SIM 202

Simulation gravitationnelle

Anthony KALAYDJIAN - Mathieu OCCHIPINTI - Juliette TREYER

Mars 2023

Table des matières

Introduction

Les simulations gravitationnelles d'un amas de particules sont des problèmes relativement simples, qui peuvent devenir complexes lorsque le nombre de particules augmente. En effet, pour pouvoir calculer les forces qui s'exercent sur une particule, il faut calculer la somme des forces qu'exercent chacune des autres particules sur celle-ci.
Calculer les forces qui s'exercent sur l'ensemble des particules créé donc une complexité en $O(n^2)$. Sachant que l'on doit répeter ce processus à chaque pas de temps de notre simulation, le temps de calcul augmente drastiquement.
Minimiser le temps de calcul des forces est donc une pratique très fructueuse. Un algorithme permettant d'effectuer une telle tâche est l'algorithme de Barnes Hut.

Cet algorithme repose sur la création d'un arbre, permettant de classer les particules, selon leur position géographique au sein de groupes de particules qui sont des boîtes (des carrés en 2 dimmension et des cubes en 3 dimmensions).
On peut ensuite décider de calculer la force exacte qu'exerce une particule sur notre particule, ou bien, lorsqu'un groupe de particules est assez loin, ne considérer ce groupe comme n'étant qu'une particule placée au centre de masse du groupe et de masse égale à la somme des masses des particules du groupe.
Ce procédé permet ainsi de réduire la complexité du calcul des forces pour atteindre une complexité en $O(n \log n)$ ce qui est très utile lorsque l'on cherche à faire de très grandes simulations.

1 Classes utilisées

1.1 Classe particle

La classe particle représente une particule élémentaire de notre galaxie.
Ainsi, ses données membres sont :

- Sa masse qui est un réel.
- Sa position qui est un vecteur de 3 réels dont les éléments sont respectivement sa coordonnée selon l'axe x,y et z.
- Sa vitesse qui est un vecteur de 3 réels dont les éléments sont respectivement sa vitesse selon l'axe x,y et z.
- Un vecteur speed-approx qui est le vecteur de la vitesse estimée en $kdt + \frac{1}{2}$.
- La force qui s'exerce sur la particule qui est un vecteur de 3 réels dont chaque composante correspond à la composante de la force dans une des direction du plan.
- Un vecteur de vecteur de positions. Cela représente les positions successives prises par la particule. Le premier vecteur est la position initiale de la particule, le second celui de la particule après une itération et ainsi de suite.
- Un pointeur de particule pointant sur la prochaine particule.

La classe particle est aussi dotée de plusieurs méthodes membres.

- Une fonction d'affichage des principales caractéristiques de la particule : ses coordonnées , sa vitesse ainsi que la force qui s'exerce sur la particule selon les 3 axes.
- Plusieurs constructeurs :
  - Le premier qui ne prend aucun paramètre et renvoie une particule à l'origine, sans vitesse et avec aucune force qui ne s'exerce dessus.
  - Un deuxième où on donne des coordonnées initiales, une vitesse initiale à la particule ainsi que les forces qui s'exercent dessus. Toutefois, la masse de la particule n'est pas spécifiée et est prise par défaut égale à 1.
  - Le dernier est le même que le second sauf que cette fois-ci on spécifie une masse à donnée à notre particule
- Une fonction qui nous permet de réinitialiser notre particule en remettant toutes les composantes à 0.
- Un destructeur.
- Une surcharge de l'opérateur == qui renvoie TRUE si les particules sont identiques (même masse, position, vitesse, vitesse estimée, force subie, positions successives, pointeur sur la particule suivante) et FALSE sinon.
- Une fonction set-position qui à partir d'un vecteur à 3 composantes réels fixe la position de notre particule sur ces mêmes composantes.
- Une fonction set-speed qui a le même comportement que set-position mais pour la vitesse.

1.2 Classe box

La classe box est la classe qui représente une boîte dans laquelle nous allons ranger particules et sur lesquelles nous appliquerons nos fonctions par récursivité sur celles-ci.

Ses données membres sont :

- Un entier qui représente le niveau de la boîte, la boite initiale possède le niveau 0 puis les

premières sous-boîtes le niveau 1 et ainsi de suite.
— Un vecteur de réels représentant les coordonnées du centre de la boîte.
— Un vecteur de réels représentant les coordonnées du centre de masse de la boîte.
— Un réel qui correspond à la masse de la boîte c'est à dire la masse de la particule dans la boîte ou la somme des masses des sous boîtes de notre boîte.
— Un pointeur pointant sur la particule présente dans la boîte où null si la boîte ne contient pas de particule.
— Un pointeur pointant sur le premier élément d'une liste chaînée de boîtes correspondant aux sous boîtes de la boîte, null s'il n'y a pas de sous boîtes.
— Un pointeur pointant sur le premier élément d'une liste chaînée de boîtes correspondant aux boîtes soeurs de notre boîte, null si elle n'a pas de boîtes soeurs.

La classe boîte est également dotée de méthodes,

— Une fonction pour initialiser une boîte qui remet à 0 ou à null les différentes données membres de la boîte.
— Une fonction renvoyant un vecteur de vecteur contenant les centres de toutes les sous boîtes de la boîte.
— Un premier constructeur qui crée une boîte où les données membres sont initialisées à 0.
— Un second constructeur qui prend en argument un niveau, un centre, un centre de masse, une masse ainsi que des pointeurs pour initialiser les différentes données membres.
— Un destructeur.
— Une fonction force, qui calcule une approximation de la force exercée par la boîte sur une particule donnée en argument.
— Une fonction ajoutant une particule dans une boîte ou divisant la boîte en sous boîtes si la boîte contient déjà une particule.
— Une fonction d'affichage des principales caractéristiques de la boîte.

2 Algorithme de Barnes Hut

2.1 Arbre

float En considérant la classe box qui a été introduite précédemment, un arbre sera constitué comme ceci :
Chaque boîte pointera vers un éventuel successeur de même niveau, à savoir une boîte soeur, et vers un éventuel successeur de niveau incrémenté, à savoir sa première sous boîte.
Visuellement, on découpera chaque boîtes en 2^d sous-boîtes identiques, où d représente la dimmension de notre problème (La structure que nous avons choisie permet de facilement généraliser l'algorithme à des dimmensions plus élevées !). Ainsi, en dimmension 3, chaque cube sera divisé en 8 sous-cubes identiques et en dimmension 2, chaque carré sera divisé en 4 sous-carrés identiques.
Visuellement, on obtient la figure 1.

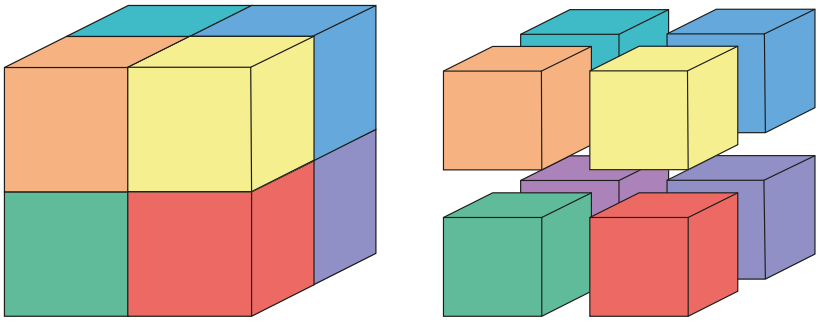

FIGURE 1 – Décomposition d'une boîte en sous-boîtes

La boîte mère ne possède pas de soeur. Le schéma de notre arbre sur les deux premier niveaux peut ainsi être le suivant :

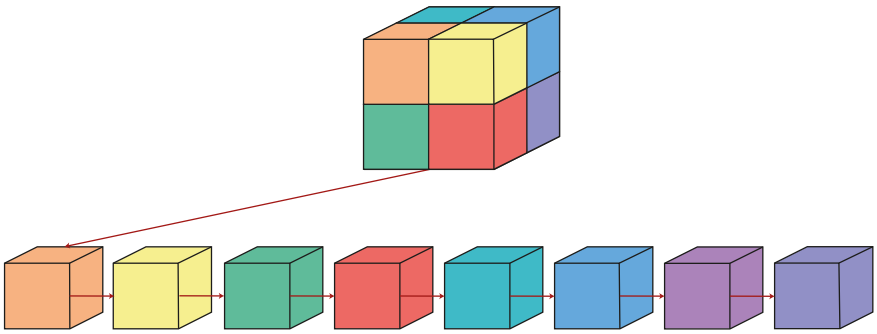

FIGURE 2 – Décomposition d'une boîte en sous-boîtes

On considèrera également qu'une boîte ne contient une particule que si elle est terminale, i.e. qu'elle ne contient pas de sous-boîtes.
Ainsi, une représentation en 2D (pour faciliter la schématisation) d'une décomposition en arbre d'un ensemble de particules est la suivante :

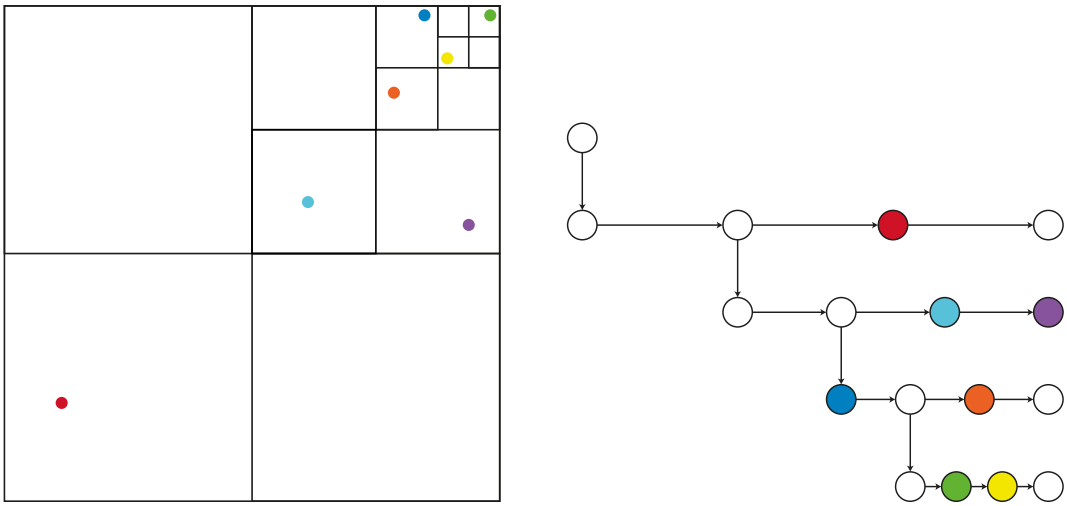

FIGURE 3 – Décomposition en arbre de particules

Dans cette figure, les noeuds du graphe représentent une boîte. Chaque noeud peut donc avoir un successeur au même niveau (boîte soeur) ou de niveau plus bas (sous-boîte). Les noeuds coloriés représentent les boîtes terminales qui possèdent une particule.

La création de l'abre s'effectuera en y ajoutant les particules une par une. Le pseudo-code de cet algorithme est le suivant :

```
//append a particle to a box
void box::append_particle(particle& part){
    //if there is no sub_box and no prior particle in the box, we append the particle to it
    if ((p_sub_box==nullptr) && (p_particle==nullptr)){
        put_particle_in_this_box();
    }

    //if the box has sub_boxes, we find the one that contains the particle and call our
    //function back on that box, we then adjust the center of mass of the current box
    else if (p_sub_box != nullptr){
        ptr = sub_box_that_contains(particle);
        if (ptr != nullptr){
            ptr->append_particle(part);
            adjust_mass(box);
            adjust_mass_center(box);
        }
    }

    //in the other case, if the box doesn't have sub_boxes, but already has a particle in it,
    //we create the sub_boxes and append the two particles to their respective sub_boxes
    else if (p_sub_box == nullptr){
        p_sub_box = create_sub_boxes();

        //we remove the particle that was in the box and the function on the two particles
        particle* tmp = p_particle;
        mass_center = center;
        mass = 0.0;
        p_particle = nullptr;
        append_particle(part);
        append_particle(*tmp);
    }

}
```

On laissera au lecteur le plaisir d'étudier la figure 4 qui montre comment s'effectue l'ajout d'une particule dans deux cas.
A chaque itération sur une boîte ne contenant pas notre particule, on vérifie si elle est assez loin pour pouvoir la considérer comme une unique particule située à son centre de masse. Si c'est le cas, on calcule la force qu'exerce ce centre de masse sur notre particule et on oublie les particules qui sont dans cette boîte.
Formellement, le critère à tester est le suivant :

$$R := \frac{||P_{particle} - P_{mass\_center}||}{box\_size} < \Theta$$

Avec :

- $P_{particle}$: la position de notre particule.
- P_{mass_center} : celle du centre de masse de la boîte.
- box_size : la taille de la boîte.
- Θ : un paramètre que l'on prendra arbitrairement égal à 0.5 ($\Theta = 0$ correspond au calcul exact de la force).

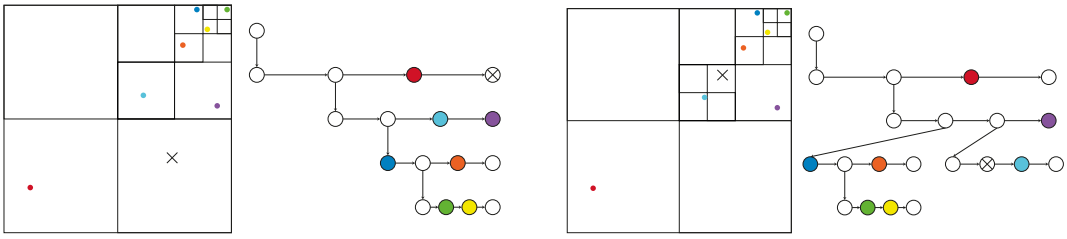

FIGURE 4 – Ajout d'une particule X dans l'arbre

— Dans le premier cas, on commence par regarder la boîte mère. Elle possède des sous boîtes donc on cherche la sous boîte qui peut contenir notre particule. Il s'agie de celle en bas à droite, qui ne possède ni de particule, ni de sous boîte. On peut donc tout simplement y ajouter notre particule.
— Dans le deuxième exemple, On réeffectue la manoeuvre, de la boîte mère on passe à la sous-boîte en haut à droite, puis de cette sous boîte on passe à sa sous-boîte en bas à gauche. Cette dernière possède déjà une particule, on la découpe alors en 4 et on ajoute nos particules dans leur sous-boîte respectives. On n'oubliera pas de retirer la particule de son ancienne boîte.

2.2 Calcul approché des forces

Le calcul des forces approchées se fera également récursivement. La récursion sera opérée sur les boîtes de l'abre. Le pseudo-code étant le suivant :

```
// Approximate calculation of the force of a box on a given particle
void box::force(particle& part){
    if (this==nullptr){
        return;
    }
    if (level==0){
        part.force = vecteur<double>(3, 0.0);
    }
    if (p_particle != nullptr){
        //we split the cases on the particle to avoid singularities
        if (p_particle != &part){
            //classic force calculation, with the addition of an EPSILON to avoid
            //singularities due to particle collisions
            vecteur<double> force_particle = usual_force_exerted_on_particle(particle);
            part.force = part.force + force_particle;
        }
    }
    //if the particle is in the box, we don't calculate the approximate force,
    //we call back the function on the sub_boxes
    else if (is_in_box(part, *this)){
        p_sub_box->force(part);
    }
    else{
        double box_size = LENGTH/pow(2, level);
        double distance = norm(part.position - mass_center);
        double ratio = box_size/distance;

        //If the following criterion is fulfilled, we assume that the box is far enough
        //to the particle to consider approximating the resulting force
        //of the center of mass of the box, rather than the individual forces
        //of each particle containned in the box
```

```
        if (ratio < THETA){
            vecteur<double> force_box = (G*part.mass*mass*(1/(pow(norm(mass_center - part.position),
            part.force = part.force + force_box;
        }
        //in the other case, we recursively call our function on the sub_boxes
        else{
            p_sub_box->force(part);
        }
    }
    //we finish by calculating the resulting force exerted by the sister boxes
    p_sister_box->force(part);
}
```

On propose dans la figure 5 un exemple de calcule de la force qui s'exerce sur la particule rouge.

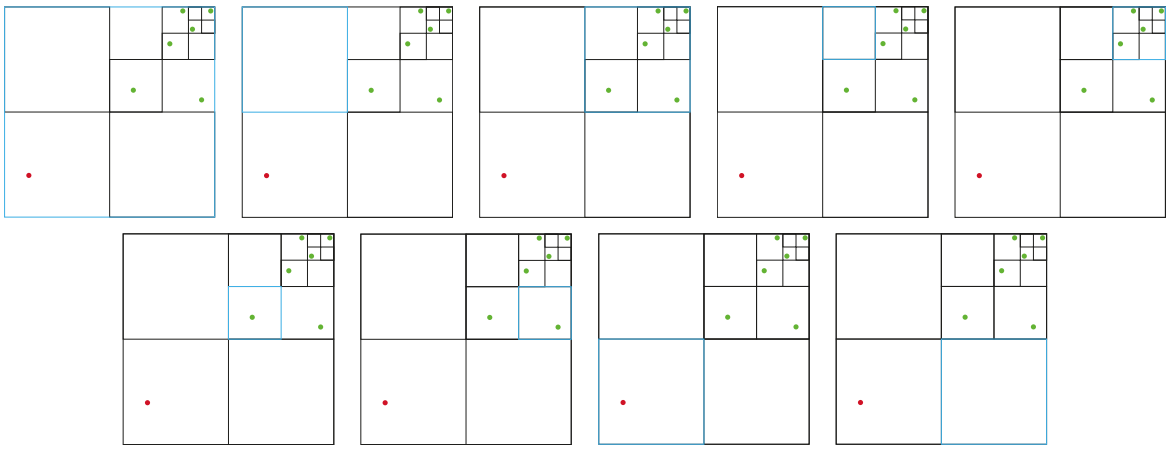

FIGURE 5 – Exemple de calcul des forces sur la particule rouge

La procédure de calcul des forces sur la particule rouge est la suivante, chaque énumération faisant référence à une étape dans la figure 5.

1. On considère d'abord la boîte mère. Elle possède des sous boîtes et contient la particule, on passe donc aux sous boîtes.
2. La première sous-boîte ne contient pas la particule et est vide. On passe à sa soeur.
3. On calcule le critère, imaginons qu'il ne soit pas vérifié. Comme cette boîte a des sous-boîtes, on passe à sa sous-boîte.
4. Cette boîte est vide, on passe à sa soeur.
5. On calcule le critère, imaginons qu'il soit vrai. On calcule alors la force qu'exerce le centre de masse de cette boîte sur la particule. On passe à sa soeur.
6. Cette boîte possède une particule, on calcule la force qu'elle exerce sur notre particule. On passe à sa soeur.
7. IDEM. Il n'y a plus de soeur, on remonte la récursion nous fait remonter d'un niveau.
8. Cette boîte contient notre particule, on ne touche à rien. On passe à la boîte soeur.
9. Cette boîte est vide, on ne touche à rien. Comme cette boîte n'a pas de soeur, la récursion nous fait remonter d'un niveau. Comme la boîte mère n'a pas de soeur, le programme est fini.

2.3 Evolution Dynamique

Afin de simuler le comportement dynamique de l'ensemble des particules, nous discrétisons la deuxième loi de Newton et utilisons un schéma saute-mouton. Plus précisément, en introduisant un pas de temps constant Δt, et en notant $t_k = k\Delta t$ et $t_{k+\frac{1}{2}} = (k+\frac{1}{2})\Delta t$, on a :

— X_i^k correspond à une approximation de la position de la particule i à l'instant t_k

— $V_i^{k+\frac{1}{2}}$ correspond à une approximation de la vitesse de la particule i à l'instant $t_{k+\frac{1}{2}}$

Le schéma saute-mouton consiste donc à calculer à chaque itération :

— $V_i^{k+\frac{1}{2}} = V_i^{k-\frac{1}{2}} + \frac{\Delta t}{m_i} F_i^k$

— $X_i^{k+1} = X_i^k + \Delta t V_i^{k+\frac{1}{2}}$

Pour l'initialisation, on calcule $V_i^{\frac{1}{2}}$ à l'aide de la formule : $V_i^{\frac{1}{2}} = V_i^0 + \frac{\Delta t}{2m_i} F_i^{k0}$

La simulation du comportement dynamique de l'ensemble des particules se fait à partir de notre fonction dynamic-iteration (située dans le fichier plummer.hpp). Cette fonction est appelée directement dans le main lors de chaque itération et prend comme argument la première particule créée ainsi que le numéro de l'itération. Lors de chaque appel, dynamic-iteration effectue les calculs suivants :

— On détruit l'arbre précédent et on crée un nouvel arbre avec les positions actualisées de chacune des particules. Cette étape correspond à la mise à jour de la boîte lors de chaque itération.

— Pour chacune des particules :
  — On calcule les forces exercées sur la particule en appelant la fonction box :force sur la particule.
  — On effectue l'itération saute-mouton en calculant la nouvelle vitesse et la nouvelle position de la particule suivant les formules ci-dessus
  — on enregistre la nouvelle position de la particule dans le vecteur successive-positions de la particule. Cela nous permettera d'avoir accès à toutes les positions succesives de la particule et de réaliser le film Matlab par la suite.

Ce schéma saute-mouton est un schéma d'ordre 2 stable si Δt est assez petit. En pratique, il est nécessaire de choisirΔt plus petit que 10^{-3} pour obtenir une solution cohérente.

3 Système auto-gravitant

Nous souhaitons utiliser cette modélisation afin de simuler l'interaction gravitationelle de galaxies. Une galaxie étant un ensemble d'étoiles dans un état auto-gravitant, on choisit de construire un système auto-gravitant sphérique en utilisant le modèle de Plummer.

Le modèle de Plummer nous permet de construire à partir de tirages aléatoires selon une loi uniforme entre 0 et 1 un couple (position, vitesse) pour chaque particule tout en obtenant une densité isotrope dans l'espace de la forme : $\rho = \frac{3}{4\pi} M R^{-3} (1 + \frac{r}{R})^{-\frac{5}{2}}$,

où M est la masse totale et R un paramètre qui détermine la dimension caractéristique du système de particules.

Nous nous sommes placés dans le cas où M = 1, R = 1, G = 1 et où la masse de chaque particule est $m = \frac{1}{N}$ avec N le nombre de particules.

L'initialisation des couples (position, vitesse) de chacune des particules en suivant le modèle de Plummer se fait au sein de notre fonction plummer-initialisation (dans le fichier plummer.hpp). Cette fonction crée N particules (N étant une constante défini dans constants.hpp) et retourne un pointeur sur la première particule. Les particules sont crées en sens inverse, c'est-à-dire que l'on crée d'abord la dernière des particules, et la première particule est créée en dernière. Nous utilisons les classes default-random-engine et uniform-real-distribution afin d'obtenir des tirages aléatoires selon une loi uniforme. Le pseudo-code de cette fonction s'écrit ainsi :

1. création du pointeur vers la dernière particule
   création du pointeur vers la particule précédente
   création du pointeur vers la particule actuelle
2. initialisation de la dernière particule en accord avec le modèle de Plummer :
   — construction du rayon r à partir du tirage aléatoire X1
   — construction de la position à partir des tirages aléatoires X2 et X3
   — construction de la vitesse d'échappement de la particule à partir des tirages aléatoires X4 et X5
   — construction de la vitesse de la particule à partir des tirages aléatoires X6 et X7
   — enregistrement des valeurs de position et vitesses dans les vecteurs position, speed et successive-positions[0] de la particule
3. Pour i allant de 0 à N-1 :
   — p-previous-particle = p-current-particle
   p-current-particle = new particle()
   p-current-particle->p-next-particle = p-previous-particle
   — initialisation de la particule actuelle en accord avec le modèle de Plummer :
      — construction du rayon r à partir du tirage aléatoire X1
      — construction de la position de la particule actuelle à partir des tirages aléatoires X2 et X3
      — construction de la vitesse d'échappement de la particule actuelle à partir des tirages aléatoires X4 et X5
      — construction de la vitesse de la particule actuelle à partir des tirages aléatoires X6 et X7
      — enregistrement des valeurs de position et vitesses dans les vecteurs position, speed et successive-positions[0] de la particule actuelle
4. création de la boîte mère
5. ajout de chacune des particules à la boîte mère
6. calcul des forces appliquées à chaque particule
7. calcul de la vitesse initiale de chaque particule
8. retour de la première particule

4 Représentation avec Maltlab

On va utiliser MATLAB pour générer un film à l'aide des positions successives des particules obtenues précédemment. Pour ce faire, on commence par importer les données stockées au format csv dans une matrice à l'aide de la fonction readmatrix. On crée ensuite une figure qui est pour l'instant vide.
On va ensuite faire deux boucles for : la première sur les itérations de notre algorithme puis la

seconde qui est imbriquée dans la première sur les particules de notre simulation.

Ainsi, pour une itération k et une particule i, on récupére dans notre matrice les coordonnées x_k^i, y_k^i, z_k^i. Ensuite, à l'aide la fonction plot3 qui nous permet de représenter un point en 3 dimensions nous allons sur une même frame (en utilisant les fonctions hold on / hold off) placer les N particules à une itération donnée k.
Enfin en sortie de boucle sur les particules mais dans la boucle sur les itérations, nous prenons une snapshot à l'aide de la fonction getframe que nous stockons dans un vecteur frame.

Une fois que nous avons obtenu la suite de nos images, nous réalisons un film. Pour ce faire nous commençons par créer un fichier au format avi grâce à la fonction VideoWriter. Enfin nous ouvrons le fichier puis écrivons dedans en utilisant la fonction writeVideo et le vecteur frame obtenu précédemment. Nous terminons par fermer le fichier.

5 Résultats obtenus

Nos codes C++ ainsi que notre code sur Matlab nous permettent de générer un film pour un nombre N de particules et un nombre N-ITER d'itérations, avec un pas de temps Δt.

5.1 Résultat pour deux particules

Dans un premier temps, nous avons choisi de tester nos codes en observant le résultat obtenu pour N=2 partiocules et N-ITER= 1000 itérations. Avec un pas de temps Δt =0.001 secondes, on obtient le résultat suivant :

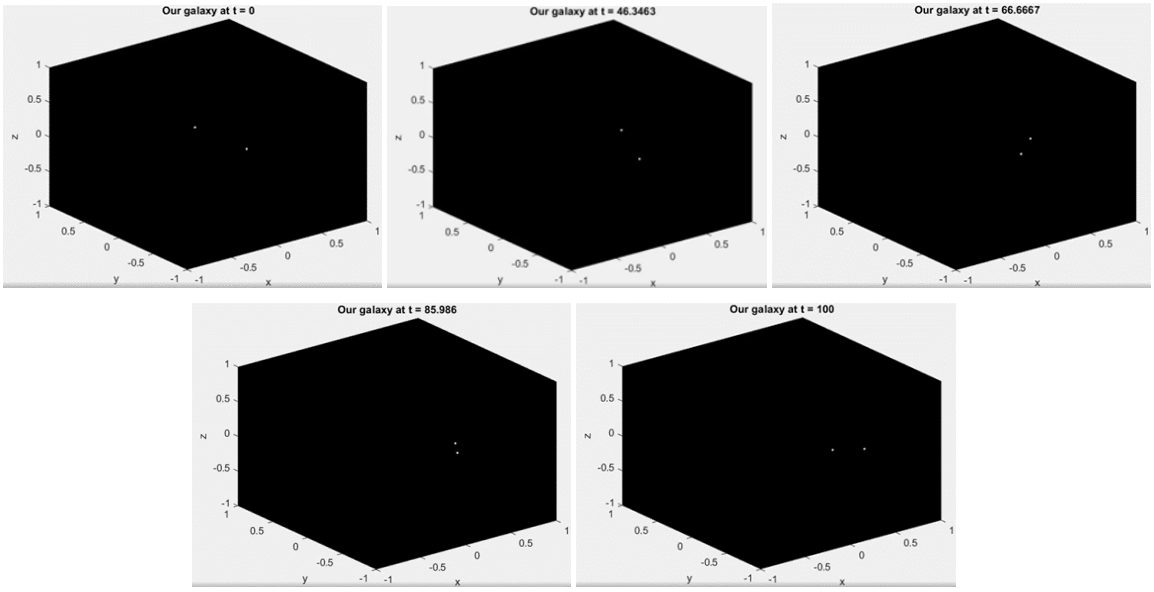

FIGURE 6 – N=2, NITER=1000, Δt=0.001

On voit sur cet exemple que le résultat est assez cohérent sur le début de la vidéo (jusqu'environ t=85). En effet, les deux particules se rapprochent en tournant l'une autour de l'autre, ce qui correspond bien au résultat que l'on avait prévu.

Effectuer ce test nous a également permis de voir l'importance qu'à la valeur du pas de temps Δt. En effet, pour Δt =0.1 s par exemple, les deux particules ne se rapprochent pas mais, au contraire,

s'écartent. Cela est évidemment absurde, la force gravitationelle étant une force attractive. On peut expliquer cela par le fait que le modèle de Plummer n'est stable uniquement pour des petites valeurs de Δt. En testant plusieurs valeurs pour Δt, nous avons commencé à obtnenir des résultats cohérents pour Δt inférieur à 10^{-3}. Cependant, même sur ce test, où Δt=0.001 s, les deux particules s'écartent à la fin (entre t=85 et t=100). On en conclut que notre valeur de Δt n'est toujours pas assez petite, ou qu'il persiste encore un autre problème. Nous n'avons cependant pas pu effectuer les autres tests possibles par manque de temps.

5.2 Résultat pour 100 particules

Une fois notre code à peu près validé, nous avons dans un deuxième temps exécuté notre code pour des valeurs de N=100 et N-ITER=1000. Le résultat obtenu est le suivant :

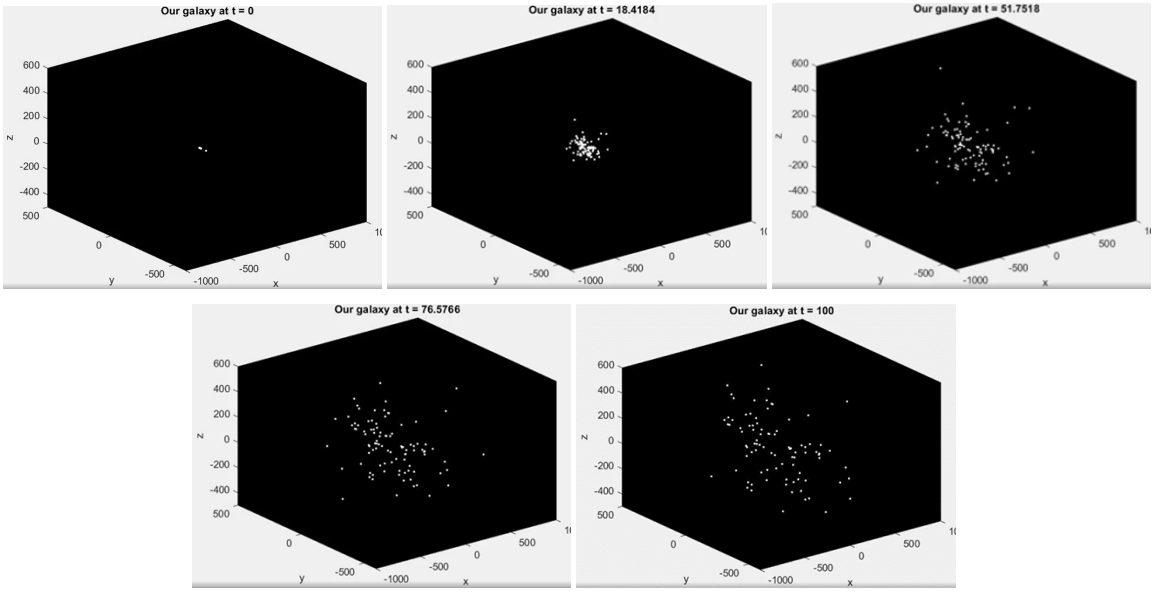

FIGURE 7 – N=100, NITER=1000, Δt=0.1

Cette fois-ci, les particules s'écartent les unes des autres au début de la vidéo, ce qui ne correspond pas à nos attentes. Cependant, à la fin de la vidéo, (à partir de t=75 environ), les particules cessent presque de s'écarter et le système semble être stable.

6 Conclusion

La réalisation de ce projet nous a beaucoup appris, d'abord sur le plan de l'utilisation du langage C++ et de l'algorithmie, mais également niveau de l'organisation de travail de groupe.

Tout d'abord, ce projet nous as permis d'appliquer nos connaissances de SIM201 et de bien progresser quant à l'usage du C++. Plus encore, nous avons pu découvrir grâce à ce projet de nouvelles méthodes de simulation et modélisation. En effet, utiliser l'algorithme de Barnes Hut nous a permis de réduire fortement la complexité des calculs de force et ainsi d'obtenir des résultats satisfaisants avec un temps d'exécution assez faible.

Enfin, ce projet nous a également appris à nous organiser de manière a travailler efficacement en groupe. En désignant des responsables pour chaque classe ou pour certaines fonctions et en mettant bien notre travail en commun sur github, notre travail a pu devenir de plus en plus productif au fil du projet!