Dynamique moléculaire

2021

1 Principe

La dynamique moléculaire classique consiste à définir un potentiel (une énergie potentielle) empirique d'interaction entre des atomes à la dériver pour définir une force d'interaction. On peut alors calculer pour chaque atome les forces que les autres atomes exercent sur lui pour appliquer le principe fondamental de la dynamique pour calculer son accélération, sa vitesse et donc pour un pas de temps défini, sa position à l'itération suivante.

Le potentiel empirique le plus courant est le potentiel de Lennard-Jones. On va travailler en 2 dimensions, non pas qu'on ne puisse pas travailler en 3D, mais la 2D est plus "visuelle" elle permettra aux étudiants de mieux comprendre et de mieux voir les structures cristallines, les défauts, la déformations, etc...

1.1 Potentiel de Lennard-Jones

Le potentiel de Lennard-Jones s'écrit :

$$E_p(r) = 4 \times E_0 \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right) \tag{1}$$

E_0 est l'énergie de liaison, σ la position (autre que $r \to +\infty$) où $E_p = 0$. Le minimum se situe en $r = r_e = 2^{\frac{1}{6}}\sigma$, et vaut $E_p = -E_0$. r_e est le rayon d'équilibre où l'énergie est minimum.

1.2 Force associée

En dérivant l'énergie potentielle, on obtient une force :

$$F(r) = -4 \times E_0 \left(6 \times \frac{\sigma^6}{r^7} - 12 \times \frac{\sigma^{12}}{r^{13}} \right) \tag{2}$$

et donc :

$$F_x(r) = F(r) \, \frac{\vec{x}}{r} \tag{3}$$

$$F_y(r) = F(r) \, \frac{\vec{y}}{r} \tag{4}$$

1.3 Raideur associée

On peut calculer une raideur de la force à la position d'équilibre. Elle s'écrit (c'est la dérivée de la force à la position d'équilibre) :

$$k = 36 \times \frac{2^{2/3} \, E_0}{\sigma^2} \tag{5}$$

1.4 Période d'oscillation

Cette raideur permet de calculer la période d'oscillation de l'atome dans son puits de potentiel. Cela permettra de définir un pas de temps de calcul tel qu'il soit très inférieur à cette période.

La pulsation d'oscillation inter-atomique est de :

$$\omega = \sqrt{\frac{k}{m}} \tag{6}$$

m la masse de l'atome. On en déduit la période d'oscillation :

$$T_{oscill} = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}} \tag{7}$$

$$T_{oscill} = \frac{\pi\sqrt{\frac{m\ \sigma^2}{E_0}}}{3 \times 2^{1/3}} \tag{8}$$

1.5 Paramètre de l'argon

On utilisera pour les simulations les paramètres empirique de l'argon (parce qu'ils sont connus) :

$$\sigma = 3.4 \times 10^{-10}\ m \tag{9}$$

$$E_0 = 1.65 \times 10^{-21}\ J \tag{10}$$

$$m = 6.69 \times 10^{-26}\ kg \tag{11}$$

1.6 Rayon de coupure et énergie

Le rayon de coupure permet de négliger les interactions entre deux atomes "trop" distants, c'est-à-dire si loin l'un de l'autre que la force qu'ils exercent entre eux est négligeable. Usuellement, le rayon de coupure est défini comme deux fois le rayon d'équilibre, soit :

$$r_{cut} = 2^{\frac{7}{6}}\sigma \tag{12}$$

L'énergie potentielle au point de coupure vaut :

$$E_p(r_{cut}) = -\frac{127}{4096}\ E_0 \tag{13}$$

Pour éviter d'avoir une énergie non-nulle au point de coupure, on "remonte" le potentiel de cette valeur, et donc :

$$E_p(r) = E_0 \times \left(4\ \left(\ \left(\frac{\sigma}{r}\right)^{12}\ -\ \left(\frac{\sigma}{r}\right)^{6}\right) + \frac{127}{4096}\right) \tag{14}$$

2 Programme Python

Je décris ici le programme Initialisation.py.

Chargement des packages utiles aux programmes :

```
import numpy as np
import matplotlib.pyplot as plt
import itertools # positionner les atomes
```

Définition du nombre de pas de temps calculés et du nombre d'atomes. Les atomes seront positionnés au départ sur une "grille" carrée dont on définit le nombre d'atomes par arrêtes :

```
####### paramètres de simulation: nombre d'atomes, de pas
# nombre de pas
npas = 1000
# nombre d'atomes par arrête
nba = 8
# nombre total d'atomes
nbtot = nba**2 ;
```

Définition des paramètres du potentiel de Lennard-Jones. J'ai mis un rayon de coupure à $2.5 * r_e$ ce sera à modifier pour respecter l'équation 12 :

```
# parametres du potentiel de Lennard-Jones (unite SI, pour l'argon)
sigma = 3.4e-10 #metre
epsilon = 1.65e-21 # joules
m = 6.69e-26 # kilogrammes
# distance du minimum de potentiel
re = 2.0**(1.0/6.0)*sigma
print("distance interatomique: %s Angstrom"%(re*1e10))
#
# rayon de coupure
rcut = 2.5*re
```

On va mettre les atomes dans une boîte, c'est à dire que quand ils vont toucher les bords, ils vont rebondir dessus. Cette boîte est définit par la paramètre *fronti*, un entier. La boîte est plus large que le carré d'origine où sont mis les atomes, et est plus large de $fronti \times r_e$ de chaque côté du carré :

```
# frontière de boite
fronti = 2
```

On calcule la période d'oscillation d'un atome et on définit le pas de temps comme très inférieur à ça (50 à 100 fois inférieur en règle général) :

```
# periode d'oscillation
puls0 = np.sqrt((57.1464*epsilon/(sigma**2.0))/m)
freq0 = puls0/(2.0*3.14159)
peri0 =1/freq0
print("periode oscillation atomique: %s s"%(peri0))
print("periode oscillation atomique: %s ps"%(peri0*1e12))
##########################################################################
# pas de temps
dt = peri0/75
print("pas de temps: %s ps"%(dt*1e12))
#
```

On va exporter des images de la simulation. Le booléen *Film* définit si on le fait ou non (c'est gourmand en temps de calcul si on le fait !) et *pfilm* définit qu'on exporte une image tous les *pfilm* pas de temps.

```
# paramètres décidant du "film"
pfilm = 10
klist = range(0,npas,pfilm)
Film = False
```

On calcule l'expression de $F(r)$ (équation 2) :

```
# definition de la force
def F(r):
    val = 4.0*epsilon*(6*sigma**6/(r**7.0)-12.0*sigma**12.0/(r**13.0))
    return val
```

On positionne les atomes sur la grille de départ. On les espace d'une distance légèrement différente de r_e, sinon ils seraient à l'équilibre en terme de force et ne bougeraient pas. On voit là que le programme est écrit pour faire de la 3D s'il faut, mais on fonctionne en 2D. Le résultat est présenté sur la Figure 1 :

```
# rayon intiale
rini = 1.1*re
#
# liste de positions
listplos = np.arange(0,nba)
# placement des atomes
coor = np.array(list(itertools.product(listplos, listplos)))
coor = coor*rini ;
# coor[i]= les trois coordonnées de la particule i
# Pour tracer:
coort=np.transpose(coor) # coort[0] tous les x, coor[1] tous les y, coor[2] tous les
                                                    z
# positions initiales  en figure
plt.figure(0)
plt.plot(coort[0],coort[1] ,"ro")
plt.title("Position initiale")
plt.ylim(-fronti*re,(nba-1)*re+fronti*re)
plt.xlim(-fronti*re,(nba-1)*re+fronti*re)
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)
plt.show()
```

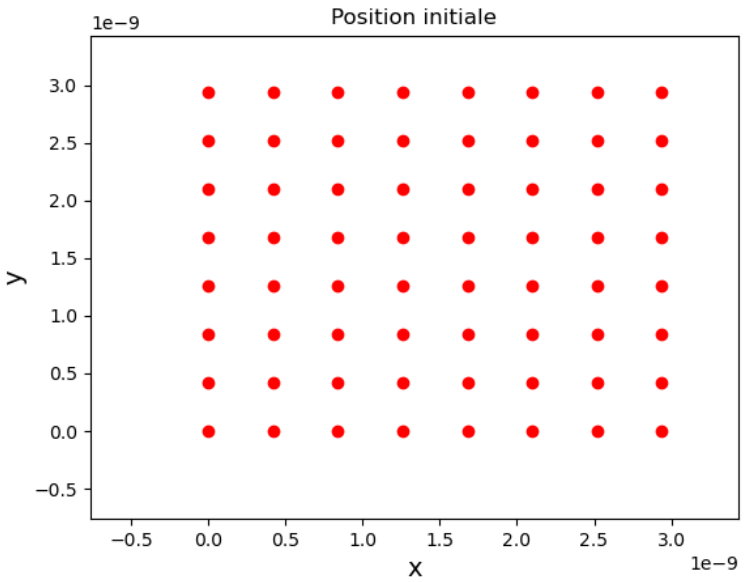

FIGURE 1 – Placement initial des atomes.

On va maintenant définir une fonction pour calculer les forces que subit un atome de la part de tous les autres. Un atome j subit une force de la part des autres atomes dont l'intensité est donné par la distance entre lui et les autres atomes. On crée d'abord une liste indexée par i des distances r_i entre l'atome j et tous les autres atomes i :

```
def Fj(j) :
    Fxi = 0
    Fyi = 0
    disti = coor-coor[j] # contient les distance x et y de la particule j aux autres
    posti = np.transpose(disti)
    xi = posti[0] # distance selon x
    yi = posti[1] # distance selon y
    disti2 = disti**2 # mets ces distances au carré
    ri2 = np.sum(disti2,1) # contient les distances ri au carré de la particle j aux
                                                    autres
    ri = ri2**0.5 # et les ri
```

Le soucis c'est que cette liste contient la distance entre l'atome j et... lui-même... soit une distance nulle, qui, si on calcule la force associée, donne une force infinie. Je change donc : si jamais la distance est nulle je la remplace par la valeur de r_{cut} (équation 12). Ensuite, pour tous les autres atomes dont la distance à l'atome j est plus grande ou égale à r_{cut} je ne vais pas calculer de force (je la mettrais nulle). Donc autant faire comme si tous ces atomes étaient à une distance r_{cut} pour bien les identifier. Je remplace donc toutes les distances plus grandes que r_{cut} par r_{cut} :

```
    ri[ri==0] = rcut # remplace les distances nulles par rcut
    ri[ri>rcut] = rcut #  remplace les distances >rcut par rcut
```

Ainsi je ne calcule de force ni pour les atomes trop loin de l'atome j ni l'interaction de l'atome avec lui-même.

Calcul des forces selon chaque direction :

```
    Fi = F(ri) # toutes les forces subie par j des particules i
    Fi[Fi==F(rcut)] = 0 # remplace les forces pour r=rcut par 0
    Fxi = xi/ri*Fi
    Fyi = yi/ri*Fi
    Fx = np.sum(Fxi)
    Fy = np.sum(Fyi)
    return np.array([Fx,Fy])
```

On peut donner une vitesse initiale aux atomes, mais ici je la mets nulles, en créant le vecteur vitesse par la multiplication des vecteurs positions par 0 :

```
v = coor*0 ;
v2 = v # vitesse au demi pas
```

v_2 est une vitesse au demi-pas de temps pour la méthode de Verlet que je présente ensuite. Peu importe sa valeur, c'est juste pour créer un array à la bonne dimension.

J'initialise des listes pour y stocker des valeurs ensuite :

```
Ec = np.zeros(nbtot) # initialisation de l'énergie cinétique de chaque particule
liaison = np.zeros(nbtot) # initialisation de liaison chaque particule
pk = np.zeros(npas) # liste des pas de temps
Eck = np.zeros(npas) # énergie cinétique totale à chaque pas
liaisonk = np.zeros(npas) # nombre total d'atomes liés à chaque pas
```

2.1 Mouvement global

Avant de poursuivre le programme, il faut que j'explique certains points. Le premier est le problème du mouvement global. L'ensemble des atomes forme un corps qui peut se mettre globalement à bouger dans l'espace : mouvement de translation et mouvement de rotation. C'est d'une part assez pénible visuellement et d'autre part très gênant pour calculer la température. La température se calcule d'après l'énergie cinétique moyenne des atomes mouvement du corps exclu (en retirant la composante de vitesse des atomes qui correspond au mouvement global de l'ensemble formé par les atomes).

Il existe des stratégie propre mais lourde en terme de calcul pour déterminer la part d'énergie cinétique des atomes qui n'est pas de l'agitation thermique, mais un mouvement global du corps. Ma stratégie est plus "bourrine", mais suffisante pour une simulation à visée pédagogique (qualitative pas quantitative), légère en calcul et efficace :

Je prends deux atomes au hasard et je les immobilise. Ca bloque la translation et la rotation. Il n'y a plus de mouvement global du corps possible.

2.2 Méthode de Verlet

Le second point est la méthode de calcul des nouvelles positions des atomes au pas suivant. La méthode de Verlet est très bien expliqué sur Wikipedia. C'est la méthode standard de la dynamique

moléculaire classique. Elle se base sur un développement limité de l'accélération. Soit $\vec{v}(t)$ la vitesse de l'atome à l'instant t. On calcule la vitesse au demi-pas de temps supérieur :

$$\vec{v}(t + \Delta t/2) = \vec{v}(t) + \frac{\vec{F}(t)}{2m}\Delta t \tag{15}$$

Δt le pas de temps. Puis on calcule la position au pas de temps $t + \Delta t$:

$$\vec{OM}(t + \Delta t) = \vec{OM}(t) + \vec{v}(t + \Delta t/2) \times dt \tag{16}$$

On recalcule la force pour ce nouveau pas de temps $\vec{F}(t + \Delta t)$ par les positons des atomes, pour en déduire la vitesse à ce nouveau pas de temps :

$$\vec{v}(t + \Delta t) = \vec{v}(t + \Delta t/2) + \frac{\vec{F}(t + \Delta t)}{2m}\Delta t \tag{17}$$

2.3 Phase principale du programme

On arrive à la partir principale du programme où on va calculer pas par pas les positions des atomes dans le temps. k est l'itérant des pas de temps allant de 0 au nombre de pas de temps :

```
for k in range(npas):
```

j est l'itérant des atomes allant de l'atome 0 au dernier

```
  for j in range (nbtot) :
```

Je réalise les boucle de Verlet en appelant ma fonction de force pour calculer à chaque pas k pour chaque atome j sa nouvelle position et sa nouvelle vitesse. J'immobilise l'atome $j = 0$ et l'atome $j =$ nombre d'atomes/2 (parfaitement arbitrairement !) pour supprimer le mouvement de corps solide :

```
  v2[j] = v[j]+Fj(j)/(2*m)*dt
         if j==int(nbtot/2) :
              v2[j] = 0*v2[j] # immobilise une particule (empeche les mouvements de
                                                    corps solide)
          coor[j] = coor[j]+v2[j]*dt
          Fjval = Fj(j) # stocke la valeur de Fj(j) pour ne pas la recalculer
          v[j] = v2[j]+Fjval/(2*m)*dt
          if j==0 :
              v[j] = 0*v[j] # immobilise une particule (empeche les mouvements de
                                                    corps solide)
```

Je prends en compte la boîte et je fais rebondir les atome sur la paroi. Le principe est simple : si une des coordonnées de l'atome est hors de la boîte, j'inverse le signe de la vitesse qui correspond à cette coordonnée.

```
        ### création de la boîte
        if coor[j,0]>(nba-1)*re+fronti*re or coor[j,0]<-fronti*re :
            v[j,0] = - v[j,0]
        if coor[j,1]>(nba-1)*re+fronti*re or coor[j,1]<-fronti*re :
            v[j,1] = - v[j,1]
        # fin boite
```

La fin de la boucle calcule des valeurs intéressantes : le nombre de liaisons dans le système et l'énergie cinétique. Pour un atome, je regarde si jamais il ne subit aucune force (il s'est échappé de la "masse" des autres atomes et est trop loin) et est considéré donc comme n'ayant pas de liaison. Dans ce cas, je l'exclue du calcul d'énergie cinétique, sinon, je le prends en compte et je le compte que 1 atome ayant des liaisons avec au moins un autre :

```
        if np.sum(Fjval**2) == 0 :
            Ec[j] = 0 # ne prend pas l'EC de la particule en compte si elle n'a plus
                                                    de liaison
            liaison[j] = 0
        else:
```

```
            Ec[j] = 0.5*m*(v[j,0]**2+v[j,1]**2)
            liaison[j] = 1
    # Eck
    Eck[k] = np.sum(Ec)
    liaisonk[k] = np.sum(liaison)
    pk[k] = k
```

Je crée le film de la simulation. On va créer les graphs de positions des atomes à chaque pas demandé, et les enregistrer en fichier png avec un nom incrémenté par le pas k :

```
    if Film : # booléen vrai ou faux
        if k in klist:
            coort=np.transpose(coor)
            plt.ioff() # pour ne pas afficher les graphs)
            fig = plt.figure()
            plt.plot(coort[0],coort[1] ,"ro")
            plt.title("Position initiale")
            plt.ylim(-fronti*re,(nba-1)*re+fronti*re)
            plt.xlim(-fronti*re,(nba-1)*re+fronti*re)
            plt.xlabel("x", fontsize=14)
            plt.ylabel("y", fontsize=14)
            fig.savefig('StorePic/MD-picture%s.png' % k) # sauvegarde incrementale
            plt.close(fig) # fermeture du graph
```

La figure est créée et enregistrée, mais jamais ouverte dans une fenêtre par python.

Je trace des jolies figures de différentes choses à la fin de ma boucle de pas de temps :

```
 ################### Figures ###################################################
coort=np.transpose(coor)
plt.figure(1)
plt.plot(coort[0],coort[1] ,"ro")
plt.title("Position Finale")
plt.ylim(-fronti*re,(nba-1)*re+fronti*re)
plt.xlim(-fronti*re,(nba-1)*re+fronti*re)
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)
plt.show()
#
plt.figure(2)
plt.plot(pk,Eck ,"b")
plt.title("Energie")
plt.xlabel("Pas", fontsize=14)
plt.ylabel("Energie cinétique", fontsize=14)
plt.show()
#
plt.figure(3)
plt.plot(pk,liaisonk ,"b")
plt.title("Liaison")
plt.xlabel("Pas", fontsize=14)
plt.ylabel("Nombre atomes en liaison", fontsize=14)
plt.show()
#
plt.figure(4)
plt.plot(np.arange(nbtot),Ec ,"bo")
plt.title("Distribution")
plt.xlabel("Particule", fontsize=14)
plt.ylabel("Ec", fontsize=14)
plt.show()
```

J'exporte les dernières positions des atomes, leurs dernières vitesses et les paramètres de simulation utilisés (paramètres de Lennard-Jones, pas de temps, r_{cut}...) pour pouvoir redémarrer une nouvelle simulation avec ces paramètres :

```
# sauvegarde
```

```
np.save('/Users/yannguegueen/Documents/Boulot/DEM/MD2DSimple/Etapes/2-Stabi/coordonne.
                                        npy', coor)
np.save('/Users/yannguegueen/Documents/Boulot/DEM/MD2DSimple/Etapes/2-Stabi/vitesse.
                                        npy', v)
# paramètre
parameter = np.array([sigma,epsilon,m,re,rcut,dt])
np.save('/Users/yannguegueen/Documents/Boulot/DEM/MD2DSimple/Etapes/2-Stabi/parameter.
                                        npy', parameter)
```