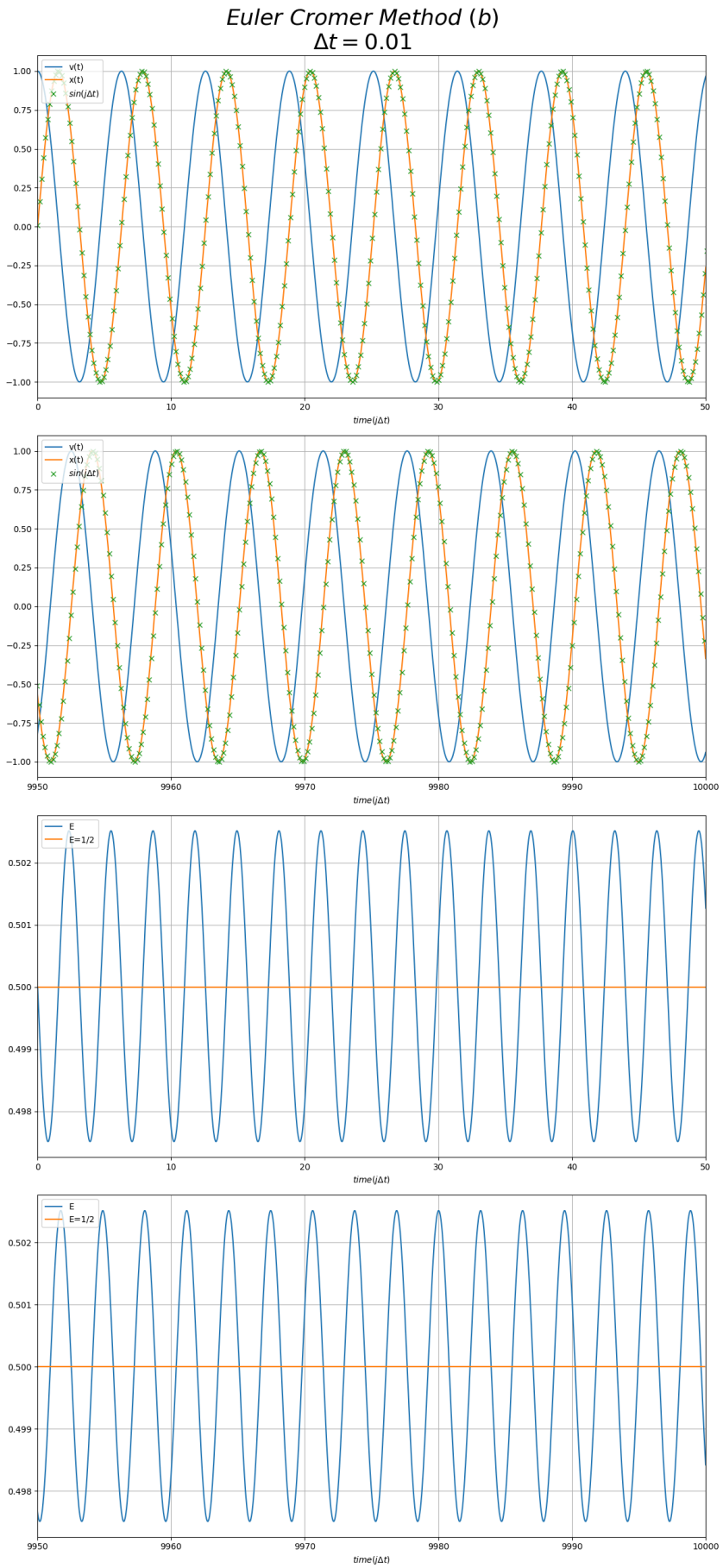

Figure 8: Single harmonic oscillator's position and velocity simulated using variation (b) of the Euler-Cromer algorithm with $\Delta t = 0.01$.

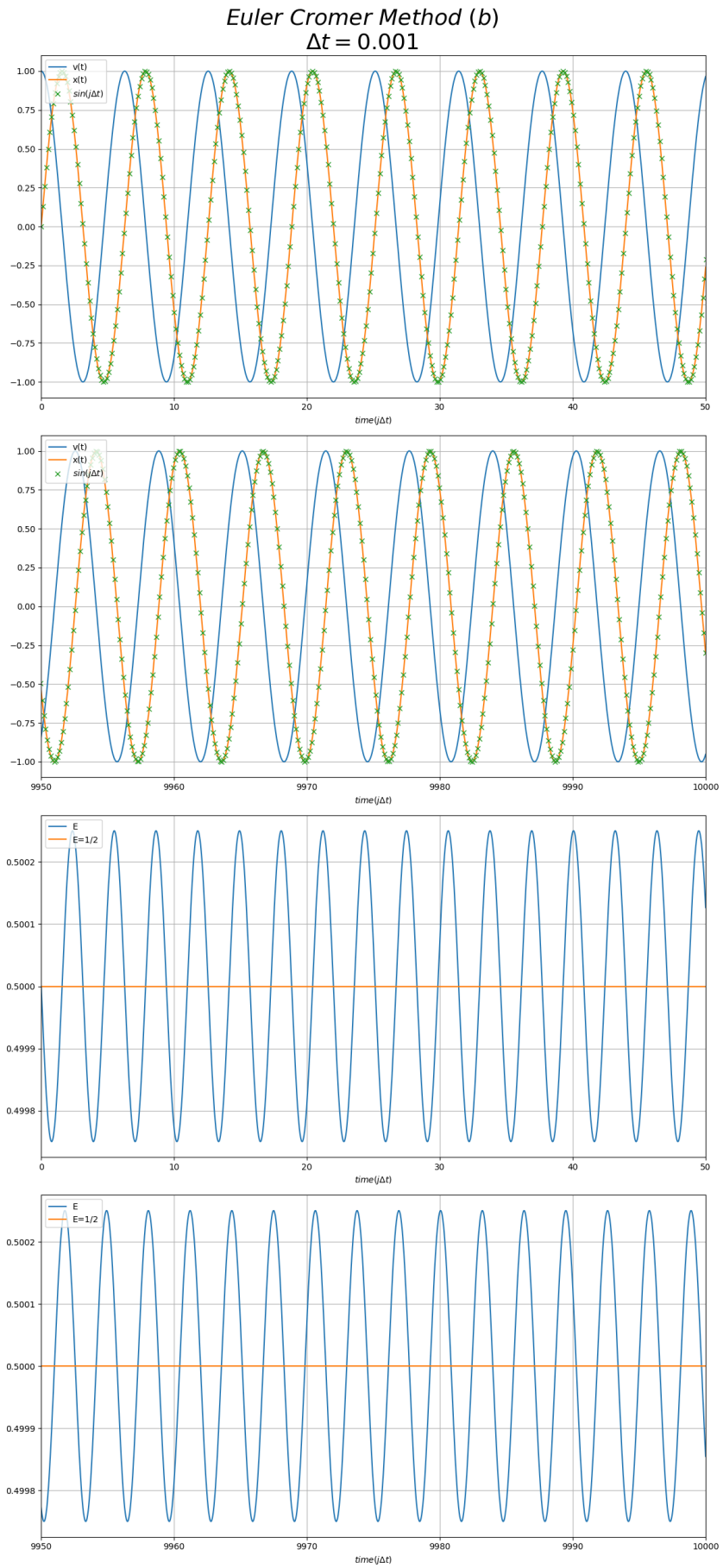

Figure 9: Single harmonic oscillator's position and velocity simulated using variation (b) of the Euler-Cromer algorithm with $\Delta t = 0.001$.

Velocity Verlet Algorithm

Figure 10 and Figure 11 show the results of velocity verlet algorithm with $\Delta t = 0.1, 0.01$ respectively. When $\Delta t = 0.1$, the behaviors of position and velocity are identical to Euler-Cromer algorithm. Initially, they closely match the analytical values, while at the end, time shift happens. The energy still remains around 0.5, but the balanced points of the fluctuation are not precisely at 0.5. This discrepancy may be due to smaller algorithmic errors in the velocity Verlet method, resulting in a noticeable shift in the average value around 0.5. However, when comparing the amplitudes with $\Delta t = 0.1$ in these two algorithms, the value for Euler is 0.03, while for velocity Verlet, it is 0.0006.

When $\Delta t = 0.01$, the time shift in the ending position and velocity vanishes. The amplitude now is $6 \cdot 10^{-6}$, showing that the error is on Δt^2 order.

In general, the velocity Verlet algorithm presents lower deviation from the analytical value and exhibits lower accumulating error as time evolves, providing more stable and precise results.

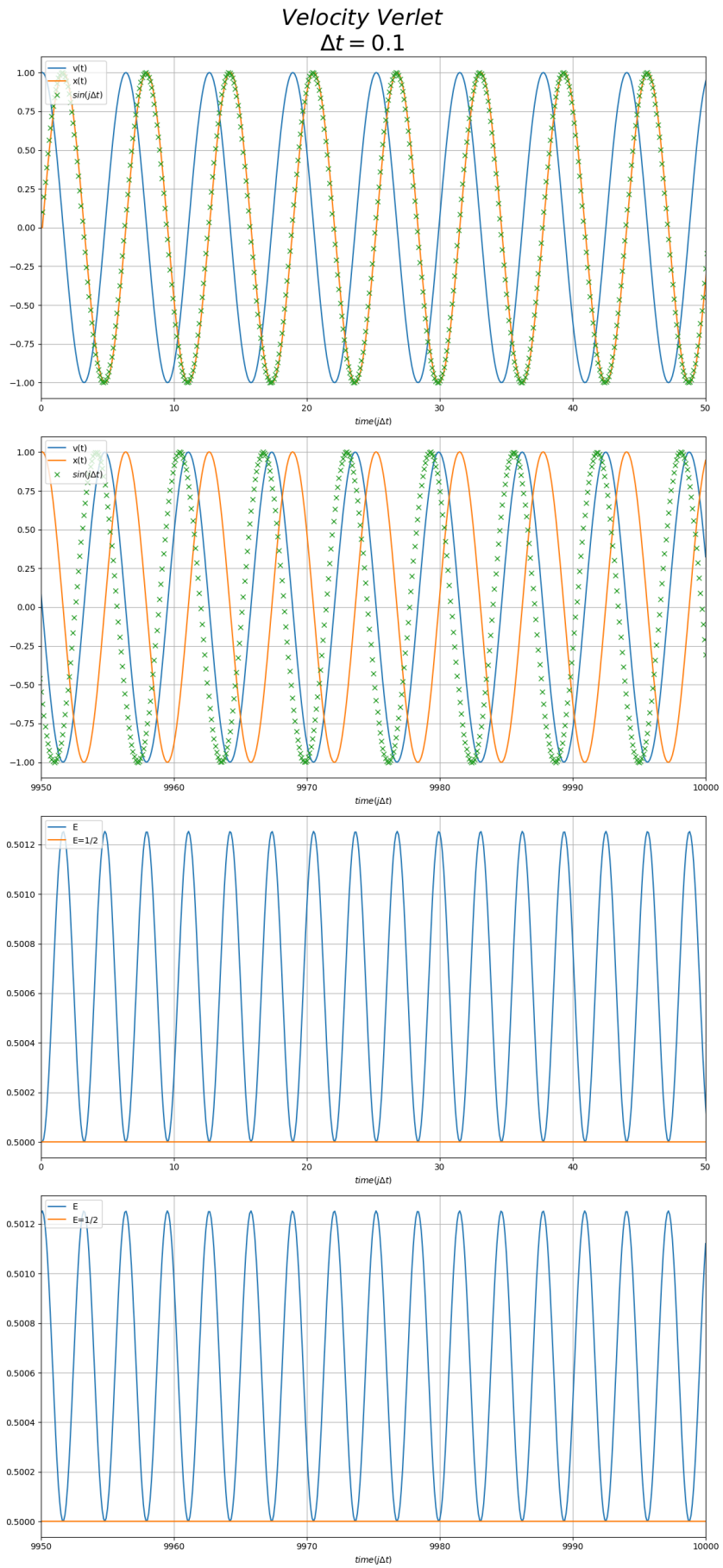

Figure 10: Single harmonic oscillator's position and velocity simulated using the Verlet algorithm with $\Delta t = 0.1$.

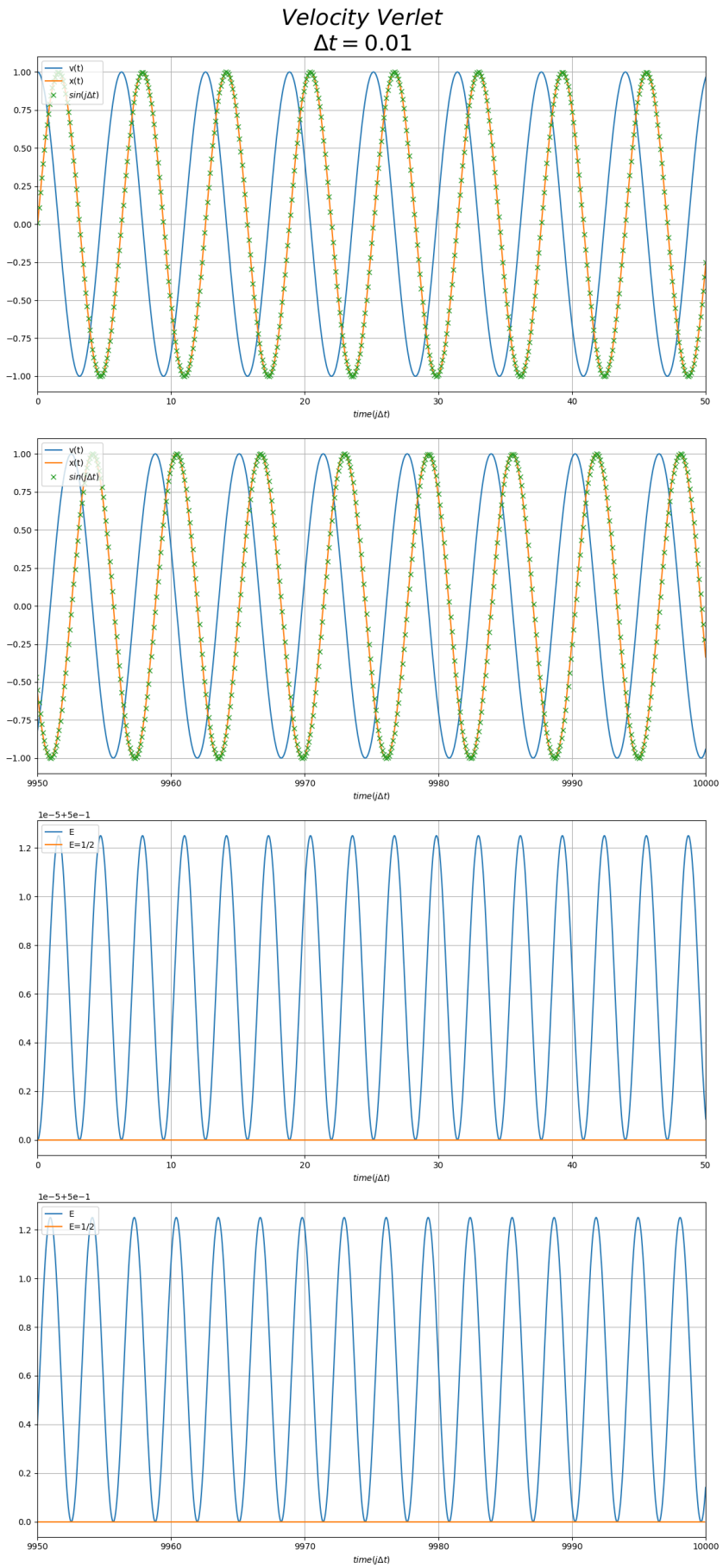

Figure 11: Single harmonic oscillator's position and velocity simulated using the Verlet algorithm with $\Delta t = 0.01$.

Velocity-Verlet on Coupled Oscillator

Figure 12 to Figure 29 show the results of applying velocity Verlet algorithm on N=4,16,128 coupled oscillators with $\Delta t = 0.1, 0.01$ and with three different initial configurations:

1. $$v_1(0), ..., v_N(0) = 0$$
$$x_1(0), ..., x_N(0) = 0, \text{except } x_{\frac{N}{2}}(0) = 1$$

2. $$v_1(0), ..., v_N(0) = 0$$
$$x_k(0) = \sin\left(\frac{\pi jk}{N+1}\right), \text{for } k = 1, ..., N \text{ and } j = 1$$

3. $$v_1(0), ..., v_N(0) = 0$$
$$x_k(0) = \sin\left(\frac{\pi jk}{N+1}\right), \text{for } k = 1, ..., N \text{ and } j = \frac{N}{2}$$

In general, the differences in position and velocity between the two time steps are not noticeable, but the amplitude of the oscillations, which represents the algorithmic error, is smaller with a smaller time step.

Due to energy conservation, the oscillators exchange energy, but the total energy remains almost constant. In the first configuration, as N increases, it takes longer for the oscillations to propagate through all the oscillators. In the second configuration with j=1, we initialize them in a balanced state, so they oscillate symmetrically without needing time for the oscillations to propagate. In the second configuration with j=N/2, some initializations are near zero, so it takes some time for the oscillations to propagate, but this time is shorter than in the first configuration. Overall, deriving an analytical result is challenging, but intuitive physics principles can validate these observations.

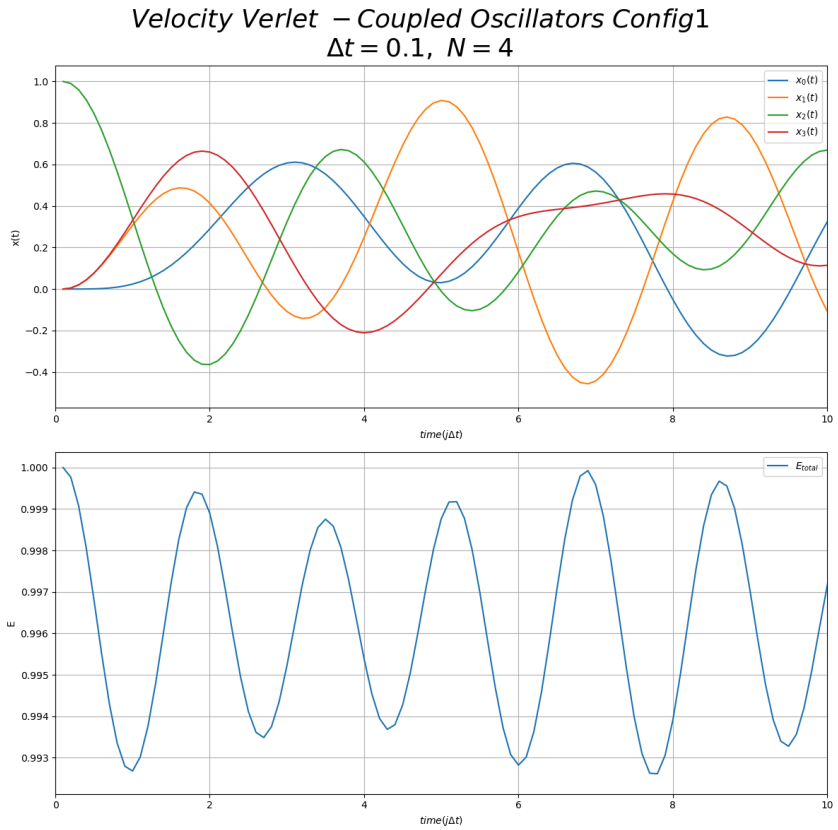

Figure 12: N = 4 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 1.

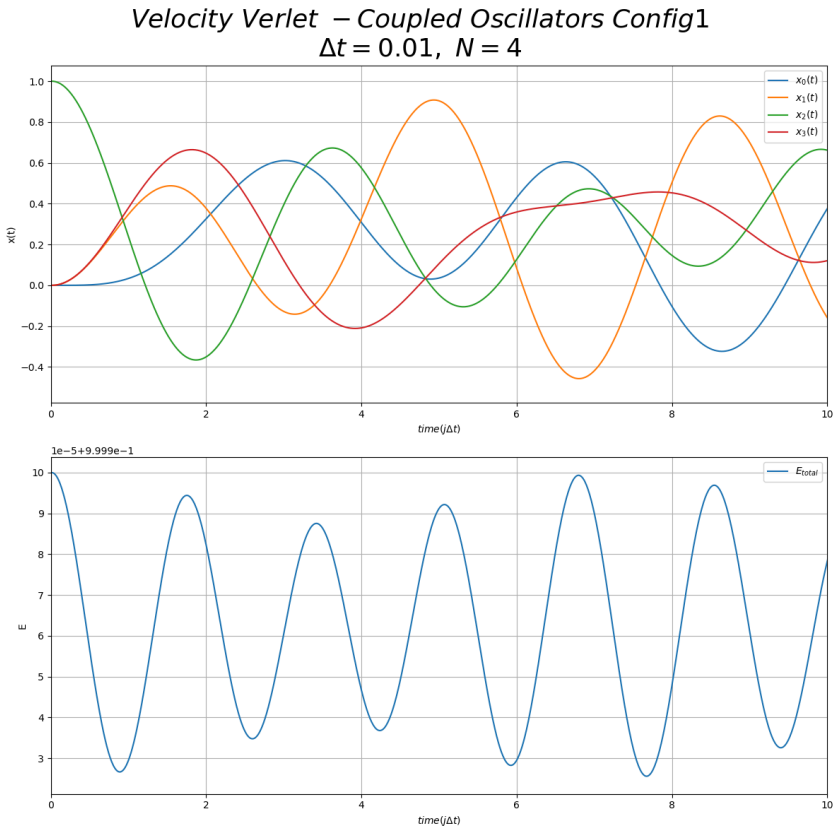

Figure 13: N = 4 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 1.

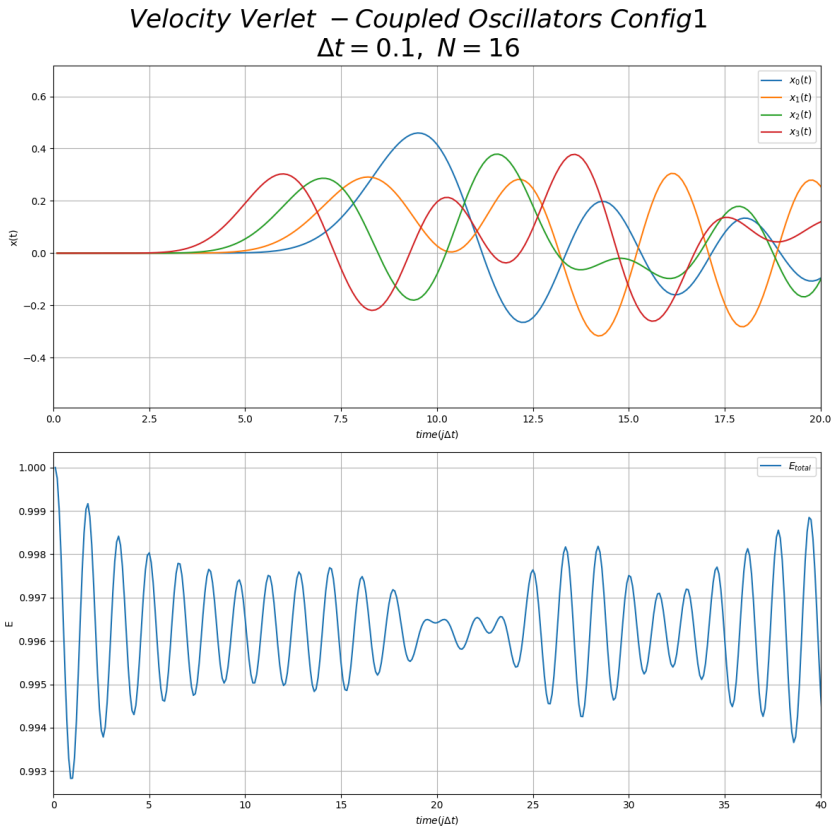

Figure 14: N = 16 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 1.

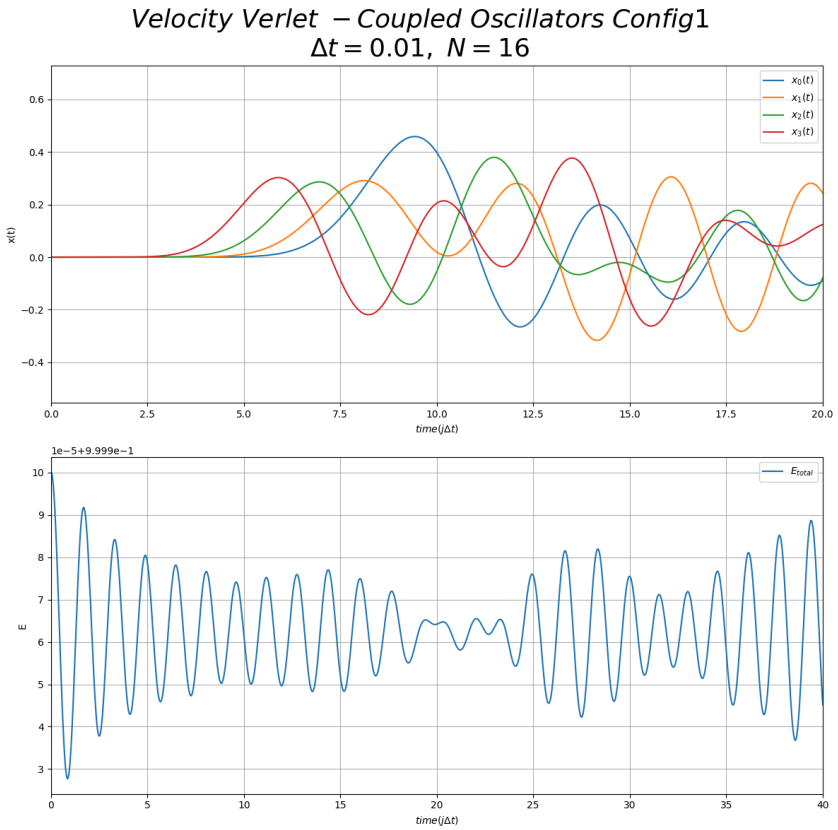

Figure 15: N = 16 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 1.

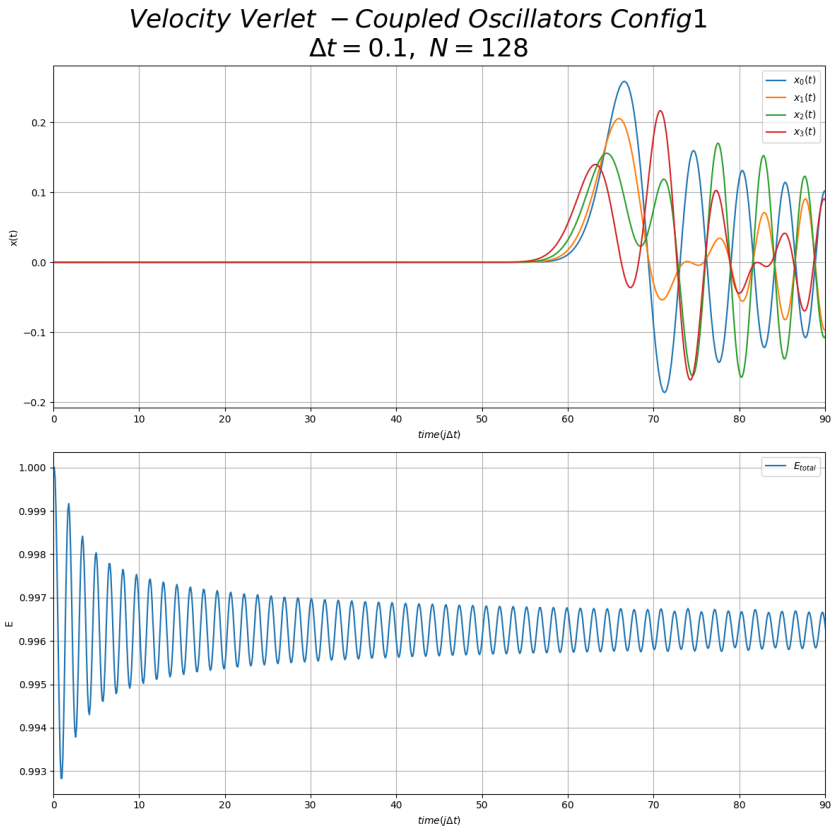

Figure 16: N = 128 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 1.

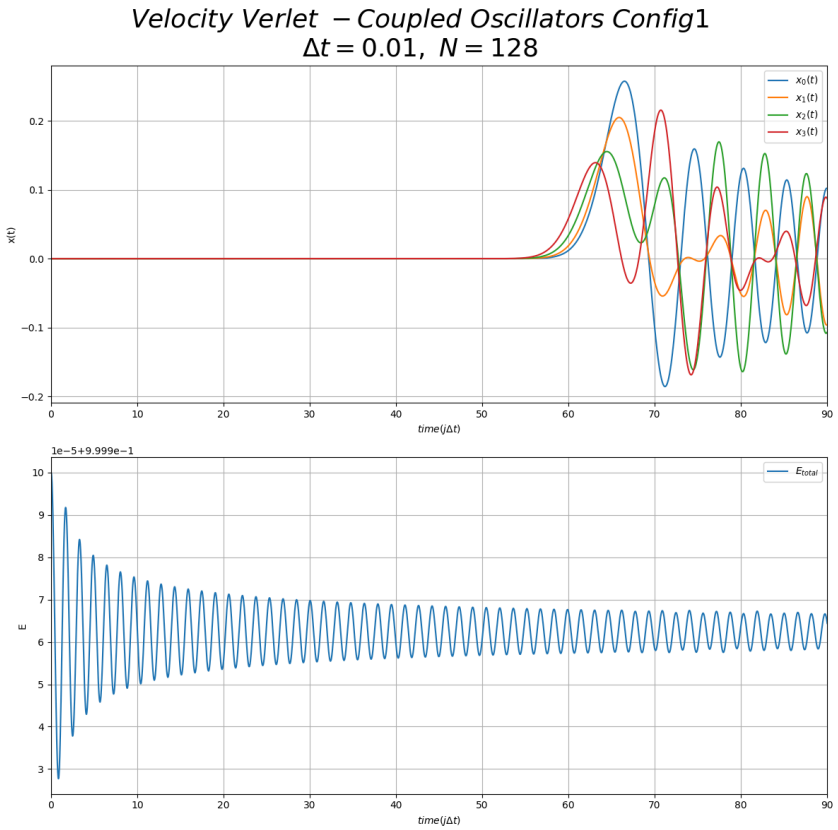

Figure 17: N = 128 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 1.

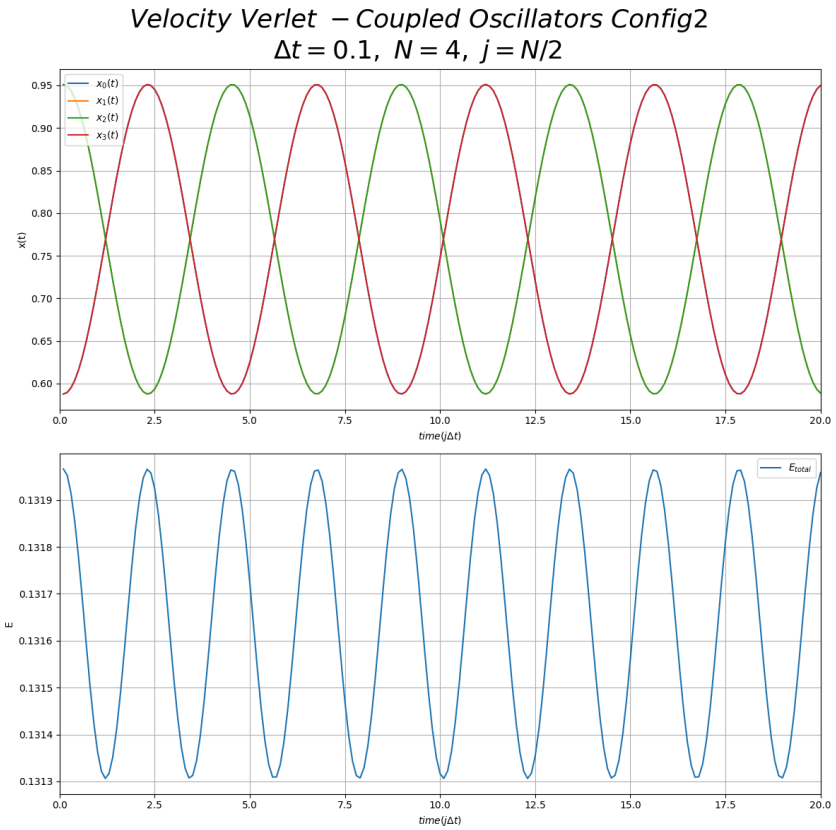

Figure 18: N = 4 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 2 (j=1).

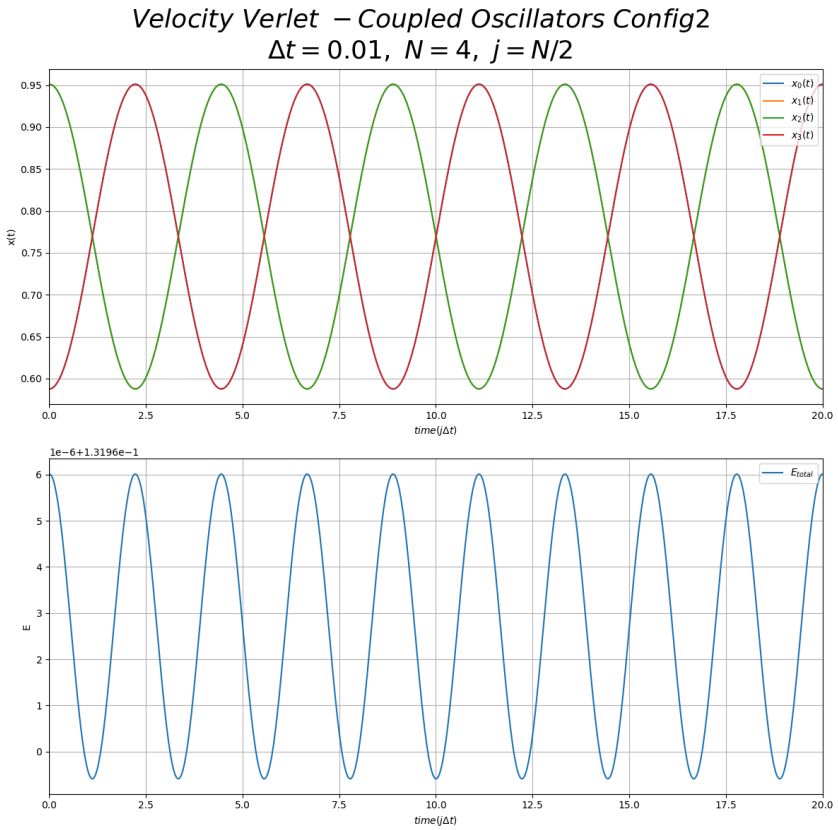

Figure 19: N = 4 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 2 (j=1).

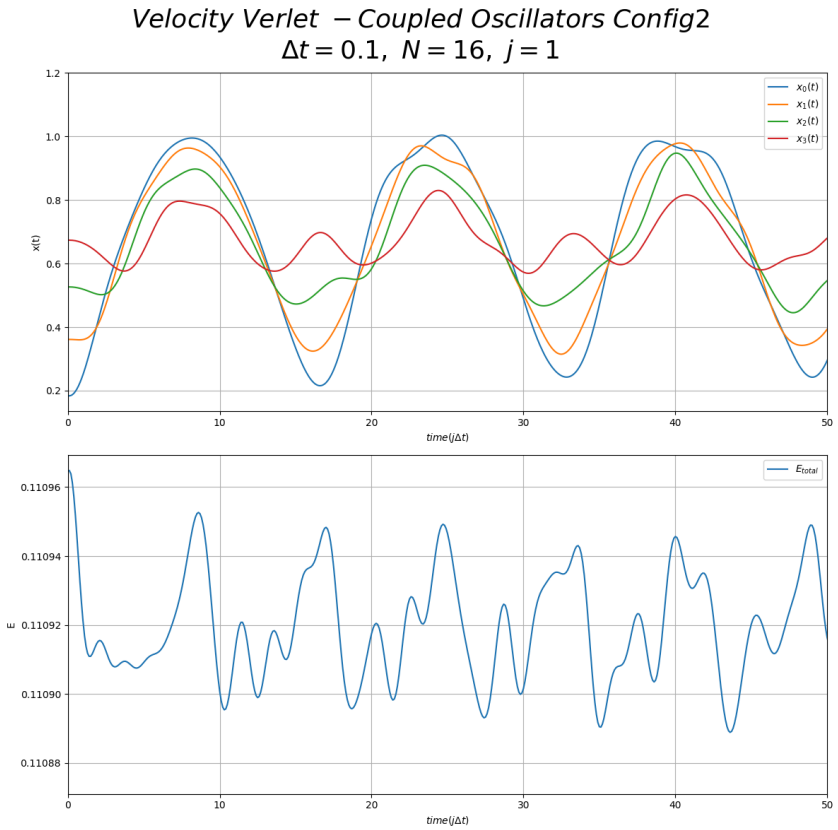

Figure 20: N = 16 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 2 (j=1).

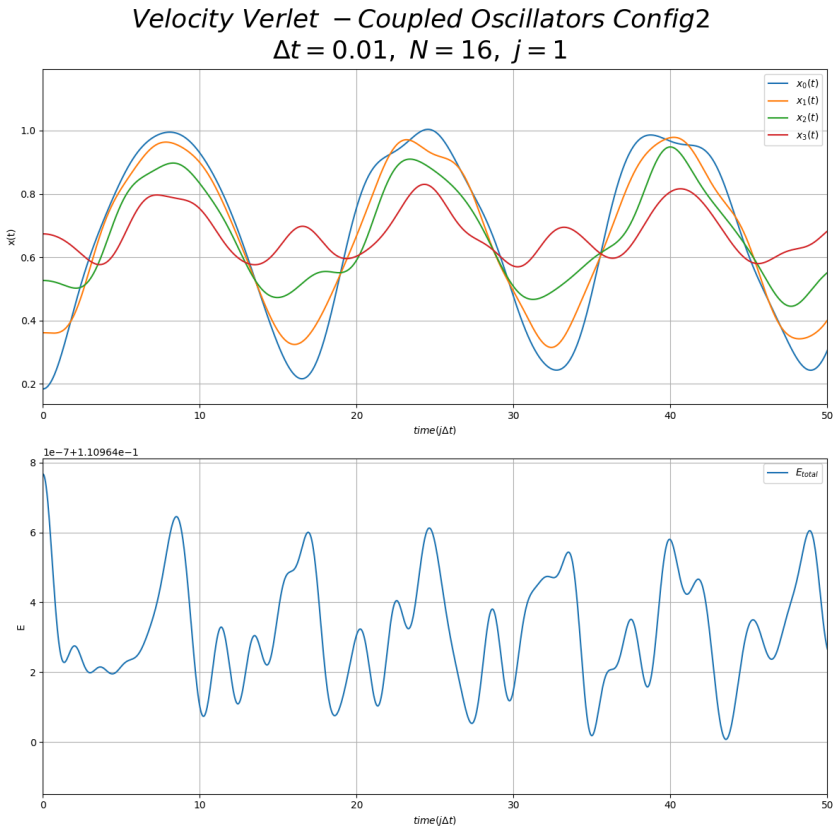

Figure 21: N = 16 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 2 (j=1).

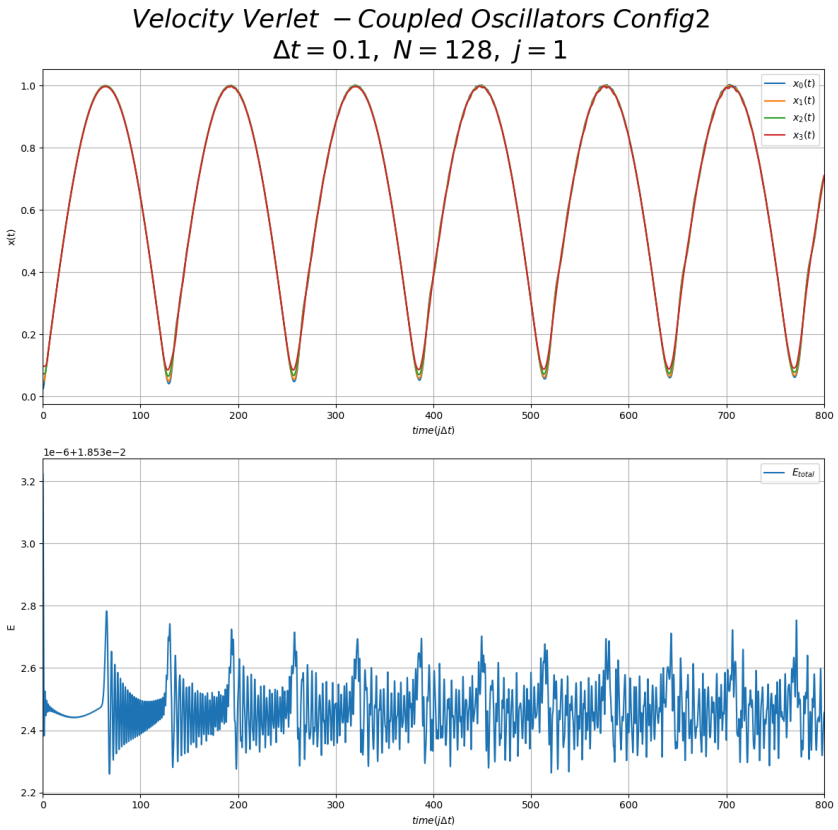

Figure 22: N = 128 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 2 (j=1).

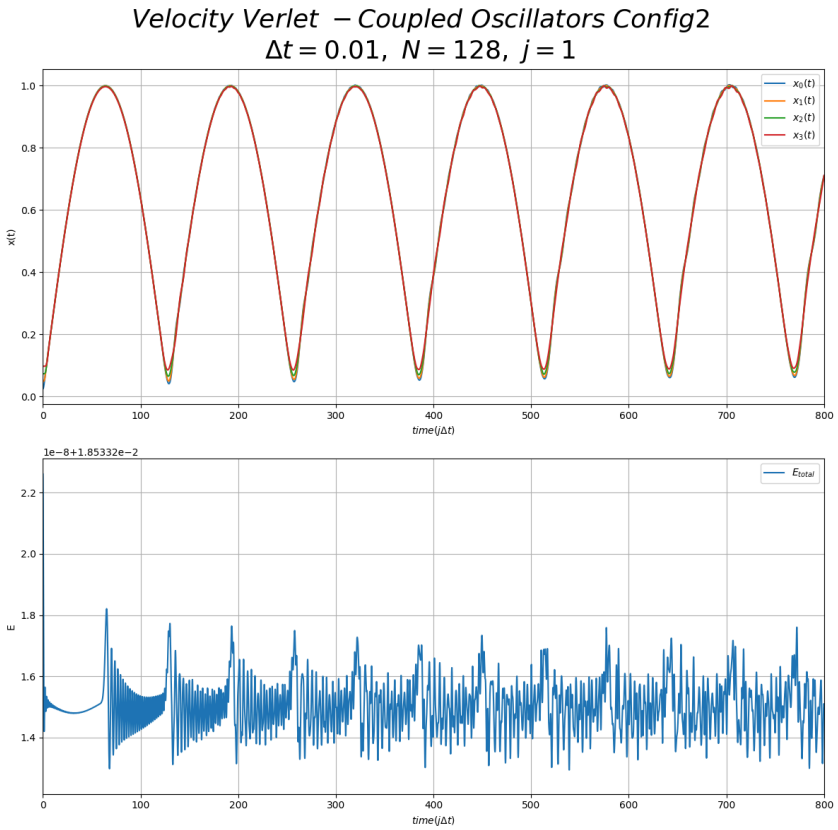

Figure 23: N = 128 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 2 (j=1).

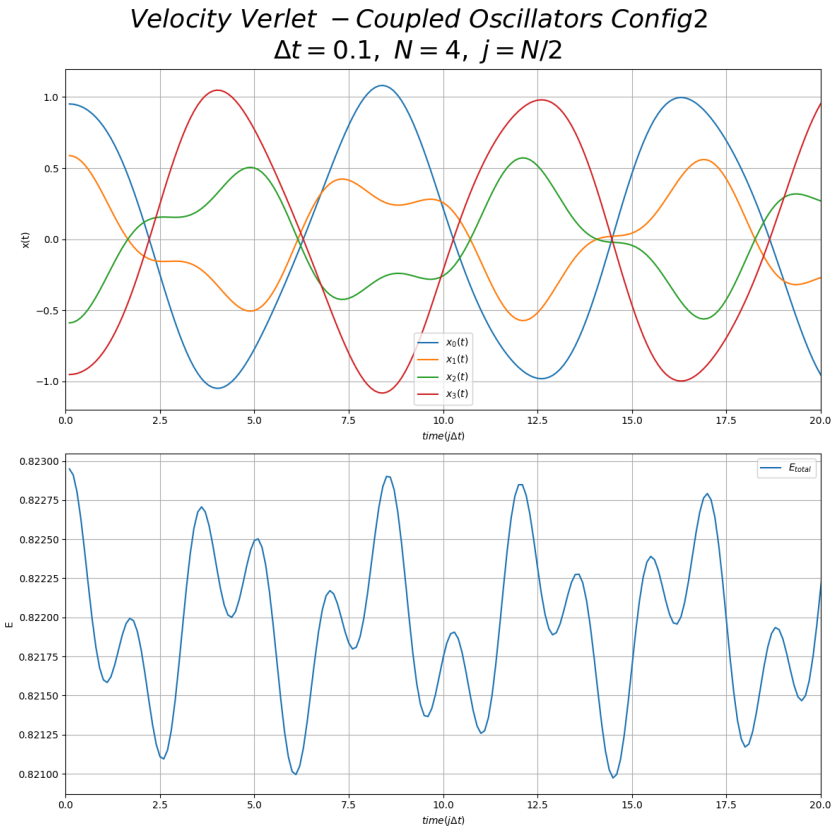

Figure 24: N = 4 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 2 (j=N/2).

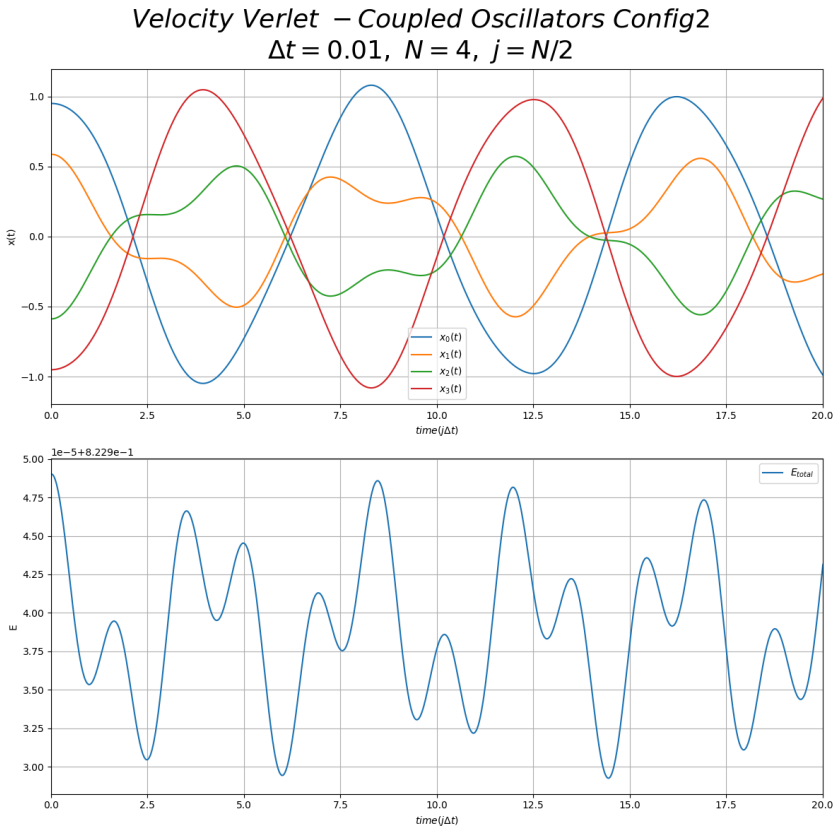

Figure 25: N = 4 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 2 (j=N/2).

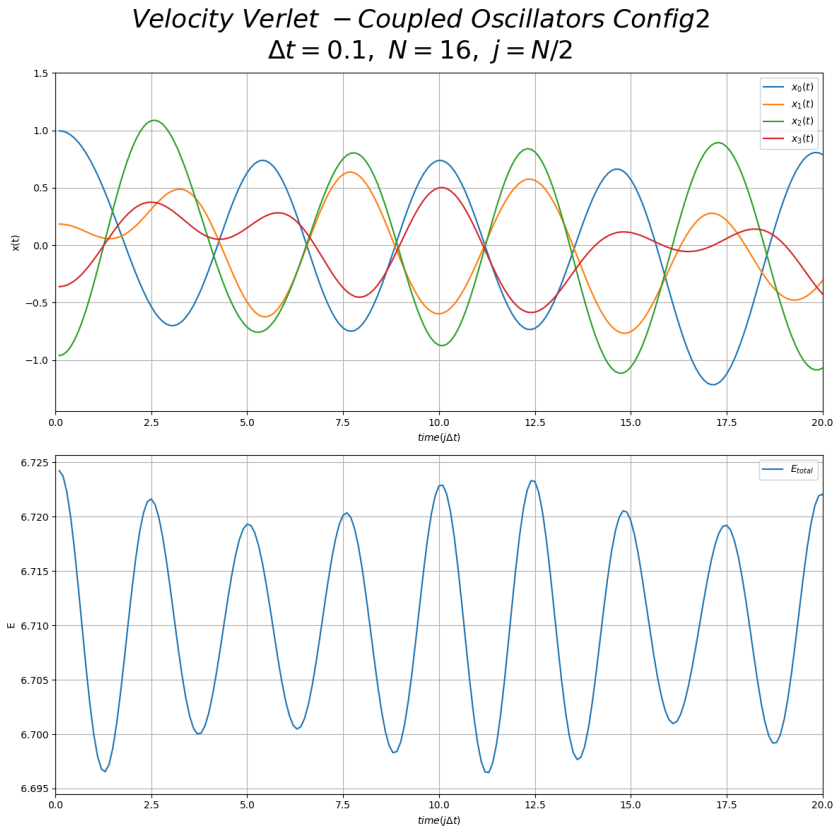

Figure 26: N = 16 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 2 (j=N/2).

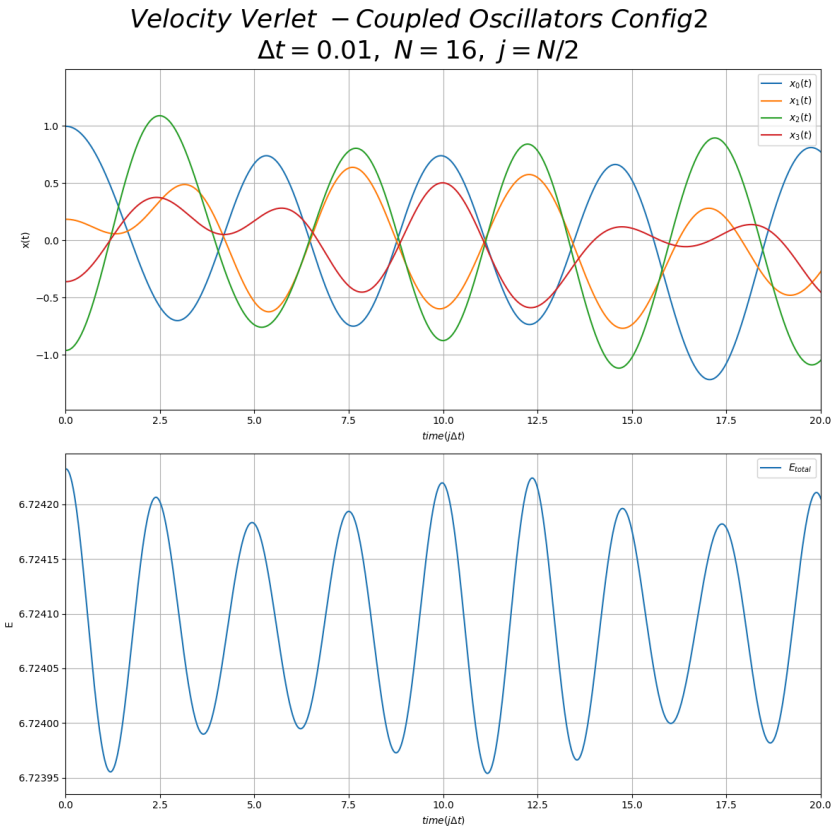

Figure 27: N = 16 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 2 (j=N/2).

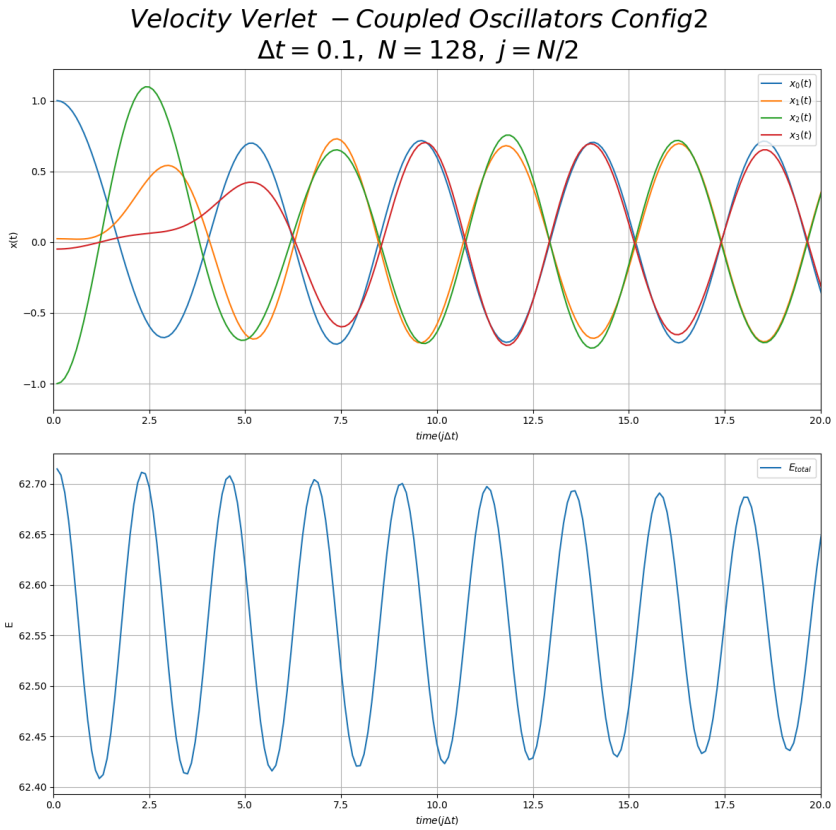

Figure 28: N = 128 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.1$ and configuration 2 (j=N/2).

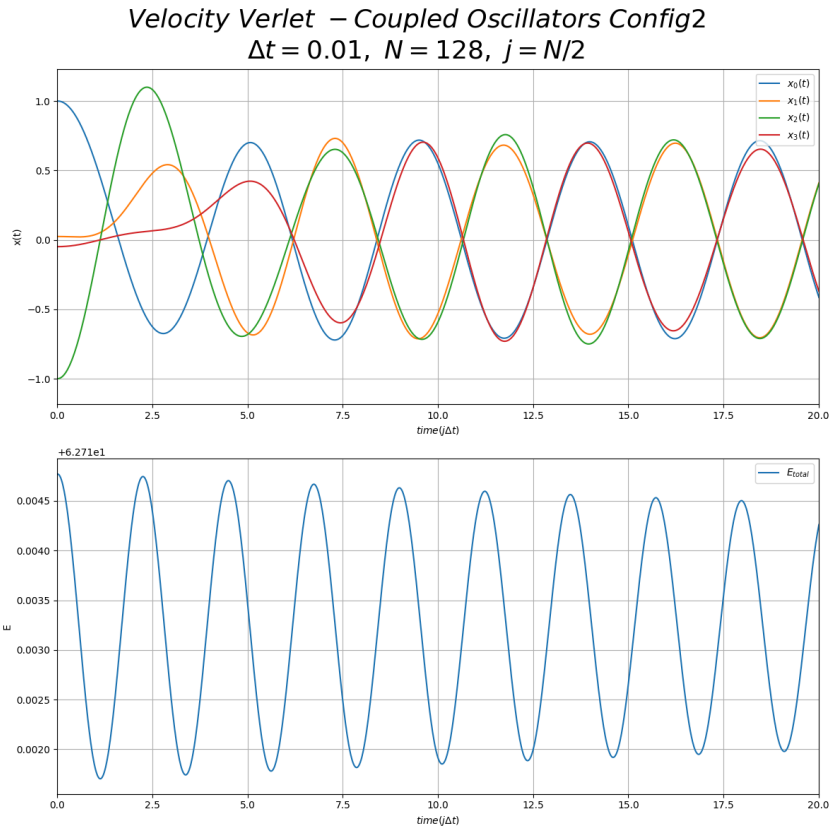

Figure 29: N = 128 coupled harmonic oscillators simulated by the velocity Verlet algorithm with $\Delta t = 0.01$ and configuration 2 (j=N/2).

Discussion

The Euler Method is not a suitable algorithm for numerical integration. As it is unstable, it already diverges very early into the simulation, even for very small time steps. For larger time-steps it diverges so heavily, that special handling for the large numbers was required to plot the results correctly. In contrast to this, the Euler-Cromer algorithm presented the first major improvement as it is a stable and easy algorithm. While we did see the minor errors, that were predicted from theory, in the position from its theoretical value, its energy remained mostly constant over time which also caused it not to gain or lose amplitude over time. While the error is minor, we still see the energy fluctuating because of it, which shows that, while very accurate, it is still not perfect. Neither is the Velocity Verlet Algorithm. While it did improve the accuracy at the same time step size compared to the Euler-Cromer Algorithm as predicted, it still did not conserve energy perfectly. This is

because of the discrete nature of such algorithms. The errors are however so small, that they are acceptable, depending on the application. We could realistically simulate the system of many coupled oscillators, even though we technically have this inaccuracy.

In conclusion, we have seen a few examples of numerical integration algorithms. We noticed that we can get significantly better results and stability at virtually no additional cost, by switching from the Euler to the Velocity Verlet algorithm. By comparing to the analytical results of the harmonic oscillator, we could confirm that the error of numerical integration is small and that numerical integration is a reasonable approach to solving molecular dynamics problems.

Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use('agg')  # Use the 'agg' backend to prevent from memory overflow
```

Euler Method

```
def Euler(step):
    # Determine the number of steps based on the step size
    len = int(10000 / step)

    # Initialize arrays for velocity (v) and position (x)
    v, x = np.ones(len), np.zeros(len)

    # Perform the Euler method integration
    for i in range(len - 1):
        v[i + 1] = v[i] + step * (-x[i])  # Update velocity
        x[i + 1] = x[i] + step * (v[i])   # Update position

    return v, x  # Return the arrays of velocity and position

# Step size for the simulation
step = 0.01

# Calculate velocity (v) and position (x) using the Euler method
v, x = Euler(step)

# Time array scaled by the step size
time = np.arange(1, 10000 / step + 1, 1) * step

# Reduced time array for plotting fewer data points
time_reduced = np.arange(1, 10000 / step + 1, 470) * step

# Create a figure and three subplots
fig, axs = plt.subplots(3, 1, figsize=(12, 18))

# Loop through the subplots to plot the data
for i in range(3):
```

```
    axs[i].plot(time, v, label='v(t)')
    axs[i].plot(time, x, label='x(t)')
   axs[i].plot(time_reduced, np.sin(time_reduced), label='$\sin(j\Delta t)$', linestyle='None',
marker='x')
    axs[i].legend(loc='upper left')  # Add legend to the upper left corner
    axs[i].set_xlabel('$time(j\Delta t)$')  # Label the x-axis with LaTeX syntax
    axs[i].grid()  # Add grid lines to the plot

# Adjust the limits for the second subplot
axs[1].set_xlim(0, 10)  # Set x-axis limits from 0 to 10
axs[1].set_ylim(-15, 15)  # Set y-axis limits from -15 to 15

# Adjust the limits for the third subplot
axs[2].set_xlim(9950 * step, 10000 * step)  # Set x-axis limits to the end of the time array

# Set the title for the entire figure with LaTeX syntax
fig.suptitle('$Euler\ Method$\n$\Delta t={}$'.format(step), fontsize=26, y=0.99)

# Adjust the layout of the subplots to fit in the figure area
fig.tight_layout()

# Save the figure to a file named "Euler_<step>.png"
plt.savefig("Euler_{}.png".format(step))

# Close the figure to free up memory
plt.close(fig)
```

Euler-Comer Method

```
# Define the Euler-Cromer method (variant a)
def Euler_Cromer_a(step):
    # Determine the number of steps based on the step size
    len = int(10000 / step)

    # Initialize arrays for velocity (v) and position (x)
    v, x = np.ones(len), np.zeros(len)

    # Perform the Euler-Cromer integration (variant a)
    for i in range(len - 1):
        v[i + 1] = v[i] + step * (-x[i])  # Update velocity
        x[i + 1] = x[i] + step * (v[i + 1])  # Update position using new velocity

    return v, x  # Return the arrays of velocity and position

# Define the Euler-Cromer method (variant b)
def Euler_Cromer_b(step):
    # Determine the number of steps based on the step size
    len = int(10000 / step)

    # Initialize arrays for velocity (v) and position (x)
    v, x = np.ones(len), np.zeros(len)

    # Perform the Euler-Cromer integration (variant b)
    for i in range(len - 1):
        x[i + 1] = x[i] + step * (v[i])  # Update position
        v[i + 1] = v[i] + step * (-x[i + 1])  # Update velocity using new position

    return v, x  # Return the arrays of velocity and position

# Step size for the simulation
step = 0.001

# Calculate velocity (v) and position (x) using Euler-Cromer method (variant b)
```

```
v, x = Euler_Cromer_b(step)

# Time array scaled by the step size
time = np.arange(1, 10000 / step + 1, 1) * step

# Reduced time array for plotting fewer data points
time_reduced = np.arange(1, 10000 / step + 1, 130) * step

# Create a figure and four subplots
fig, axs = plt.subplots(4, 1, figsize=(12, 26))

# Loop through the first two subplots to plot velocity, position, and sine function
for i in range(2):
    axs[i].plot(time, v, label='v(t)')
    axs[i].plot(time, x, label='x(t)')
    axs[i].plot(time_reduced, np.sin(time_reduced), label='$sin(j\Delta t)$', linestyle='None',
marker='x')

# Loop through the last two subplots to plot energy
for i in range(2, 4):
    axs[i].plot(time, 0.5 * (v**2 + x**2), label='E')
    axs[i].plot(time_reduced, 0.5 * np.ones(np.shape(time_reduced)[0]), label='E=1/2')

# Add legends, grid lines, and x-axis labels to all subplots
for i in range(4):
    axs[i].legend(loc='upper left')
    axs[i].grid()
    axs[i].set_xlabel('$time(j\Delta t)$')

# Adjust the limits for specific subplots
axs[0].set_xlim(0, 50)  # Set x-axis limits from 0 to 50 for the first subplot
# axs[0].set_ylim(-15, 15)  # Optionally set y-axis limits for the first subplot
axs[1].set_xlim(9950 * step, 10000 * step)  # Set x-axis limits to the end of the time array
for the second subplot
axs[2].set_xlim(0, 50)  # Set x-axis limits from 0 to 50 for the third subplot
axs[3].set_xlim(9950 * step, 10000 * step)  # Set x-axis limits to the end of the time array
for the fourth subplot

# Set the title for the entire figure with LaTeX syntax
fig.suptitle('$Euler\\  Cromer\\  Method\\  (b)$\n$\\Delta  t={}$'.format(step),  fontsize=26,
y=0.99)

# Adjust the layout of the subplots to fit in the figure area
fig.tight_layout()

# Save the figure to a file named "Euler_Cromer_b_<step>.png"
plt.savefig("Euler_Cromer_b_{}.png".format(step))

# Close the figure to free up memory
plt.close(fig)
```

Velocity Verlet

```
# Define the Velocity-Verlet method
def Velocity_Verlet(step):
    # Determine the number of steps based on the step size
    len = int(10000 / step)

    # Initialize arrays for velocity (v) and position (x)
    v, x = np.ones(len), np.zeros(len)

    # Perform the Velocity-Verlet integration
    for i in range(len - 1):
```

```
        x[i+1] = x[i] + v[i]*step + 0.5*step*step*(-x[i])
        v[i+1] = v[i] + 0.5*step*(-x[i]-x[i+1])

    # Trim the position array to match the length of the velocity array
    x = x[:-1]

    return v, x  # Return the arrays of velocity and position

# Step size for the simulation
step = 0.01

# Calculate velocity (v) and position (x) using the Velocity-Verlet method
v, x = Velocity_Verlet(step)

# Time array scaled by the step size
time = np.arange(1, 10000 / step + 1, 1) * step

# Reduced time array for plotting fewer data points
time_reduced = np.arange(1, 10000 / step + 1, 10) * step

# Create a figure and four subplots
fig, axs = plt.subplots(4, 1, figsize=(12, 26))

# Loop through the first two subplots to plot velocity, position, and sine function
for i in range(2):
    axs[i].plot(time, v, label='v(t)')
    axs[i].plot(time, x, label='x(t)')
    axs[i].plot(time_reduced, np.sin(time_reduced), label='$sin(j\Delta t)$', linestyle='None',
marker='x')

# Loop through the last two subplots to plot energy
for i in range(2, 4):
    axs[i].plot(time, 0.5 * (v**2 + x**2), label='E')
    axs[i].plot(time_reduced, 0.5 * np.ones(np.shape(time_reduced)[0]), label='E=1/2')

# Add legends, grid lines, and x-axis labels to all subplots
for i in range(4):
    axs[i].legend(loc='upper left')
    axs[i].grid()
    axs[i].set_xlabel('$time(j\Delta t)$')

# Adjust the limits for specific subplots
axs[0].set_xlim(0, 50)  # Set x-axis limits from 0 to 50 for the first subplot
axs[1].set_xlim(9950 * step, 10000 * step)  # Set x-axis limits to the end of the time array
for the second subplot
axs[2].set_xlim(0, 50)  # Set x-axis limits from 0 to 50 for the third subplot
# axs[2].set_ylim(0.3, 0.6)  # Optionally set y-axis limits for the third subplot
axs[3].set_xlim(9950 * step, 10000 * step)  # Set x-axis limits to the end of the time array
for the fourth subplot
# axs[3].set_ylim(0.3, 0.6)  # Optionally set y-axis limits for the fourth subplot

# Set the title for the entire figure with LaTeX syntax
fig.suptitle('$Velocity\\ Verlet$\n$\\Delta t={}$'.format(step), fontsize=26, y=0.99)

# Adjust the layout of the subplots to fit in the figure area
fig.tight_layout()

# Save the figure to a file named "Velocity_Verlet_<step>.png"
plt.savefig("Velocity_Verlet_{}.png".format(step))

# Close the figure to free up memory
plt.close(fig)
```

Coupled Oscillators

```
# Define the Velocity-Verlet method for coupled oscillators
def Velocity_Verlet_CP(step, N, config=1):
    # Determine the number of steps based on the step size
    len = int(1000 / step)

    # Initialize velocity (v) and position (x) arrays based on the selected initial condition
(config)
    if config == 1:
        # First initial condition: single initial displacement in the middle
        v, x = np.zeros((N, len)), np.zeros((N, len))
        x[int(N / 2)][0] = 1
    elif config == 2:
        # Second initial condition: sinusoidal initial displacement
        v, x = np.zeros((N, len)), np.zeros((N, len))
        for j in range(N):
            x[j][0] = np.sin(np.pi * 1 * (j + 1) / (N + 1))
    elif config == 3:
        # Third initial condition: different sinusoidal initial displacement
        v, x = np.zeros((N, len)), np.zeros((N, len))
        for j in range(N):
            x[j][0] = np.sin(np.pi * (N / 2) * (j + 1) / (N + 1))

    # Perform the Velocity-Verlet integration
    for i in range(len-1):

      for j in range(N):

        if j != 0 and j != N-1:

          x[j][i+1] = x[j][i] + v[j][i]*step + 0.5*step*step*-( 2*x[j][i] - x[j-1][i] - x[j+1]
[i] )

        elif j == 0:

          x[j][i+1] = x[j][i] + v[j][i]*step + 0.5*step*step*-( x[j][i] - x[j+1][i] )

        else:

          x[j][i+1] = x[j][i] + v[j][i]*step + 0.5*step*step*-( x[j][i] - x[j-1][i] )

      for j in range(N):

        if j != 0 and j != N-1:

          v[j][i+1] = v[j][i] + 0.5*step*-( 2*x[j][i]-x[j-1][i]-x[j+1][i] + 2*x[j][i+1]-x[j-1]
[i+1]-x[j+1][i+1] )

        elif j == 0:

          v[j][i+1] = v[j][i] + 0.5*step*-( x[j][i]-x[j+1][i] + x[j][i+1]-x[j+1][i+1] )

        else:

          v[j][i+1] = v[j][i] + 0.5*step*-( x[j][i]-x[j-1][i] + x[j][i+1]-x[j-1][i+1] )


    return v, x  # Return the arrays of velocity and position

# Simulation parameters
step = 0.1
```

```
N = 16

# Calculate velocity (v) and position (x) using the Velocity-Verlet method with coupled
oscillators
v, x = Velocity_Verlet_CP(step, N, config=3)

# Time array scaled by the step size
time = np.arange(1, 1000 / step + 1, 1) * step

# Calculate the total energy
E_tot = np.zeros(np.shape(time)[0])
for i in range(N - 1):
    E_tot += 0.5 * (v[i, :]**2 + (x[i, :] - x[i + 1, :])**2)
E_tot += 0.5 * v[N - 1, :]**2

# Create a figure and two subplots
fig, axs = plt.subplots(2, 1, figsize=(12, 12))

# Plot position (x) for the first four oscillators
for i in range(4):
    axs[0].plot(time, x[i, :], label='$x_{}(t)$'.format(i))

# Plot total energy
axs[1].plot(time, E_tot, label='$E_{total}$')

# Add legends, grid lines, and axis labels to the subplots
for i in range(2):
    axs[i].legend()
    axs[i].grid()
    axs[i].set_xlabel('$time(j\Delta t)$')
    axs[i].set_xlim(0, 20)

# Add y-axis labels
axs[0].set_ylabel('x(t)')
axs[1].set_ylabel('E')

# Set the title for the entire figure with LaTeX syntax
fig.suptitle('$Velocity\\ Verlet\\ -Coupled\\ Oscillators\\ Config\\ 3$\n$\\Delta t={},\\ N={},
\\ j=N/2$'.format(step, N), fontsize=26, y=0.98)

# Adjust the layout of the subplots to fit in the figure area
fig.tight_layout()

# Save the figure to a file named "CP3_N<step>.png"
plt.savefig("CP3_N{}_{}.png".format(N, step))

# Close the figure to free up memory
plt.close(fig)
```