

Gravitation à N corps

Anne Schliebach, Zack Ribeiro

7 mai 2024

Résumé

Le problème à N corps est depuis longtemps abordé numériquement ainsi les algorithmes/optimisations le traitant sont en grand nombres avec des degrés de complexité divers et des approches différentes. Ce document présente les bases du problème et de sa résolution numérique la plus “naïve” : Particle-Particle (PP), mais aussi de deux des grandes approches moins coûteuses pour le traiter : Tree-Code/QuadTree (TC/QT) & Particle-Mesh (PM/PIC).

Le code est open-source¹ et est en Python, des rendus de simulation sont disponibles sur Youtube².

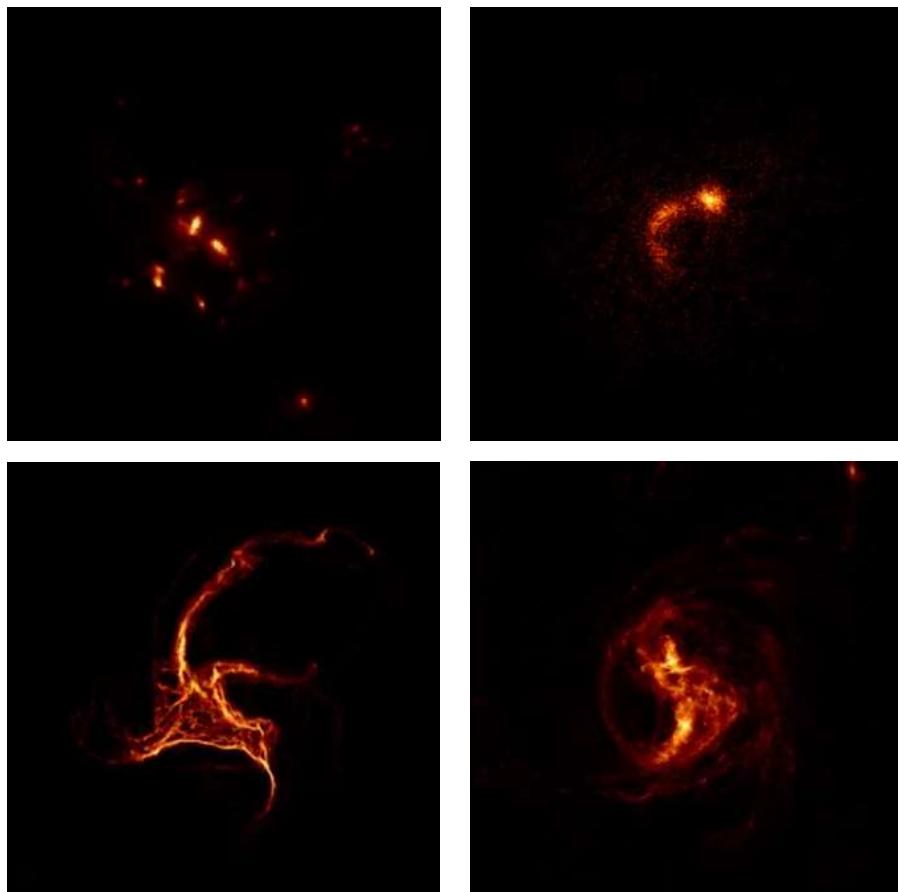


TABLE 1 – Divers simulations (QT & PIC)

1. <https://github.com/EazyEnder/GravSim>

2. <https://www.youtube.com/@zackrib485>

Table des matières

I Approche directe	3
1 Problème physique et Outils à la résolution	3
1.1 Théorie du problème	3
1.2 Méthodes d'intégration & de résolutions	3
1.2.1 Euler & Verlet	3
1.2.2 Leapfrog	3
1.2.3 Runge-Kutta	4
1.2.4 Gauss-Seidel	4
1.3 Objets de la simulation	4
2 Approche Particle-Particle (PP)	5
2.1 Mise en place	5
2.1.1 Algorithme	5
2.1.2 Observations	5
2.2 Problèmes	6
2.2.1 Divergence de la vitesse des particules	7
2.2.2 Conservation de l'énergie	8
2.3 Résultats	10
II Approche avancée	11
3 Tree-Code (TC)	11
3.1 QuadTree	11
3.1.1 Principe	11
3.1.2 Construction	11
3.2 Utilisation	13
3.2.1 Centre de masse	13
3.2.2 Calcul de la Force	13
3.3 Résultats	15
3.4 Axes d'améliorations possibles	16
4 Particle-Mesh (PM)	16
4.1 Mise en place 1D	16
4.2 Mise en place 2D	17
4.2.1 Système à résoudre	17
4.2.2 Calcul de ρ	18
4.3 Résultats	18
4.4 Axes d'améliorations possibles	18
A Pseudo-Code → Python	20
B MultiProcessing CPU	21

Première partie

Approche directe

1 Problème physique et Outils à la résolution

1.1 Théorie du problème

Soient N masses m_i en interaction gravitationnelle, quelle est la trajectoire des particules ?

La difficulté de la résolution analytique du problème vient de sa grande sensibilité aux conditions initiales. En effet, les équations données par la seconde loi de Newton forment un système couplé entre les positions \vec{q}_i des particules. Ce couplage est dû à l'expression de la force gravitationnelle : $(i, j \in \mathbb{N}, 0 \leq i, j < N)$:

$$\vec{F}_i \stackrel{\text{Force}}{=} -G \times \sum_{j \neq i} \frac{m_j m_i (\vec{q}_i - \vec{q}_j)}{\|\vec{q}_i - \vec{q}_j\|^3} \stackrel{\text{Newton}}{=} m_i \frac{d^2 \vec{q}_i}{dt^2} \quad (1)$$

, ce qui donne $3N$ équations pour $3N$ paramètres (positions en 3D). Il y a donc une unique solution mais le couplage des positions donne une trop grande complexité pour les expliciter. Pour des cas particuliers il est cependant possible de traiter le problème en utilisant une approche perturbative. Par exemple si il y a $N=3$ corps et l'un d'eux a une masse bien inférieure aux deux autres.

L'intérêt de l'approche numérique est qu'il suffit de l'expression analytique de la force et d'une méthode d'intégration, c'est à dire un algorithme, pour obtenir l'évolution des positions dans le temps via les équations. Il est possible d'obtenir ces trajectoires sur un intervalle de temps avec une certaine précision/un certains pas de temps.

1.2 Méthodes d'intégration & de résolutions

Pour faire évoluer les particules sur le temps, et donc résoudre le système d'équations, il faut faire appel à des méthodes de résolution d'équations différentielles (~EDP). Ces méthodes sont multiples et possèdent tous un certain degré d'erreur, de stabilité et de convergence. Une méthode plus précise va permettre d'être plus proche de la solution analytique. La convergence concerne de la solution vers la solution analytique pour des pas de discrétisation infiniment petits.

Nous utiliserons la méthode des différences finies pour approximer les dérivées. Les méthodes seront explicites.

1.2.1 Euler & Verlet

La méthode d'Euler est la plus simple des méthodes d'intégrations. En effet c'est un développement limité à l'ordre 1 : $q(t) = q_0 + \dot{q}t + o(t)$, où q_0 est la position initiale et $\dot{q} = \frac{dq}{dt} = v(t)$ la vitesse de la particule au temps t . En discrétilisant le temps nous obtenons : $q_{t+1} = q_t + v_t \times \Delta t$ avec Δt le pas de discrétisation du temps. Ce qui revient à faire $\Delta q = \frac{dq}{dt} \Delta t$. De même pour la vitesse : $v_{t+1} = v_t + a_t \times \Delta t$ avec a_t l'accélération au temps t . Cette accélération qui se calcule avec la force : $a_t = \sum_m F_m$. Ainsi, pour le cas de la gravitation, la méthode d'Euler consiste à calculer les forces, puis la vitesse avec l'accélération (la force)

et enfin la position avec la vitesse (du temps précédent). En partant d'une condition initiale (positions au temps t_0), en fixant un pas de temps Δt et un nombre d'itérations (temps final t_1 / nombre de pas), la méthode d'Euler permet de faire évoluer les positions initiales jusqu'au temps t_1 avec un certain nombre d'itérations $\sim \frac{t_1-t_0}{\Delta t}$ et en approximant les dérivées.

Algorithme 1 #EulerExplicite

```
#Return tuple: (positions,velocities)
def EulerExplicite(q0,v0,t0,t1,dt):
    list v = [v0]
    list q = [q0]
    for i in range((t1-t0)/dt):
        vec a = calcForce(...args)/m
        q.append(q[-1] + v[-1]*dt)
        v.append(v[-1] + a*dt)
    return (q,v)
```

Le gros problème de la méthode d'Euler, outre son ordre de précision, est son manque de stabilité. En effet, un changement du pas de temps peut mener à des gros changements sur la trajectoire : En plus de l'erreur que cela engendre la trajectoire peut avoir une forme complètement différente. Il est possible de réduire l'erreur et d'augmenter la stabilité en faisant une somme astucieuse entre 2 développements de Taylor à l'ordre 3 :

$$\begin{cases} (1) & q(t + \Delta t) = q(t) + \dot{q}(t)\Delta t + \frac{\ddot{q}(t)\Delta t^2}{2} + \frac{\dddot{q}(t)\Delta t^3}{6} + o(\Delta t^3) \\ (2) & q(t - \Delta t) = q(t) - \dot{q}(t)\Delta t + \frac{\ddot{q}(t)\Delta t^2}{2} - \frac{\dddot{q}(t)\Delta t^3}{6} + o(\Delta t^3) \end{cases}$$

$$(1)+(2) \Rightarrow q_{t+1} = 2q_t - q_{t-1} + a_t \Delta t^2.$$

Donc, dans l'intégration de Verlet, nous n'enregistrons plus la vitesse mais l'ancienne position et si nous souhaitons avoir la vitesse il suffit de faire la dérivée centrée : $\frac{q_{t+1}-q_{t-1}}{2\Delta t}$.

Algorithme 2 Intégration de #Verlet

```
#Return positions
def Verlet(q0,v0,t0,t1,dt):
    list q = [q0-v0*dt,q0]
    for i in range((t1-t0)/dt):
        vec a = calcForce(...args)/m
        q.append(2q[-1]-q[-2]+a*dt**2)
    return q
```

1.2.2 Leapfrog

L'intégration leapfrog (ou saute-mouton...) est une autre manière pour faire la résolution. Elle est, comme l'intégration de Verlet, un ordre plus précis en erreur, toujours pour le même nombre d'évaluations de a . Cependant, l'intégration de Verlet et la méthode d'Euler évaluent la vitesse à un seul temps t tandis que l'intégration de leapfrog (la forme kick-drift-kick) utilise une vitesse à « mi-pas » $v_{t+\frac{1}{2}}$ pour estimer la position suivante (plutôt que d'utiliser la vitesse précédente ou actuelle). (*Il est aussi possible de rajouter une évaluation de l'accélération à la fin en faisant une perturbation pour être un peu plus précis.*)

Le schéma numérique est donc :

Algorithme 3 Integration #Leapfrog

```
#Return tuple: (positions, velocities)
def Leapfrog(q0,v0,t0,t1,dt):
    list q = [q0]
    list v = [v0]
    for i in range((t1-t0)/dt):

        vec a = calcForce(...args)/m
        vec v_s = v[-1]+a*dt/2
        q.append(q[-1]+v_s*dt)
        v.append(v_s+a*dt/2)

    return (q,v)
```

1.2.3 Runge-Kutta

Si nous souhaitons diminuer l'erreur et son ordre alors il existe des méthodes plus poussées (demandant plus d'évaluations). Les méthodes de Runge-Kutta permettent cela. Elles forment un ensemble de méthodes reposant sur un principe d'interpolation entre divers évaluations. Pour presque tous les cas nécessitant un bon degré de précision, la méthode de RK à l'ordre 4 est utilisée.

Nous n'entrerons pas plus dans les détails car l'objectif n'est pas de décrire ici « complètement » les méthodes RK. De plus, il existe beaucoup de documents sur le sujet.

Cependant voici un exemple pour la méthode explicite d'ordre 2 (explicite = elle ne fait intervenir que des évaluations sur des temps $t_k < t_{k+1}$) :

Algorithme 4 Runge-Kutta 2

```
#f is the function system (derivatives equations)
# as lambda functions!
#h = step, t0 & y0 = initial conditions
def RK2(y0, t0, h, n_steps, f):
    vec t = zeros(n_steps)
    t[1] = t0
    #solution matrix
    matrix y = zeros((len(y0),n_steps))
    for i in range(len(y0)):

        y[i,0] = y0[i]

        for i in range(n_steps):

            t[i+1]=t[i]+h
            #Use a general form of the Euler method
            #The idea is to use this to estimate
            # the function and its derivatives
            # to a half-step time further
            (a,b) = EulerExplicite(y[:,i],t[i],h/2,1,f)
            for j in range(len(f)):

                y[j,i+1]=y[i,j]+h*f[j](t(i)+h/2, b[:n,2])

    return (t,y)
```

1.2.4 Gauss-Seidel

Cette dernière sous-section est un peu particulière, car elle ne concerne pas une méthode d'intégration mais une méthode de résolution de système linéaire $Ax = b$. En effet, nous en aurons besoin plus loin dans le document. Même si nous utiliserons une méthode de scipy (spssolve), il est pertinent de connaître une méthode de résolution de système linéaire. Notons que la méthode de Jacobi est aussi un choix astucieux car elle permet la parallélisation des opérations facilement.

Algorithme 5 #Gauss-Seidel

```
def GaussSeidel(A,b,x0,epsi,Nmax):
    int n = A.shape[0]
    vec X = x0
    int N = 0
    while N < Nmax:

        vec X2 = zeros(n)
        for i in range(0,len(N)):

            sum = 0
            for j in range(i):

                sum = sum + (A[i,j] @ X2[j])

            sum2 = sum2 + (A[i,j] @ X[j])
            X2[i] = (1/A[i,i])*(b[i]-sum-sum2)

            b2 = A @ X2.T
            bool flag = True
            for i in range(len(b)):

                if(abs(b2[i]-b2[i])/abs(b[i])>epsi):
                    flag = False
                    break
                X=X2
                if(flag):
                    break
                N+=1
            return X
```

Les méthodes présentées ici permettent un choix assez large sur la précision et la stabilité recherchée. Elles fournissent aussi les outils de parallélisation des tâches utiles pour le calcul sur multi-coeur, qui est actuellement très importante.

1.3 Objets de la simulation

Toujours dans l'optique de comprendre toutes les choses en jeu dans le code, nous allons créer notre propre objet Vecteur/Vector (pour notre cas en 2D) ainsi qu'un objet Particule/Particule. Des méthodes pour simplifier le travail (faire des interpolations, centrer sur le centre de masse, faire le rendu...) devront aussi être codées. Comme ce sont des outils, ils seront présentés avec le code au moment où ils rentrent en jeu.

- L'objet vecteur³ doit contenir les méthodes d'addition et de multiplication : par un autre vecteur ou par un scalaire. Il peut aussi être utile d'ajouter une fonction permettant de vérifier l'égalité avec un autre vecteur ou encore une fonction permettant de calculer sa norme.
- Une Particule⁴ est un objet contenant, à l'état le moins complexe, sa position (vecteur), son ancienne position(vecteur~vitesse), ainsi que sa masse :

Algorithme 6 Particle

```
#Args: vector, vector, float
class Particle(position,old_position,mass):
    constructor(self):
        self.position = position
        self.old_position = old_position
        self.mass = mass
```

Elle peut également contenir des fonctions utiles comme une fonction permettant de renvoyer une copie de la particule (dans une nouvelle instance) :

Algorithme 7 Particle#clone()

```
class Particle(...args):
    def clone(self):
        return Particle(...self.props)
```

ou encore une fonction permettant de vérifier si la particule est égale à une autre (même objet) :

Algorithme 8 Particle>equals(particle)

```
class Particle(...args):
    def equals(self, particle):
        return particle.pos == self.pos and ...
```

Nous avons maintenant tous les outils en main pour bien commencer la mise en place de la simulation.

2 Approche Particle-Particle (PP)

La première approche du problème que nous ferons est la plus « naïve », c'est à dire calculer toutes les forces des particules entre elles. Ainsi si il y a N particules, cela nous donne $O(N^2)$

2.1.2 Observations

La situation initiale est telle que les particules sont immobiles et placées aléatoirement et uniformément dans un carré d'une certaine taille, nous obtenons :

3. Vector2D : <https://github.com/EazyEnder/GravSim/blob/main/Vector2.py>
 4. Particle : <https://github.com/EazyEnder/GravSim/blob/main/Particle.py>

opérations donc une baisse très rapide des performances en fonction du nombre de particules. Cette approche sera néanmoins utile pour avoir une référence pour le débogage des autres approches.

2.1 Mise en place

Comme énoncé précédemment, la méthode Particle-Particle est la plus directe et naïve des méthodes car elle consiste à calculer toutes les interactions particule par particule.

Cependant elle est très simple à coder. D'abord, il y a N particules à mettre à jour il faut une première boucle « for » pour itérer sur toute la liste de particules. Cette boucle doit contenir un outil pour obtenir la somme des forces agissant sur la particule itérée « i » ainsi qu'une méthode d'intégration utilisant cette force pour faire évoluer la particule jusqu'au temps d'après.

Dans cette approche direct la force finale utilisée est la somme des interactions avec les autres particules. Pour la calculer, il faut une deuxième boucle « for » avec les N particules (d'où le $O(N^2)$ opérations). Cette boucle doit vérifier le fait que la particule « j » n'est pas la même que « i » car une particule n'intéagit pas avec elle-même.

Enfin afin d'arriver jusqu'à un temps t_1 choisi, soit un nombre de pas fixé, il faut envelopper cette double-boucle par exemple par un « while » pour faire l'évolution jusqu'à atteindre t_1 .

2.1.1 Algorithme

L'algorithme présenté ici est le cœur de l'approche et permet d'avoir l'accélération/somme des forces sur une particule :

Algorithme 9 Particle-Particle (PP)

```
#In a while loop
for p1 in particles:

    vec sum_forces = new Vector(0,...,0)
    for p2 in particles:
        if p1.equals(p2):
            continue
        sum_forces += G*p1.mass*p2.mass

    /({distance between p1 and p2})
    *{Normalized vector from p1 to p2}

#Integration (as verlet) can come here
#or you'll need to save the sums as a list
```

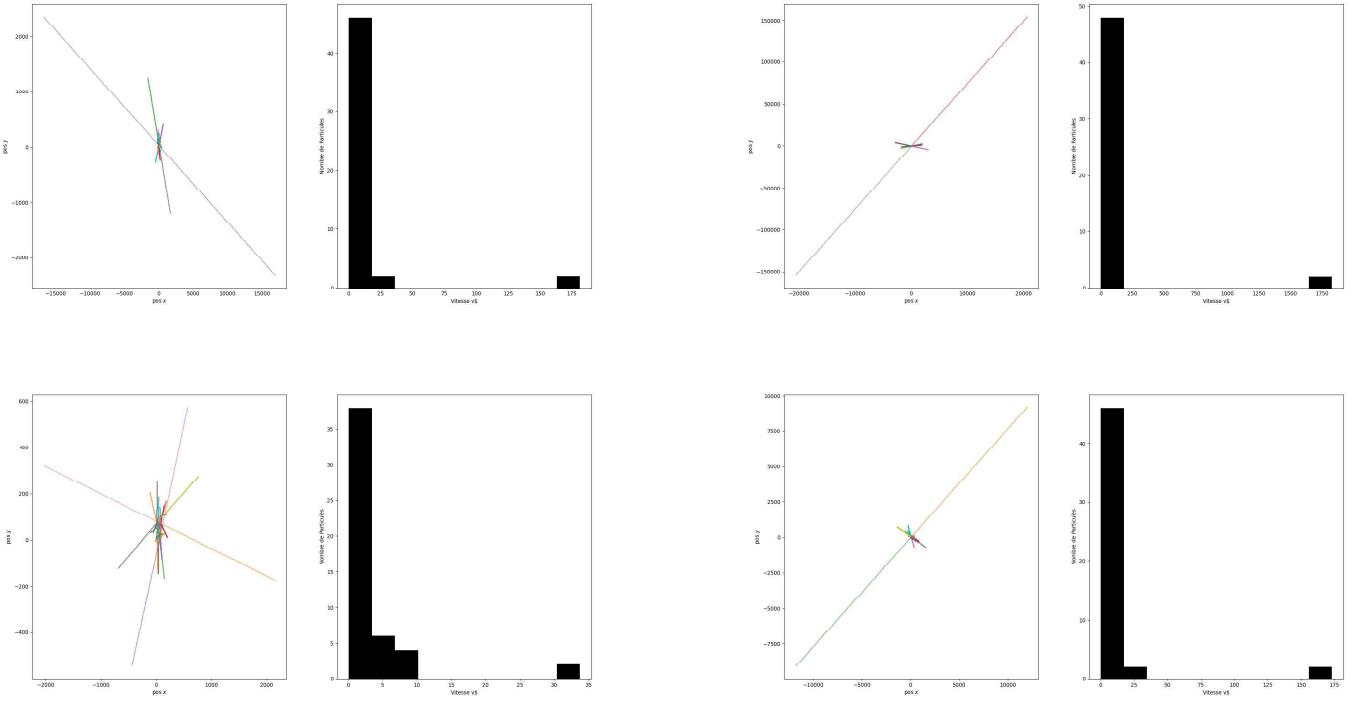


TABLE 2 – PP Simulations : Trajectoires | Histogramme vitesses

Les observations soulignent que certaines particules atteignent des vitesses excessives. En effet sur ces simulations les particules ont une masse $m = 1$, sont générées dans une zone de taille $L = 100$ avec une constante de gravitation $G = 0.25$. Le pas de temps est $\Delta t = 0.5$.

Si nous supposons que la particule gagne cette grande vitesse en un seul pas de temps (nous verrons un peu plus tard que le problème réside justement là) alors : $v \sim a \times \Delta t$. Si en plus nous faisons l'hypothèse que l'accélération est dominée par une seule interaction avec une particule alors v devient : $|v| \sim \frac{G}{d^2} \Delta t$ avec d la distance entre les 2 particules. Ainsi $d \sim \frac{\sqrt{G \Delta t}}{|v|}$. Les vitesses peuvent être très élevées : souvent entre 100 et 1000. Donc pour $v = 100$: $d \sim 10^{-3}$ soit $\frac{d}{L} = 10^{-5}$.

La conclusion est que le pas de temps est trop grand pour gérer des interactions à si faible distance. Or pour réduire ce souci non physique il faudrait grandement diminuer le pas de temps ce qui reviendrait à énormément augmenter le temps de calcul.

Si maintenant nous faisons abstraction de ces particules à haute vitesse, en zoomant sur les particules à faible vitesse comprises dans la zone au centre, alors nous obtenons :

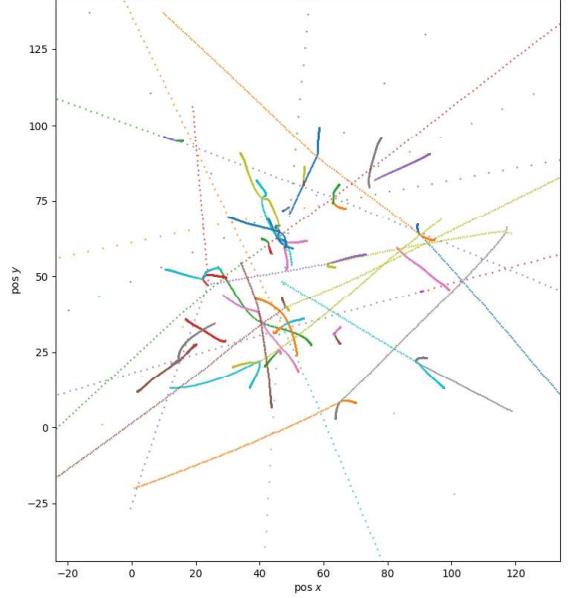


FIGURE 1 – PP : Trajectoires des particules

Ces trajectoires permettent au moins de vérifier qu'il y a bien des interactions qui ressemblent à vue d'oeil à ce que pourrait produire la gravité. En effet, il aurait été éliminatoire pour l'algorithme si par exemple les particules se repoussaient.

2.2 Problèmes

Néanmoins comme nous l'avons vu plus tôt cette simulation comporte des problèmes évidents.

2.2.1 Divergence de la vitesse des particules

Le plus visible de ces problèmes est la divergence dans la vitesse des particules. Comment expliquer cela ?

Détaillons l'explication du pas de temps trop court. Si le temps est discrétisé, alors la vitesse et l'accélération sont aussi discrétisées. Et que donc certe dans la réalité un corps peut aussi passer à très faible distance mais il va cependant subir une intensité très forte pendant un temps aussi court que l'intensité est grande. Ce qui n'est pas le cas ici.

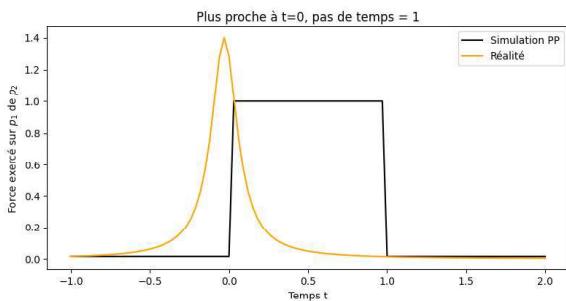


FIGURE 2 – PP : Effet de la discréttisation sur la vitesse

En effet, au plus proche de la particule p_2 où elle va subir l'interaction, la force va atteindre son maximum pendant un court temps dt . Or dans la simulation il suffit que le pas Δt soit trop long pour que cette force, considérée comme constante sur le pas de temps dans les méthodes d'intégration, soit intégrée sur tout ce temps. Il s'en suit un excès anormal d'accélération et donc de vitesse pour la particule p_1 .

Le problème peut aussi simplement venir de la propriété ponctuelle des particules. Il n'y a en effet aucunes répulsions ou effets de contact entre elles.

Une des méthodes les plus simples pour pallier ce soucis, si on ne souhaite pas diminuer le pas de temps, est d'utiliser un facteur d'amortissement a et de le mettre dans l'expression de la force $F \propto \frac{1}{r^2}$ afin d'obtenir $F \propto \frac{1}{r^2+a^2}$ (au carré pour que cela soit aussi compatible avec le potentiel). Cet ajout permet d'éviter la divergence aux distances nulles.

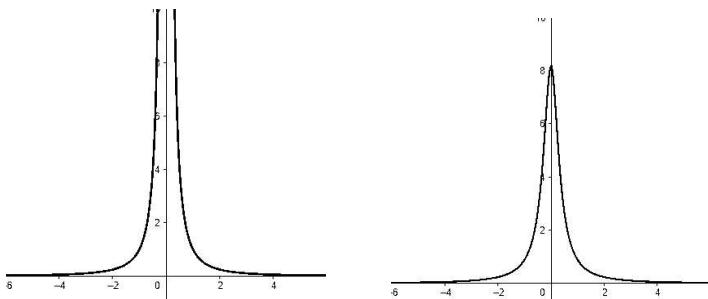


TABLE 3 – $f(r) = \frac{1}{r^2+a^2}$: À gauche sans amortissement, À droite $a = 0.35$

Néanmoins ce paramètre est assez sensible car si il est trop élevé alors il n'y a plus de sens physique dans la simulation (et le mouvement est stoppé) Au contraire si sa valeur est trop faible alors le problème n'est pas supprimé. Un bon choix peut être cependant trouvé en étudiant et prenant en compte les distances (\sim une distance moyenne) et les vitesses initiales (\sim une vitesse moyenne) entre particules ainsi que la taille de l'environnement de simulation.

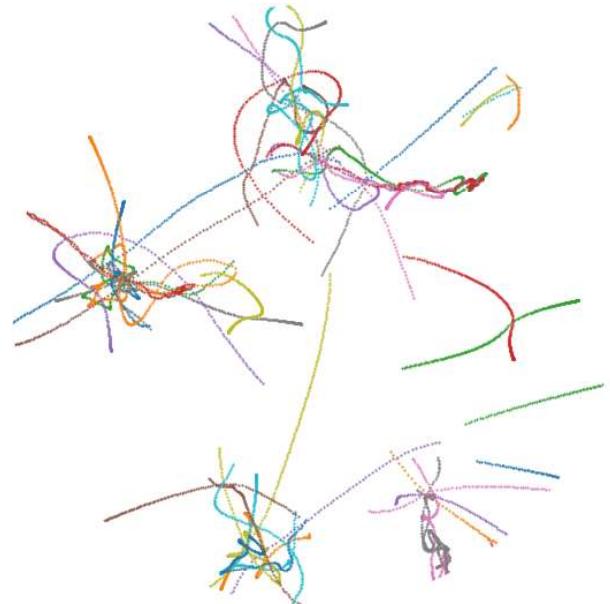


FIGURE 3 – PP : Trajectoires des particules en utilisant un facteur d'amortissement

Une autre solution pourrait être de fusionner les particules lorsqu'elles sont trop rapprochées. Elles forment alors une particule unique plus massive $m = m_1 + m_2$ d'impulsion $\vec{p}_1 + \vec{p}_2 = \vec{p} \iff m_1 \vec{v}_1 + m_2 \vec{v}_2 = \vec{v}$ donc $\vec{v} = \frac{m_1 \vec{v}_1 + m_2 \vec{v}_2}{m_1 + m_2}$.

Pour mettre en oeuvre cette solution simplement, il faut le faire dans la première boucle "for". En effet, c'est avant le calcul des forces et après Verlet. Néanmoins cela rajoute encore n^2 d'opérations (si pas QuadTree) donc double le temps de calcul.

Algorithme 10 Collision: Merge 2 particles

```

list new_particles = []
#in a double for like the one used for the forces
if Math.abs((p1.r - p2.r).length()) <= threshold:
    Particle new_p =
        new Particle(p1.position,  $\frac{m_1 \vec{v}_1 + m_2 \vec{v}_2}{m_1 + m_2}$ 
        with  $\begin{cases} \vec{v}_1 = (p1.r - p1.old_r)/dt \\ \vec{v}_2 = (p2.r - p2.old_r)/dt \\ ,p1.m+p2.m \end{cases}$ 
    new_particles.append(new_p)
    particles = new_particles

```

Nous pourrions aussi envisager une collision rigide élastique entre les particules considérées alors comme des disques. Mais cela rajoute aussi des opérations de calcul, ce que nous ne souhaitons pas faire dans le code.

2.2.2 Conservation de l'énergie

Une des nombreuses lois vérifiées par le problème à n corps est la conservation de l'énergie si le système est isolé et sans forces dissipatives.

L'énergie potentielle d'une interaction entre 2 particules est $\Phi_{12} = -G \frac{m_1 m_2}{\sqrt{r_{12}^2 + a^2}}$ car la force doit respecter : $\vec{F}_{12} = -\vec{\nabla} \Phi_{12}$. Ainsi l'énergie potentielle totale d'un système composé de N particules est une somme de ces Φ .

Cependant, il faut bien faire attention à ne pas compter deux fois la même interaction, c'est à dire Φ_{12} et Φ_{21} . Donc l'énergie potentielle du système est : $E_p = \sum_{i=1}^N \sum_{j=i+1}^N \Phi_{ij} = -G \sum_{i=1}^N \sum_{j=i+1}^N \frac{m_i m_j}{\sqrt{r_{ij}^2 + a^2}}$ avec a le facteur d'amortissement.

L'autre contribution en énergie du système est évidemment l'énergie cinétique : $E_c = \frac{1}{2} \sum_{i=1}^N m_i v_i^2$.

Finalement,

$$E_{tot} = E_c + E_p = \sum_{i=1}^N \left(\frac{m_i v_i^2}{2} - G \sum_{j=i+1}^N \frac{m_i m_j}{\sqrt{r_{ij}^2 + a^2}} \right) \quad (2)$$

Pour le moment, considérons $a = 0$, c'est à dire sans régler le problème précédent de la divergence des vitesses :

Appliquons maintenant la solution trouvée lors de la précédente section, c'est à dire insérer un facteur d'amortissement a (« softening factor »). Pour un plus grand nombre de particules, observons l'évolution de l'énergie avec différentes valeurs de a .

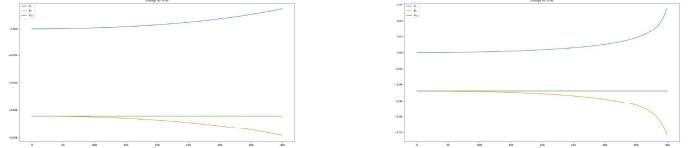


TABLE 4 – PP : Energie du système pour 2 et 5 particules pour temps courts. (vert= E_{tot} , jaune= E_p , bleu= E_c)

Avec peu de particules et sur des temps courts, nous observons que l'énergie est bien conservée. Le temps court est corrélé au nombre de particules, c'est à dire que plus il y a de particules, plus vite il va y avoir une particule qui va provoquer une divergence. Néanmoins, si nous attendons assez longtemps pour avoir un rapprochement et la divergence alors :

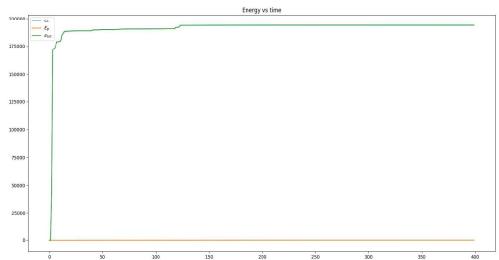


FIGURE 4 – PP : Energie du système aux temps longs, sans facteur d'amortissement

La vitesse étant fausse, l'énergie cinétique l'est aussi et donc l'énergie du système n'est pas conservée durant ce « contact ».

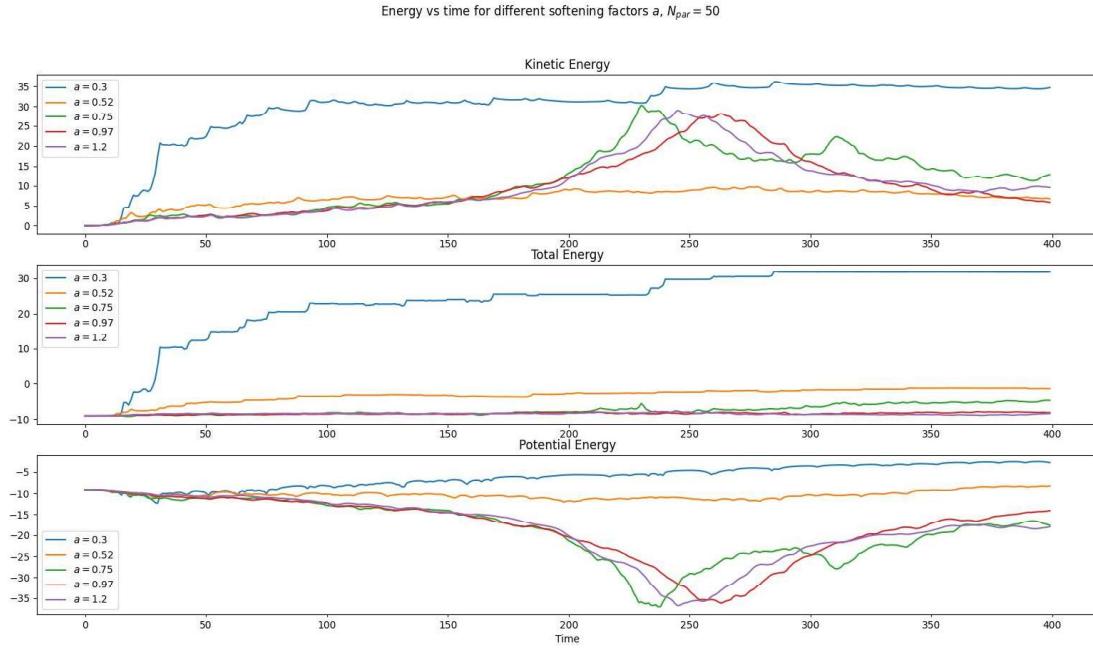


FIGURE 5 – PP : Energie pour différents a pour $N=50$

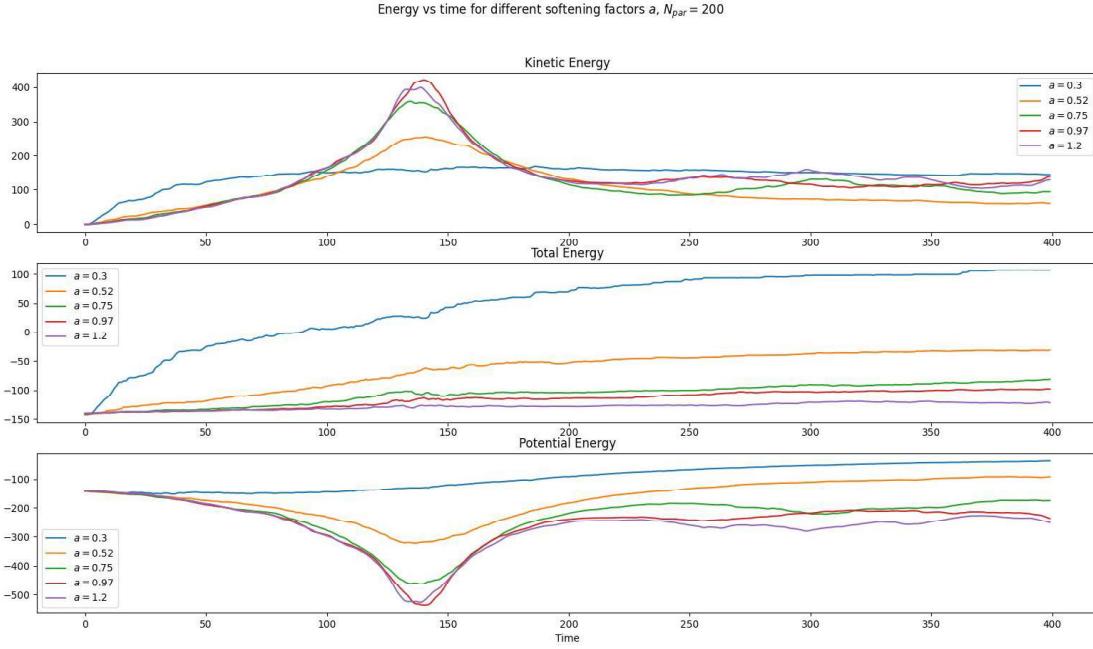


FIGURE 6 – PP : Energie pour différents a pour $N=200$

Les graphes mettent en évidence plusieurs choses :

- L'énergie n'est pas correctement conservée mais elle s'améliore si le facteur d'amortissement augmente.
- L'amortissement provoque à partir d'un seuil un maximum d'énergie cinétique / minimum d'énergie potentielle. Ce maximum a une amplitude max pour une certaine valeur de a_{max} (borne supérieur). *Notons que la courbe avec $N=200$ particules est à privilégier car les particules étant générées uniformément et aléatoirement dans un espace confiné défini alors plus il y a de particules, plus l'espace est uniformément peuplé : loi des grands nombres.*
- Cependant ce a_{max} ne permet pas la meilleure conservation de l'énergie.

Le soucis est qu'il n'est pas possible de mettre un a infini ou très grand car cela reviendrait à geler les particules (les forces tendent vers 0) donc à perdre le sens physique. Ainsi, il faudra se contenter d'une certaine valeur de a choisi judicieusement en

prenant en compte le sens physique et une certaine conservation de l'énergie, par exemple pour $N=200$: $a \sim 0.75$ semble être un choix correct.

De plus, les méthodes d'intégration peuvent ne pas conserver l'énergie. En général elles ne le font pas sauf si il y a une attention particulière pour que ça soit le cas. Mais, le faire implique d'autres problèmes. Par construction, la méthode de Verlet devrait permettre une meilleure conservation de l'énergie que la méthode d'Euler. Mais, même avec la méthode de Verlet qui ne provoque pas un gain ou une perte net d'énergie, il est toujours possible que la valeur de l'énergie oscille autour d'une constante (et donc soit fausse d'un certain facteur). Cependant, avec cette méthode, nous obtenons une valeur de l'énergie proche de la vraie valeur.

2.3 Résultats

Maintenant que nous avons une simulation qui fonctionne nous pouvons tester différentes situations avec. Cependant nous n'irons pas plus loin en utilisant cette approche. En effet, le temps de calcul est trop grand pour un grand nombre de particules (et donc pour des résultats intéressants). Même en faisant du multi-processing (voir Annexe B) le nombre de particules ne peut pas être très élevé à cause du coût en N^2 .

Il est donc nécessaire de repenser notre façon de faire.

Deuxième partie

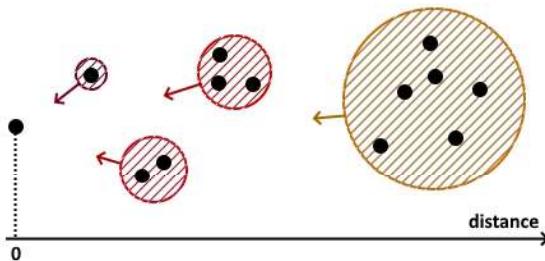
Approche avancée

Nous avons vu précédemment que l'algorithme le plus direct consiste à simuler toutes les forces. Cependant, il a un coût $O(N^2)$ en opérations avec N le nombre de particules simulées. L'objectif de cette partie est de chercher différentes méthodes qui permettent de réduire fortement ce coût tout en conservant une simulation correcte d'un point de vue physique.

Nous aborderons ici deux de ces méthodes : « Tree-Code » (TC) ainsi que « Particle-In-Cell »(PIC) ~ « Particle-Mesh »(PM) pour des cas 2D.

3 Tree-Code (TC)

Pour une particule p_0 qui subit deux forces, une pour chacune de deux particules éloignées p_1 et p_2 et si la distance séparant p_1 et p_2 est faible par rapport à leur distance à p_0 alors, l'hypothèse que nous pouvons faire est de considérer ces deux particules comme une seule. Ainsi p_0 ne subira plus qu'une force venant de $p_{12} = p_1 + p_2$ le centre de masse du système des 2 particules. Cette hypothèse, repose sur le fait que la force est proportionnelle à l'inverse de la distance au carré. Au final elle peut être appliquée aux N particules. Donc, elle permet de diminuer le nombre de forces à calculer en regroupant les particules entre elles.



Disques noirs = particules ; zones en couleurs = groupes de particules ; flèches = forces exercées par ces groupes sur la particule à gauche en $d = 0$.

FIGURE 7 – TC - Hypothèse

La structure permettant de regrouper ces particules est nommée : « QuadTree » (2D) ou « OctTree » (3D). L'utilisation de cette structure et de cette hypothèse donne le nom d'approximation de « Barnes-Hut » à la simulation et permet de passer de $O(N^2)$ à $O(N \ln(N))$ opérations. C'est à dire pour $N = 10\,000$: $\begin{cases} N^2 = 10^8 \\ N \ln(N) = \sim 10^5 \end{cases}$.

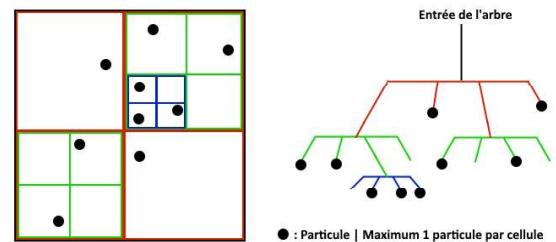
3.1 QuadTree

Dans l'objectif de réduire au maximum le nombre d'opérations, il faudrait que la structure numérique qui contient les particules ait une donnée de la position des particules directement. C'est

à dire que les particules soient rangées dans cette structure en fonction de leur position. Il est alors possible de trouver les plus proches voisins de ces particules avec un minimum d'opérations. Dans la liste où sont stockées les particules actuellement, ce n'est pas le cas. En effet, pour trouver les voisins d'une particule de cette liste il faut itérer sur toute la liste. L'objectif ici est au contraire de ne pas itérer sur toute la structure pour faire ce genre d'opérations.

3.1.1 Principe

La structure « QuadTree » répond à ses attentes. Cette dernière peut-être formée récursivement via un seul objet : une cellule qui se divise dans l'espace en 4 nouvelles lorsqu'un certain nombre de particules est atteint. Lors de la division, elle redistribue toutes ses particules à ses « enfants ». À chaque nouvelle particule insérée, elle va la redistribuer à un de ces enfants en fonction de sa position. Ainsi nous avons récursivement une chaîne parent-enfants.



QuadTree avec pour condition qu'une cellule ne peut contenir + qu'une particule sinon elle se subdivise. Vision en cellules et en arbre.

FIGURE 8 – TC - Schéma de la structure

3.1.2 Construction

Le code de cette structure est composé de deux objets : le premier contient la forme de la cellule (sa position et sa taille), et le deuxième est l'arbre lui-même. Il s'agit d'une structure récursive donc le deuxième objet s'appellera lui même et ainsi de suite.

Le premier objet que nous nommons : « QRegion » contient la position du centre de la cellule, sa demi-largeur et une fonction qui permet de savoir si une particule est dans la région en utilisant sa position.

Algorithme 11 QRegion

```
#Args: vector, float
class QRegion(center, half_length):
    constructor(self, center, half_length):
        self.center = center
        self.half_length = half_length
```

Ainsi que sa fonction :

Algorithme 12 QRegion#containsParticle

```
#Args: particle
def containsParticle(self, particle):
    return isInBox(particle.position,
                   self.center, self.half_length)
```

« isInBox » est simplement une fonction générique permettant de vérifier si un vecteur est dans une cellule : pour chaque composante, la fonction vérifie si elle est inclut dans l'intervalle : [center-h_length,center+h_length]. Il y a 3 composantes \times 2 booléans primaires = 6 booléans à vérifier.

Passons à la construction de l'objet « QuadTree »(QT) . Il est créé à partir d'une région. Il contient dès sa création la liste de ses enfants « subqt », son parent, la liste des particules dans le QT, et de son centre de masse. Néanmoins ces propriétés sont initialement nulles ou vides.

Algorithme 13 QuadTree

```
#Args: QRegion
class QuadTree(region):
    constructor(self, region):
        self.region = region
        #NE-NW-SE-SW
        self.subqt = [None, None, None, None]
        #tuple with qt parent & child id
        self.parent = None
        self.particles = []
        #masscenter : [mass, vec2(x,y)]
        self.masscenter = None
```

Ses 2 fonctions élémentaires sont : celle lui permettant de se subdiviser « subdivide » et celle permettant d'insérer « insert » une particule à l'intérieur.

Algorithme 14 QuadTree#insert

```
#Args: particle
#return if it's a success
def insert(self, particle):
    if(not(self.region.containsParticle(particle))):
        return False
    if(len(self.particles) < QT_PARTICLE_CAPACITY
       and self.subqt[0] == None):
        particle.host = self
        self.particles.append(particle)
        return True
    if(self.subqt[0] == None):
        self.subdivide()
    for qt in self.subqt:
        if(qt.insert(particle)):
            return True
    return False
```

La fonction est une succession de conditions :

1. Si la particule est déjà dans le QT il n'est pas nécessaire de la réinsérer
2. Si le QT n'est pas déjà au maximum de particules hébergées et qu'il n'a pas d'enfants alors la particule est insérée avec succès
3. Sinon le QT est subdivisé (si ce n'est pas déjà le cas) et la particule est donnée aux enfants.

Notons plusieurs points :

- L'objet « Particle » a maintenant un attribut « host » qui permet d'avoir une référence à l'intérieur de la particule sur le QT qui l'héberge.
- La fonction peut aussi échouer si la particule n'est spatialement pas dans le QT. Si c'est le cas et que le code en arrive à cette étape cela révèle un problème majeur du code.
- La fonction s'appelle elle-même et c'est ce type de structure qui rend plus accessible la structure en arbre.

La fonction « subdivide » a la même structure récursive car elle peut appeler « insert » :

Algorithme 15 QuadTree#subdivide

```
#Subdivide the quadtree into 4 smaller trees
def subdivide(self):
    self.subqt[1:4] = new QuadTree(
        new QRegion with an offset
        and smaller size
    )
    for i in range(len(self.subqt)):

        self.subqt[i].parent = (self,i)

        for p in self.particles:
            for qt in self.subqt:

                if(qt.region.containsParticle(p)):
                    qt.insert(p)
                    self.particles.remove(p)
                    break

                if(len(self.particles) > 0):
                    print("QT node not in subtree")
```

Les étapes de cette fonction sont :

1. Création des QT fils avec un offset pour le centre dans la direction $\begin{pmatrix} \pm 1 \\ \pm 1 \end{pmatrix}$ et une taille deux fois plus petite.
2. La référence du parent est mise dans ces nouvelles instances.
3. Redistribution de chaque particule du parent dans les QT fils.

Si il reste une particule dans le QT parent c'est alors signe de la même erreur vu précédemment dans la fonction « insert ».

Le QuadTree peut aussi avoir une fonction « query » afin de rechercher les particules à l'intérieur d'une région ou d'autres fonctions mineurs non élémentaires.

La structure détaillée ici est la version la plus fondamentale et universelle. Elle est utilisée aussi bien dans le traitement d'image que dans des simulations comme la notre.

3.2 Utilisation

Utiliser le QuadTree comme mémoire à la place d'une liste plate permet de réduire fortement le nombre d'opérations. La construction du QuadTree via les particules se fait en peu d'opérations et peut donc être refaite à chaque itération de temps. En effet, les particules se déplacent et elles peuvent quitter leur région d'origine pour en rejoindre une autre. Ce qui amène donc à un QuadTree différent.

3.2.1 Centre de masse

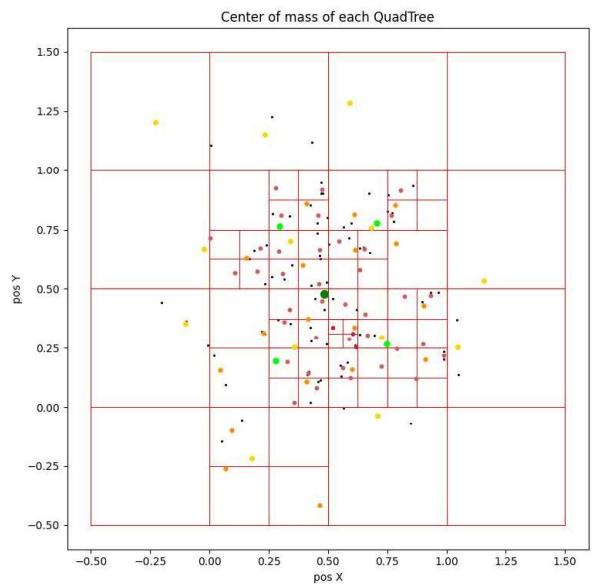
Comme annoncé précédemment, le calcul de la force utilisera les proches voisins de la particule pour laquelle est calculée la force, ainsi que les centres de masses des QTs. Dans un premier temps, il faut donc les calculer avec la formule :

$$\vec{p}^* = \frac{\sum_i m_i \vec{p}_i}{\sum_i m_i} \quad (3)$$

Algorithme 16 QuadTree#computeMassCenter

```
#Compute the inertial center of all the particles
#in the QuadTree and his children
def computeMassCenter(self):
    masscenter = [0,Vector2(0,0)]
    #Si le qt a des enfants, on utilise directement
    #les CDMs de ses enfants
    if(self.subqt[0] != None):
        for qt in self.subqt:
            m_c = qt.masscenter
            #Si le CDM n'existe pas on le calcule
            if(m_c == None):
                m_c = qt.computeMassCenter()
                #Calcul du CDM via la formule
                masscenter[0] += m_c[0]
                masscenter[1] = masscenter[1].add(
                    m_c[1].multiplyScalar(m_c[0]))
            masscenter[1] = masscenter[1]
            .multiplyScalar(1/masscenter[0])
    #Sinon on le fait avec les particules du QT
    if(len(self.particles) > 0 and
       self.subqt[0] == None):
        p_masscenter = [0, Vector2(0,0)]
        #même chose que plus haut
        #mais sur toutes les particules
        ...
        self.masscenter = masscenter
    return masscenter
```

Nous pouvons vérifier qualitativement sur un faible nombre de particules en les faisant apparaître sur un schéma :



$N=100$ particules réparties aléatoirement via une loi normale ; limite de 3 particules par QT ; particules représentées via les points noirs et les différents niveaux de centre de masse avec les points en couleurs et leurs tailles.

FIGURE 9 – TC - Centres de masse d'un QuadTree

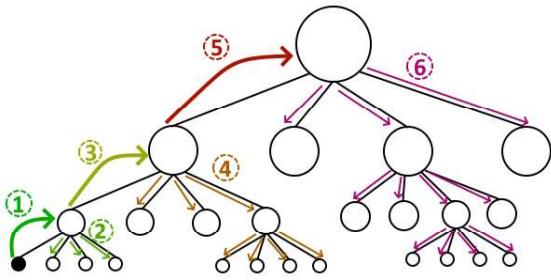
En utilisant une loi de probabilité avec symétrie centrale, pour une génération, nous observons bien que le centre de masse total du QuadTree est au centre.

3.2.2 Calcul de la Force

Pour calculer la force en utilisant le QuadTree nous allons procéder en deux étapes et utiliser un critère pour savoir si il faut appliquer l'approximation ou non.

Ce critère est le rapport entre la taille du QuadTree où sont les particules et sa distance avec la particule sur laquelle le calcul est en cours. Si ce critère $\theta = \frac{L}{d}$ est inférieur à un ϵ que l'utilisateur choisit alors l'approximation est faite. Nous ne calculons alors que la force avec le centre de masse du quadtree cible. Si le critère est supérieur à ϵ alors nous parcourons, les enfants du QuadTree (si il y en a) et le critère est recalculé pour ces derniers. Si il n'y a pas de fils alors la somme des forces de toutes les particules est calculée.

Afin de ne pas calculer deux fois les interactions avec une même région, il ne faut pas faire une descente du QuadTree principal. Il faut au contraire partir du QuadTree de la particule et remonter jusqu'au plus grand parent. À chaque niveau nous appliquons l'algorithme du critère en faisant attention de ne pas faire à nouveau le calcul sur le QuadTree fils duquel nous remontons.



La particule sur laquelle nous calculons les forces est le point noir. Les flèches sont les montées et descendentes, leur couleur et leur numéro correspondent à l'ordre de l'étape (En rajoutant évidemment l'étape où les forces directes du QuadTree, où se situe la particule, sont rajoutées).

FIGURE 10 – TC - Calcul des forces exercées sur une particule du QuadTree

L'algorithme peut donc être explicité en deux fonctions : une qui permet la montée et lance la deuxième fonction qui gère la descente.

Algorithme 17 QuadTree#getForce

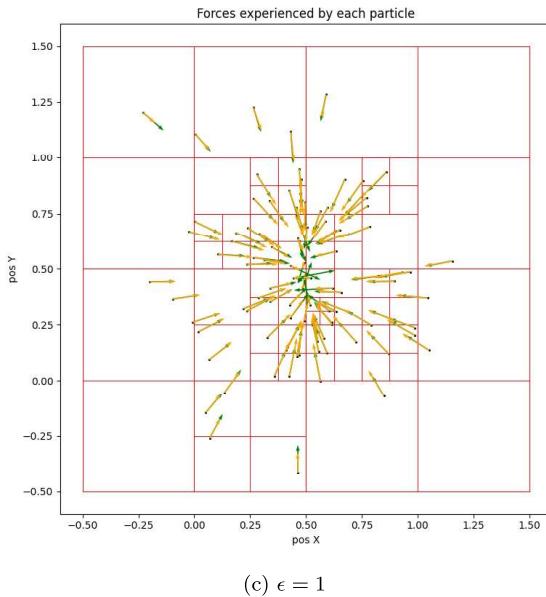
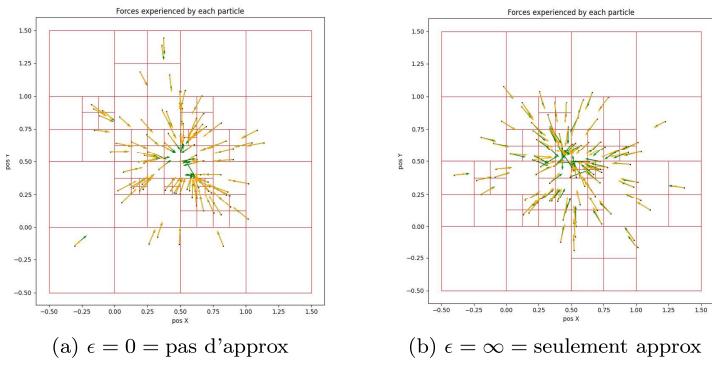
```
def getForce(self,particle,eps):
    if(self.masscenter == None):
        self.computeMassCenter()
    vec2 force = Vector2(0,0)
    #Sur toutes les particules du QT
    #Calcul de force direct
    for p in self.particles
        #Idem partie précédente
        force = ...
    QuadTree layer = self
    #Permet de ne pas retomber dans ce QT
    int parent_index = -1
    #Tant que le QT itéré a un parent
    while(layer.parent!=None):

        layer,parent_index = layer.parent
        for i in range(len(layer.subqt)):
            if(i==parent_index):
                continue
            #Descente
            force = force.add(
                layer.subqt[i]
                .getSubForce(particle,eps))
    return force
```

Et la fonction de descente :

Algorithme 18 QuadTree#getSubForce

```
def getSubForce(self,particle,eps):
    if(self.masscenter == None):
        self.computeMassCenter()
    #Utilisation du centre de masse
    #pour la distance
    vec2 distToQt=self.masscenter[1].sub(
        particle.R).length()
    double sizeQt=self.region.half_length*2
    #Critère, si oui -> approximation non faite
    if(distToQt==0 or sizeQt/distToQt>eps):
        vec2 f_sum = Vector2(0,0)
        if(self.subqt[0] !=None):
            for qt in self.subqt:
                f_sum = f_sum.add(
                    qt.getSubForce(particle,precision))
        return f_sum
    for p in self.particles:
        #Calcul direct vu que pas d'approx
        f_sum = ...
    return f_sum
    #Sinon on fait l'approx
    else:
        #Calcul de la force mais en utilisant
        #le centre de masse du QT
        return ...
```



Flèches jaunes = Approximations ; Flèches vertes = Réalité Seule la direction compte, l'affichage peut mener à croire que la norme change mais ce n'est pas forcément le cas (visible en zoomant).

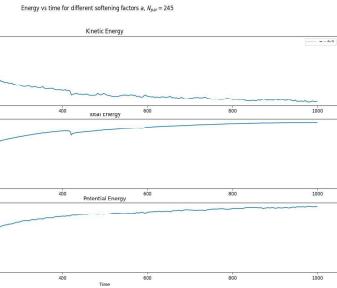
TABLE 5 – TC - Forces pour chaque particule

Il nous faut encore à changer le déroulé de la simulation. Pour cela, il nous suffit de changer le calcul de force de la méthode directe par celle du QT, puis de reconstruire un nouveau QuadTree à chaque pas de temps.

3.3 Résultats

Observons l'évolution d'un disque tournant uniformément dense. Cette situation permet un bon mélange aléatoire des particules sans qu'elles partent à l'infini (sans qu'elles prennent une vitesse trop élevée). Par exemple, si il n'y avait pas de vitesse orthoradiale initiale, les particules chuteraient toutes au centre du disque ce qui amèneraient à des vitesses très grandes. Le nombre de particules est faible par soucis de mémoire. En effet tous les états par lesquels les particules sont passés sont sauvegardés.

Par ailleurs, le nombre de particules dans la simulation peut ne pas être constant. En effet, si elles sortent de la région définie par le QuadTree initiale, elles seront automatiquement supprimées du processus.



$N=500$ particules, $\epsilon = 1$

FIGURE 11 – TC - Evolution de l'énergie

La chute d'énergie brutale au temps ~ 400 peut être expliquée par les particules quittant la région du QT. Sinon nous observons que l'énergie totale du système converge vers une valeur. L'énergie cinétique et l'énergie potentielle convergent aussi vers des valeurs, ce qui indique qu'il existe une distance et une vitesse moyenne des particules qui sont constantes.

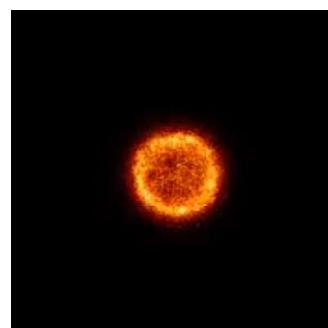
L'approximation ne brise donc pas la conservation de l'énergie, du moins pas plus que la méthode d'intégration.

Le temps de calcul est d'ailleurs très similaire à celui de l'approche naïve pour ce nombre de particules. C'est seulement à partir de 10 000 particules que l'approximation a de l'intérêt.

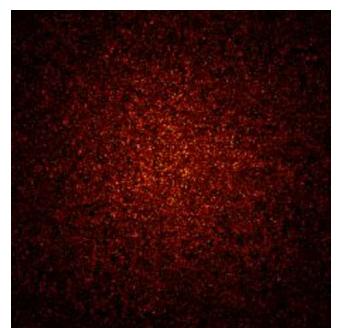
Deux vidéos d'une même situation initiale mais avec des facteurs d'amortissements différents et avec plus de particules sont disponibles sur Youtube. Elles permettent de voir qu'en effet les particules s'éloignent jusqu'à une certaine distance et convergent vers une même vitesse.

Cependant le facteur d'amortissement permet la formation d'amas sur un spectre plus large en taille et plus stables.

Les deux simulations convergent aussi différemment vers la situation finale. En effet avec le facteur d'amortissement il y a bien un effondrement comme on pourrait le prévoir alors que ce n'est pas le cas sans. Sans amortissement, les particules sont dès le départ expulsées du fait de leur trop grande proximité.



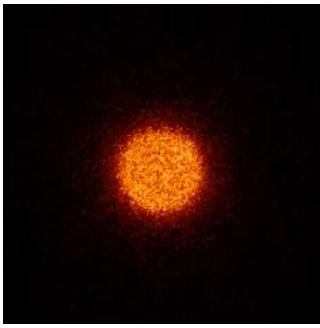
(a) Début de la simulation - Effondrement



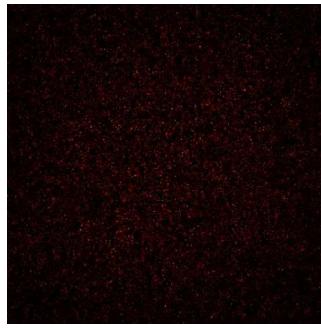
(b) Fin de la simulation - Densité hétérogène

Au début de la simulation, nous pouvons remarquer qu'un effondrement a lieu et qu'une couche plus dense apparaît, c'est la région de choc. Il s'en suit une expulsion des particules aux temps longs, néanmoins la région au centre est plus dense.

TABLE 6 – TC - Simulation $a = 0.3$



(a) Début de la simulation



(b) Fin de la simulation - Densité homogène

Au début de la simulation, il n'y a pas de couche plus dense. Les particules sont directement expulsées du centre à grande vitesse. Puis, aux temps longs il y a une densité homogène à grande échelle des particules, voir une plus faible densité au centre pendant un moment. Cette différence peut être expliquée par le fait qu'initiallement les particules prennent une vitesse vers l'extérieur.

TABLE 7 – TC - Simulation $a = 0$.

3.4 Axes d'améliorations possibles

Pour améliorer le temps de calcul, nous pouvons évidemment nous pencher sur le GPU et la parallélisation des tâches. Il y a plusieurs moyens d'y arriver.

Par exemple, dans un premier cas la construction du QuadTree peut être faite sur le CPU puis envoyé au GPU qui va répartir les particules sur les différents threads. Il va ensuite revenir sur le CPU pour la construction du nouveau QuadTree et répéter ensuite les mêmes étapes. Ce n'est pas optimum (nous ne profitons pas de la pleine puissance de calcul du GPU) mais c'est facile à mettre en place.

Il est aussi possible plutôt que d'utiliser un QuadTree d'utiliser différents niveaux de détails (LOD). En effet nous pouvons transformer le système en une texture d'une certaine résolution puis, par des opérations de floutage et d'interpolations, générer des textures de plus faible résolution. Ces textures reviennent à calculer les centres de masses des différentes régions. Le calcul sur une particule utilise alors les différents niveaux de détails avec un facteur d'importance différent.

Sinon il existe d'autres méthodes, par exemple la méthode multipolaire rapide (FMM).

Il est aussi possible d'utiliser une prédiction des positions afin de ne reconstruire que les sous-parties en ayant besoin plutôt que de reconstruire la totalité du QuadTree à chaque itération.

4 Particle-Mesh (PM)

Jusqu'à maintenant nous avons calculé la force à partir de son expression. Que ce soit avec des approximations, des modifications ou de manière directe. Cependant, il est aussi possible d'utiliser le fait que la force gravitationnelle est conservative et dérive donc d'une énergie potentielle ϕ_G : $\vec{F} = -\nabla\phi_G$. Le problème est ainsi déplacé sur le calcul du potentiel et non plus celui de la force.

De ce fait, il est nécessaire d'avoir une équation nous permettant de calculer le potentiel. Cette équation est l'équation de

Poisson qui relie le laplacien du potentiel à la densité de matière :

$$\Delta\phi_G = 4\pi G\rho \quad (4)$$

Cette équation nécessite de calculer une densité. Vu que la résolution numérique de cette équation va se faire à l'aide des différences finies, il nous faut définir une grille ou encore un maillage (« mesh ») avec une résolution n . Il s'agira d'une matrice de taille $n \times n$ pour le cas 2D et d'un vecteur de taille n pour le cas 1D. La densité pourra ensuite être calculée en comptant le nombre de particules dans chaque cellule ou bien par une interpolation sur la position de chaque particule pour plus de précision.

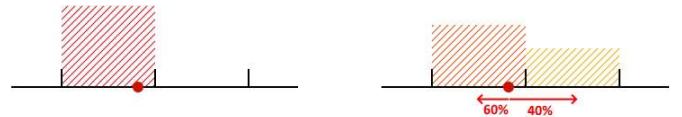
De manière classique, une fois la force obtenue, il est nécessaire d'utiliser ensuite une méthode d'intégration. Ici, nous utiliserons la méthode « leapfrog » pour fournir de la diversité au document.

4.1 Mise en place 1D

Commençons par le cas 1D afin de voir comment l'équation se résout numériquement.

Regardons chaque élément et opérateur qui compose l'équation de Poisson :

- ϕ_G est le potentiel, il a la même dimension que l'espace. C'est à dire ici 1 dimension, ainsi ϕ_G est un vecteur à n composantes. Dans la suite nous noterons ses composantes : ϕ_i avec i son indice spatiale, $0 < i \leq n$.
- ρ est aussi un vecteur, dans le cas 1D, avec n composantes : ρ_i . La densité ρ_i est calculée en comptant le nombre de particules dans le sous-espace i délimité par $r \in [x_i, x_{i+1}]$ avec x_i les positions des noeuds du maillage. Dans le cas 1D, il est aisément de faire une extrapolation afin de distribuer la densité dans plusieurs cellules du maillage à la place d'une seule. En effet si nous comptons les particules naïvement cela pose des problèmes au niveau des frontières. Par exemple, si une particule est proche de la limite avec une autre cellule elle ne sera comptée que dans une seule des cellules avec un poids de 1. Avec une extrapolation les densités des deux cellules voisines seraient incrémentées d'un poids de 0.501 et 0.499 à la place de 1.000 et 0.000. Ainsi, les particules sont comptées de façon plus correcte.



(a) Sans extrapolation

(b) Avec extrapolation

Ces schémas mettent bien en évidence que sans extrapolation la position « locale » de la particule n'a pas d'importance.

C'est à dire que cela revient à par exemple arrondir les positions des particules à des entiers. Il n'y a donc aucune différence entre des particules à des positions : 3.71 et 3.16.

TABLE 8 – PM - 1D - Extrapolation Densité

L'incrémentation de la densité pour une particule de position r s'écrit donc : $\begin{cases} \rho_i+ = \frac{x_{i+1}-r}{x_{i+1}-x_i} \\ \rho_{i+1}+ = -\frac{x_i-r}{x_{i+1}-x_i} \end{cases}$, et si nous défini-

nissons $\Delta x = x_{i+1} - x_i$ le pas de discréétisation de l'espace :

$$\begin{cases} \rho_i+ = \frac{x_{i+1}-r}{\Delta x} \\ \rho_{i+1}+ = -\frac{x_i-r}{\Delta x} \end{cases}$$

— À une dimension, l'opérateur laplacien Δ est : $\frac{d^2}{dx^2} = d_x^2$.

En discréétisant l'espace avec un pas Δx nous trouvons l'opérateur gradient : $d_x \phi_i \approx \frac{\phi_{i+1}-\phi_i}{\Delta x} \sim \frac{\phi_i-\phi_{i-1}}{\Delta x} \sim \frac{\phi_{i+1}-\phi_{i-1}}{2\Delta x}$. Si nous appliquons l'opérateur gradient à nouveau : $d_x d_x \phi_i = \frac{1}{\Delta x^2} \times (\phi_{i+1}-2\phi_i+\phi_{i-1})$. (*L'astuce pour avoir ce résultat est de dire qu'il est équivalent analytiquement de prendre un espace avec un pas deux fois plus petit et donc $d_x \phi_i = \frac{2}{\Delta x}(\phi_{i+0.5}-\phi_i)$ et de prendre une combinaison de la dérivée droite et de la dérivée gauche.*)

En écriture matricielle :

$$d_x = \nabla = \begin{pmatrix} 0 & 1 & & \dots & -1 \\ -1 & 0 & 1 & & \\ & -1 & 0 & 1 & \\ \vdots & & & \ddots & \\ & & -1 & 0 & 1 \\ 1 & & & -1 & 0 \end{pmatrix} \quad (5)$$

Les deux éléments dans les coins sont non nuls car nous avons besoin de conditions aux limites pour résoudre le système. Nous prenons ici la périodicité donc $\phi_1 = \phi_{i+1}$. De la même manière, pour le laplacien :

$$\Delta = \begin{pmatrix} -2 & 1 & & \dots & 1 \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ \vdots & & & \ddots & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{pmatrix} \quad (6)$$

Finalement, en 1D, l'équation de Poisson s'écrit :

$$\begin{pmatrix} -2 & 1 & & \dots & 1 \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ \vdots & & & \ddots & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_{n-1} \\ \phi_n \end{pmatrix} = 4\pi G \begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \vdots \\ \rho_{n-1} \\ \rho_n \end{pmatrix} \quad (7)$$

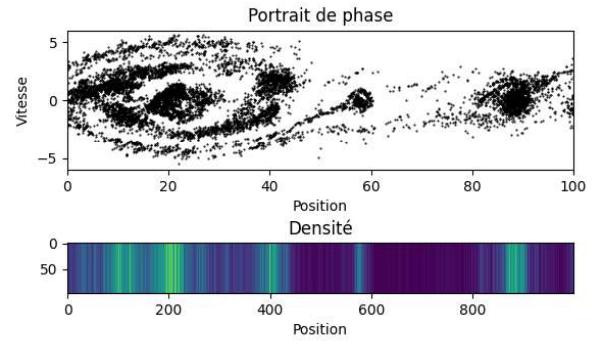
Il faut donc résoudre cette équation matricielle en utilisant une méthode comme celle de Gauss-Seidel présentée au début de ce document. Cette étape de résolution est celle qui présente le plus d'opérations. C'est donc l'étape la plus lourde numériquement.

L'étape qui vient après est de calculer le gradient de ϕ pour obtenir la force :

$$\begin{pmatrix} 0 & 1 & & \dots & -1 \\ -1 & 0 & 1 & & \\ & -1 & 0 & 1 & \\ \vdots & & & \ddots & \\ & & -1 & 0 & 1 \\ 1 & & & -1 & 0 \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_{n-1} \\ \phi_n \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{n-1} \\ F_n \end{pmatrix} \quad (8)$$

La dernière étape est l'intégration sur le temps.

Comme il s'agit d'un cas en une dimension, nous pouvons tracer le portrait de phase des particules :



$N=10000$ particules, $n = 1000$

FIGURE 12 – PM - 1D - Simulation

Nous pouvons observer la formation de 3 ou 4 attracteurs stables. Les particules prennent bien une vitesse importante lorsqu'elles s'approchent de ces attracteurs. Le portrait de phase souligne aussi qu'il y a des échanges de particules entre les attracteurs. Les particules échangées sont celles à grande vitesse. Ceci est logique car ce sont celles qui ont une vitesse suffisamment importante pour se libérer de leur attracteur.

Ce cas 1D fonctionnant bien, nous pouvons généraliser ce cas à 2 dimensions, voire à trois dimensions.

4.2 Mise en place 2D

4.2.1 Système à résoudre

En 2 dimensions ρ et ϕ deviennent des matrices et l'opérateur laplacien devient un tenseur d'ordre 3 tels que $\Delta\phi = (d_x^2 + d_y^2)\phi = \frac{1}{\Delta x^2}(\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1} - 4\phi_{i,j})$ pour un maillage avec un pas Δx sur les axes y et x. (Il existe plusieurs manières de l'exprimer, par exemple en mettant un poids non nul aux voisins diagonaux, exemple : $\phi_{i+1,j+1}$.)

Or nous pouvons remarquer que $4\Delta x^2\pi G\rho$ $\underset{\text{collapse}}{\sim} \phi_{i,j-1} - K\phi_i + \phi_{i,j+1}$ avec $K = \begin{pmatrix} -4 & 1 & & \dots & 1 \\ 1 & -4 & 1 & & \\ & 1 & -4 & 1 & \\ \vdots & & & \ddots & \\ & & 1 & -4 & 1 \\ 1 & & & 1 & -4 \end{pmatrix}$ similaire au laplacien 1D.

Il faudrait donc pouvoir réduire ce tenseur d'ordre 3 à une matrice et ρ et ϕ à des vecteurs afin de résoudre ce système de la même manière que pour le cas 1D.

En utilisant le produit de kronecker, il vient :

$$— \Delta\phi = I_n \otimes K + K \otimes I_n = \begin{pmatrix} K & I_n & & \dots & I_n \\ I_n & K & I_n & & \\ & I_n & K & I_n & \\ \vdots & & & \ddots & \\ & & & I_n & K & I_n \\ I_n & & & & I_n & K \end{pmatrix}$$

avec I_n l'identité et K la matrice similaire au laplacien 1D. Tous deux sont des matrices $n \times n$ où n est la résolution du maillage. Ce qui donne finalement une matrice $\Delta\phi : n^2 \times n^2$

— Ceci implique également que $\rho = \begin{pmatrix} \rho_{1,1} \\ \rho_{1,2} \\ \vdots \\ \rho_{n,n} \end{pmatrix}$ et $\phi = \begin{pmatrix} \phi_{1,1} \\ \phi_{1,2} \\ \vdots \\ \phi_{n,n} \end{pmatrix}$. Il s'agit donc de vecteurs à n^2 composantes.

Le système à résoudre en utilisant une méthode numérique est finalement :

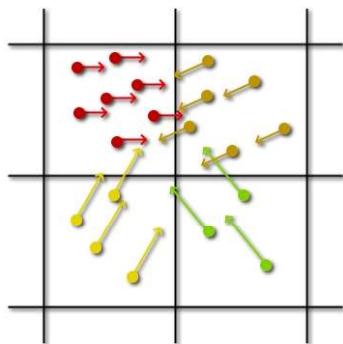
$$\begin{pmatrix} K & I_n & & \dots & I_n \\ I_n & K & I_n & & I_n \\ & I_n & K & I_n & \\ & & \ddots & & \\ & & & I_n & K & I_n \\ & & & & I_n & K \end{pmatrix} \begin{pmatrix} \phi_{1,1} \\ \phi_{1,2} \\ \vdots \\ \phi_{n,n} \end{pmatrix} = 4\pi G \begin{pmatrix} \rho_{1,1} \\ \rho_{1,2} \\ \vdots \\ \rho_{n,n} \end{pmatrix} \quad (9)$$

Le module « `scipy` » ayant une fonction permettant de réaliser le produit de kronecker entre deux matrices, il n'est pas compliqué de construire la matrice du laplacien.

Le temps de calcul dépend essentiellement de la résolution du maillage et moins du nombre de particules. En effet, la matrice $n^2 \times n^2$ croît très vite avec n la résolution.

4.2.2 Calcul de ρ

Comme pour le cas 1D, nous pouvons naïvement compter le nombre de particules par cellule et normaliser en fonction du nombre total de particules. Cependant cette méthode implique que lors du calcul des forces, l'ensemble des particules d'une cellule subit les mêmes forces.

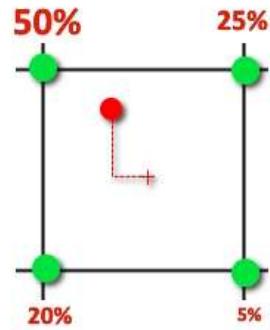


Les couleurs permettent de différencier les différentes cellules.

FIGURE 13 – PM - 2D - ρ calculé naïvement

Afin d'améliorer ce point, il est possible de distribuer un certain poids aux noeuds du réseau en fonction de la position de la particule dans la cellule. Ces poids sont données par :

$$\begin{cases} w_\nwarrow = (1-x)(1-y) \\ w_\nearrow = (1-x)y \\ w_\swarrow = (1-y)x \\ w_\searrow = xy \end{cases} \quad \text{avec } x, y \text{ les coordonnées locales de la particule dans la cellule, allant de 0 à 1.}$$



En réalité pour que ρ soit distribué avec les cellules voisines, il faut prendre leurs centres comme noeuds du réseau.

FIGURE 14 – PM - 2D - Distribution de poids pour le calcul de ρ

Si la résolution est bonne (soit un grand temps de calcul) alors la différence peut être négligeable. Cependant faire cette extrapolation ne coûte presque rien en opérations et peut rajouter de la précision à la simulation.

4.3 Résultats

Une simulation avec une résolution $n = 500$ peut prendre jusqu'à 10h de calcul sur un seul cœur. Cette résolution, bien que semblant basse, fournit néanmoins des résultats riches. Nous pouvons ainsi observer des jets de matière, des corps denses stables ou encore la formation de filaments.

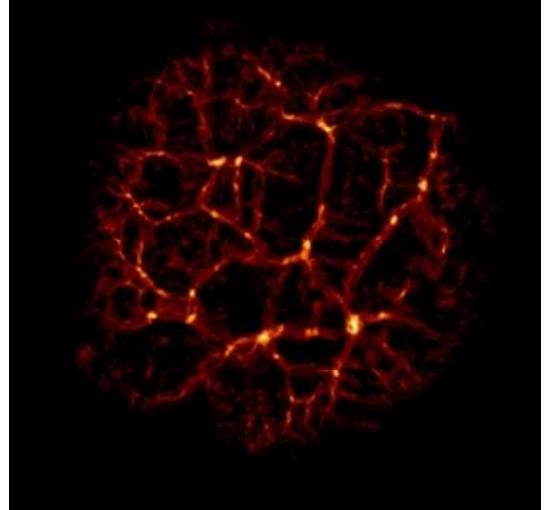


FIGURE 15 – PM - 2D - Filaments de matière après expansion

4.4 Axes d'améliorations possibles

Plusieurs améliorations sont possibles aussi bien sur le calcul de ρ que sur le maillage en lui-même. En effet, il est possible de faire un maillage adaptatif en utilisant un QuadTree afin d'avoir une meilleure résolution aux endroits où la densité est élevée et une basse résolution aux endroits vides de la simulation pour économiser des opérations. Cependant cette méthode nécessite de réviser la façon dont la matrice est construite.

Par ailleurs, résoudre l'équation de Poisson de manière directe n'est pas la seule manière pour accéder au potentiel. Nous pouvons aussi le trouver en passant dans l'espace de Fourier et en utilisant une fonction de Green. Cette méthode est plus utilisée aujourd'hui que la résolution directe.

Conclusion

Finalement, dans ce document, nous ne faisons qu'effleurer les possibilités et les méthodes numériques reliées à notre sujet. Ce problème physique est exceptionnel sur le plan numérique de par la pluralité et les différents niveaux de complexité des méthodes. Cette diversité permet par exemple d'apprendre des méthodes numériques en suivant un seul et même problème et ensuite d'appliquer ces méthodes et outils sur d'autres sujets. Ce document aurait pu compter quelques centaines de pages supplémentaires sans être hors sujet ou vide d'approches nouvelles à explorer. Nous n'avons même pas exploré plus en détail l'évolution de différentes situations initiales. C'est à dire que nous n'avons pas réellement sondé les possibilités de l'outil que nous avons codé. Mais, là où le nombre d'approches numériques et de situations physiques ne semblent pas être limitées, le temps lui l'est.

Enfin, le GPU n'est ici pas sollicité. Or il peut réduire considérablement le temps de calcul. Paralléliser Gauss-Seidel n'est cependant pas chose aisée mais reste possible. Il serait également possible d'utiliser d'autres méthodes de résolution de système matricielle.

A Pseudo-Code → Python

Cette annexe contient une aide pour convertir le pseudo-code utilisé dans ce document ainsi que les conventions associées à un code python.

Saut de texte/code

Pour ne pas avoir à réécrire un morceau de code contenu dans une partie que l'on modifie, il peut y avoir des “...” qui apparaissent. Ils signifient simplement qu'il y a à cet endroit (ou peut y avoir) un morceau de code existant.

Il peut aussi y avoir “...LISTE” qui apparaît avec à la place de “LISTE” le nom d'une variable. Cela signifie que nous faisons le tour des variables contenues dans la liste. Par exemple “...args” contient les paramètres d'une fonction ou d'un objet. Cela ne fonctionne pas en Python et il faut donc écrire tous les paramètres avec la “,” entre chacun d'eux.

Déclarer une variable

Les variables sont déclarées avec leur type précisé avant le nom. Cela permet par exemple de mieux distinguer les nombres des booleans ou encore d'objets plus complexes comme un vecteur. En python il n'y a pas besoin de préciser le type de la variable donc il doit être enlevé.

Quand il s'agit d'un objet complexe, il y a aussi un “new” qui apparaît avant de créer l'instance. Il est là pour montrer que nous créons bien une nouvelle instance d'un objet et que ce n'est pas une fonction que nous appellons.

En résumé cela veut dire que par exemple :

Algorithme 19 Annexe A : 1.a

```
vec sum_forces = new Vector(0,...,0)
```

devient :

Algorithme 20 Annexe A : 1.b

```
sum_forces = Vector(0,...,0)
```

Fonctions & Objets

Pour convertir une fonction venant du pseudo-code utilisé dans ce document en une fonction python, il faut tout simplement changer “function” par “def”.

Pour créer un objet en Python il faut remplacer “constructor” par une fonction “__init__” et mettre les arguments que nous donnons à l'objet dans la fonction constructor et non dans “class”.

En résumé cela veut dire que par exemple :

Algorithme 21 Annexe A : 2.a

```
class Particle(...args):  
    constructor(self):  
        ...
```

devient

Algorithme 22 Annexe A : 2.b

```
class Particle:  
    def __init__(self,...args):  
        ...
```

B MultiProcessing CPU

Les simulations présentées dans ce document n'utilisent qu'un seul cœur du processeur ou les CPUs ont plusieurs coeurs qui permettent d'exécuter des tâches en parallèle. En général lorsque du calcul en parallèle est réalisé, il faut que les tâches ne soient pas interdépendantes (c'est à dire qu'une tâche ne dépend pas des résultats d'une autre qui est exécutée en même temps). C'est à dire que les tâches exécutées en parallèle ne doivent pas communiquer entre elles.

Pour l'approche directe ou celle utilisant le QuadTree, la solution est plutôt simple pour mettre en place ce processus. En effet faire l'intégration de Verlet sur une particule pour le temps $t + \Delta t$ ne dépend que des états au temps t . Toutes les particules utilisent une seule et même matrice/liste de donnée qui est celle au temps t .

Ainsi, il nous suffit de connaître le nombre de coeurs que nous pouvons utiliser et de diviser le nombre de particules par le nombre de coeurs afin de donner à chaque cœur une fraction des particules à calculer. Lorsque chaque cœur a fini nous récupèrons les matrices résultantes pour en former une seule que nous passerons pour le prochain temps et ainsi de suite. Ici, notre matrice est simplement la liste des particules (plus compliqué pour un QT).

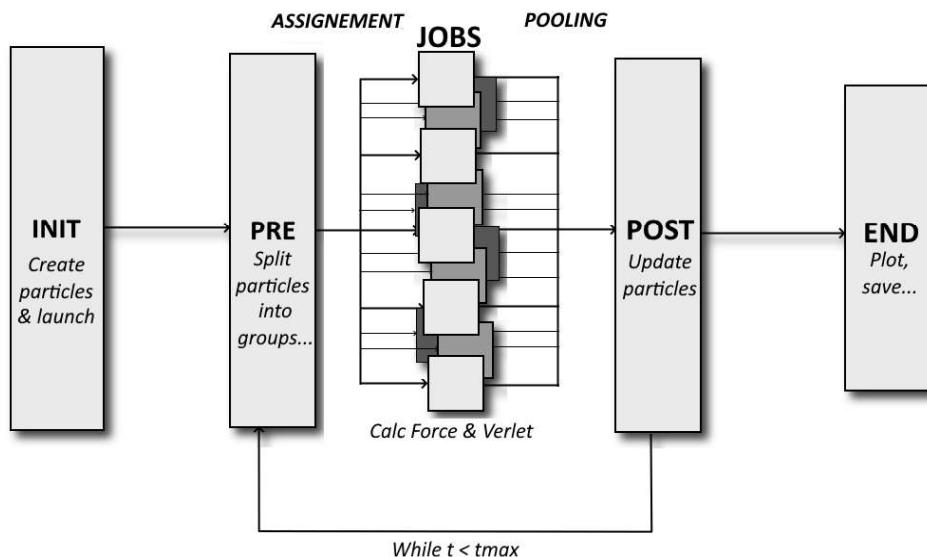


FIGURE 16 – Annexe B : Schéma Multiprocessing

Sur Python il est assez simple de l'intégrer :

Algorithme 23 Annexe B : Multiprocessing CPU - Forme général

```

import multiprocessing
#your task
def process(...args,i,DATA):
    ...
    #update the result
    DATA[i] = ...

#Send the tasks to each core
manager = multiprocessing.Manager()
DATA = manager.dict()
jobs = []
for i in range(16):
    p = multiprocessing.Process(target=process,args=(...,i,DATA))
    jobs.append(p)

#Start the tasks
for j in jobs:
    j.start()

#Join the tasks
for j in jobs:
    j.join()

```
