

# MD Simulation

## Table of contents

Atom Class . . . . .	1
----------------------	---

```
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))
```

## 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

```

```

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

```

```

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 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_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)

```

```

def create_grid_atoms(num_atoms, box_size, mass=1.0, random_offset=0.1):
    box_size = np.array(box_size)

    # Calculate grid dimensions
    n = int(np.ceil(np.sqrt(num_atoms)))
    spacing = np.min(box_size) / n

    atoms = []
    for i in range(num_atoms):
        # Calculate grid position

```

```
row = i // n
col = i % n

# Base position
pos = np.array([col * spacing + spacing/2,
               row * spacing + spacing/2])

# Add random offset
pos += (np.random.rand(2) - 0.5) * spacing * random_offset

# Create atom
atoms.append(Atom(i, 'H', pos, mass=mass))

return atoms
```