Step-by-Step Development of a Molecular Dynamics Simulation

Frank Cichos

Implementations

The Atom Class

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

```
class Atom:
    dimension = 2

    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(dimension)
        self.mass = mass
        self.force = np.zeros(dimension)
```

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:

`add_force(force)`: Adds a force acting on the atom

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

`reset_force()`: Resets the force acting on the atom

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

`update_position(dt)`: Updates the position of the atom

```
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
```

`update_velocity(dt)`: Updates the velocity of the atom

```
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
```

`apply_periodic_boundaries(box_size)`: Applies periodic boundary conditions to the atom

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

Complete Atom class

```
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
```

This would be a good time to do something simple with the atom class. Let's create a bunch of atoms and plot them in a 2D space.

```
#| 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
```

```
atoms = [
    Atom(0, 'C', np.array([0.0, 0.0]), velocity=np.array([1.0, 1.0]), mass=1.0),
    Atom(1, 'C', np.array([2.0, 2.0]), velocity=np.array([-1.0, 1.0]), mass=1.0)
]

# Visualize positions
plt.figure(figsize=(6,6))
for atom in atoms:
    plt.plot(atom.position[0], atom.position[1], 'o')
    # Add velocity arrows
    plt.arrow(atom.position[0], atom.position[1],
              atom.velocity[0], atom.velocity[1],
              head_width=0.1)
plt.axis('equal')
plt.show()
```

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': 1.0, 'sigma': 3.4},
            'H': {'epsilon': 1, 'sigma': 1},
            'O': {'epsilon': 0.8, 'sigma': 3.0},
        }
        self.box_size = None  # Will be set when initializing the simulation
```

`get_pair_parameters`: **Apply mixing rules when needed**

```
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
```

`minimum_image_distance`: **Apply minimum image convention**

```
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_lj_force`: **Calculate the Lennard-Jones force between two atoms**

```
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
```

Complete ForceField class

```python
class ForceField:
    def __init__(self):
        self.parameters = {
            'C': {'epsilon': 1.0, 'sigma': 3.4},
            'H': {'epsilon': 1, 'sigma': 1},
            'O': {'epsilon': 0.8, '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
```