python で超簡単な分子動力学プログラムを書く

出典： https://qiita.com/ulabiisutokyo/

～ 課題 ～ MD のコードを作成・改変してください.

a. 下記のプログラム (crudemd1.py ~ crudemd3.py) が上手く動くことを確認してください.

b. 初期条件 (初期配置，初期速度，原子数など) を変更し，MD を実行してみてください.

c. 元々のプログラム (crudemd3.py) では計算セルの外に出た原子は戻ってきません.
このプログラムを改変し，セルの枠で原子が跳ね返って戻ってくるようにしてください.
※ 計算セルの一辺の大きさは，l/scl = 500/10 = 50 Å です.

0．下準備：原子配置のファイルを用意する

原子 3 つの初期配置と速度をファイル (initial.d) に書いておく。

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

initial. d の各行は 原子の *x* 座標、*y* 座標、速度の *x* 成分、*y* 成分の 4 つの項目からなり、半角スペースで区切られている必要がある。

※ 注意：各項目間を区切るスペースは 1 個．また，行の先頭にはスペースを入れない．これらを守らないと実行時にエラーになる．また，initial.d は python プログラムを実行するフォルダに置いておかなければならない.

以下では，３段階で MD のプログラムを作成してゆく：(1) キャンバスに原子を配置する，(2) 原子を移動させる，(3) MD を実行する．前のステップで作ったプログラムに新たな処理を追加して次のプログラムを作るという形になっており，crudemd1.py, crudemd2.py, crudemd3.py, というファイルを順番に作ってゆく．ただし，各プログラムは独立なので，必ずしも全てを順番どおりに作らなければいけないわけではない. 数値計算のプログラミングに慣れている人は (1) (2) を飛ばしていきなり (3) から始めても構わない.

1. キャンバスに原子を配置する (crudemd1.py)

物体を描画できるようにキャンバスを開き、initial.d ファイルから原子の位置と速度の情報を読み取って原子を初期位置に配置するプログラム例を crudemd1.py (次ページ) に示す。後々のため、ダミーのボタンも用意している（ここではボタンが押されても特に何もしない)。ここでは initial.d が 3 行なので原子

数は３つとなるが、行数を増やすことで原子数が増やせる。プログラム中では各行を読み込む度に配列サイズを増やしている (事前に配列のサイズを決めておく必要は無い)。座標の単位はÅ、速度の単位はÅ/s としている。

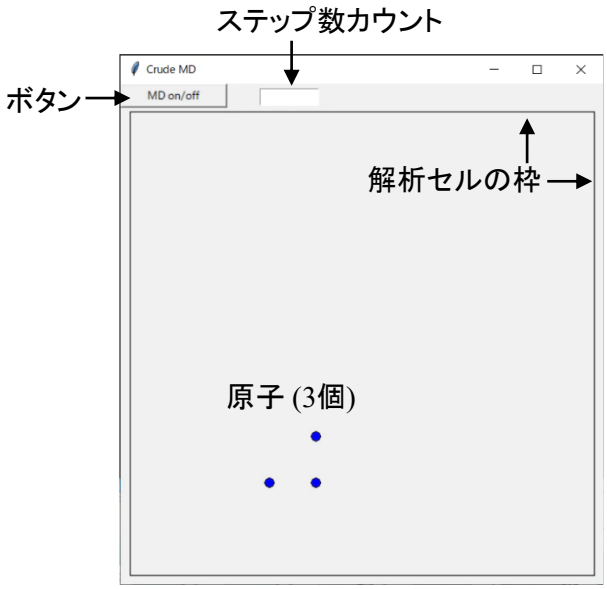

図 1. このようなウィンドウを作る.

```
#### crudemd1.py (#記号の後はコメント扱いになる)
# ← python なので，各行のインデント (行頭スペース) の量に注意.

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. ボタンイベントを取得して原子を移動 (crudemd2.py)

上記のプログラムを元に機能を追加。ボタンが押されたイベントを検出して、原子を斜め上方向にただ連続的に動かすようにしたもの。

※ 原子同士の相互作用に基づいて原子を動かしているわけではないので，まだ “MD” ではない.

```
#### crudemd2.py

import tkinter as tk

def dummy(event): # Button to switch on/off
    global imd    # to substitute value into 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. ボタンが押されたらＭＤをオン・オフする (crudemd3.py)

上記をさらに改変。ボタンが押されたら MD の on/off を切り替えるようにした。原子同士の相互作用を Morse ポテンシャルで計算し、Verlet 法で原子を動かすようになっている。

※ Morse ポテンシャルと Verlet 法については，末尾の Note 1, 2 を参照.

```
#### crudemd3.py

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

def dummy(event): # Switch MD on/off
    global imd   # to substitute value into 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 ポテンシャル

Morse ポテンシャルは原子間ポテンシャル関数の一種であり，原子間の相互作用を規定する．Morse ポテンシャルの関数形は以下のとおり．2 個の原子 i-j 間の距離 r_{ij} に応じてポテンシャルエネルギー$U(r_{ij})$が決まる．

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

描画すると下図 2 のようになる．$\varepsilon, \rho, \alpha$ はポテンシャルパラメータで，それぞれ，結合によるエネルギー利得，最安定距離，バネ定数 (安定点での曲率) に対応している．ポテンシャル U を r_{ij} で微分すると原子 i が原子 j から受ける力の大きさ f_{ij} が計算できる．また，f_{ij} の作用する方向は $\mathbf{r}_{ij}$ の方向と一致するので，$\mathbf{f}_{ij}$ が決まる．

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

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

原子 i が受ける正味の力 (合力) $\mathbf{f}_i$ は，全ての原子 j に対する $\mathbf{f}_{ij}$ の総和である．

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

各原子に働く力 $\mathbf{f}_i$ が分かれば，Newton の運動方程式 $\mathbf{f}_i = m_i\mathbf{a}_i$ から原子の加速度 $\mathbf{a}_i$ が得られ，それを数値積分することで，原子の運動 (＝位置 $\mathbf{r}_i$ と速度 $\mathbf{v}_i$ の時間変化) の様子を記述することができる．なお，初期条件 (位置と速度) は事前に明示的に与えておかなければならない．また，今回扱うのは単元系なので，質量 m_i は全ての原子 i で同一である．

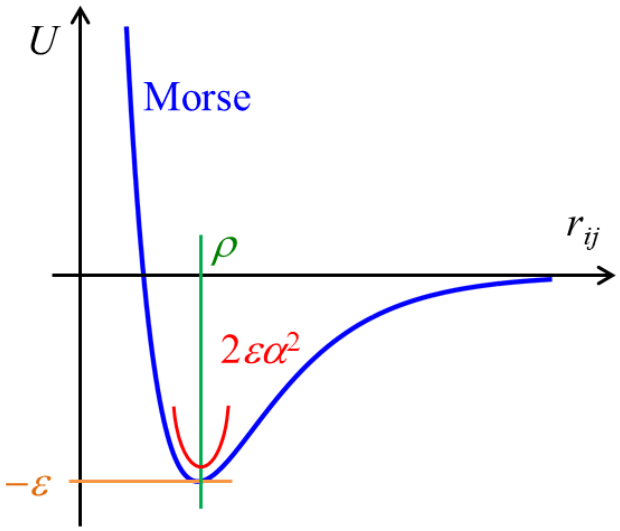

図 2．Morse ポテンシャルの関数形．

Note 2: Verlet 法

Verlet (ヴェルレ) 法は，微分方程式の数値解法の一種である．分子動力学法では，各原子の位置と速度の時間発展を計算するときに用いられる．時刻 t での位置 $\mathbf{r}_i(t)$ と速度 $\mathbf{v}_i(t)$ から微小時間Δt 経過後の $\mathbf{r}_i(t+\Delta t)$ と $\mathbf{v}_i(t+\Delta t)$ を求めるのが目的である．

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

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

プログラム crudemd3.py のメインループでは，この 2 式を使って $\mathbf{r}_i$, $\mathbf{v}_i$ を毎ステップ更新している．なお，$\mathbf{r}_i$ の式 (2-1) では右辺が時刻 t の情報しか含まない (時刻 t の情報のみで $\mathbf{r}_i(t+\Delta t)$ を決定できる) のに対して，$\mathbf{v}_i$ の式 (2-2) では時間が進む前の力 $\mathbf{f}_i(t)$ とΔt 進んだ後の力 $\mathbf{f}_i(t+\Delta t)$ の両方を使っている．そのため，プログラムでは以下の順序で処理を行っている．

1. $\mathbf{r}_i(t)$ を更新 [式 2-1, “crudemd3.py” の Verlet(1)].
2. $\mathbf{v}_i(t)$ の $\mathbf{f}_i(t)$ の部分を反映 [式 2-2, Verlet(1)].
3. $\mathbf{f}_i(t)$ を更新 ($t \rightarrow t+\Delta t$).
4. $\mathbf{v}_i(t)$ の $\mathbf{f}_i(t+\Delta t)$ の部分を反映 [式 2-2, Verlet(2)].

プログラム中で Verlet 法の処理が (1) (2) に分かれているのは，速度 $\mathbf{v}$ を更新する途中で力 $\mathbf{f}$ を更新する必要があることに起因している．

***

参考までに，$\mathbf{r}_i(t+\Delta t)$ の式を導出する．まず，$\mathbf{r}_i(t+\Delta t)$ をΔt で展開する．

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

この式に対して 3 次以上の項を切り捨てて，$\mathbf{v} = d\mathbf{r}/dt$, $\mathbf{a} = d^2\mathbf{r}/dt^2$ と $\mathbf{f} = m\mathbf{a}$ を使うと，式 2-1 が得られる．なお, $\mathbf{v}$ の式2-2 の導出はやや技巧的なので, ここでは省略する (興味があればトライしてみてください).