Writing an Ultra-Simple Molecular Dynamics Program in Python

Source: https://qiita.com/ulabiisutokyo/

Assignment ~ Create or modify a Molecular Dynamics (MD) code

1.1. Confirm that the following programs (`crudemd1.py` to `crudemd3.py`) run correctly.

1.2. Try running the MD simulation after changing the initial conditions (initial positions, initial velocities, number of atoms, etc.).

1.3. In the original program (`crudemd3.py`), atoms that exit the simulation cell do not return. Modify the program so that atoms **bounce back** from the cell boundaries and return.

※ The length of one side of the simulation cell is `l/scl = 500/10 = 50 Å`.

0. Preparation ~ Create a File for Initial Atomic Configuration

0.1. Write the initial positions and velocities of three atoms in a file named `initial.d`.

```
#### initial.d
15 10 0 0
20 10 0 0
20 15 0 0
#### End of initial.d
```

0.2. Each line in initial.d contains four values: the x and y coordinates of an atom, and the x and y components of its velocity. These should be separated by a single half-width space.

- Notes:
  - Use only one space between values.
  - Do not insert a space at the beginning of each line.
  - Failing to follow these rules may cause errors during execution.
  - Also, `initial.d` must be placed in the same folder as the Python program being run.

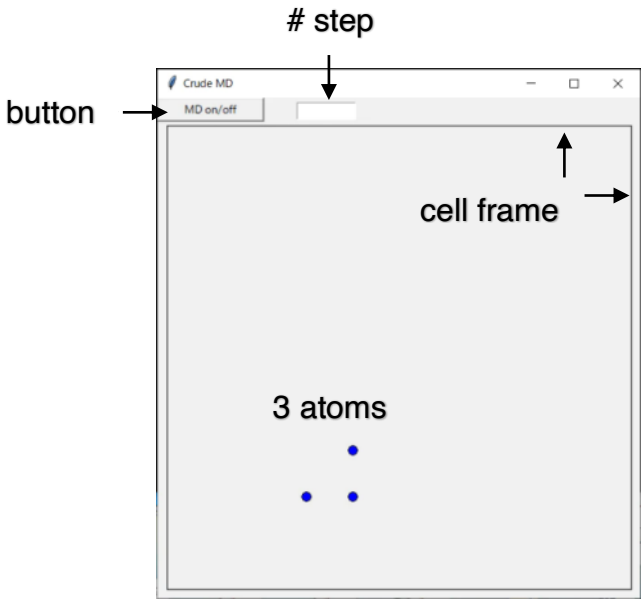

Figure 1 – Create the window as shown in the figure

1. Placing Atoms on the Canvas (`crudemd1.py`)

1.1. This step opens a canvas for drawing and reads the atomic positions and velocities from `initial.d`, placing the atoms at their initial positions.

1.2. An example program, `crudemd1.py`, is shown on the next page.

1.3. A dummy button is also included for later use (currently, pressing the button does nothing).

1.4. Since `initial.d` contains 3 lines, the number of atoms will be 3. You can increase the number of atoms by adding more lines.

```
#### crudemd1.py (anything after the # symbol is treated as a comment)
# ← In Python, # is considered as commented

import tkinter as tk

def dummy(event):       # Dummy button
        print('button is pressed')

def drawatom(x,y):      # Draw atom
        scl=10  # Scaling (magnification) factor
        rad=5   # Radius of sphere (atom)
        x1=x0+x*scl
        y1=y0+l-y*scl
        canvas.create_oval(x1-rad,y1-rad,x1+rad,y1+rad, fill='blue')

# creation of main window and canvas
win = tk.Tk()
win.title("Crude MD")
win.geometry("520x540")
x0=10   # Origin
y0=10   # Origin
l=500   # Size of canvas
canvas = tk.Canvas(win,width=l+x0*2,height=l+y0*2)
canvas.place(x=0, y=20)
canvas.create_rectangle(x0,y0,l+x0,l+y0)

# Declaration of arrays
rx = [] # Position of atom
ry = []
vx = [] # Velocity of atom
vy = []
fx = [] # Force on atom
fy = []

# button sample (dummy)
button_dummy = tk.Button(win,text="dummy",width=15)
button_dummy.bind("<Button-1>",dummy)
button_dummy.place(x=0,y=0)
```

```
# read initial position and velocity
f = open('initial.d', 'r')
i = 0
for line in f:
    xy = line.split(' ')
    rx = rx + [float(xy[0])]
    ry = ry + [float(xy[1])]
    vx = vx + [float(xy[2])]
    vy = vy + [float(xy[3])]
    print(rx)
    drawatom(rx[i],ry[i])
    i = i + 1

# Main Loop
win.mainloop()

#### End of crudemd1.py
```

2. Responding to Button Events and Moving Atoms (`crudemd2.py`)

2.1. This program extends the functionality of `crudemd1.py` with the following features:

- Detects button press events.
- When the button is pressed, all atoms begin moving diagonally upward in a continuous motion.

2.2. Note:

- This is **not yet a Molecular Dynamics (MD) simulation**, as the atoms are not moving due to interatomic forces or physical laws—only by a fixed directional displacement.

```
#### crudemd2.py

import tkinter as tk

def dummy(event):       # Button to switch on/off
    global imd          # to substitute the value into the variable
    if (imd == 0 ):
        imd = 1# Run
    else:
        imd = 0# Stop

def drawatom(x,y):      # Draw atom
    x1=x0+x*scl
    y1=y0+l-y*scl
    return canvas.create_oval(x1-rad,y1-rad,x1+rad,y1+rad, fill='blue')

def moveatom(obj,x,y): # Move atom
    x1=x0+x*scl
    y1=y0+l-y*scl
    canvas.coords(obj,x1-rad,y1-rad,x1+rad,y1+rad)

# creation of main window and canvas
win = tk.Tk()
win.title("Crude MD")
win.geometry("520x540")
x0 =10  # Origin
y0 =10  # Origin
l  =500 # Size of canvas
scl=10  # Scaling (magnification) factor
rad=5   # Radius of sphere

canvas = tk.Canvas(win,width=l+x0*2,height=l+y0*2)
canvas.place(x=0, y=20)
canvas.create_rectangle(x0,y0,l+x0,l+y0)

imd = 0 # MD on/off
```

```
# declaration of arrays
rx = [] # Position
ry = []
vx = [] # Velocity
vy = []
fx = [] # Force
fy = []
obj = []

# button sample (dummy)
button_dummy = tk.Button(win,text="dummy",width=15)
button_dummy.bind("<Button-1>",dummy)
button_dummy.place(x=0,y=0)

# Entry box (MD step)
entry_step = tk.Entry(width=10)
entry_step.place(x=150,y=5)

# read initial position and velocity
f = open('initial.d', 'r')
n = 0
for line in f:
    xy = line.split(' ')
    rx = rx + [float(xy[0])]
    ry = ry + [float(xy[1])]
    vx = vx + [float(xy[2])]
    vy = vy + [float(xy[3])]
    print(rx)
    obj = obj + [drawatom(rx[n],ry[n])] # obj[] holds pointer to object
    n = n + 1 # number of total atoms
print("number of atoms = ",n)

# Main Loop
step = 0
stepend = 1000

while step < stepend:
    if imd == 1:
        for i in range(n):
            rx[i] = rx[i] + 0.01
            ry[i] = ry[i] + 0.01
            moveatom(obj[i],rx[i],ry[i])
        step = step + 1
        entry_step.delete(0,tk.END)
        entry_step.insert(tk.END,step)
    win.update()

win.mainloop()

#### End of crudemd2.py
```

3. Toggle Molecular Dynamics with Button Press (`crudemd3.py`)

3.1. This program builds upon `crudemd2.py` and introduces actual Molecular Dynamics functionality:

- **Pressing the button toggles the MD simulation on and off**.
- Atom positions are updated based on **interatomic forces**, calculated using the **Morse potential**.
- Atom movement is computed using the **Verlet integration method**, a standard approach in MD simulations.

3.2. Note:

- This marks the transition to an actual MD simulation, where atoms interact physically rather than move arbitrarily.
- Please refer to Notes 1 and 2 at the end for details on the Morse potential and the Verlet method.

```
#### crudemd3.py

import tkinter as tk   # to create a window and a canvas
import math

def dummy(event):      # Switch MD on/off
    global imd         # to substitute the value into the variable
    if (imd == 0 ):
        imd = 1
    else:
        imd = 0

def drawatom(x,y):     # Draw atoms on canvas
    x1=x0+x*scl
    y1=y0+l-y*scl
    return canvas.create_oval(x1-rad,y1-rad,x1+rad,y1+rad, fill='blue')

def moveatom(obj,x,y): # Move atoms on canvas
    x1=x0+x*scl
    y1=y0+l-y*scl
    canvas.coords(obj,x1-rad,y1-rad,x1+rad,y1+rad)

def v(rang):    # Morse potential
    ep = 0.2703# epsilon [eV]
    al = 1.1646# alpha [1/ang]
    ro = 3.253  # rho [ang]
    ev = 1.6021892e-19 # eV -> J (energy unit conversion)
    return ep*(math.exp(-2.0*al*(rang-ro))-2.0*math.exp(-al*(rang-ro))) *ev

def vp(rang):  # Derivative of Morse potential (= Force)
```

```
    ep = 0.2703
    al = 1.1646
    ro = 3.253
    ev = 1.6021892e-19
    return -2.0*al*ep*(math.exp(-2.0*al*(rang-ro))-math.exp(-al*(rang-ro)))
*ev*1.0e10

# creation of main window and canvas
win = tk.Tk()
win.title("Crude MD")
win.geometry("520x540")
x0=10   # Origin
y0=10   # Origin
l=500   # Size of canvas
scl=10  # Scaling (magnification) factor
rad=5   # Radius of sphere

canvas = tk.Canvas(win,width=l+x0*2,height=l+y0*2)
canvas.place(x=0, y=20)
canvas.create_rectangle(x0,y0,l+x0,l+y0)

imd = 0 # MD on/off

# declaration of arrays
rx = [] # Position [ang]
ry = []
vx = [] # Velocity [ang/s]
vy = []
fx = [] # Force [N]
fy = []
epot = []       # Potential energy [J]
obj = []        # Object (for visualization)
dt = 1.0e-16    # Time step [s]
wm = 1.67e-37   # Mass [1e-10 kg]

# button sample (dummy)
button_dummy = tk.Button(win,text="MD on/off",width=15)
button_dummy.bind("<Button-1>",dummy)
button_dummy.place(x=0,y=0)

# Entry box (MD step)
entry_step = tk.Entry(width=10)
entry_step.place(x=150,y=5)

# read initial position and velocity
f = open('initial.d', 'r')
n = 0
for line in f:
    xy = line.split(' ')
    rx = rx + [float(xy[0])]
    ry = ry + [float(xy[1])]
    vx = vx + [float(xy[2])]
    vy = vy + [float(xy[3])]
    fx = fx + [0]
    fy = fy + [0]
    epot = epot + [0]
    obj = obj + [drawatom(rx[n],ry[n])] # obj[] holds pointer to object
```

```
    n = n + 1 # number of total atoms
print("number of atoms = ",n)

# Main Loop of MD
step = 0
stepend = 100000

while step < stepend:
    if imd == 1:
        # Verlet(1)
        for i in range(n):
            rx[i] = rx[i] + dt * vx[i] + (dt*dt/2.0) * fx[i] / wm
            ry[i] = ry[i] + dt * vy[i] + (dt*dt/2.0) * fy[i] / wm
            vx[i] = vx[i] + dt/2.0 * fx[i] / wm
            vy[i] = vy[i] + dt/2.0 * fy[i] / wm
        # Force and energy
        for i in range(n):
            fx[i] = 0
            fy[i] = 0
            epot[i] = 0
        for i in range(n):
            for j in range(n):
                if (i != j):
                    rr = math.sqrt((rx[i]-rx[j])**2 + (ry[i]-ry[j])**2)
                    drx = rx[i] - rx[j]
                    dry = ry[i] - ry[j]
                    fx[i] = fx[i]-vp(rr)/rr*drx
                    fy[i] = fy[i]-vp(rr)/rr*dry
                    epot[i] = epot[i]+v(rr)/2.0
        # Verlet(2)
        for i in range(n):
            vx[i] = vx[i] + dt/2.0 * fx[i] / wm
            vy[i] = vy[i] + dt/2.0 * fy[i] / wm
            moveatom(obj[i],rx[i],ry[i])
        step = step + 1
        entry_step.delete(0,tk.END)
        entry_step.insert(tk.END,step)
    win.update()

win.mainloop()

#### End of crudemd3.py
```

Note 1: Morse Potential

The Morse potential is a type of interatomic potential function that defines the interaction between atoms. Its functional form is as follows: the potential energy $U(r_{ij})$is determined based on the distance r_{ij} between a pair of atoms i and j.

$$U(r_{ij}) = \varepsilon\left[\exp\left(-2\alpha(r_{ij}-\rho)\right) - 2\exp\left(-\alpha(r_{ij}-\rho)\right)\right]. \tag{1-1}$$

When plotted, the Morse potential appears as shown in Figure 2. The parameters e, r, and α are potential parameters corresponding to the **bond energy**, **equilibrium (most stable) distance**, and **spring constant** (curvature at the equilibrium point), respectively. By differentiating the potential U with respect to r_{ij}, the **magnitude of the force** f_{ij} exerted on atom i by atom j can be calculated. Since the force f_{ij} acts in the direction of $\mathbf{r}_{ij}$, its **direction is aligned** with the vector connecting the two atoms.

$$f_{ij} = -\frac{dU(r_{ij})}{dr_{ij}} = 2\alpha\varepsilon\left[\exp\left(-2\alpha(r_{ij}-\rho)\right) - \exp\left(-\alpha(r_{ij}-\rho)\right)\right], \tag{1-2}$$

$$\mathbf{f}_{ij} = f_{ij}\frac{\mathbf{r}_{ij}}{r_{ij}}. \tag{1-3}$$

The net force (resultant force) $\mathbf{f}_i$ acting on atom i is the sum of $\mathbf{f}_i$ for all atoms j.

$$\mathbf{f}_i = \sum_{j\neq i}\mathbf{f}_{ij}. \tag{1-4}$$

If the force $\mathbf{f}_i$ acting on each atom is known, the atomic acceleration $\mathbf{a}_i$ can be obtained using Newton's equation of motion: $\mathbf{f}_i = m_i\mathbf{a}_i$. By numerically integrating this equation, we can describe the motion of the atom—that is, how its **position** $\mathbf{r}_{ij}$ and **velocity** $\mathbf{v}_{ij}$ change over time. It is noteworthy that the **initial conditions** (initial position and velocity) must be explicitly provided beforehand. Since we are treating a **uniform system**, the mass m_i is assumed to be the **same for all atoms** i.

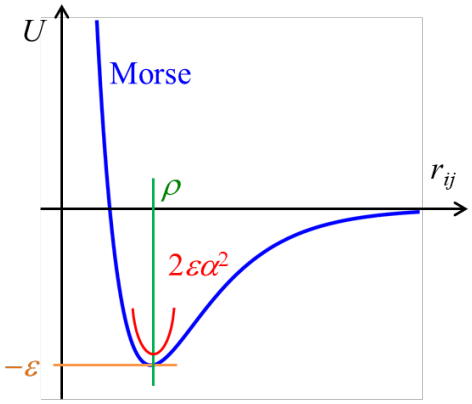

Figure 2 – Morse potential function form

Note 2: Verlet method

The Verlet method is a numerical solution method for differential equations. In molecular dynamics, it is used to calculate the time evolution of each atom's position and velocity. The purpose is to calculate $\mathbf{r}_i(t+\Delta t)$ and $\mathbf{v}_i(t+\Delta t)$ after a small amount of time Δt has passed from the position $\mathbf{r}_i(t)$ and velocity $\mathbf{v}_i(t)$ at time t.

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{1}{2m_i}\mathbf{f}_i(t)(\Delta t)^2, \tag{2-1}$$

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \frac{1}{2m_i}\left[\mathbf{f}_i(t) + \mathbf{f}_i(t+\Delta t)\right]\Delta t\,. \tag{2-2}$$

In the main loop of the program `crudemd3.py`, these two equations are used to update $\mathbf{r}_i$ and $\mathbf{v}_i$ at every step. Note that in the Eq. (2-1) for $\mathbf{r}_i$, the right-hand side only contains information about time t ($\mathbf{r}_i(t+\Delta t)$ can be determined using only the information about time t), whereas the Eq. (2-2) for $\mathbf{v}_i(t)$ uses both the force $\mathbf{f}_i(t)$ before time advances and the $\mathbf{f}_i(t+\Delta t)$ after time has advanced by Δt. Therefore, the program processes in the following order:

1. Update $\mathbf{r}_i(t)$ [Eq. (2-1), Verlet(1) in "`crudemd3.py`"]
2. Reflect the $\mathbf{f}_i(t)$ part of $\mathbf{v}_i(t)$ [Eq. (2-2), Verlet(1)]
3. Update $\mathbf{f}_i(t)$ ($t \rightarrow t+\Delta t$)
4. Reflect the $\mathbf{f}_i(t+\Delta t)$ part of $\mathbf{v}_i(t)$ [Eq. (2-2), Verlet(2)]

The Verlet method processing is divided into (1) and (2) in the program because the force $\mathbf{f}$ needs to be updated in the middle of updating the velocity $\mathbf{v}$.

***

For reference, let us derive the formula for $\mathbf{r}_i(t+\Delta t)$. First, expand $\mathbf{r}_i(t+\Delta t)$ in terms of Δt.

$$\cdot\,\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \frac{d\mathbf{r}_i(t)}{dt}\Delta t + \frac{1}{2}\frac{d^2\mathbf{r}_i(t)}{dt^2}(\Delta t)^2 + \ldots \tag{2-3}$$

By discarding the third-order and higher terms in this equation and using $\mathbf{v} = d\mathbf{r}/dt$, $\mathbf{a} = d^2\mathbf{r}/dt^2$ and $\mathbf{f} = m\mathbf{a}$, we obtain Eq. (2-1). Note that the derivation of Eq. (2-2) for $\mathbf{v}$ is somewhat tricky, so we will omit it here (though if you are interested, feel free to give it a try).