Report 5: Numerical integration in molecular dynamics simulations

Don Winter[1] **and Madhulan Suresh**[2]

[1]431380, don.winter@rwth-aachen.de
[2]429481, madhulan.suresh@rwth-aachen.de

Introduction

In this exercise, we simulate the motion of harmonic oscillators using different intergration algorithms. We implemented the Euler, Euler-Cromer and Velocity Verlet algorithms and record the position, velocity and energy for each time step which we subsequently compare to the analytical solution. Finally, we simulate the motion of coupled oscillators via Velocity Verlet. Our task in this exercise is to compare the algorithms and argue why the Velocity Verlet algorithm is superior.

Simulation Model and Method

We use three different integration algorithms to simulate the position, velocity and energy of a harmonic osciallator of fixed spring konstant k at different times. Afterwards, we compare the results of each algorithm to the theoretical results.

By classical mechanics we know that objects move due to forces exerted on them via Newton's equation and that forces are derived from mechanical potentials. As we are only considering movement in one dimension the general equation of motion becomes:

$$F = ma = -\frac{\partial V}{\partial x}, \tag{1}$$

where F is the force acting on a particle of mass m inducing it to come into motion with acceleration a as consequence of the (negative) change in potential at the particle's current location x. In this exercise we assume $V = V(x) = \frac{1}{2}kx^2$ to be a harmonic potential, which is the only source influencing our particle's motion. Thus the particle moves according to:

$$F = -\frac{dV}{dx} = -kx, \tag{2}$$

where the partial derivative became a regular derivative as our potential is time-independent. Only for the last part, where we model a (monoatomic) chain of coupled harmonic oscillators we have to consider the forces acting on each particle individually at each time step. In this case the equations get a subscript to identify the corresponding particle. For our simulations we consistently set $m = k = 1$ for simplicity.

Algorithm 1: Euler

The Euler algorithm is the simplest of the algorithms we discuss here. It comes directly from the (first-order accurate) forward difference equations of $a(t)$ and $v(t)$:

$$a(t) \approx \frac{v(t+\Delta t) - v(t)}{\Delta t} \implies v(t+\Delta t) = v(t) + a(t)\Delta t + \mathcal{O}((\Delta t)^2) \tag{3}$$

$$v(t) \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \implies x(t+\Delta t) = x(t) + v(t)\Delta t + \mathcal{O}((\Delta t)^2), \tag{4}$$

which define the update equations for the Euler algorithm. We simulate for 10000 time steps of length $\Delta t = 0.1, 0.01, 0.001$. The initial conditions of position and velocity are $x(0) = 0$ and $v(0) = 1$. Iterating over time, we record position and velocity by means of Eqs. 3 and 4. Then, we plot the position as function of time along with its analytical solution $x(j\Delta t) = \sin(j\Delta t)$ for comparison, which we get from solving the differential equation

$$a(t) = \frac{d^2x(t)}{dt^2} = -x(t) \implies x(t) = \sin(t), \tag{5}$$

where we substitute $t \to j\Delta t$ for discretized time.

Algorithm 2: Euler-Cromer

The Euler-Cromer algorithm is a simple variation of the Euler algorithm which produces, opposed to the regular Euler method, stable solutions. We use the same forward finite difference equations as in the Euler scheme, but change the time steps in the update sequence. Thus, we inform our update of one of the variables $x(t)$ or $v(t)$ by the previously calculated current time step solution of the respective other variable. We can do this for either variable, which yields two different (but equivalent) update rules

$$\begin{aligned} v(t+\Delta t) &= v(t) + a(t)\Delta t + \mathcal{O}((\Delta t)^2) \\ x(t+\Delta t) &= x(t) + v(t+\Delta t)\Delta t + \mathcal{O}((\Delta t)^2), \end{aligned} \tag{6}$$

and

$$\begin{aligned} x(t+\Delta t) &= x(t) + v(t)\Delta t + \mathcal{O}((\Delta t)^2) \\ v(t+\Delta t) &= v(t) + a(t+\Delta t)\Delta t + \mathcal{O}((\Delta t)^2), \end{aligned} \tag{7}$$

respectively. Furthermore, we calculate the total energy of our oscillator system at each time step by

$$E(t) = K.E. + P.E. = \frac{1}{2}v^2(t) + \frac{1}{2}x^2(t), \tag{8}$$

again as $m = k = 1$. We use the same initial conditions as we used for Euler algorithm. For both the Euler and the Euler-Cromer scheme we have the same local first-order accuracy, i.e. errors in position and velocity scale with Δt^2, while the global error scales with Δt.

Algorithm 3: Velocity Verlet

A much better integration algorithm is the second-order accurate Velocity Verlet algorithm. It is stable as the Euler-Cromer algorithm while also predicting correct behavior of the system energy. Instead of forward differences, the Velocity Verlet algorithm makes use of central differences, thus calculates velocity at half time steps in order to inform both (full time step) position and velocity updates. Thus, our update rules become:

$$\begin{aligned} x(t+\Delta t) &= x(t) + v(t+\frac{\Delta t}{2})\Delta t + \mathcal{O}((\Delta t)^3) \\ v(t+\Delta t) &= v(t+\frac{\Delta t}{2}) + \frac{1}{2}a(t+\Delta t)\Delta t + \mathcal{O}((\Delta t)^3), \end{aligned} \tag{9}$$

where we make use of the velocity calculated at half time steps which we always calculate on the fly and not keep track of:

$$v(t+\frac{\Delta t}{2}) = v(t) + \frac{1}{2}a(t)\Delta t. \tag{10}$$

Furthermore, for the velocity update, we calculate the force anew after updating the position yielding our new intermediate acceleration $a(t+\Delta t)$. For the simulation we use the same initial conditions as above and time steps $\Delta t = 0.1, 0.01$. We plot $x(t), v(t)$ and total energy $E(t)$ along with the analytical results (from above) for comparison.

Algorithm 4: N-particle Velocity Verlet

In this task we use the Velocity Verlet algorithm to simulate the motion of a n-particle monoatomic chain where each particle has identical mass and is connected to its nearest neighbors by the same spring constant, i.e. $m_i = k_i = 1$. The system can be completely described by the interactions between the particles:

$$\frac{\partial V}{\partial x_k} = \begin{cases} x_1 - x_2 & k = 1 \\ 2x_k - x_{k-1} - x_{k+1} & 1 < k < N \\ x_N - x_{N-1} & k = N. \end{cases} \tag{11}$$

For this exercise there were two sets of initial conditions given. First, the velocity and position are initialized to 0 for all oscillators except at position $N/2$, where we have $x_{N/2} = 1$. The second set defines the velocity to initially be zero but for the positions we set $x_k(0) = \sin\frac{\pi jk}{N+1}$ for $k = 1, ..., N$ and $j = 1, N/2$. We run the simulation for $N = 4, 16$ and 128 oscillators in the chain with time step $\Delta t = 0.1, 0.01$. We plot the position as function of time, the velocity and total energy for the systems of N oscillators.

Simulation results

In the following, we present our simulation results, compare them to the above mentioned analytical solutions and argue what are the pros and cons of each method.

Algorithm 1: Euler

We can clearly see from the Fig. 1 that the oscillations are exponentially increasing with each time step in comparison to the analytical solution. Thus, the algorithm is not stable in time, which renders this algorithm not useful for this problem.

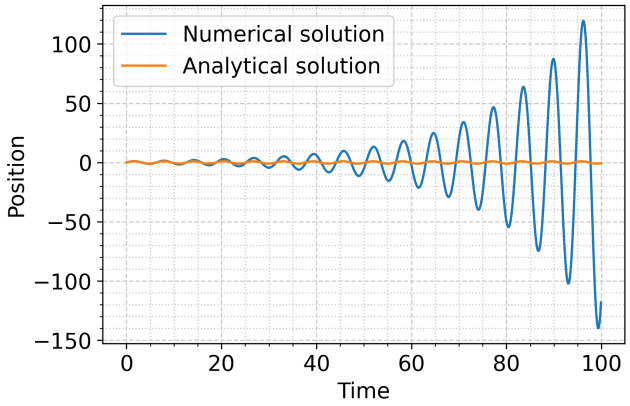
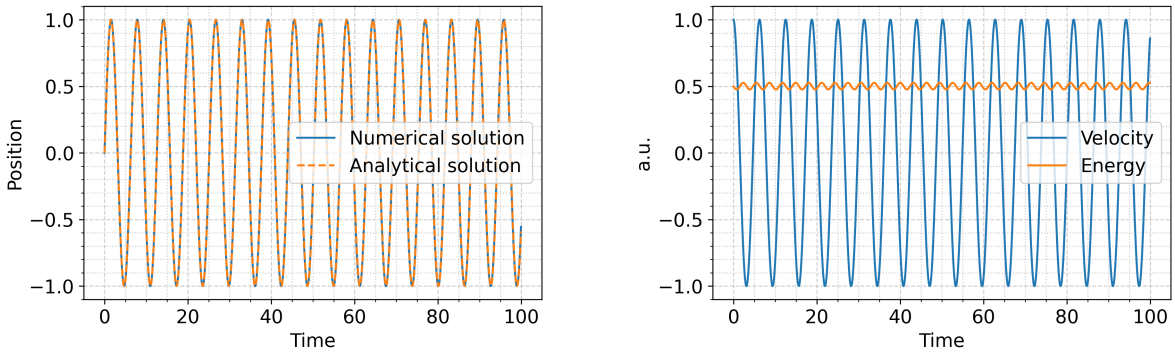


Figure 1. Position as function of time for the Euler algorithm. In blue we show the numerical simulation results while the analytical solution are shown in orange.

Algorithm 2: Euler-Cromer

In Fig. 2 we can see that the position and velocity solutions of the Euler-Cromer algorithm are stable compared to Euler algorithm. However, the system energy is fluctuating. This is an unphysical result, as we have a closed system in which we would expect the energy to be constant. Therefore, although in this simple simulation the numerical values for the position match the analytical results well, the Euler-Cromer algorithm might not be useful for more complex problems. This is the case for both update schemes in Eqs. 6 and 7, which yield the same results.



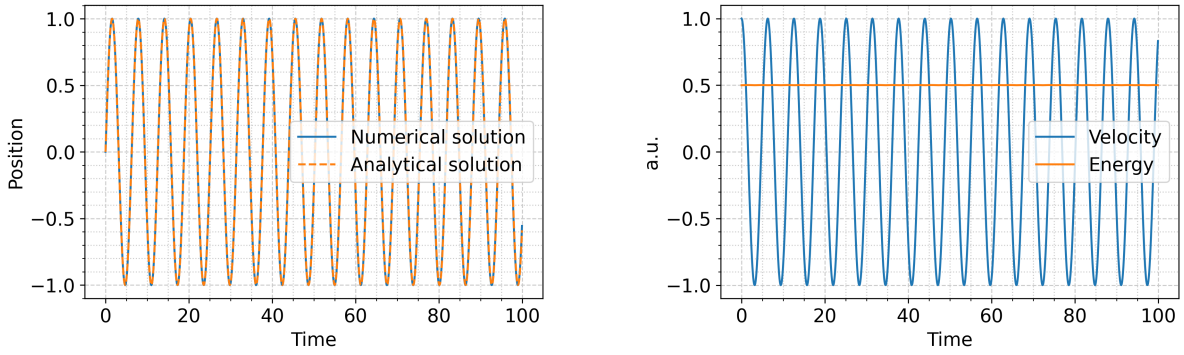
(a) Position as function of time

(b) Velocity and energy as functions of time

Figure 2. Position as function of time for the Euler-Cromer algorithm together with system energy and velocity as functions of time.

Algorithm 3: Velocity Verlet

The Velocity Verlet method also yields stable results with constant energy. Although we cannot see it from our simulation results the time-reversibility and the higher accuracy make the Velocity Verlet method the superior integration algorithm. Our simulation results can be seen in Fig. 3.



(a) Position as function of time

(b) Velocity and energy as functions of time

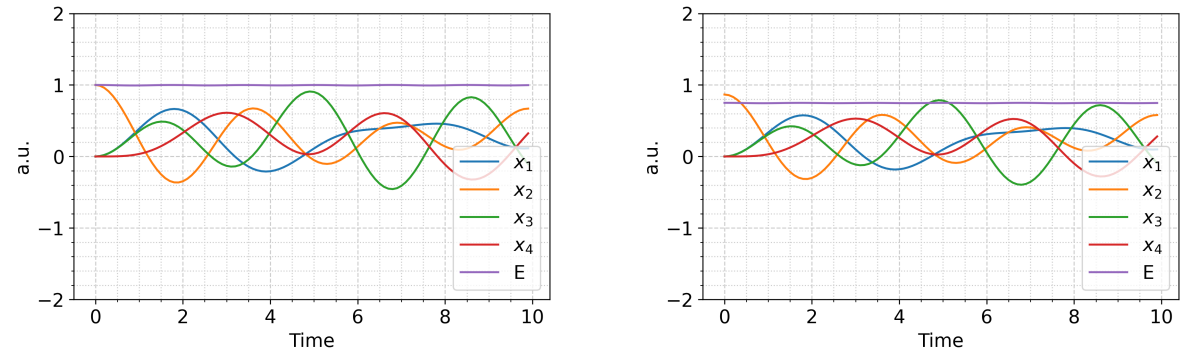
Figure 3. Position as function of time for the Velocity Verlet algorithm together with system energy and velocity as functions of time.

Algorithm 4: N-body Velocity Verlet

Finally, we simulated a monoatomic chain of $N = 4, 16$ and 128 oscillators for two different initial conditions:

1. $v_k(0) = 0$ and $x_k(0) = 0$ except $x_{N/2}(0) = 1$.
2. $v_k(0) = 0$ and $x_k(0) = \sin(\frac{\pi jk}{N+1})$ for $k = 1, ..., N$ and $j = N/2$

In Figs. 4–6 we show our results. As we can see, the energy of the entire system of oscillators keeps a resonably constant value. This makes the Velocity Verlet method a good method for also integrating complex systems as our coupled oscillators. It is also noteworthy that the computational cost at which we achieve this accuracy and stability are very small compared to the benefits.



(a) Initial condition 1

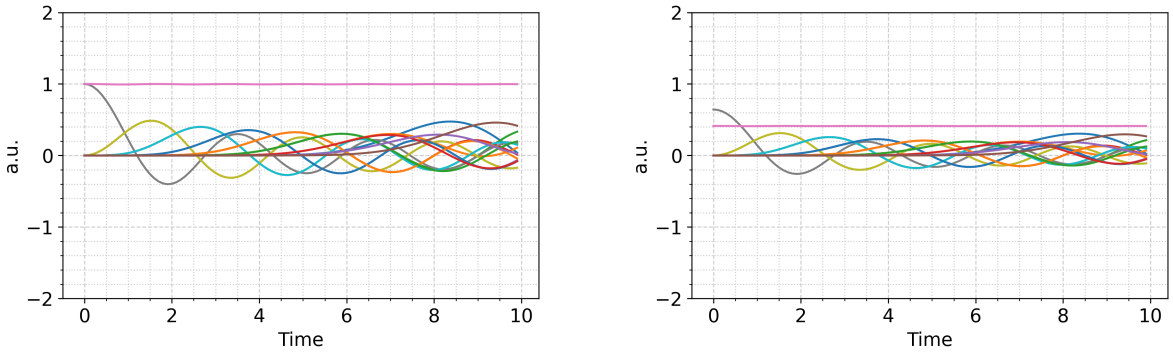
(b) Initial condition 2

Figure 4. Oscillator positions as function of time for $N = 4$ oscillators and $\Delta t = 0.1$.

We can clearly see that for inital condition 2 with increasing number of oscillators the oscillations of the whole system are dampened and the system energy vanishes as the individual oscillator velocities decrease. This can be explained as the effect similar to destructive interference, i.e. the system energy cannot increase but it decreases by oscillators *working against each other* and by offsetting the inital positions of the oscillators by different amounts in different directions we more and more achieve this situation. For initial condition 1 on the other hand we offset only one oscillator from which the chain is "activated", thus it can freely find a resonant mode in which the whole chain oscillates harmonically.

Discussion

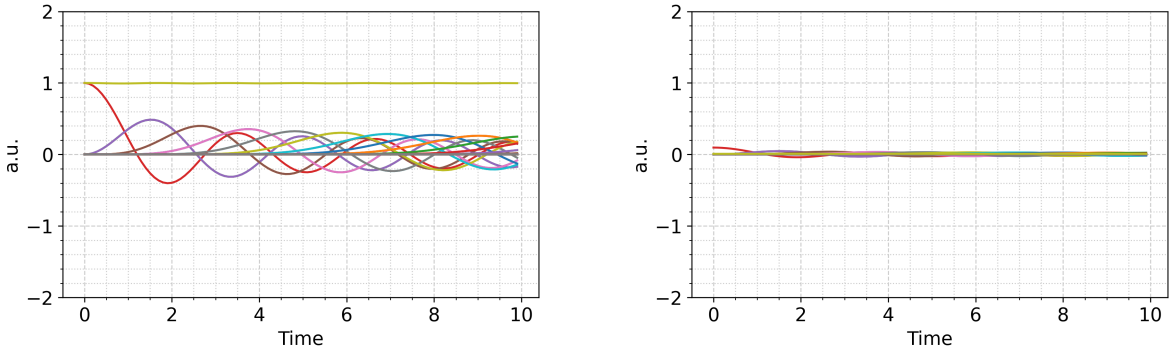
The Euler algorithm is not stable and the total energy is not conserved hence it is not a good method to solve complicated integrals numerically. The Euler-Cromer algorithm on the other hand is stable and takes the same computational cost as the Euler algorithm, thus it is the superior of the two. However, the fluctuating system energy introduce an unphysical side effect



(a) Initial condition 1

(b) Initial condition 2

Figure 5. Oscillator positions as function of time for $N = 16$ oscillators and $\Delta t = 0.1$. We refrained from labelling each oscillator for the sake of clarity.



(a) Initial condition 1

(b) Initial condition 2

Figure 6. Oscillator positions as function of time for $N = 128$ oscillators and $\Delta t = 0.1$. We refrained from labelling each oscillator for the sake of clarity.

which is especially undesirable for simulations of physical systems, as energy is on of the key property one would like to track and make statements about a system. The Velocity Verlet algorithm is more accurate, unconditionally stable and yields expected energies even for complex systems of many interacting particles. It also gives these benefits at very low computational extra cost. Therefore we should in any situation always prefer the Velocity Verlet algorithm to the others.

Appendix

Common libraries and functions

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
font = {'size': 14}
matplotlib.rc('font', **font)

def plot(x, ys, lbls=None, markers=None, xlabel='Time', ylabel='a.u.', path=None, ylim=None):
    plt.figure(figsize=(6,4))

    for i, y in enumerate(ys):
        if lbls: lbl = lbls[i]
        else: lbl = None
        if markers: plt.plot(x,y,markers[i],label=lbl)
        else: plt.plot(x,y,label=lbl)
```

```
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    plt.minorticks_on()
    plt.grid(which='major', color='#CCCCCC', linestyle='--')
    plt.grid(which='minor', color='#CCCCCC', linestyle=':')
    if ylim: plt.ylim(ylim)
    plt.tight_layout()
    if lbls: plt.legend()

    if path: plt.savefig(path, dpi=300)
    else: plt.show()
```

Euler algorithm

```
from common import *

dt = 0.1  # time spacing
N = 1000  # no. of space steps
x = np.zeros((N)) # positions
v = np.zeros((N)) # velocities
t = np.arange(0,N*dt,dt) # time

# initial conditions
x[0] = 0
v[0] = 1

for i in range(N-1):
    x[i+1] = x[i] + v[i]*dt # update position
    v[i+1] = v[i] - x[i]*dt # update velocity (f = -kx = -dVdx)

plot(t,[x,np.sin(t)],['Numerical solution', 'Analytical solution'],
        ylabel='Position', path='./1.png')
```

Euler-Cromer algorithm

```
from common import *

dt = 0.1 # time step
N = 1000 # no. of space steps
x = np.zeros((N)) # positions
v = np.zeros((N)) # velocities
t = np.arange(0,N*dt,dt) # time

x[0] = 0
v[0] = 1

# Variant 1:
# for i in range(N-1):
#     v[i+1] = v[i] - x[i]*dt # update velocity first
#     x[i+1] = x[i] + v[i+1]*dt # update position afterwards

# Variant 2:
for i in range(N-1):
    x[i+1] = x[i] + v[i]*dt # update position first
    v[i+1] = v[i] - x[i+1]*dt # update velocity afterwards

E = v**2/2 + x**2/2
x_ana = np.sin(t)
plot(t,[x,x_ana],lbls=['Numerical solution','Analytical solution'], markers=['-','--'], path='./2a.png',
     ylabel='Position')
```

Velocity Verlet algorithm

```
from common import *

dt = 0.1 # time spacing
N = 1000 # no. of space steps
x = np.zeros(N) # positions
v = np.zeros(N) # velocities
```

```
t = np.arange(0,N*dt, dt) # time

x[0] = 0
v[0] = 1

for i in range(N-1):
    a = -x[i]           # force
    v_n = v[i] + a*dt/2.0   # velocity after dt/2 (half kick)
    x[i+1] = x[i] + v_n*dt
    a = -x[i+1]         # force at new position
    v[i+1] = v_n + a*dt/2.0 # velocity from t+dt/2 (remaining half kick)

E = v**2/2 + x**2/2
x_ana = np.sin(t)
plot(t,[v,E],lbls=['Velocity','Energy'],
        markers=['-','-'], path='./4.png')
```

N-body Velocity Verlet algorithm

```
from common import *

N = 100
n = 128 # no. of oscillators
dt = 0.1
x = np.zeros((N,n))
v = np.zeros((N,n))
t = np.arange(0,N*dt, dt)

# x[0,(n-1)//2] = 1 #initial condition 1

# inital condition 2
j = (n-1) // 2 # and (n-1)//2 or 1
for k in range(n):
    x[0,j] = np.sin(np.pi*(k+1)*j / (n+2))

def force(x):  # Defining the force between oscillators
    res = np.empty_like(x)
    res[1:-1] = -2*x[1:-1] + x[:-2] + x[2:]
    res[0] = -x[0] + x[1]
    res[-1] = -x[-1] + x[-2]
    return res

for i in range(N-1):  # Velocity Verlet algorithm
    v_n = v[i,:] + force(x[i,:]) * dt / 2.0
    x[i+1,:] = x[i,:] + v_n * dt
    v[i+1,:] = v_n + force(x[i+1,:]) * dt / 2.0

E = 0.5 * np.sum(v**2, axis=1) + 0.5 * np.sum( (x[:,:-1] - x[:,1:])**2 , axis=1) # total energy

plot(t, [*(x.T), E], ylim=[-2,2], path='./10.png')
# plot(t, [*(x.T[:4]), E], lbls=[r'$x_1$', r'$x_2$', r'$x_3$', r'$x_4$', 'E'], ylim=[-2,2], path='./10.
    png')
```