Table of contents

1. Simple Atomic System with Lennard-Jones Potential

1.1 Lennard-Jones Potential

The Lennard-Jones potential describes the interaction between two atoms:

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

The corresponding force:

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

where: - ϵ is the depth of the potential well - σ is the distance at which the potential is zero - r is the distance between particles

1.2 Basic Implementation

```python
import numpy as np

def lennard_jones_force(pos1, pos2, epsilon=1.0, sigma=1.0):
    r_vec = pos2 - pos1
    r = np.linalg.norm(r_vec)
    # Force magnitude
    force_mag = 24 * epsilon * (2 * (sigma/r)**12 - (sigma/r)**6) / r

    # Force vector
    force_vec = force_mag * r_vec / r

    return force_vec
```

2. Boundary Conditions

2.1 Periodic Boundary Conditions (PBC)

Particles that exit one side of the box re-enter from the opposite side.

```
def apply_periodic_bc(positions, box_length):
    return positions - box_length * np.floor(positions/box_length)

def minimum_image_distance(pos1, pos2, box_length):
    dr = pos2 - pos1
    dr = dr - box_length * np.round(dr/box_length)
    return dr
```

2.2 Reflective Boundaries

Particles bounce off the walls:

```
def apply_reflective_bc(positions, velocities, box_length):
    for i in range(len(positions)):
        for dim in range(3):
            if positions[i,dim] < 0:
                positions[i,dim] = -positions[i,dim]
                velocities[i,dim] = -velocities[i,dim]
            elif positions[i,dim] > box_length:
                positions[i,dim] = 2*box_length - positions[i,dim]
                velocities[i,dim] = -velocities[i,dim]
    return positions, velocities
```

3. Basic Simulation Loop

3.1 Initial Implementation with Just LJ Forces

```
class MDSimulation:
    def __init__(self, positions, velocities, mass, box_length, dt):
        self.positions = positions
        self.velocities = velocities
        self.mass = mass
        self.box_length = box_length
        self.dt = dt

    def calculate_forces(self):
        n_particles = len(self.positions)
        forces = np.zeros_like(self.positions)

        for i in range(n_particles):
            for j in range(i+1, n_particles):
                r_ij = minimum_image_distance(
                    self.positions[i],
                    self.positions[j],
                    self.box_length
                )
                f_ij = lennard_jones_force(np.zeros(3), r_ij)
                forces[i] += f_ij
                forces[j] -= f_ij  # Newton's third law

        return forces

    def velocity_verlet_step(self):
        # Calculate initial forces
        forces = self.calculate_forces()

        # Update positions
        self.positions += self.velocities * self.dt + \
                         0.5 * forces / self.mass * self.dt**2

        # Apply boundary conditions
        self.positions = apply_periodic_bc(self.positions, self.box_length)

        # Calculate new forces
        new_forces = self.calculate_forces()

        # Update velocities
        self.velocities += 0.5 * (forces + new_forces) / self.mass * self.dt
```

4. Adding Molecular Forces

4.1 Bond Forces

Adding harmonic bond potential:

$$V_{bond}(r) = \frac{k}{2}(r - r_0)^2$$

```
def bond_force(pos1, pos2, k_bond, r0):
    r_vec = pos2 - pos1
    r = np.linalg.norm(r_vec)
    force_mag = -k_bond * (r - r0)
    return force_mag * r_vec / r
```

4.2 Angle Forces

Three-body angle potential:

$$V_{angle}(\theta) = \frac{k_\theta}{2}(\theta - \theta_0)^2$$

```
def angle_force(pos1, pos2, pos3, k_angle, theta0):
    # Calculate vectors
    v1 = pos1 - pos2
    v2 = pos3 - pos2

    # Calculate angle
    cos_theta = np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
    theta = np.arccos(np.clip(cos_theta, -1.0, 1.0))

    # Calculate forces (simplified version)
    force_magnitude = -k_angle * (theta - theta0)

    return force_magnitude * v1, -force_magnitude * v2
```

4.3 Enhanced Simulation Class

```python
class MolecularMDSimulation(MDSimulation):
    def __init__(self, *args, bonds=None, angles=None):
        super().__init__(*args)
        self.bonds = bonds or []  # [(i, j, k, r0), ...]
        self.angles = angles or []  # [(i, j, k, k_angle, theta0), ...]

    def calculate_forces(self):
        # Start with non-bonded forces
        forces = super().calculate_forces()

        # Add bond forces
        for bond in self.bonds:
            i, j, k_bond, r0 = bond
            f_ij = bond_force(
                self.positions[i],
                self.positions[j],
                k_bond, r0
            )
            forces[i] += f_ij
            forces[j] -= f_ij

        # Add angle forces
        for angle in self.angles:
            i, j, k, k_angle, theta0 = angle
            f_i, f_k = angle_force(
                self.positions[i],
                self.positions[j],
                self.positions[k],
                k_angle, theta0
            )
            forces[i] += f_i
            forces[k] += f_k
            forces[j] -= (f_i + f_k)

        return forces
```

5. Example Usage

```
# Initialize system
n_particles = 100
box_length = 10.0
positions = np.random.rand(n_particles, 3) * box_length
velocities = np.zeros((n_particles, 3))
mass = 1.0
dt = 0.001

# Create simulation
sim = MolecularMDSimulation(
    positions, velocities, mass, box_length, dt,
    bonds=[(0, 1, 1000.0, 1.0)],  # Example bond
    angles=[(0, 1, 2, 100.0, np.pi)]  # Example angle
)

# Run simulation
n_steps = 1000
for step in range(n_steps):
    sim.velocity_verlet_step()
```

Here's a suggested basic structure:

```
class Atom:
    def __init__(self, atom_id, atom_type, position, velocity=None, mass=None):
        self.id = atom_id          # unique identifier
        self.type = atom_type      # e.g., 'H', 'C', 'O'
        self.position = position   # numpy array [x, y, z]
        self.velocity = velocity if velocity is not None else np.zeros(3)
        self.mass = mass
        self.force = np.zeros(3)   # current force on atom

        # Optional attributes that might be useful later:
        self.bonded_atoms = []     # list of atoms this atom is bonded to
        self.charges = 0.0         # for electrostatic interactions
```

Then you can later create classes for: 1. Bond (connects two Atom objects)

```python
class Bond:
    def __init__(self, atom1, atom2, k, r0):
        self.atom1 = atom1
        self.atom2 = atom2
        self.k = k      # force constant
        self.r0 = r0    # equilibrium distance
```

2. Angle (three Atom objects)

```python
class Angle:
    def __init__(self, atom1, atom2, atom3, k, theta0):
        self.atoms = [atom1, atom2, atom3]
        self.k = k          # force constant
        self.theta0 = theta0  # equilibrium angle
```

3. Dihedral (four Atom objects)

This modular approach makes it easier to: - Add features incrementally - Debug each interaction type separately - Keep track of connectivity - Calculate forces systematically

```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)

    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(3)

    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

class ForceField:
    def __init__(self, sigma, epsilon):
        self.sigma = sigma
        self.epsilon = epsilon

    def calculate_lj_force(self, atom1, atom2):
        """Calculate LJ force between two atoms"""
        r = atom1.position - atom2.position
        r_mag = np.linalg.norm(r)
        # LJ force calculation
        force = 24 * self.epsilon * (2 * (self.sigma/r_mag)**12
                                     - (self.sigma/r_mag)**6) * r/r_mag**2
        return force

class MDSimulation:
    def __init__(self, atoms, forcefield):
        self.atoms = atoms
        self.forcefield = forcefield

    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
```

New lecture

class Atom: def **init**(self, atom_id, atom_type, position, velocity=None, mass=None): self.id = atom_id self.type = atom_type # e.g., 'A', 'B', etc. self.position = position self.velocity = velocity if velocity is not None else np.zeros(2) self.mass = mass self.force = np.zeros(2)

class ForceField: def **init**(self): # Dictionary to store interaction parameters between atom types self.pair__parameters = {}

```
def add_pair_parameters(self, type1, type2, epsilon, sigma):
    """Add interaction parameters for a pair of atom types"""
    # Store parameters symmetrically
    key = tuple(sorted([type1, type2]))
    self.pair_parameters[key] = {'epsilon': epsilon, 'sigma': sigma}

def get_pair_parameters(self, type1, type2):
    """Get interaction parameters for a pair of atom types"""
    key = tuple(sorted([type1, type2]))
    if key not in self.pair_parameters:
        raise ValueError(f"No parameters defined for atom types {type1} and {type2}")
    return self.pair_parameters[key]

def calculate_lj_force(self, atom1, atom2, r_vec, r_mag):
    """Calculate LJ force with type-specific parameters"""
    params = self.get_pair_parameters(atom1.type, atom2.type)
    epsilon = params['epsilon']
    sigma = params['sigma']

    force_mag = 24.0 * epsilon * (2.0 * (sigma/r_mag)**13
                                 - (sigma/r_mag)**7)
    return force_mag * r_vec / r_mag
```

```
    def setup_simulation():
        # Create force field and add parameters
        ff = ForceField()

        # Add parameters for different combinations
        ff.add_pair_parameters('A', 'A', epsilon=1.0, sigma=1.0)  # A-A interaction
        ff.add_pair_parameters('B', 'B', epsilon=0.5, sigma=1.2)  # B-B interaction
        ff.add_pair_parameters('A', 'B', epsilon=0.7, sigma=1.1)  # A-B interaction

        # Create atoms of different types
        atoms = [
            Atom(0, 'A', np.array([1.0, 1.0]), mass=1.0),
            Atom(1, 'A', np.array([2.0, 2.0]), mass=1.0),
            Atom(2, 'B', np.array([3.0, 3.0]), mass=1.5),
            Atom(3, 'B', np.array([4.0, 4.0]), mass=1.5)
        ]

        return ff, atoms
```