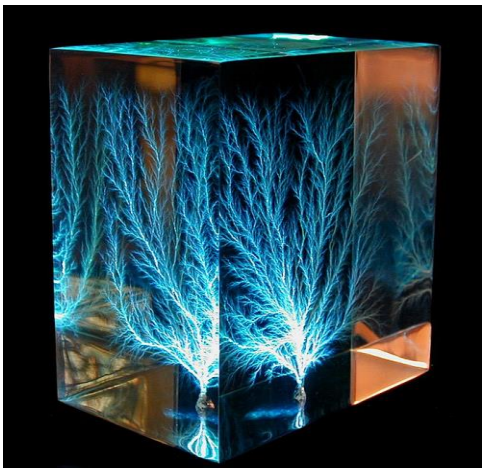
DIFFUSION-LIMITES AGGREGATION

ABDALLAH NASSIB/ HARRISON DASHIELL

Dans ce rapport on va expliquer la modélisation de la diffusion limitée par agrégation (DLA), c'est un phénomène où les particules subissant une marche aléatoire, à cause leur mouvement Brownien, se regroupent pour faire des agrégats.

Cette théorie est applicable à l'agrégation dans tout système où la diffusion est le principal moyen de transport des particules. C'est un phénomène qu'ont peut observer dans des nombreux domaines tel que le claquage diélectrique.



Des figures de Lichtenberg modernes, dans un bloc d'acrylique transparent. Une hypothèse est que le modèle de décharge en fractale s'applique jusqu'au niveau moléculaire.

Dans notre modélisation simplifiée, nous prenons une boite fermée dont une particule de vitesse nulle et de masse infinie est positionnée au centre, autour de laquelle des particules entrent en collision.

Quand ces particules tapent la particule centrale elles y seront collées.

Du fait que la particule centrale est beaucoup plus grande en masse que les particules qui la tapent, une figure de fractale se formera à la fin de la simulation.

La dernière particule entrera en collision avec la figure de fractal sans s'y coller.

Enfin nous analyserons notre simulation sous l'angle de la formation des planétésimales, les composantes primaires des planètes, qui fonctionnent exactement de ma même manière (formation de fractal puis rejet de particules)

La modélisation de ce phénomène :

Pour commencer, nous changeons le code déjà manipulé pendant les séances de simulation numérique qui traite des monomères qui se tapent entre elle dans une boite fermée.

Nous nous servons de la Class Monomères qui contient tous les variables et les fonctions nécessaires pour notre programme comme le nombre des monomères, les limites de la boîte, les moments de collisions…

Ainsi, pour appeler ces variables dans la suite du code il suffit de taper le nom de la variable précédé par SELF, d'où l'intérêt de la Class comme Objet.

On définit notre système de monomères avec la fonction suivante :

```
  def assignRadiaiMassesVelocities(self, NumberMono_per_kind = np.array([4]), Radiai_per_kind
= 0.5*np.ones(1), Densities_per_kind = np.ones(1), k_BT = 1 ):

        assert( sum(NumberMono_per_kind) == self.NM )
        assert( isinstance(Radiai per kind,np.ndarray) and (Radiai per kind.ndim == 1) )
        assert( (Radiai_per_kind.shape == NumberMono_per_kind.shape) and (Radiai_per_kind.shap
e == Densities per kind.shape))

        total=0
        for i, number in enumerate(NumberMono per kind): #we are filling up radii and masses,
the infinite density will give an infinite mass
           self.rad [total:total+number]=Radiai per kind[i]
           self.mass[total:total+number]=Densities_per_kind[i]*(np.pi*(Radiai_per_kind[i]))
           total+=number
```

On définit aussi la masse, le rayon et la vitesse des monomères en utilisant des boucles qui indexent les monomères après chaque mouvement aléatoire défini également avec la fonction assignRandomMonoPos :

```
 total=0

        for i, number in enumerate(NumberMono per kind): #we are filling up radiai and masses,
 the infinite density will give an infinite mass
           self.rad [total:total+number]=Radiai_per_kind[i]
           self.mass[total:total+number]=Densities per kind[i]*(np.pi*(Radiai per kind[i]))
           total+=number


           assert( k_BT > 0 )

        for i in range(self.NM):
            rand=np.random.random()*2*np.pi #we take random angles
            self.vel[i]=[np.cos(rand),np.sin(rand)]
            if self.mass[i] >1000:
               self.vel[i] = [0.000001,0.000001]  #to avoid having a division by zero later on
 we give the seed particle a very very slow speed but not zero
            else:
                vitesse = np.sqrt( k BT *2 / self.mass[i]) #formula of the speed
                self.vel[i] *= vitesse        #random angles gives random speed directions to
the particles


def assignRandomMonoPos(self, start index = 0 ):

        assert ( min(self.rad) > 0 ) #otherwise not initialized
        new_mono, infiniteLoopTest = start_index, 0
        if self.mass[new_mono]>1000: #if it is the seed particle then put it in the middle
           self.pos[new_mono,:]= [(L_xMax-L_xMin)/2 ,(L_yMax-L_yMin)/2 ]
```

```
        else: #otherwise assign a random position
            self.pos[new_mono,:] = np.random.rand(1,2)*[ L_xMax - self.rad[new_mono], L_yMax -
 self.rad[new mono]] + [self.rad[new mono], self.rad[new mono]]
        new mono +=1 #initialize at 1 to compare with "old mono"
        while new_mono < self.NM and infiniteLoopTest < 10**4:
            if self.mass[new mono]>1000:
               self.pos[new_mono,:]= [(L_xMax-L_xMin)/2 ,(L_yMax-L_yMin)/2 ]
            else:
                self.pos[new_mono,:] = np.random.rand(1,2)*[ L_xMax -
 self.rad[new_mono], L_yMax - self.rad[new_mono]] + [self.rad[new_mono], self.rad[new_mono]]
            NoOverlap = True
            old_mono = 0
            while old mono < new mono and NoOverlap:
                dist = np.sqrt((self.pos[new_mono,0]-
self.pos[old mono,0])**2+(self.pos[new mono,1]-self.pos[old mono,1])**2)
                doubleR = self.rad[new_mono]+self.rad[old_mono]
                if dist < doubleR: #the sum of the particles radii must be superior to the d
istance that separes their center
                   NoOverlap = False
                else:
                   old_mono += 1
            if NoOverlap:
                new_mono += 1
                infiniteLoopTest = 0
            else:
                infiniteLoopTest += 1
```

Nous essayerons de mettre le SELF_VEL à zéro pour les particules qui touchent la particule centrale dans la suite de notre code.

Avec la function Wall_time et Mono_Paire_time on définie les conditions de collisions avec les murs de la boîte et des monomères entre eux :

```
 def Wall_time(self):


        coll_condition = np.where( self.vel > 0, self.BoxLimMax-
self.rad[:,np.newaxis], self.BoxLimMin+self.rad[:,np.newaxis]) #the particle touches the wall
(radius touching the wall)
        dt_List = ( coll_condition -
 self.pos) / self.vel #time before impact following the position and the wall
        MinTimeIndex = np.argmin(  dt List ) #finding the smallest time to know which is the f
irst particle to hit the wall
        collision_disk = MinTimeIndex // 2 # index of new position after collision
        wall direction = MinTimeIndex % 2 #index of new direction after collision

        self.next wall coll.dt = dt List[collision disk][wall direction] # time before impact
        self.next wall coll.mono 1 = collision disk
        self.next wall coll.w dir = wall direction

        print(self.next_wall_coll)


    def Mono pair time(self):

        mono_i = self.mono_pairs[:,0]
        mono j = self.mono pairs[:,1]

        d vx=self.vel[mono i,0]-self.vel[mono j,0]
        d_vy=self.vel[mono_i,1]-self.vel[mono_j,1]
        d x0=self.pos[mono i,0]-self.pos[mono j,0]
        d_y0=self.pos[mono_i,1]-self.pos[mono_j,1]

        a=d vx**2+d vy**2
        b=2*(d_vx*d_x0+d_vy*d_y0)
        c=d x0**2+d y0**2-(self.rad[mono i]+self.rad[mono j])**2

        delta = b**2-
4*a*c                                          #all of this calculus has be done during clas
s, it can be looked up in the collaboratory

        condition = np.empty((1,len(b)))
        for i in range(len(delta)):   #our condition for the np.where
            condition[0,i] = delta[i] >=0 and b[i] < 0
```

```
        deltat = np.where(condition ,(-b-
np.sqrt(delta))/(2*a),np.array([np.inf]*len(condition))) #a simple physical analysis shows tha
t we need b<0

        index_min = np.argmin(deltat[0])

        self.next_mono_coll.dt = deltat[0][index_min]
        self.next_mono_coll.mono_1 = self.mono_pairs[index_min,0]
        self.next_mono_coll.mono_2 = self.mono_pairs[index_min,1]

        print(self.next_mono_coll)
```

Les conditions de collisions avec le mur ont été étudiées pendant les séances de simulations afin de trouver les signes des coefficients de l'équation du second degré du temps de collision.

```
 def compute_next_event(self):

        self.Mono_pair_time()
        self.Wall_time()
        if self.next_wall_coll.dt < self.next_mono_coll.dt :
            return self.next_wall_coll
        else :
            return self.next_mono_coll
```

Une fois qu'on a défini des variables et des fonctions qui décrivent les conditions des états statiques dans la boîte, nous allons définir une fonction de dynamisme qui modélise le mouvement des monomères quand ils tapent un mur ou une particule.

Pour définir la nouvelle position de la particule, il faut définir aussi une nouvelle vitesse du monomère après collision :

Nous imposons une vitesse de 0.00001 pour les particules qui touchent la particule centrale pour ne pas avoir une division par 0 dans le code, ce qui était le cas pour une vitesse nulle.

```
 def compute_new_velocities(self, next_event):


        if next_event.Type == 'wall' :
                self.vel[next_event.mono_1,next_event.w_dir]= self.vel[next_event.mono_1,next_
event.w_dir]*-1

        else:


            mono_1 = next_event.mono_1
            mono_2 = next_event.mono_2

            if self.NM>2 : #as long as we didn't reach the last particle
                if (self.mass[mono_1]>1000) or (self.mass[mono_2]>1000) : #every particle that
 hits the fractal must stay stuck to it
                    self.vel[mono_1]=[0.000001,0.000001]
                    self.vel[mono_2]=[0.000001,0.000001]
                    self.mass[mono_1]=10000
                    self.mass[mono_2]=10000
                    self.NM -= 1

                else : #if they just hit each other outside the seed particle then it's an une
lastic collison
                    diff = self.pos[mono_2] - self.pos[mono_1]
                    diff=diff/np.linalg.norm(diff)
                    m1=self.mass[mono_1]
```

```
                    m2=self.mass[mono_2]
                    Dv=self.vel[mono_1]-self.vel[mono_2]
                    self.vel[mono_1]=self.vel[mono_1] - (2*m2)/(m1+m2)*np.inner(diff,Dv)*diff
                    self.vel[mono_2]=self.vel[mono_2] + (2*m1)/(m1+m2)*np.inner(diff,Dv)*diff

            else : #once we arrive to the last particle, it all happens like a simple collisio
n, without sticking to the fractal
                diff = self.pos[mono_2] - self.pos[mono_1]
                diff=diff/np.linalg.norm(diff)
                m1=self.mass[mono_1]
                m2=self.mass[mono_2]
                Dv=self.vel[mono_1]-self.vel[mono_2]
                self.vel[mono_1]=self.vel[mono_1] - (2*m2)/(m1+m2)*np.inner(diff,Dv)*diff
                self.vel[mono_2]=self.vel[mono_2] + (2*m1)/(m1+m2)*np.inner(diff,Dv)*diff
```

Et la fonction snapshot est la fonction d'animation de notre simulation.

```
def   def snapshot(self, FileName = './snapshot.png', Title = '$t = $?'):
```

Le code expliqué ci-dessus est le Particule_class.

Nous allons maintenant expliquer les principales fonctions et boucles de event_class_monomers

La fonction Molecular_dynamics_loop impose le pas de temps constant :

On a une simulation avec un pas de temps constant dt= 0.02, donc chaque nouveau frame est obtenu à dt, mais le changement de vitesse est lié à next-event_dt, donc si le nouveau changement de vitesse a lieu avant le next_frame_dt, il faut le ramener à ce next_frame_dt. Ainsi la boucle while dit que si le nouveau time_change_Vel est inférieur à time du next frame, on ramène la position du monomère à la position qui correspond au vel_change puis on le décale vers la position qui correspond au time du next_frame en calculant le time entre les deux(time remaining).

On veille à remettre le next event à sa place en retranchant ce time remaining, sinon le next event sera décalé d'un time remaining dans la boucle.

```
t = 0.0
dt = 0.02
NumberOfFrames = 1500

next_event = mols.compute_next_event()
def MolecularDynamicsLoop( frame ):

    global t, mols, next_event

    next_time_vel_change = t + next_event.dt
    future time next frame = t + dt
    while next_time_vel_change < future_time_next_frame :
        mols.pos += mols.vel * next_event.dt
        t += next_event.dt
        mols.compute_new_velocities(next_event)
        next event=mols.compute next event()
        next_time_vel_change=t+next_event.dt

    timeremaining = future_time_next_frame - t
    mols.pos += timeremaining*mols.vel
    t += timeremaining
```

```
    next_event.dt -= timeremaining

    if mols.NM > 2:
        plt.title( '$t = %.4f$, remaining frames = %d, Planetesimal is forming! ' % (t, Number
OfFrames-(frame+1)) )
    else:
        plt.title( '$t = %.4f$, remaining frames = %d, Saffronov number higher than 1 ' % (t,
NumberOfFrames-(frame+1)) )
    collection.set_offsets( mols.pos )
    return collection
```

Results and conclusions in other pdf