

# Molecular Dynamics

February 1, 2020

## 1 Molecular Dynamics

For a single particle moving in an external potential, we write the equation of motion as

$$\frac{d\vec{r}}{dt} = \vec{v}(t), \quad (1)$$

$$m \frac{d\vec{v}}{dt} = -\nabla U(\vec{r}), \quad (2)$$

where  $\vec{a}(t) = -\frac{\nabla U(\vec{r})}{m}$  is the acceleration of the particle.

We can then apply the velocity **Verlet algorithm** to solve this problem numerically.

In many simulations, the particle is not moving in the static potential of an external agent, but is interacting with many particles in the same system volume.

If there are  $N$  particles in the system, then the equations of motion become

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i(t), \quad (3)$$

$$m_i \frac{d\vec{v}_i}{dt} = -\sum_{j \neq i} u_{ij}(\vec{r}_i, \vec{r}_j). \quad (4)$$

In the above equation, we assume that the forces acting on particle  $i$  are derived from pair potentials  $u_{ij}(\vec{r}_i, \vec{r}_j)$ .

This is only an assumption.

In principle, it is possible (and indeed suggested by **molecular mechanics that we will talk about later in this course**) that the force acting on particle  $i$  also receive contributions from  $n$ -body interactions  $u_{i_1 i_2 \dots i_n}(\vec{r}_{i_1}, \vec{r}_{i_2}, \dots, \vec{r}_{i_n})$ , the simplest beyond pair potentials would be three-body potentials  $u_{ijk}(\vec{r}_i, \vec{r}_j, \vec{r}_k)$ .

In molecular dynamics, the nature and functional forms of the interaction potentials are almost never derived from first principles. Instead, they are **chosen based on intuition**, tested in simulations, and then retained or discarded. Interaction potentials obtained this way are called semi-empirical potentials.

In this part of the course, we will start simple, and then progressively go to more and more sophisticated molecular dynamics problems.

## 1.1 Hard Disks and Hard Spheres

The simplest interaction potential we can use would be ... **no interaction potential**. That is to say, particles **do not feel the presence of each other, until they collide**.

If we assume that the particles are identical with mass  $m_i = m_j = m$ , have a finite radius  $b$ , and **do not deform in the collisions**, the particles can be treated as hard disks (2D) or hard spheres (3D).

Between collisions, all particles move along straight paths.

When particles  $i$  and  $j$  collide, as shown above, we assume that the collision is **perfectly elastic**. This means that

$$\vec{v}_{i\perp,\text{after}} = \vec{v}_{i\perp,\text{before}}, \quad (5)$$

$$\vec{v}_{j\perp,\text{after}} = \vec{v}_{j\perp,\text{before}}, \quad (6)$$

along the direction perpendicular to the line joining the two centres, and exchange velocity components

$$\vec{v}_{i\parallel,\text{after}} = \vec{v}_{j\parallel,\text{before}}, \quad (7)$$

$$\vec{v}_{j\parallel,\text{after}} = \vec{v}_{i\parallel,\text{before}} \quad (8)$$

along the line joining the two centres.

## 1.2 Initialization

To start such a simulation, we choose random initial positions for the  $N$  particles.

If any pair of particles overlap, that is, we find  $i$  and  $j$  such that


$$|\vec{r}_i - \vec{r}_j| < 2b, \quad (9)$$

we move them apart along the line joining the two centres until they are just touching.

We repeat this process until there are no more overlapping particles.

[Python code below]

```
In [24]: import numpy as np
         # N = 10 particles inside a unit box
         N = 10
         # b = 0.1
         b = 0.1
         # random initial positions for particles
         x = np.random.random(N)
         y = np.random.random(N)
         # check for overlap
         for i in range(N-1):
             for j in range(i+1,N):
                 if (x[i]-x[j])**2 + (y[i]-y[j])**2 < 4*b**2:
                     print(i,j)
```



0 6  
7 8

To do this, let us observe that if particle  $i$  (position  $\vec{r}_i$ ) and particle  $j$  (position  $\vec{r}_j$ ) overlap, the distance between the centres of the two particles is

$$|\vec{r}_i - \vec{r}_j| < 2b, \quad (10)$$

or equivalently,

$$|\frac{1}{2}(\vec{r}_i + \vec{r}_j)| < b. \quad (11)$$

To move particle  $i$  and particle  $j$  back along the line joining the two centres, so that they are just touching, we need to add  $b\hat{n}_{ij}$  to  $\frac{1}{2}(\vec{r}_i + \vec{r}_j)$ , and  $-b\hat{n}_{ij}$  to  $\frac{1}{2}(\vec{r}_i + \vec{r}_j)$ , i.e.

$$\vec{r}'_i = \frac{1}{2}(\vec{r}_i + \vec{r}_j) + b\hat{n}_{ij} = \frac{1}{2}(\vec{r}_i + \vec{r}_j) + b \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|}, \quad (12)$$

$$\vec{r}'_j = \frac{1}{2}(\vec{r}_i + \vec{r}_j) - b\hat{n}_{ij} = \frac{1}{2}(\vec{r}_i + \vec{r}_j) - b \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|}. \quad (13)$$

[Python code]

```
In [25]: # check for overlap
        for i in range(N-1):
            for j in range(i+1,N):
                if (x[i]-x[j])**2 + (y[i]-y[j])**2 < 4*b**2:
                    # move overlapping particles apart
                    # find unit vector along line joining two centres
                    xnij = x[i] - x[j]
                    ynij = y[i] - y[j]
                    normnij = np.sqrt(xnij**2 + ynij**2)
                    xnij = xnij/normnij
                    ynij = ynij/normnij
                    # find centre of overlap
                    xc = 0.5*(x[i] + x[j])
                    yc = 0.5*(y[i] + y[j])
                    # move i and j
                    x[i] = xc + b*xnij
                    y[i] = yc + b*ynij
                    x[j] = xc - b*xnij
                    y[j] = yc - b*ynij
```

After we have moved the two overlapping particles, we should check that they are not overlapping anymore.

```
In [26]: # check for overlap
        for i in range(N-1):
            for j in range(i+1,N):
                if (x[i]-x[j])**2 + (y[i]-y[j])**2 < 4*b**2:
                    print(i,j)
```

0 1

We see that after moving the overlapping pairs, we created a new overlapping pair. To completely remove overlaps in the initial configuration, let us write a Python function to return the list of overlapping pairs, so that we can later write a while loop to handle it.

```
In [28]: def check_overlap(x, y):
         # first find the number of particles
         N = len(x)
         # next initialize an empty array for the overlapping pairs
         l_overlap = []
         # check for overlaps
         for i in range(N-1):
             for j in range(i+1,N):
                 if (x[i]-x[j])**2 + (y[i]-y[j])**2 < 4*b**2:
                     l_overlap.append([i,j])
         # return l_overlap
         return(l_overlap)
```

Let us test this function.

```
In [30]: l_overlap = check_overlap(x,y)
         len(l_overlap)
```

```
Out[30]: 1
```

```
In [31]: for p in l_overlap:
         i = p[0]
         j = p[1]
         # move particles i and j
         # find unit vector along line joining two centres
         xnij = x[i] - x[j]
         ynij = y[i] - y[j]
         normnij = np.sqrt(xnij**2 + ynij**2)
         xnij = xnij/normnij
         ynij = ynij/normnij
         # find centre of overlap
         xc = 0.5*(x[i] + x[j])
         yc = 0.5*(y[i] + y[j])
         # move i and j
         x[i] = xc + b*xnij
         y[i] = yc + b*ynij
         x[j] = xc - b*xnij
         y[j] = yc - b*ynij
```

```
In [32]: l_overlap = check_overlap(x,y)
         l_overlap
```

Out[32]: [[0, 1], [0, 6]]

```
In [47]: def move_overlapping_particles(x, y, l_overlap):
        f = 1.001
        for p in l_overlap:
            i = p[0]
            j = p[1]
            # move particles i and j
            # find unit vector along line joining two centres
            xnij = x[i] - x[j]
            ynij = y[i] - y[j]
            print(xnij**2 + ynij**2)
            normnij = np.sqrt(xnij**2 + ynij**2)
            xnij = xnij/normnij
            ynij = ynij/normnij
            # find centre of overlap
            xc = 0.5*(x[i] + x[j])
            yc = 0.5*(y[i] + y[j])
            # move i and j
            x[i] = xc + f*b*xnij
            y[i] = yc + f*b*ynij
            x[j] = xc - f*b*xnij
            y[j] = yc - f*b*ynij
            # print testing
            print((x[i]-x[j])**2 + (y[i]-y[j])**2)
```

```
In [48]: move_overlapping_particles(x,y,l_overlap)
        l_overlap = check_overlap(x,y)
        l_overlap
```

0.0399722254339  
0.04008004  
0.039975236315  
0.04008004

Out[48]: []

```
In [35]: (x[0],y[0], x[1], y[1])
```

Out[35]: (0.87096962883559437,  
0.1658211781413117,  
0.72554478671840228,  
0.30121083976870644)

```
In [36]: xnij = x[0] - x[1]
        ynij = y[0] - y[1]
        (xnij, ynij)
```

Out[36]: (0.14542484211719209, -0.13538966162739474)

```

In [37]: normnij = np.sqrt(xnij**2 + ynij**2)
          normnij

Out[37]: 0.19869258964639491

In [38]: xnij = xnij/normnij
          ynij = ynij/normnij
          (xnij, ynij)

Out[38]: (0.73190873588189043, -0.68140267268317456)

In [39]: xnij**2 + ynij**2

Out[39]: 1.0000000000000004

In [46]: move_overlapping_particles(x,y,l_overlap)
          l_overlap = check_overlap(x,y)
          l_overlap

0.04
0.04
0.0398793530886
0.04

Out[46]: [[0, 1], [0, 6]]

In [41]: (x[0],y[0], x[1], y[1])

Out[41]: (0.87144808136518725,
          0.16537574168669161,
          0.72506633418880928,
          0.30165627622332652)

In [42]: xc = 0.5*(x[0] + x[1])
          yc = 0.5*(y[0] + y[1])
          (xc, yc)

Out[42]: (0.79825720777699827, 0.23351600895500907)

In [43]: xc + b*xnij

Out[43]: 0.87144808136518725

In [44]: b

Out[44]: 0.1

```

It turns out that if we move the overlapping particles apart until they are just overlapping, the `check_overlap()` function gets confused by truncation errors. Therefore, it is safer to move the overlapping particles apart until they are no longer overlapping. We do not need this space to be much, and in fact multiplication by  $f = 1.001$  was enough.

```
In [ ]: # set l_overlap to a non-empty list
        l_overlap = check_overlap(x, y)
        safety_counter = 0
        while (len(l_overlap) > 0):
            move_overlapping_particles(x,y,l_overlap)
            safety_counter += 1
            if safety_counter > 1000:
                break
        l_overlap = check_overlap(x, y)
```

Then we choose random velocities  $\vec{v}_i(0)$  for the  $N$  particles, by sampling from a Maxwell velocity distribution. We will discuss this sampling problem in the Monte Carlo part of the course.

For now just assign random velocity components between 0 and 1.

```
In [ ]: for i in range(N):
        vx[i] = 2.0*np.random.random() - 1.0
        vy[i] = 2.0*np.random.random() - 1.0
```

### 1.3 Boundary Conditions

If our  $N$  particles are in infinite space, then after some number of collisions, the particles will fly apart to infinity.

To prevent this from happening, we need to impose boundary conditions.

For example, if we allow the particles to take on positions within a box  $(L_x, L_y)$  or  $(L_x, L_y, L_z)$ , then in addition to the list of collision times  $\{\Delta t_{ij}\}$  between particles we also need to keep a list of collision times  $\{\Delta t_{iw}\}$  between particles and the walls of the box. If the boundary conditions are designed to keep particles within the box, they are called fixed boundary conditions.

In this case, to decide what the next event is we need to determine the minimum  $\Delta t_{ij}$ , and also the minimum  $\Delta t_{iw}$ . If  $\Delta t_{ij} < \Delta t_{iw}$ , the next event is a collision between particles. Else if  $\Delta t_{ij} > \Delta t_{iw}$ , the next event is a collision between a particle and the wall of the system.

However, even with the top-of-the-line computing resources, the largest hard-disk or hard-sphere MD simulation we can do contains 10-100 billion particles. In other words,  $10^{10}$ - $10^{11}$  particles. This is very small compared to  $10^{23}$  particles we expect to find in a thermodynamic system.

However, if we allow a particle crossing a boundary to reappear on the opposite boundary, like that shown above, we are dealing with a periodic boundary conditions. Periodic boundary conditions are frequently employed to mimic infinite systems (or large thermodynamic systems).

[Python code]

```
In [2]: def PBC(x, y, Lx=1.0, Ly=1.0):
        N = len(x)
        for i in range(N):
            if x[i] < 0.0:
                x[i] = x[i] + Lx
            if x[i] > Lx:
                x[i] = x[i] - Lx
            if y[i] < 0.0:
                y[i] = y[i] + Ly
            if y[i] > Ly:
```

$$y[i] = y[i] - Ly$$

## 1.4 Updates

For hard disks or hard spheres, it is silly to update the positions and velocities one time step  $\Delta t = h$  at a time, because the velocities change only during collisions, and the particles move along straight lines until then.

Therefore, we compute the times  $t_{ij}$  at which the collisions will take place between particle  $i$  and particle  $j$ .

Suppose at time  $t$ , the two particles are as shown below.

After a time interval  $\Delta t$ , we will have

$$\vec{r}_i(t + \Delta t) = \vec{r}_i(t) + \vec{v}_i \Delta t, \quad (14)$$

$$\vec{r}_j(t + \Delta t) = \vec{r}_j(t) + \vec{v}_j \Delta t. \quad (15)$$

To solve for when the two particles collide, we set

$$|\vec{r}_i(t + \Delta t) - \vec{r}_j(t + \Delta t)| = 2b, \quad (16)$$

or equivalently,

$$|\vec{r}_i(t + \Delta t) - \vec{r}_j(t + \Delta t)|^2 = 4b^2. \quad (17)$$

The latter is a quadratic equation in  $\Delta t$ ,

$$[(v_{xi} - v_{xj})^2 + (v_{yi} - v_{yj})^2] \Delta t^2 + 2[(x_i - x_j)(v_{xi} - v_{xj}) + (y_i - y_j)(v_{yi} - v_{yj})] \Delta t + (x_i - x_j)^2 + (y_i - y_j)^2 - 4b^2 = 0, \quad (18)$$

which we can solve to get two answers,  $\Delta t_-$  and  $\Delta t_+$ .

But why are there two answers?

Let me illustrate by making the second particle stationary.

Therefore, if  $\Delta t_- > 0$ , we see that  $t + \Delta t_-$  is the time that particle  $i$  collides with particle  $j$ , and  $t + \Delta t_+$  is the time that particle  $i$  would just be in contact with particle  $j$ , if it had 'passed through' particle  $j$  without collision at  $t + \Delta t_+$ .

This tells us that we should take  $\Delta t_-$  as the time to the collision, and set  $\Delta t_{ij} = \Delta t_-$ .

Sometimes when we solve the quadratic equation, we find that  $\Delta t_-$  and  $\Delta t_+$  are both negative. This tells us that there is no collision in the future (because the 'collision' is in the past). We should therefore set  $\Delta t_{ij} = \infty$ .

We should not be able to find  $\Delta t_- < 0$  and  $\Delta t_+ > 0$ , because this only happens when the particles are overlapping. We must have eliminated this problem before starting the simulation.

[Python code]

```
In [ ]: # x[i], y[i], x[j], y[j] known
        # vx[i], vy[i], vx[j], vy[j] known
        # radius of hard disk = b known
        A = (vx[i] - vx[j])**2 + (vy[i] - vy[j])**2
        B = 2.0*((x[i] - x[j])*(vx[i] - vx[j]) + (y[i] - y[j])*(vy[i] - vy[j]))
        C = (x[i] - x[j])**2 + (y[i] - y[j])**2 - 4.0*b*b
```



```

D = B*B - 4.0*A*C
if D <= 0.0:
    dtminus = T
else:
    dtminus = (- B - np.sqrt(D))/(2.0*A)

```

After this calculation, we have a list of collision times  $\{\Delta t_{ij} > 0\}$ . How do we proceed?  
At the beginning of the simulation, we have  $t = 0$ .  
From the list of collision times, we find the minimum collision time

$$\Delta t_1 = \min_{i,j} \Delta t_{ij}. \quad (19)$$

This is the time to the first collision in the simulation. Suppose it involves particles  $i_1$  and  $j_1$ .  
We therefore advance the simulation time to this first collision, by setting  $t \rightarrow t + \Delta t_1$ .  
We also subtract  $\Delta t_1$  from the rest of the collision times,  $\Delta t_{ij} \rightarrow \Delta t_{ij} - \Delta t_1$ .  
Next, we update the velocities of particles  $i_1$  and  $j_1$ .  
First, we need to determine the unit vector

$$\hat{n}_{i_1 j_1} = \frac{\vec{r}_{i_1}(t + \Delta t_1) - \vec{r}_{j_1}(t + \Delta t_1)}{|\vec{r}_{i_1}(t + \Delta t_1) - \vec{r}_{j_1}(t + \Delta t_1)|} \quad (20)$$

that points from particle  $j_1$  to particle  $i_1$ .  
Second, we need to compute the perpendicular and parallel components

$$\vec{v}_{i_1 \parallel, \text{before}} = (\vec{v}_{i_1} \cdot \hat{n}_{i_1 j_1}) \hat{n}_{i_1 j_1}, \quad (21)$$

$$\vec{v}_{j_1 \parallel, \text{before}} = (\vec{v}_{j_1} \cdot \hat{n}_{i_1 j_1}) \hat{n}_{i_1 j_1}, \quad (22)$$

$$\vec{v}_{i_1 \perp, \text{before}} = \vec{v}_{i_1} - \vec{v}_{i_1 \parallel, \text{before}}, \quad (23)$$

$$\vec{v}_{j_1 \perp, \text{before}} = \vec{v}_{j_1} - \vec{v}_{j_1 \parallel, \text{before}}. \quad (24)$$

Finally, we update the velocities of particles  $i_1$  and  $j_1$  to be

$$\vec{v}_{i_1, \text{after}} = \vec{v}_{i_1 \perp, \text{before}} + \vec{v}_{j_1 \parallel, \text{before}}, \quad (25)$$

$$\vec{v}_{j_1, \text{after}} = \vec{v}_{j_1 \perp, \text{before}} + \vec{v}_{i_1 \parallel, \text{before}}. \quad (26)$$

[Python code]

After this first collision, two velocities have changed, so we cannot simply look into  $\{\Delta t_{ij}\}$  for the next smallest time to collision  $\Delta t_2$ .

Instead, we must update the list of collision times!

Even so, there is no need to go through all  $(i, j)$  again, because only the velocities of  $i_1$  and  $j_1$  changed. Therefore, we only update  $\Delta t_{ij}$  if  $i = i_1, j_1$  or  $j = i_1, j_1$ .

Once this is done, we can choose the next minimum collision time  $\Delta t_2$ , and note the particles  $i_2$  and  $j_2$  involved.

[Python code]

## 1.5 Measurements

In MD, we perform microscopic simulations so that we can carry out macroscopic measurements.

For example, one of the macroscopic quantity that is commonly measured in MD simulations is the total energy

$$E = \sum_{i=1}^N \frac{1}{2} m v_i^2 + \frac{1}{2} \sum_{i=1}^N \sum_{j \neq i}^N u(\vec{r}_i, \vec{r}_j). \quad (27)$$

In our events-driven MD simulation of hard disks or hard spheres,  $E$  does not change, so no need to measure.

We can also determine the equation of state for the system, and in order to do so we must measure the pressure  $p$ , the volume  $V$ , and the temperature  $T$ .

In our MD simulations, the volume  $V$  is up to us to determine. To measure  $p$ , we can change the volume of our simulation system slightly, or measure the rate of momentum change over an imaginary surface. Similarly, to measure the temperature  $T$ , we need to change the total energy of the system slightly.

We can also measure mechanical properties like Young's modulus  $Y$ , or the Poisson ratio  $\nu$ . However, to perform such measurements we must run simulations with different boundary conditions to determine strains and stresses.

Perhaps the easiest quantity to measure in MD simulations of hard disks or hard spheres is the pair correlation function  $g(\vec{r})$ , which is the probability of finding a second particle at  $\vec{r}$  given a particle at  $\vec{0}$ . The Fourier transform  $\tilde{g}(\vec{k})$  of this quantity can be measured experimentally using X-ray diffraction.

Another easy quantity to measure in such MD simulations is the velocity-velocity correlation function.

## 1.6 Timestep-Driven MD

The first timestep-driven MD simulated the Lennard-Jones potential

$$u(r_{ij}) = 4\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^6 \right], \quad (28)$$

which model noble gasses with only van der Waals interaction, i.e. instantaneous dipoles attracting/repelling instantaneous dipoles. This work was done by Aneesur Rahman on a CDC 3600 mainframe computer in 1964.

Aneesur Rahman

CDC 3600 Computer

The Lennard-Jones potential consists of a repulsive core, and an attractive tail, as shown below. This is also a semi-empirical potential.

## 1.7 Force on Particle from Potential

Suppose a particle with position  $x$  moves in a potential  $U(x)$ . The force  $F$  it experiences will be

$$F = -\frac{dU}{dx}. \quad (29)$$

This is so that, if the potential increases from  $x = 0$  to  $x > 0$ , the slope  $dU/dx > 0$  but the particle must expend kinetic energy to gain potential energy. Thus the force must be acting opposite to the direction of increasing  $x$ .

More generally, if the particle moves in an  $N$ -dimensional space, so that its position is  $\vec{x} = (x_1, x_2, \dots, x_N)$  and the potential it experiences is  $U(x_1, x_2, \dots, x_N)$ , the force  $\vec{F}$  that the particle experiences would be a  $N$ -dimensional vector. We can write the force vector in terms of the potential function as

$$\vec{F} = -\nabla U(x_1, x_2, \dots, x_N), \quad (30)$$

and its component in the  $k$ th dimension would be

$$F_k = -\frac{\partial U(x_1, x_2, \dots, x_N)}{\partial x_k}. \quad (31)$$

## 1.8 Force on Particle $i$ Due to Particle $j$

Next, suppose there are  $N$  particles moving in the presence of each other. If there is no external potential, the potential due to interactions is given by

$$U(x_1, x_2, \dots, x_N) = \frac{1}{2} \sum_i^N \sum_{j \neq i}^N u_{ij}(x_i, x_j). \quad (32)$$

In a typical physics problem, the  $N$  particles are identical. They have the same mass, and interact with each other through the same pair potential  $u_{ij}(x_i, x_j) = u(x_i, x_j)$ .

From our discussion above, we know that the force  $F_k$  acting on the  $k$ th particle is

$$F_k = -\frac{\partial U(x_1, \dots, x_N)}{\partial x_k}. \quad (33)$$

Bringing the partial derivative inside the sums, we then find that

$$F_k = -\frac{1}{2} \sum_i^N \sum_{j \neq i}^N \frac{\partial u(x_i, x_j)}{\partial x_k} = -\frac{1}{2} \sum_i^N \sum_{j \neq i}^N \left[ \frac{\partial u}{\partial x_i} \frac{\partial x_i}{\partial x_k} + \frac{\partial u}{\partial x_j} \frac{\partial x_j}{\partial x_k} \right], \quad (34)$$

making use of the chain rule of partial differentiation.

Now, the partial derivative  $\partial x_i / \partial x_k$  is zero if  $i \neq k$ , because the coordinates of different particles are independent, and is one if  $i = k$ . Therefore, we can write it as a Kronecker delta

$$\frac{\partial x_i}{\partial x_k} = \delta_{ik} = \begin{cases} 1, & i = k; \\ 0, & i \neq k. \end{cases} \quad (35)$$

The same is true for  $\partial x_j / \partial x_k$ , i.e.

$$\frac{\partial x_j}{\partial x_k} = \delta_{jk} = \begin{cases} 1, & j = k; \\ 0, & j \neq k. \end{cases} \quad (36)$$

Putting these into the expression for  $F_k$ , we have

$$\begin{aligned}
F_k &= -\frac{1}{2} \sum_i^N \sum_{j \neq i}^N \left[ \frac{\partial u(x_i, x_j)}{\partial x_i} \delta_{ik} + \frac{\partial u(x_i, x_j)}{\partial x_j} \delta_{jk} \right] \\
&= -\frac{1}{2} \sum_{j \neq k}^N \frac{\partial u(x_k, x_j)}{\partial x_k} - \frac{1}{2} \sum_{i \neq k}^N \frac{\partial u(x_i, x_k)}{\partial x_k}.
\end{aligned} \tag{37}$$

In general, the pair potential  $u(x_i, x_k) = u(x_k, x_i)$  depends only on the separation  $r_{ik} = |x_i - x_k|$  between the two particles, and not on which particle we call  $i$  and which other particle we call  $k$ . Therefore,  $\partial u(x_k, x_j)/\partial x_k = \partial u(x_j, x_k)/\partial x_k$ . Also, since  $i$  and  $j$  are summed over, they are dummy variables, and we can always rename  $j$  as  $i$ , to get

$$F_k = - \sum_{i \neq k}^N \frac{\partial u(x_k, x_i)}{\partial x_k} = \sum_{i \neq k} f_{ki}. \tag{38}$$

In the above equation,

$$f_{ki} = - \frac{\partial u(x_k, x_i)}{\partial x_k} \tag{39}$$

is the force acting on particle  $k$  due to particle  $i$ .

For two particles  $i$  and  $j$  interacting via the Lennard-Jones potential in three dimensions, the force  $\vec{f}_{ij}$  acting on particle  $i$  due to particle  $j$  can then be written as

$$\vec{f}_{ij} = \left( -\frac{\partial u(\vec{r}_i, \vec{r}_j)}{\partial x_i}, -\frac{\partial u(\vec{r}_i, \vec{r}_j)}{\partial y_i}, -\frac{\partial u(\vec{r}_i, \vec{r}_j)}{\partial z_i} \right). \tag{40}$$

Since  $u(\vec{r}_i, \vec{r}_j) = u(r_{ij})$  is a function only of the separation

$$r_{ij} = [(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{1/2}, \tag{41}$$

we can also write

$$\frac{\partial u(r_{ij})}{\partial x_i} = \frac{du(r_{ij})}{dr_{ij}} \frac{\partial r_{ij}}{\partial x_i} = \frac{du(r_{ij})}{dr_{ij}} \frac{x_i - x_j}{r_{ij}}. \tag{42}$$

Similarly,

$$\frac{\partial u(r_{ij})}{\partial y_i} = \frac{du(r_{ij})}{dr_{ij}} \frac{\partial r_{ij}}{\partial y_i} = \frac{du(r_{ij})}{dr_{ij}} \frac{y_i - y_j}{r_{ij}}, \tag{43}$$

$$\frac{\partial u(r_{ij})}{\partial z_i} = \frac{du(r_{ij})}{dr_{ij}} \frac{\partial r_{ij}}{\partial z_i} = \frac{du(r_{ij})}{dr_{ij}} \frac{z_i - z_j}{r_{ij}}. \tag{44}$$

Putting them all together, we obtain

$$\vec{f}_{ij} = - \frac{du(r_{ij})}{dr_{ij}} \frac{\vec{r}_i - \vec{r}_j}{r_{ij}}. \tag{45}$$

## 1.9 Explicit Form of $\vec{f}_{ij}$

Given

$$u(r_{ij}) = 4\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^6 \right], \quad (46)$$

we find that

$$\frac{du}{dr_{ij}} = 4\epsilon \left[ -12 \frac{\sigma^{12}}{r_{ij}^{13}} + 6 \frac{\sigma^6}{r_{ij}^7} \right], \quad (47)$$

and hence

$$\vec{f}_{ij} = \frac{48\epsilon}{r_{ij}^2} \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{\sigma}{r_{ij}} \right)^6 \right] (\vec{r}_i - \vec{r}_j). \quad (48)$$

Since  $\vec{f}_{ji} = -\vec{f}_{ij}$ , we need compute only for distinct pairs.

The (non-executable) Python code for doing so is shown below:

```
In [ ]: for i in range(N-1):
        for j in range(i+1, N):
            Rij = R[i,:] - R[j,:]
            rij2 = np.linalg.norm(Rij)**2
            sij2 = sigma*sigma/rij2
            sij6 = sij2*sij2*sij2
            sij12 = sij6*sij6
            fij = 48*(epsilon/rij2)*(sij12 - 0.5*sij6)*Rij
            F[i,:] = F[i,:] + fij
            F[j,:] = F[j,:] - fij
```

## 1.10 Memory Allocation

As we can see, the above Python code for computing the forces requires us to use the numpy arrays

```
R = np.zeros((N, 3))
```

```
V = np.zeros((N, 3))
```

```
F = np.zeros((N, 3))
```

to allocate the memory space for the positions, velocities, and forces on the  $N$  particles.

## 1.11 Initialization

Unlike for the hard disk or hard sphere simulations, we do not need to be very careful about the initialization.

If we are simulating  $N$  particles in a unit cube, then we can use

```
In [ ]: R = np.random.random((N, 3))
```

Similarly, we can randomly assign positive and negative velocities as follows:

```
In [ ]: V = 2.0*np.random.random((N,3)) - 1.0
```

Because of the way the forces are computed,  $n$ -body interactions are guaranteed, and the simulation should eventually achieve thermodynamic equilibrium.

This reassurance notwithstanding, it is perhaps a good time to talk about how to sample equilibrium velocity distributions for our timestep-driven simulation.

To do so, we need to first write down the Maxwell velocity distribution

$$f(\vec{v}) = \left(\frac{m}{2\pi kT}\right)^{3/2} \exp\left(-\frac{mv^2}{2kT}\right) = f(v_x)f(v_y)f(v_z), \quad (49)$$

where

$$f(v_x) \sim \exp\left(-\frac{mv_x^2}{2kT}\right). \quad (50)$$

This tells us that we can sample  $v_x$  from a normal distribution with mean  $\mu_x = 0$  and variance  $\sigma_x^2 = kT/m$ . Similarly, we can sample  $v_y$  and  $v_z$  from normal distributions with  $\mu = 0$  and  $\sigma^2 = kT/m$ .

```
In [ ]: V = (T/m)*np.random.randn((N,3))
```

if we set units such that the Boltzmann constant is  $k = 1$ .

## 1.12 Updates

For the Lennard-Jones MD simulation, we perform the numerical integration using velocity Verlet.

However, before we do so, there is an important question we need to answer: do we save all the trajectories  $(x_i(t), y_i(t), z_i(t))$ ?

For  $N = 10$  particles, and say 1,000 time steps, this poses no problem.

However, if we do MD simulation of protein folding where water molecules are simulated explicitly, then we are easily talking about  $10^6$  to  $10^9$  coordinates over  $10^6$  to  $10^9$  time steps.

If we store them all, we will need memory space for  $10^{12}$  to  $10^{18}$  doubles. Each double uses 64 bits  $\equiv$  8 bytes of memory. So saving the trajectors of  $10^6$  to  $10^9$  coordinates over  $10^6$  to  $10^9$  time steps would require 1 GB of RAM/disk space to 1 PB of RAM/disk space. The former is still manageable, but the latter is not possible.

If we save the trajectory every 1,000 time steps, then in the latter case we will end up with 1 TB files, which is still possible to manage. Alternatively, if we save only measurements, like the total energy  $E(t)$  over  $10^9$  time steps, then we would end up with files about 100 kB in size.

[Python code]

## 1.13 Acceleration Schemes

With modern supercomputers (i.e. clusters), we can do  $10^{12}$ -atom MD simulations, but the run time will be too long.

In general, the long run time is because of the large number of force evaluations,  $(10^{12})^2 = 10^{24}$ !

In such a simulation, most particles are too far apart for the Lennard-Jones potential to be important. That is to say,  $|\vec{f}_{ij}|$  would be smaller than the machine epsilon, and thus there is no need to evaluate such forces.

A common way to speed up the MD simulation is to introduce the truncated Lennard-Jones potential, which we evaluate if  $r_{ij} < r_c$ , but skip otherwise. Unfortunately, if we only truncate the Lennard-Jones potential, we will have to go through  $10^{24}$  pairs to check.

Therefore, the truncated Lennard-Jones potential must be combined with breaking the simulation system into cells, such that each cell is slightly larger than  $r_c \times r_c \times r_c$ . Then,

- for particles in the same cell, we evaluate the Lennard-Jones force;

- for particles in neighboring cells, we also evaluate the Lennard-Jones force;

- for particles in cells further than nearest neighbors, there is no need to evaluate the Lennard-Jones force.

This organization of the simulation system into cells greatly reduces the number of pairs we need to perform force evaluation over, and hence the total simulation time.