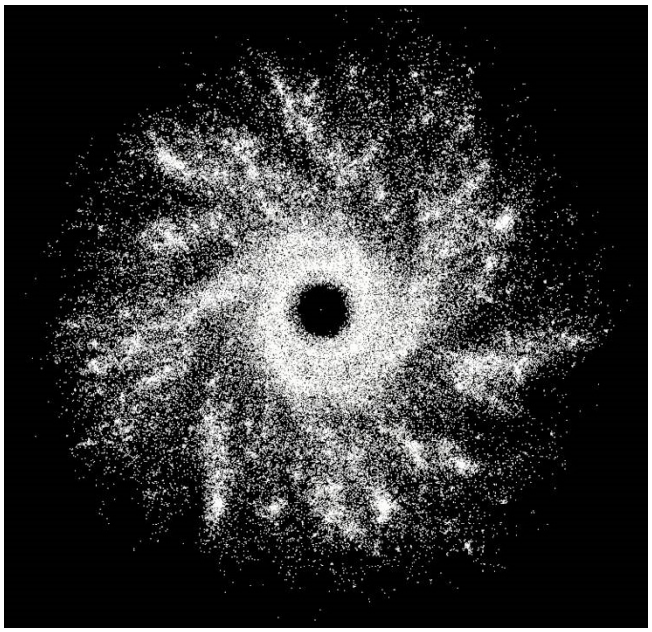
POLYTECH SORBONNE

Problème à N-corps : algorithmes et parallélisations



Auteur :
Rudio FIDA CYRILLE
Noah VEYTIZOUX
BEN-SOUSSEN
Elyas ASSILI
Camille HASCOET

Référent :
Roméo MOLINA

16 mars 2022

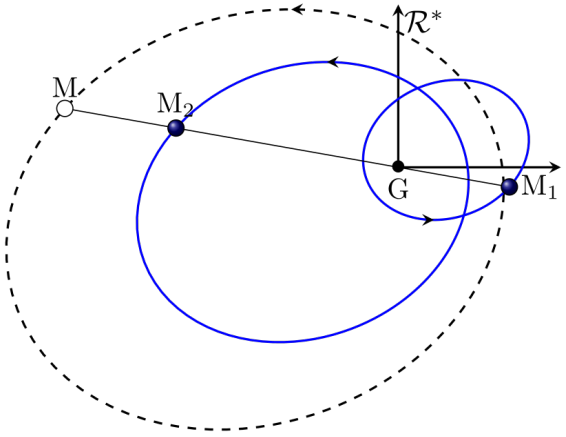
Table des matières

Chapitre 1

Présentation du projet

1.1 Sujet

Le problème a N-corps consiste à calculer le mouvement de N particules en connaissant leur masse et leur vitesse respective. Il s'agit donc de résoudre les équations du mouvement de Newton pour ces N particules. Le problème à N-corps est une simulation classique et importante pour l'astronomie, étant donné qu'il permet d'étudier la mécanique de corps céleste interagissant gravitationnellement. Le problème à deux corps et le problème à trois sont résolubles analytiquement. Pour N>3, il n'existe pas de résolution analytique, il faut donc utiliser des solutions approchées.



$$\begin{cases} a_1 = \dfrac{m_2}{m_1+m_2}a \\ a_2 = \dfrac{m_1}{m_1+m_2}a \end{cases}$$

FIGURE 1.1 – Solution du problème à 2 corps

1.2 Objectif : résolution la plus efficace possible du problème à N-corps

L'objectif du projet est donc de résoudre le plus efficacement possible le problème à n-corps. Pour cela, nous utiliserons des solutions approchées calculées à partir de l'algorithme de Barnes-Hut. Nous étudierons également l'application de la parallélisation à notre programme. En pratique, le projet consiste à continuer et améliorer un projet déjà bien entamé afin d'acquérir de nouvelles compétences en C++, sur les algorithmes hiérarchiques , en parallélisation et plus généralement en optimisation de code.

1.3 Démarche

Un code C++ implémentant une interface graphique nous a été fourni pour le projet. Nous allons dans un premier temps calculer la force qui s'applique à chaque particules à partir de la formule de l'interaction gravitationnelle. La position se calcule alors en utilisant un intégrateur numérique (Méthode d'Euler, saute mouton...). Dans un deuxième temps, nous utiliserons l'algorithme de Barnes-Hut pour le calcul des forces. Enfin, nous utiliserons la parallélisation multi-thread pour accélérer nos calculs.

Chapitre 2

Analyse du code

Notre projet s'effectue dans la continuité d'un projet déjà bien entamé, ainsi, la première étape est de comprendre et d'assimiler le travail qui a déjà été effectué.

2.0.1 Vector.cpp - Vector.h

Le fichier **Vector.cpp** permet la création de vecteurs. On y retrouve un constructeur pour les vecteur de deux dimensions et un deuxième pour les vecteurs de trois dimensions.

```
1  //Exemple d'un des constructeurs :vecteurs en 2 dimensions
2  Vec2D::Vec2D(double a_x, double a_y)
3          :x(a_x)
4          ,y(a_y)
5      {}
```

Le fichier Vector.h associé contient les différentes classes mettant en places les vecteurs.

2.0.2 Types.cpp - Types.h

Ce code ci possède des constructeurs nécessaires pour la mise en place des données liés aux particules. Nous avons :

- Un constructeur à vide des particules
- Un constructeur de particules avec une initialisation par 2 valeurs
- Un constructeur utilisant la structure d'une particule situé dans le Types.h
- Un constructeur servant d'overload, c'est-à-dire qu'elle spécifie plusieurs fonctions possédant le même nom
- Une fonction permettant la réinitialisation à 0 de nos données de particules
- Une fonction booléenne vérifiant si une donnée de particule est vide

Voici par exemple, le constructeur à vide de particules

```
1  \\Constructeur à vide de ParticleData */
2  ParticleData::ParticleData()
3     :m_pState(NULL)
4     ,m_pAuxState(NULL)
5  {}
```

Quand au fichier Types.h, on y retrouve différentes structures :

- **PODState** pour la position et la vitesse d'une particule
- **PODAuxState** pour la masse d'une particule
- **PODDeriv** pour la vitesse et l'accélération d'une particule
- **ParticleData** définissant une particule

2.0.3 ModelNBody.cpp - ModelNbody.h

Ce fichier contient les fonctions permettant la création de notre modèle à N-corps. Nous avons ainsi :

— Une classe permettant la création de notre modèle à N-corps. Il y figure un ensemble de donnée comme par exemple la masse du Soleil ou encore la constante gravitationnelle
— Un destructeur de la classe précédente, dans le but de libérer l'espace alloué par notre classe
— une fonction définissant la région d'étude
— une fonction obtenant le temps des étapes
— une fonction récupérant la région d'étude lors de la visualisation
— une fonction récupérant le centre de masse
— une fonction permettant d'obtenir la direction de la caméra et une autre sa position lors de la visualisation
— Une fonction récupérant l'état initial de la simulation
— une fonction permettant de calculer la vitesse orbitale d'une particule par rapport à une autre particule
— une classe permettant la reintialisation du modèle à N-corps
— une classe qui initialise la simulation du modèle
— une autre classe qui initialise la simulation du modèle mais cette fois avec seulement 3 particules
— une classe permettant l'initialisation de black holes (=trou noir) accompagné de particules aléatoires
— une classe permettant la mise en place de l'arbre(cette notion d'arbre se clarifiera par la suite)
— une classe qui sert à obtenir le noeud racine de l'arbre
— Une classe récupérant les différents noeud
— une classe permettant de régler thêta et une autre classe le récupérant.Thêta est une variable qui va être utiliser lors de la mise en place de l'algorithme de Barnes-Hut
— une classe récupérant la masse des particules
— une classe calculant l'accélération et la vitesse sur chaque particule du modèle
— une fonction booléenne qui permet de savoir lorsque la mise en place du modèle est terminé

Par exemple, voici le destructeur du modèle à N-corps

```
1 ModelNBody::~ModelNBody()
2 {
3   delete m_pInitial;
4   delete m_pAux;
5 }
```

Quant au ModelNbody.h , elle contient la classe permettant le traitement du modèle à N-corps

2.0.4 BHTree.cpp - BHTree.h

Ce fichier permet la création de fonction et de classe agissant sur un arbre. Un arbre est une structure contenant des informations(ici les particules) situés dans des éléments qu'on appelle noeuds.
On retrouve ainsi dans ce fichier la création et la reintilisation de l'arbre, des fonctions booléenne nous indiquant la racine (= noeud initial) de l'arbre, si un noeud est externe (=noeud sans descendant) ou encore si deux noeuds sont trop proche de l'un de l'autre. Des fonctions permettant d'obtenir les coordonnées du bord supérieur et inférieur d'un noeud, son centre de masse et sa variable thêta y sont définit.
S'y trouve aussi une fonction permettant de récupérer le nombre de particule situé dans les noeud et une autre nous indiquant le nombre des noeuds non assignés.
Une fonction calcule l'accélération causé par la gravitation d'une particule sur une autre et une autre calcule la force exercé par un noeud sur une particule.

Quand au BHTree.h, on y retrouve une classe implémentant l'arbre de Barnes-Hut avec un seul noeud. Par exemple dans cete classe, on retrouve plusieurs données situé dans un noeud comme le montre l'extrait de code suivant :

```
1    double m_mass;                  // Masse de toutes les particules situés dans un noeud
2    Vec2D m_cm;                     // Centre de masse
3    Vec2D m_center;                 // Centre d'un noeud
```

2.0.5 Les intégrateurs

Le code contient également un fichier dans lequel seront écrits les différents intégrateurs permettant de calculer les vitesses et positions de chaque particules. Pour l'instant, seul un simple intégrateur d'Euler est implémenté.

2.0.6 La visualisation

Des classes et méthodes gérant l'affichage des particules est aussi implémenté à partir des bibliothèques SDL et OpenGL, ce qui nous permet de pouvoir nous concentrer sur les calculs qui doivent être effectués.

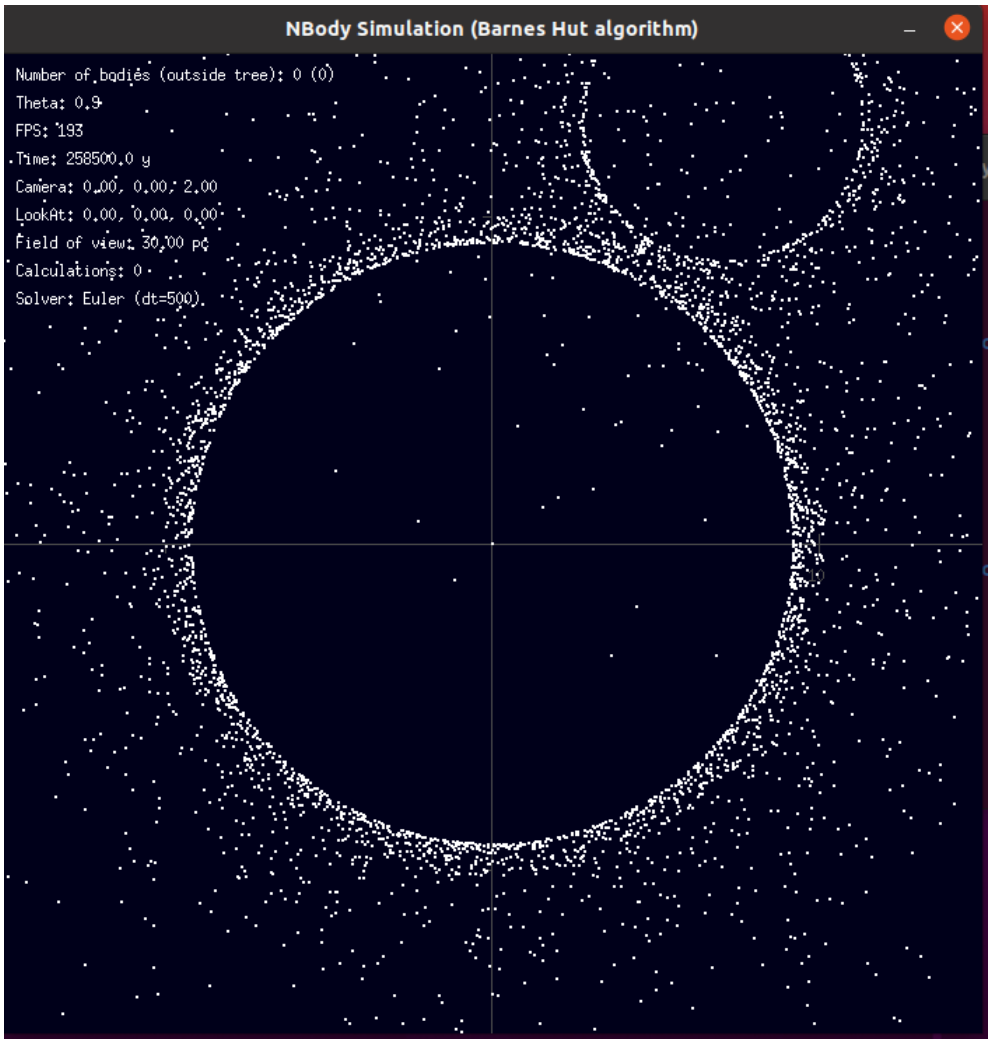


FIGURE 2.1 – Visualisation (aucun calcul effectué)

Chapitre 3

Résolution physique et mathématiques du problème et implémentation "naïve"

La résolution "naïve" du problème consiste à calculer pour toutes les particules la force exercée sur elles par toutes les autres à un instant t. La force prise en compte est l'interaction gravitationnelle d'un corps sur un autre.

Soit p_1 et p_2 deux particules, la force appliquée par p_2 sur p_1 est :

$$\vec{F}_{2\to1} = \frac{-Gm_1m_2}{\|\vec{p}_{2\to1}\|^2}\vec{p}_{2\to1} \tag{3.1}$$

où
— G est la constante de la gravitation $G = 6.672.10^{-11}Nm^2/kg^2$.
— m_1 est la masse de p_1.
— m_2 est la masse de p_2.

On peut donc calculer l'accélération d'une particule avec la 2ème loi de Newton :

$$a(t) = \Sigma\vec{F} = m\frac{dv}{dt} \tag{3.2}$$

où v est la vitesse de la particule, m sa masse et $\vec{F}$ les forces qui s'appliquent.

On obtient donc l'équation différentielle sur la position $OP_1(t)$:

$$\frac{d^2\vec{OP_1}(t)}{dt^2} = \vec{a(t)} \tag{3.3}$$

L'intégrateur permettra alors de résoudre numériquement cette équation différentielle.
On pourra utiliser les méthodes d'Euler ou la méthode Saute mouton.

Conclusion intermédiaire

Projection pour la deuxième moitié du projet

Jusque-là, nous avons étudié le code qui nous a été fourni afin de commencer une implémentation naïve du problème à N-corps. Avant de résoudre naïvement le problème, nous allons recréer les fonctions nécessaires à l'initialisation de la simulation (nombre de trous noirs galactiques, taille des galaxies ou encore autre forme de départ plus singulière).

Nous reimplémenterons par la suite un intégrateur saute-mouton afin de remplacer celui qui était présent dans le code puis nous coderons tout ce qui permettra à la simulation naïve de tourner.

Dans un deuxième temps, nous allons nous servir de l'algorithme de Barnes-Hut afin d'améliorer l'efficacité. Enfin nous allons nous servir de la bibliothèque OpenMP pour faire du multi-thread et ainsi rendre la simulation beaucoup plus efficace.

Annexes

Annexe 1

1 Lien vers le repository github

`https://github.com/Rudiio/Projet-N-corps.git`