Problème à N corps

Simuler un (petit) univers

version 0

Ismail Bennani
<ismail.lahkim.bennani@ens.fr>

17 mars 2020

Mesures exceptionnelles
Étant donnée la situation actuelle, vous pourrez me joindre sur slack pour me poser vos questions en plus de mon mail (envoyez moi votre mail pour que je vous invite).
Je serai aussi disponible via Skype ou Zoom pour faire des points réguliers avec vous.

1 Projet

1.1 Description du problème

Ce problème consiste à résoudre les équations du mouvement de N corps quelconques interagissant dans un champs gravitationnel en dimension $n \in \{2, 3\}$. Soient $(c_i)_{i\in[0,N]}$ nos corps. Pour chaque corps c_i et chaque instant $t \in \mathbb{R}$, on note $m_i \in \mathbb{R}$ sa masse, $\mathbf{p_i}(t) \in \mathbb{R}^n$ sa position, $\mathbf{v_i}(t) \in \mathbb{R}^n$ sa vitesse et $\mathbf{a_i}(t) \in \mathbb{R}^n$ son accélération.
Pour chaque autre corps c_j, on note $r_{i,j}(t) = \|\mathbf{p_j}(t) - \mathbf{p_i}(t)\|$ la distance euclidienne entre ces corps et $\mathbf{u_{i,j}}(t) = \frac{1}{r_{i,j}}(\mathbf{p_j}(t) - \mathbf{p_i}(t))$ le vecteur unitaire allant de $\mathbf{p_i}(t)$ à $\mathbf{p_j}(t)$.

La force gravitationnelle appliquée par un corps $j \in [0, N]$ sur un corps $i \in [0, N], i \neq j$ s'écrit :

$$\mathbf{F_{i,j}}(t) = G\frac{m_i m_j}{r_{i,j}(t)}\mathbf{u_{i,j}}(t)$$

La dynamique du système est alors décrite par les équations suivantes :

$$\forall i \in [0, N], \begin{cases} \dot{\mathbf{p_i}}(t) = \mathbf{v_i}(t) \\ \dot{\mathbf{v_i}}(t) = \mathbf{a_i}(t) \\ \dot{\mathbf{a_i}}(t) = \dfrac{1}{m_i} \displaystyle\sum_{j\in[0,N], j\neq i} \mathbf{F_{i,j}}(t) \end{cases}$$

1.2 Travail attendu et évaluation

Vous écrirez un simulateur capable de calculer l'évolution des positions (en 2D **OU** 3D) des N corps dans le temps, ce simulateur devra utiliser un solveur d'équations différentielles que vous aurez écrit. Vous écrirez aussi un affichage graphique pour visualiser cette évolution.
Le projet est volontairement peu cadré pour que vous puissiez approfondir **les parties qui vous intéressent le plus**. Vous pourrez par exemple :

- implémenter des optimisations pour accélérer le calcul du système d'équations différentielles (cf. section 2.1)
- implémenter un solveur d'équations différentielles plus précis que la méthode d'Euler (cf. section 2.2)
- implémenter des fonctionnalités pour votre interface graphique (cf. section 2.3)

Je vous demande à tous d'écrire au moins une version naïve du moteur physique, du solveur et de l'interface graphique qui n'utilise aucune librairie externe. Vous pourrez ensuite remplacer des parties de votre code par des interfaces vers un code existant, par exemple utiliser `numpy` pour vos opérations sur les vecteurs et vos opérations matricielles.
Je vous conseille donc de penser votre architecture logicielle de manière modulaire, pour qu'il soit facile de changer d'échanger les modules. Par exemple, il faudrait que vous puissiez passer d'une représentation interne des vecteurs à une représentation `numpy` en changeant simplement votre instruction `import .. from ...`

Le rendu attendu est une archive contenant

- l'intégralité du code source
- un fichier requirement.txt
- un rapport concis expliquant votre travail, en particulier :
  - la répartition des taches dans votre binôme
  - les librairies externes utilisées
  - le fonctionnement de votre programme
  - ce que vous avez approfondi
  - les difficultés rencontrées et vos solutions, les choix techniques effectués

Je vous suggère d'utiliser Python comme langage de programmation, mais vous pouvez me consulter si vous désirez utiliser autre chose.
En section 2.5, je vous parle de quelques librairies utiles pour Python, vous êtes libres d'utiliser les librairies que vous voulez. Par contre, veillez à bien noter dans votre rapport tout ce que vous avez implémenté vous-même et tout ce qui vient d'une librairie externe.

Attention, la valeur d'un code ne se mesure pas au nombre de lignes ! Faites du code commenté (pas besoin de trop en faire) et clair. Veillez par exemple à choisir des noms de variable explicites et informatifs.

2 Implémentation

Dans cette section, je vous décrit une façon d'implémenter le simulateur en trois parties bien délimitées et indépendantes les unes des autres. Vous **n'êtes pas** tenus de respecter cette implémentation.
Je vous conseille vivement de commencer par implémenter les version les plus naïves des différents modules. Cela vous permettra en particulier de réfléchir aux interfaces entre ces modules, de commencer à écrire des tests unitaires, de mettre en place votre dépôt git, et surtout d'avoir une meilleure vision d'ensemble du projet. Contactez moi à ce moment là pour qu'on discute de ce que vous comptez faire par la suite.

2.1 Moteur Physique

Le rôle du moteur physique est de calculer une approximation de la dynamique du problème. Il s'occupe d'une part de fournir au solveur d'équations différentielles les valeurs à intégrer, et d'autre part de fournir à l'interface graphique les données nécessaires à l'affichage des corps (positions, vitesses, ...).
Ce moteur peut gérer les collisions entre corps, considérer que deux corps qui se touchent fusionnent, où ignorer les collisions. Les méthodes que je donne plus bas ignorent les collisions.

Notez que l'efficacité de ce module sera cruciale si vous voulez pouvoir simuler un grand nombre de corps. En effet, la dynamique du système doit être calculée au moins une fois à chaque pas de calcul du solveur d'équations différentielles.

— **Méthode naïve** $\mathcal{O}(N^2)$
Une approximation simple dans notre cas est celle où les réelles sont représentés par des flottants machines, et où l'accélération d'un corps est obtenue en calculant les $N-1$ forces qu'il subit.
Cependant, on peut être plus efficace en remarquant que

$$\mathbf{F_{i,j}} = -\mathbf{F_{j,i}}$$

Grâce à ça, on peut obtenir le même résultat que précédemment en économisant la moitié des calculs.

— **Simulation de Barnes-Hut** $\mathcal{O}(Nlog(N))$
On peut réduire le nombre de calculs à effectuer en sacrifiant en partie la précision de leurs résultats. Cette méthode consiste à construire un arbre (octree) dont les feuilles sont vos corps, et dont les nœuds intermédiaires représentent des portions de l'espace contenant leurs fils. Dans chaque zone on peut définir un corps virtuel en son centre dont la masse est égale à la somme des masses des corps s'y trouvant.
Ensuite, pour calculer l'interaction gravitationnelle que subit un corps en particulier, on pourra remplacer la contributions d'un groupe de corps éloigné par celle de son corps virtuel.
https://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation
http://arborjs.org/docs/barnes-hut

— **Fast multiple methods $\mathcal{O}(N)$**
https://en.wikipedia.org/wiki/Fast_multipole_method
A Hierarchical $\mathcal{O}(N)$ Force Calculation Algorithm

Note
Lors de l'implémentation de ce module, il sera judicieux d'écrire une classe de vecteurs pour représenter les positions, vitesses, accélérations et forces.
Vous pourrez aussi surcharger les opérateurs arithmétiques ($+$, $*$, ...) pour manipuler les instances de cette classe.
Surcharge opérateurs : explication, liste des opérateurs

2.2 Solveur d'équations différentielles

Le rôle du solveur d'équations différentielles (**O**rdinary **D**ifferential **E**quation) est de calculer la dynamique du système. Il reçoit du moteur physique les positions initiales des variables et leurs dérivées puis calcule la nouvelle valeur des variables à un horizon donné.
Résoudre une ODE, c'est calculer les valeurs d'une fonction $\mathbf{y} : \mathbb{R} \to \mathbb{R}^k$ définie par

$$\forall t \geq 0, \dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)) \qquad \mathbf{y}(0) \text{ connue}$$

à des temps donnés. On note $(t_n) \in \mathbb{R}^{\mathbb{N}}$ la suite des temps auxquels on veut calculer la valeur de $\mathbf{y}$, et $\mathbf{y_n} = y(t_n)$.

Notez qu'ici, notre système est bien une ODE :

$$\forall i \in [0, N], \begin{cases} \dot{\mathbf{p_i}}(t) = \mathbf{v_i}(t) \\ \dot{\mathbf{v_i}}(t) = \mathbf{a_i}(t) \\ \dot{\mathbf{a_i}}(t) = \dfrac{1}{m_i} \displaystyle\sum_{j \in [0,N], j \neq i} \mathbf{F_{i,j}}(t) \end{cases}$$

se réécrit en

$$\forall i \in [0, N], \begin{bmatrix} \dot{\mathbf{p_i}}(t) \\ \mathbf{v_i}(t) \\ \mathbf{a_i}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{v_i}(t) \\ \mathbf{a_i}(t) \\ \dfrac{1}{m_i} \displaystyle\sum_{j \in [0,N], j \neq i} \mathbf{F_{i,j}}(t) \end{bmatrix}$$

— **Méthode d'Euler explicite**
Approximation du premier ordre :

$$\forall n \geq 0, \mathbf{y_{n+1}} = \mathbf{y_n} + (t_{n+1} - t_n) * \mathbf{f}(t_n, \mathbf{y_n}) \qquad \mathbf{y_0} = \mathbf{y}(0)$$

La fonction qui implémente cette méthode pourrait s'écrire
`def integrate(y, h, y'): return y + h * y'`
et être appelé par `new_y = integrate(y, h, f(t,y))`.

Cependant, les méthodes qui suivent auront besoin d'évaluer f en plusieurs points entre t et $t + h$, pour produire des résultats plus précis. Pour cette raison, la façon la plus classique d'implémenter une fonction d'intégration est de lui passer en argument la fonction **f** :
`def integrate(f, y0, t, h): return y0 + h * f(t, y0)`

— **Leapfrog algorithm**
— **Runge Kutta family**

2.3 Interface graphique

2.4 Orbites stables

Si vous voulez faire de beaux dessins, vous pouvez simuler des orbites stables (appelées *chorégraphies*) ; c'est à dire des positions initiales des corps telles que, au bout d'un moment, leur mouvement devienne périodique. Trouver ces positions initiales est loin d'être facile, il y a de nombreux travaux de recherche dessus. Je vous met ici une petite liste de pointeur qui peuvent vous être utiles si vous voulez approfondir cette partie là.

— Une explication accessible `http://www.scholarpedia.org/article/N-body_choreographies`
— Papier : Classification of symmetry groups for planar n-body choreographies, une famille de solutions pour N quelconque, illustrée ici
— Thèse: Periodic Solutions to then n-Body Problem, quelques solutions au chapitre 8

2.5 Outils