Step-by-Step Development of a Molecular Dynamics Simulation

Frank Cichos

```
#| autorun: true
#| edit: false
#| echo: false
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 8,
                     'lines.linewidth': 1,
                     'lines.markersize': 10,
                     'axes.labelsize': 10,
                     'axes.titlesize': 10,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})

def get_size(w,h):
    return((w/2.54,h/2.54))
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.random.randn(2)*20
        self.mass = mass
        self.force = np.zeros(2)


    def add_force(self, force):
        """Add force contribution to total force on atom"""
        self.force += force

    def reset_force(self):
        """Reset force to zero at start of each step"""
        self.force = np.zeros(2)

    def update_position(self, dt):
        """First step of velocity Verlet: update position"""
        self.position += self.velocity * dt + 0.5 * (self.force/self.mass) * dt**2

    def update_velocity(self, dt, new_force):
        """Second step of velocity Verlet: update velocity using average force"""
        self.velocity += 0.5 * (new_force + self.force)/self.mass * dt
        self.force = new_force

    def apply_periodic_boundaries(self, box_size):
            """Apply periodic boundary conditions"""
            self.position = self.position % box_size
```

In the last seminar we have defined the class `Atom` that represents an atom in the simulation. This time, we would like to a force field to the simulation. We will use for out simulations the Lennard-Jones potential that we have had a look at initiall. We will implement this force field in a class `ForceField` that will contain the parameters of the force field and the methods to calculate the forces between the atoms.

The ForceField Class

The force field is a class that contains the parameters of the force field and the methods to calculate the forces between the atoms. The class `ForceField` has the following attributes:

- `sigma`: The parameter sigma of the Lennard-Jones potential
- `epsilon`: The parameter epsilon of the Lennard-Jones potential

These parameters are specific for each atom type. We will store these parameters in a dictionary where the keys are the atom types and the values are dictionaries containing the parameters sigma and epsilon. The class `ForceField` also contains the box size of the simulation. This is needed to apply periodic boundary conditions.

```
class ForceField:
    def __init__(self):
        self.parameters = {
            'C': {'epsilon': 1.615, 'sigma': 1.36},
            'H': {'epsilon': 1.0, 'sigma': 1.0 },
            'O': {'epsilon': 1.846, 'sigma': 3.0},
        }
        self.box_size = None  # Will be set when initializing the simulation
```

You will have certainly noticed that the parameters I defined do not correspond to the real values of the Lennard-Jones potential. Remember that the values for the hydrogen atom are typically

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

These are all small numbers and we will use larger values to make the simulation more stable. Actually, the Lenard-Jones potential provides a natural length and energy scale for the simulation. The length scale is the parameter σ and the energy scale is the parameter ϵ. We can therefore set $\sigma_{LJ} = 1$ and $\epsilon_{LJ} = 1$ and scale all other parameters accordingly. This is a common practice in molecular dynamics simulations.

Due to this rescaling energy, temperature and time units are also not the same as in the real world. We will use the following units:

- Energy: $\epsilon_{LJ} = \epsilon_H/\epsilon_H = 1$
- Length: $\sigma_{LJ} = 1$
- Mass: $m_{LJ} = 1$

This means now that all energies, for example, have to be scales by _{H} also the thermal energy. As thermal energy is related to temperature, then the temperature of the Lennard-Jones system

$$T_{LJ} = \frac{k_B T}{\epsilon_{LJ}}$$

which is, in the case of using the hydrogen energy scale, $T_{LJ} = 3.571.$ for $T = 300\,K$. For the time scale, we have to consider the mass of the hydrogen atom. The time scale is given by

$$t_{LJ} = \frac{t}{\sigma}\sqrt{\frac{\epsilon}{m_H}}$$

Thus a time unit of $1\,fs$ corresponds to $t_{LJ} = 0.099$. Thus using a timestep of 0.01 in reduced units would correspond to a real world timestep of just 1 fs. The table below shows the conversion factors for the different units. Simulating a Lennard-Jones system in reduced units therefore allows you to rescale to a real systems with the help of these conversion factors.

r*	rσ^{-1}
m*	mM^{-1}
t*	t$\sigma^{-1}\sqrt{\epsilon/M}$
T*	k$_B T\epsilon^{-1}$
E*	Eϵ^{-1}
F*	F$\sigma\epsilon^{-1}$
P*	P$\sigma^3\epsilon^{-1}$
v*	v$\sqrt{M/\epsilon}$
ρ^*	N$\sigma^3 V^{-1}$

Apply mixing rules when needed

`get_pair_parameters`

When we looked at the Lennard-Jones potential we realized that it reflects the pair interaction between the same atoms. However, in a molecular dynamics simulation, we have different atoms interacting with each other. We need to define the parameters of the interaction between different atoms. This is done using mixing rules. The most common mixing rule is the Lorentz-Berthelot mixing rule. The parameters of the interaction between two different atoms are calculated as follows:

```
def get_pair_parameters(self, type1, type2):
    # Apply mixing rules when needed
    eps1 = self.parameters[type1]['epsilon']
    eps2 = self.parameters[type2]['epsilon']
    sig1 = self.parameters[type1]['sigma']
    sig2 = self.parameters[type2]['sigma']

    # Lorentz-Berthelot mixing rules
    epsilon = np.sqrt(eps1 * eps2)
    sigma = (sig1 + sig2) / 2

    return epsilon, sigma
```

We therefore introduce the method `get_pair_parameters` that calculates the parameters of the Lennard-Jones potential between two different atoms. The method takes the atom types as arguments and returns the parameters epsilon and sigma of the Lennard-Jones potential between these two atoms. The method applies the Lorentz-Berthelot mixing rules to calculate the parameters. The method returns the parameters epsilon and sigma of the Lennard-Jones potential between the two atoms.

Apply minimum image convention

`minimum_image_distance`

Similarly, we already realized that using a finite box size requires to introduce boundary conditions. We decided that periodic boundary conditions are actually most convinient. However, this is introducing a new problem. When we calculate the distance between two atoms, we have to consider the minimum image distance. This means that we have to consider the distance between two atoms in the nearest image. This is done by applying the minimum image convention. The method `minimum_image_distance` calculates the minimum image distance between two positions. The method takes the positions of the two atoms as arguments and returns the minimum image distance between the two positions. The method applies the minimum image convention to calculate the minimum image distance.

```python
def minimum_image_distance(self, pos1, pos2):
    """Calculate minimum image distance between two positions"""
    delta = pos1 - pos2
    # Apply minimum image convention
    delta = delta - self.box_size * np.round(delta / self.box_size)
    return delta
```

Calculate the Lennard-Jones force between two atoms

`calculate_lj_force`

Finally we can calculate the Lennard-Jones force between two atoms. The method `calculate_lj_force` calculates the Lennard-Jones force between two atoms. The method takes the two atoms as arguments and returns the force between the two atoms. The method calculates the Lennard-Jones force between the two atoms using the Lennard-Jones potential. The method returns the force between the two atoms.

```python
def calculate_lj_force(self, atom1, atom2):
    epsilon, sigma = self.get_pair_parameters(atom1.type, atom2.type)
    r = self.minimum_image_distance(atom1.position, atom2.position)
    r_mag = np.linalg.norm(r)

    # Add cutoff distance for stability
    if r_mag > 2.5*sigma:
        return np.zeros(2)

    force_mag = 24 * epsilon * (
        2 * (sigma/r_mag)**13
        - (sigma/r_mag)**7
    )
    force = force_mag * r/r_mag
    return force
```

With these parts we have now a complete force field class which we can add to our simulation code.

Complete ForceField class

```python
class ForceField:
    def __init__(self):
        self.parameters = {
            'C': {'epsilon': 1.615, 'sigma': 1.36},
            'H': {'epsilon': 1.0, 'sigma': 1.0 },
            'O': {'epsilon': 1.846, 'sigma': 3.0},
        }
        self.box_size = None  # Will be set when initializing the simulation

    def get_pair_parameters(self, type1, type2):
        # Apply mixing rules when needed
        eps1 = self.parameters[type1]['epsilon']
        eps2 = self.parameters[type2]['epsilon']
        sig1 = self.parameters[type1]['sigma']
        sig2 = self.parameters[type2]['sigma']

        # Lorentz-Berthelot mixing rules
        epsilon = np.sqrt(eps1 * eps2)
        sigma = (sig1 + sig2) / 2

        return epsilon, sigma

    def minimum_image_distance(self, pos1, pos2):
        """Calculate minimum image distance between two positions"""
        delta = pos1 - pos2
        # Apply minimum image convention
        delta = delta - self.box_size * np.round(delta / self.box_size)
        return delta

    def calculate_lj_force(self, atom1, atom2):
        epsilon, sigma = self.get_pair_parameters(atom1.type, atom2.type)
        r = self.minimum_image_distance(atom1.position, atom2.position)
        r_mag = np.linalg.norm(r)

        # Add cutoff distance for stability
        if r_mag > 2.5*sigma:
            return np.zeros(2)

        force_mag = 24 * epsilon * (
            2 * (sigma/r_mag)**13
            - (sigma/r_mag)**7
        )
        force = force_mag * r/r_mag
        return force
```

MD Simulation Class

The last thing we need to do is to implement the MD simulation class. This class will be responsible for running the simulation. It is the controller of the simulation, who coordinates everything. By keeping this in a class you may even run several simulations simultaneously. This is not the case here, but it is a good practice to keep the simulation in a class.

`MDSimulation` class

This is just the constructor of the MD Simulation class. It takes the atoms, the force field, the timestep, and the box size as input. It initializes the simulation with the given parameters and sets the initial energy of the system to None. It also initializes an empty list to store the energy history of the system. The latter ones are not used for the moment but could be important later.

```python
class MDSimulation:
    def __init__(self, atoms, forcefield, timestep, box_size):
        self.atoms = atoms
        self.forcefield = forcefield
        self.forcefield.box_size = box_size  # Set box size in forcefield
        self.timestep = timestep
        self.box_size = np.array(box_size)
        self.initial_energy = None
        self.energy_history = []
```

`calculate_forces` method

The `calculate_forces` method calculates the forces between all pairs of atoms in the system. It first resets all forces on the atoms to zero. Then, it calculates the forces between all pairs of atoms using the Lennard-Jones force calculation from the force field class. The method updates the forces on the atoms accordingly. The method does not return anything.

```
    def calculate_forces(self):
        # Reset all forces
        for atom in self.atoms:
            atom.reset_force()

        # Calculate forces between all pairs
        for i, atom1 in enumerate(self.atoms):
            for atom2 in self.atoms[i+1:]:
                force = self.forcefield.calculate_lj_force(atom1, atom2)
                atom1.add_force(force)
                atom2.add_force(-force)  # Newton's third law
```

`update_positions_and_velocities` method
The `update_positions_and_velocities` method does exactly what its name says. It first of all updates the positions by calling the specific method of the atom. Then it is applying periodic boundary conditions. After that, it stores the current forces for the velocity update. Then it recalculates the forces with the new positions. Finally, it updates the velocities using the average of the old and new forces. The method does not return anything.

```
    def update_positions_and_velocities(self):
        # First step: Update positions using current forces
        for atom in self.atoms:
            atom.update_position(self.timestep)
            # Apply periodic boundary conditions
            atom.apply_periodic_boundaries(self.box_size)

        # Store current forces for velocity update
        old_forces = {atom.id: atom.force.copy() for atom in self.atoms}

        # Recalculate forces with new positions
        self.calculate_forces()

        # Second step: Update velocities using average of old and new forces
        for atom in self.atoms:
            atom.update_velocity(self.timestep, atom.force)
```

With these methods, we have a complete simulation class that can run a molecular dynamics simulation for a given number of steps. The simulation class will keep track of the energy of the system at each step, which can be used to analyze the behavior of the system over time.

Complete MDSimulation class

```
class MDSimulation:
    def __init__(self, atoms, forcefield, timestep, box_size):
        self.atoms = atoms
        self.forcefield = forcefield
        self.forcefield.box_size = box_size  # Set box size in forcefield
        self.timestep = timestep
        self.box_size = np.array(box_size)
        self.initial_energy = None
        self.energy_history = []


    def calculate_forces(self):
        # Reset all forces
        for atom in self.atoms:
            atom.reset_force()

        # Calculate forces between all pairs
        for i, atom1 in enumerate(self.atoms):
            for atom2 in self.atoms[i+1:]:
                force = self.forcefield.calculate_lj_force(atom1, atom2)
                atom1.add_force(force)
                atom2.add_force(-force)  # Newton's third law

    def update_positions_and_velocities(self):
        # First step: Update positions using current forces
        for atom in self.atoms:
            atom.update_position(self.timestep)
            # Apply periodic boundary conditions
            atom.apply_periodic_boundaries(self.box_size)

        # Store current forces for velocity update
        old_forces = {atom.id: atom.force.copy() for atom in self.atoms}

        # Recalculate forces with new positions
        self.calculate_forces()

        # Second step: Update velocities using average of old and new forces
        for atom in self.atoms:
            atom.update_velocity(self.timestep, atom.force)
```

Now we have the atom class, the force field class, and the simulation class. We can use these classes to run a molecular dynamics simulation of a simple Lennard-Jones system. In the next seminar, we still have to find a way to

- initialize the positions of the atoms in an appropriate way
- to provide them with a velocity distribution that matches the temperature of the system
- to run the simulation and keep the temperature constant
- to trace the energy in the system over time