# Step-by-Step Development of a Molecular Dynamics Simulation

Frank Cichos

## Molecular Dynamics Simulations

Real molecular dynamics (MD) simulations are complex and computationally expensive but very cool, as they give you a glimpse into the world of atoms and molecules. Here, we will develop a simple MD simulation from scratch in Python. The goal is to understand the basic concepts and algorithms behind MD simulations and get something running which can be extended later but also what we are proud of at the end of the course.

Before we can start with implementing a simulation, we need to understand the basic concepts and algorithms behind MD simulations. The following sections will guide you through the development of a simple MD simulation.

## Basic Physical Concepts

### Newton's Equations of Motion

The motion of particles in a molecular dynamics simulation is governed by Newton's equations of motion:

$$m_i \frac{d^2\vec{r}_i}{dt^2} = \vec{F}_i$$

where:

- $m_i$ is the mass of particle $i$
- $\vec{r}_i$ is the position of particle $i$
- $\vec{F}_i$ is the force acting on particle $i$

The force acting on a particle is the sum of all forces acting on it:

$$\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij}$$

where $\vec{F}_{ij}$ is the force acting on particle $i$ due to particle $j$.

## Potential Energy Functions and Forces

The force $\vec{F}_{ij}$ is usually derived from a potential energy function and may result from a variety of interactions, such as:

- Bonded interactions
  - bond stretching
  - bond angle bending
  - torsional interactions
- Non-bonded interactions
  - electrostatic interactions
  - van der Waals interactions
- External forces

We will implement some of them but not all of them.

## Lennard-Jones Potential

The most common potential energy function used in MD simulations is the Lennard-Jones potential. It is belonging to the class of non-bonded interactions. The force and the potential energy of the Lennard-Jones potential are given by:

$$V_{LJ}(r) = 4\epsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

and

$$F_{LJ}(r) = -\frac{dV_{LJ}}{dr} = 24\epsilon \left[ 2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \frac{\vec{r}}{r^2}$$

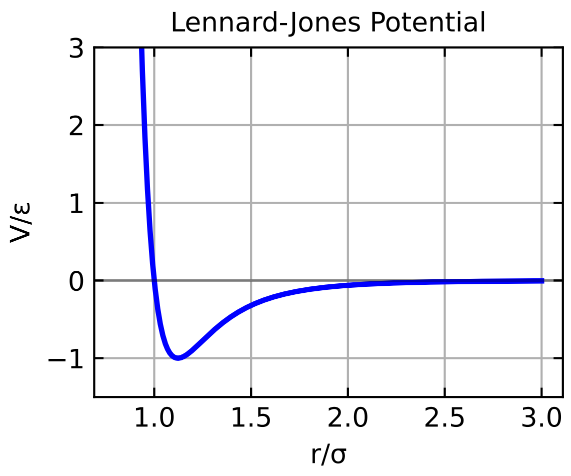
where:

- $\epsilon$ is the depth of the potential well

- $\sigma$ is the distance at which the potential is zero
- $r$ is the distance between particles

The Lenard Jones potential is good for describing the interaction of non-bonded atoms in a molecular system e.g. in a gas or a liquid and is therefore well suited if we first want to simulate a gas or a liquid.

```python
def lennard_jones(r, epsilon=1, sigma=1):
    return 4 * epsilon * ((sigma/r)**12 - (sigma/r)**6)

r = np.linspace(0.8, 3, 1000)
V = lennard_jones(r)

plt.figure(figsize=get_size(8, 6),dpi=150)
plt.plot(r, V, 'b-', linewidth=2)
plt.grid(True)
plt.xlabel('r/ ')
plt.ylabel('V/ ')
plt.title('Lennard-Jones Potential')
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.ylim(-1.5, 3)
plt.show()
```



The figure above shows the Lenard-Jones potential as a function of the distance between particles. The potential energy is zero at the equilibrium distance $r = \sigma$ and has a minimum at $r = 2^{1/6}\sigma$. The potential energy is positive for $r < \sigma$ and negative for $r > \sigma$.

Values for atomic hydrogen

For atomic hydrogen (H), typical Lennard-Jones parameters are:

- $\sigma \approx 2.38$ Å $= 2.38 \times 10^{-10}$ meters
- $\epsilon \approx 0.0167$ kcal/mol $= 1.16 \times 10^{-21}$ joules

Later, if we manage to advance to some more complicated systems, we may want to introduce:

1. force in bonds between two atoms
2. force in bond angles between three atoms
3. force in dihedral angles between four atoms

But for now, we will stick to the Lennard-Jones potential.

## Integrating Newtons Euqation of Motion

When we have the forces on a particle we have in principle its acceleration. To get the velocity and the position of the particle we need to integrate the equations of motion. There are several methods to do this, but we will start with the simplest one, the Euler method.

### Euler Method

To obtain this one first needs to know about the Taylor expansion of a function in general. The Taylor expansion of a function $f(x)$ around a point $x_0$ is providing an approximation of the function in the vicinity of $x_0$. It is given by:

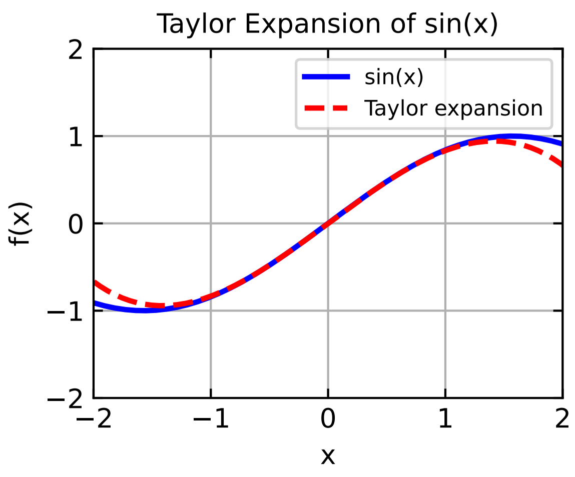
$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \cdots$$

where $f'(x_0)$ is the first derivative of $f(x)$ at $x_0$, $f''(x_0)$ is the second derivative of $f(x)$ at $x_0$, and so on. We can demonstrate that by expanding a sine function around $x_0 = 0$:

$$\sin(x) = \sin(0) + \cos(0)x - \frac{1}{2}\sin(0)x^2 + \cdots = x - \frac{1}{6}x^3 + \cdots$$

Plotting this yields:

```
x = np.linspace(-2*np.pi, 2*np.pi, 1000)
y = np.sin(x)
y_taylor = x - 1/6*x**3

plt.figure(figsize=get_size(8, 6),dpi=150)
plt.plot(x, y, 'b-', label='sin(x)', linewidth=2)
plt.plot(x, y_taylor, 'r--', label='Taylor expansion', linewidth=2)
plt.grid(True)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.xlim(-2,2)
plt.ylim(-2,2)
plt.title('Taylor Expansion of sin(x)')
plt.legend()
plt.show()
```



The expansion is therefore a good approximation in a region close to $x_0$.

## Velocity Verlet Algorithm

The velocity Verlet algorithm is a second-order algorithm that is more accurate than the Euler method. It can be derived from the Taylor expansion of the position and velocity vectors

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\frac{\mathbf{F}(t)}{m}\Delta t^2 + O(\Delta t^3)$$

The higher order terms in the Taylor expansion are neglected, which results in an error of order $\Delta t^3$. As compared to that the Euler method is obtained by neglecting the higher order terms in the Taylor expansion of the velocity vector:

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{\mathbf{F}(t)}{m}\Delta t + O(\Delta t^2)$$

and is therefore only first order accurate with an error of order $\Delta t^2$.

The velocity Verlet algorithm consists of three steps:

1. Update positions: $\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\frac{\mathbf{F}(t)}{m}\Delta t^2$
2. Calculate new forces: $\mathbf{F}(t+\Delta t) = \mathbf{F}(\mathbf{r}(t+\Delta t))$
3. Update velocities: $\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{1}{2}\frac{\mathbf{F}(t)+\mathbf{F}(t+\Delta t)}{m}\Delta t$

where: - $\mathbf{r}$ is the position vector - $\mathbf{v}$ is the velocity vector - $\mathbf{F}$ is the force vector - $m$ is the mass - $\Delta t$ is the timestep

## Simple Integration Example: Free Fall

Let's start and try to integrate the equation of motion for a particle in free fall with the help of the Velocity Verlet algorithm. The only force acting on the particle is gravity. The equation of motion is:

Newton's equation of motion: $\mathbf{F} = m\mathbf{a}$

For gravity: $\mathbf{F} = -mg\hat{\mathbf{y}}$

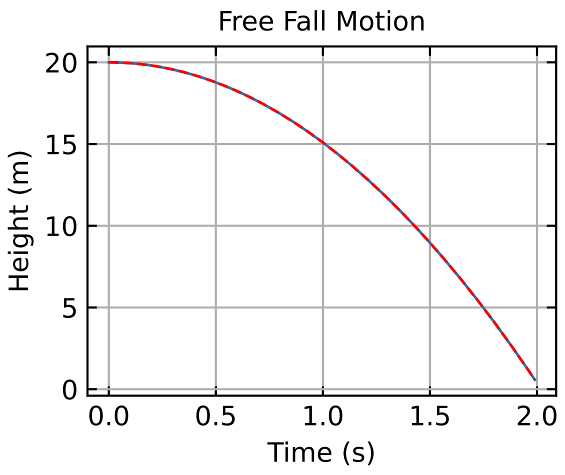
Therefore: $\ddot{y} = -g$

The analytical solution is:

- Position: $y(t) = y_0 + v_0 t - \frac{1}{2}gt^2$
- Velocity: $v(t) = v_0 - gt$

```
# Parameters

g = 9.81  # m/s^2
dt = 0.01  # time step
t_max = 2.0  # total simulation time
steps = int(t_max/dt)

# Initial conditions
y0 = 20.0  # initial height
```

```
v0 = 0.0   # initial velocity


# Arrays to store results
t = np.zeros(steps)
y = np.zeros(steps)
v = np.zeros(steps)
a = np.zeros(steps)

# Initial values
y[0] = y0
v[0] = v0
a[0] = -g

# Velocity Verlet integration
for i in range(1, steps):
    t[i] = i * dt
    y[i] = y[i-1] + v[i-1] * dt + 0.5 * a[i-1] * dt**2  # update position
    a_new = -g                                          # new acceleration (assuming constan
    v[i] = v[i-1] + 0.5 * (a[i-1] + a_new) * dt         # update velocity
    a[i] = a_new                                        # store new acceleration

y_analytical = y0 + v0*t - 0.5*g*t**2
plt.figure(figsize=get_size(8, 6), dpi=150)
plt.plot(t, y)
plt.plot(t, y_analytical, 'r--')

plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.title('Free Fall Motion')
plt.grid(True)
plt.show()
```



### Boundary Conditions

In the previous example, we have assumed that the particle is in free fall. That means eventually it would bounce against the floor. In a real simulation, we need to consider boundary conditions as well. For example, if the particle hits the ground we could implement a simple reflection rule. This is called *reflecting boundary conditions* and would introduce some additional effects to the simulation. On the other side, one could make the system "kind of" infinitely large by introducing periodic boundary conditions. This means that if a particle leaves the simulation box on one side, it re-enters on the opposite side. This is a common approach in molecular dynamics simulations.

## Implementation

The question we have to think about now is how to implement these formulas in a numerical simulation. The goal is to simulate the motion of many atoms in a box. Each atom is different and has its own position, velocity, and force. Consequently we need to store these quantities for each atom, though the structure in which we store them is the same for each atom. All atoms with their properties actually belong to the same class of objects. We can therefore use a very suitable concept of object-oriented programming, the class.

### Classes in Object-Oriented Programming

A class in object-oriented programming is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). The class is a template for objects, and an object is an instance of a class. The class defines the properties and behavior common to all objects of the class. The objects are the instances of the class that contain the actual data. :::

We define a class `Atom` that contains the properties of an atom. The class `Atom` has the following attributes:

```python
class Atom:
    def __init__(self, atom_id, atom_type, position, velocity=None, mass=None):
        self.id = atom_id
        self.type = atom_type
        self.position = position
        self.velocity = velocity if velocity is not None else np.zeros(3)
        self.mass = mass
        self.force = np.zeros(3)
```

The class `Atom` has the following attributes:

- `id`: The unique identifier of the atom
- `type`: The type of the atom (hydrogen or oxygen or …)
- `position`: The position of the atom in 3D space (x, y, z)
- `velocity`: The velocity of the atom in 3D space (vx, vy, vz)
- `mass`: The mass of the atom
- `force`: The force acting on the atom in 3D space (fx, fy, fz)

In addition, we will need some information on the other atoms that are bound to the atom. We will store this information later in a list of atoms called `boundto`. Since we start with a monoatomic gas, we will not need this information for now. Note that position, velocity, and force are 3D vectors and we store them in numpy arrays. This is a very convenient way to handle vectors and matrices in Python.

The class `Atom` should further implement a number of functions, called methods in object-oriented programming, that allow us to interact with the atom. The following methods are implemented in the `Atom` class:

- `update_position(dt)`: Updates the position of the atom
- `update_velocity(dt)`: Updates the velocity of the atom
- `update_force(force)`: Updates the force acting on the atom
- `reset_force()`: Resets the force acting on the atom