Problème à N corps

Simuler un (petit) univers

version 2

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

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 $(c_i)_{i\in[1,N]}$ nos corps. A chaque instant $t \in \mathbb{R}$ et pour chaque corps c_i, 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 $\mathbf{u_{i,j}}(t) = \frac{1}{r_{i,j}(t)}(\mathbf{p_j}(t) - \mathbf{p_i}(t))$ le vecteur unitaire allant de $\mathbf{p_i}(t)$ à $\mathbf{p_j}(t)$.

La force gravitationnelle appliquée par un corps $j \in [1, N]$ sur un corps $i \in [1, N], i \neq j$ s'écrit :

$$\mathbf{F_{i,j}}(t) = G\frac{m_i m_j}{r_{i,j}^2(t)}\mathbf{u_{i,j}}(t)$$

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

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

1.2 Travail attendu et évaluation

Vous écrirez un simulateur capable de calculer l'évolution des positions (en 2D) 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.
Ce sujet est volontairement long pour que vous puissiez approfondir **les parties**

qui vous intéressent le plus. Je n'attends pas de vous que vous fassiez tout, faites un maximum dans le temps que vous avez. Vous pourrez par exemple :

- implémenter des optimisations pour accélérer le calcul du système d'équations différentielles (cf. section 2.1)
- implémenter plusieurs solveurs d'équations différentielles (cf. section 2.2)
- implémenter des fonctionnalités supplémentaires pour votre interface graphique (cf. section 2.3)
- calculer automatiquement des orbites stables (cf. section 2.4)
- gérer les collisions (cf. section 2.5)

Je demande que chaque groupe écrive au moins une version naïve, sans librairie externe, du moteur physique, du solveur et de l'interface graphique. Pour cela je vous fournis un squelette de projet avec une architecture de classes (et surtout d'interfaces) pensée pour faciliter votre implémentation.
Vous pourrez ensuite remplacer des parties de votre code par des interfaces vers un code existant, par exemple vous pourrez utiliser `numpy` pour vos opérations sur les vecteurs et vos opérations matricielles.

Le rendu attendu est un PDF contenant

- un lien vers un git contenant l'intégralité du code source (pensez à inclure un fichier requirement.txt)
- un rapport concis expliquant votre travail, en particulier :
  - la répartition des taches dans votre groupe
  - les librairies externes utilisées
  - le fonctionnement de votre programme, l'architecture de vos classes
  - ce que vous avez approfondi, les algorithmes implémentés
  - les difficultés rencontrées et vos solutions, les choix techniques effectués

Comme vous l'a dit Natalia, ayez une version préliminaire du rapport prête à la moitié du cours pour que je puisse vous faire un premier retour.

Je vous suggère d'utiliser Python comme langage de programmation, mais vous pouvez me consulter si vous désirez utiliser autre chose.
En section 2.6, 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 correct, commenté (sans exagération) et clair, dans cet ordre. Veillez par exemple à choisir des noms explicites et informatifs pour vos variables, vos fonctions et vos classes.

2 Implémentation

Dans les sections 2.1, 2.2 et 2.3, je vous décrit **une** façon d'implémenter le simulateur à partir du squelette fourni. Vous **n'êtes pas** tenus de respecter cette implémentation.
Les sections 2.4 et 2.5 sont là pour vous donner des idées d'améliorations possibles pour votre simulateur, et des pointeurs vers des ressources qui vous aideront à implémenter ces idées.
Dans ce projet nous n'utiliserons pas les unités du système international. Les unités du SI font que les valeurs que l'on manipule ont des ordres de grandeurs très différents (vitesse terre par rapport au soleil $\approx 10^4$ m.s^{-1}, distance terre-soleil $\approx 10^{11}$ m, poids soleil $\approx 10^{30}$ kg, ...). De ce fait, les erreurs dues à l'approximation flottante et aux approximations des solveurs sont d'autant plus importantes. Si vous voulez utiliser des valeurs du monde réel dans vos simulations, je vous conseille d'utiliser les unités suivantes :

- distance : parsec (pc)
- masse : masse solaire ($M_\odot$)
- vitesse : kilomètre par seconde (km.s^{-1})
- constante gravitationnelle : pc.${M_\odot}^{-1}$.km^2.s^{-2}

Test unitaires : Le squelette que je vous fourni contient dejà quelques tests unitaires que votre code devra passer. Ajoutez vos propres tests à ceux là au fur et à mesure de votre développement.

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, ...).
En ce qui concerne les collisions entre corps, ce moteur peut soit les gérer de manière plus ou moins réaliste (faire rebondire les corps, faire fusionner les corps, ...) soit les ignorer (vous assimilerez les corps à leur centre de masse). Les méthodes que je donne plus bas ignorent les collisions. Reportez vous à la section 2.5 si vous voulez gérer les collisions.
Note : Même si vous décidez d'assimiler les corps à un point, il y a une (très) faible probabilité que deux points se retrouvent exactement à la même position, pensez à gérer ce cas là d'une manière ou d'une autre pour éviter les bugs.

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 $\mathcal{O}(N^2)$**
  La méthode la plus naïve est celle 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 (*cellule*) contenant leurs fils. Dans chaque cellule on définit un corps virtuel qui se trouve en son centre et dont la masse est égale à la somme des masses des corps se trouvant dans sa cellule.
Ensuite, pour calculer l'interaction gravitationnelle que subit un corps en particulier, on pourra remplacer la contributions d'un groupe éloigné de corps par celles du corps virtuel correspondant.
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 (cf. vector.py).
Vous pourrez aussi surcharger les opérateurs arithmétiques ($+$, $*$, ...) pour manipuler les instances de cette classe plus facilement.
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 que le système définit en 1.1 est une ODE d'ordre 1 :

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

se réécrit en

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

$\mathbf{y}(t)$ est donc un vecteur de dimension $2 * i$ contenant les positions et vitesses des différents corps.

— **Méthode naïve : Euler explicite**
Approximation du premier ordre, peu couteuse à calculer mais peu stable, très imprécise :

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

La fonction qui implémente cette méthode pourrait s'écrire

```
def integrate(y, h, y'): return y + h * y'
```

et être utilisée comme ça :

```
new_y = integrate(y, h, f(t,y))
```

(où `h` est l'horizon $(t_{n+1} - t_n)$).
Cependant, les autres méthodes d'intégration auront besoin d'évaluer $\mathbf{f}$ en plusieurs points entre t_n et t_{n+1}, 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 $\mathbf{f}$ qui calcule les dérivées de la solution :

```
def integrate(f, y0, t, h): return y0 + h * f(t, y0)
```

https://www.f-legrand.fr/scidoc/docimg/numerique/euler/euler/euler.html

— **Algorithme Leapfrog**
Approximation explicite du deuxième ordre, peu couteuse mais stable pour des mouvement oscillatoires. Elle nécessite d'avoir accès à la vitesse et à l'accélération des variables à intégrer. Ca tombe bien, c'est notre cas. Soit $i \in [1, N]$, en notant $p_n^i = p_i(t_n)$, $v_n^i = v_i(t_n)$, $a_n^i = a_i(t_n)$ et $\Delta t_n = t_{n+1} - t_n$, on a l'approximation suivante :

$$\forall i \in [1, N], \forall k \in \mathbb{N}, \begin{cases} p_{k+1}^i = p_k^i + \Delta t_n v_k^i + \dfrac{1}{2} {\Delta t_n}^2 a_k^i \\ v_{k+1}^i = v_k^i + \dfrac{1}{2} \Delta t_n (a_k^i + a_{k+1}^i) \end{cases}$$

https://en.wikipedia.org/wiki/Leapfrog_integration

— **Famille des méthodes Runge Kutta**
Approximation explicite, même algorithme pour plusieurs ordres, ces méthodes font parti des plus utilisées. Je vous met là une petite sélection de liens utiles si vous voulez en savoir plus et implémenter ces algorithmes
  — http://www.scholarpedia.org/article/Runge-Kutta_methods
  — A family of embedded Runge-Kutta formulae, Dormand and Prince
  — Coefficients for the study of Runge-Kutta integration processes, Butcher

— **Méthode d'Euler implicite**
Approximation implicite du premier ordre. Ici implicite veut dire que l'approximation au pas $n+1$ n'est pas défini comme une fonction explicite des valeurs au pas n, mais comme la solution d'une équation point fixe. Les méthodes implicites sont souvent très couteuses à calculer car elles demandent plus d'évaluations de la fonction $\mathbf{f}$, mais elles sont plus stables et généralement plus précisent :

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

On cherche y_{n+1} le point fixe de la fonction

$$\mathbf{G}(\mathbf{x}) = \mathbf{y_n} + (t_{n+1} - t_n) * \mathbf{f}(t_{n+1}, \mathbf{x})$$

Plusieurs méthodes existent pour ça, par exemple la méthode de Newton-Raphson.
https://www.f-legrand.fr/scidoc/docimg/numerique/euler/implicite/implicite.html

2.3 Interface graphique

Le rôle de l'interface est d'abord d'afficher l'état du système à l'écran. Elle peut dans un second temps servir à interagir avec le système : par exemple ajouter un corps avec un clic gauche, ou en supprimer un avec un clic droit.
Comme je l'ai dis plus haut, vous implémenterez un affichage en 2D.
Je vous suggère d'utiliser `pygame` comme librairie de dessin mais vous pouvez bien sur utiliser n'importe quelle autre librairie si vous préférez.

— **Pygame** https://www.pygame.org
Je vous laisse suivre l'excellent tutoriel sur leur site officiel

Je vous conseille aussi de séparer les unités des valeurs que manipulent votre système physique de celles des valeurs utiles à l'affichage. Écrivez des fonctions de changement de base entre votre monde et votre caméra, c'est à dire des fonctions qui convertissent les coordonnées d'un point du monde en un point à l'écran, et vice versa. Ca vous permettra de contrôler votre échelle plus facilement et vous aidera aussi à implémenter une façon de contrôler la caméra durant la simulation.

2.4 Orbites stables

Si vous voulez faire de beaux dessins, vous pouvez simuler des orbites stables (appelées *chorégraphies*) ; c'est à dire choisir des positions initiales des corps telles que, au bout d'un moment, leur mouvement devienne périodique. Trouver ces positions initiales n'est pas facile, il y a de nombreux travaux de recherche là 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 Collisions

Si vous souhaitez gérer les collisions, il faut que vos corps c_i aient une taille. Vous pouvez par exemple les assimiler à des sphères de rayon r_i.
Le problème de gestion de collisions se décompose en deux étapes : détecter les différentes collisions, puis les résoudre, c'est à dire calculer les nouvelles positions et vitesses des astres impliqués.

Détection Tout d'abord, on peut approximer l'instant de la collision en considérant qu'elle ne peut se produire qu'a des instants auxquels on échantillonne notre dynamique, c'est à dire à un des t_n. Dans ce cas, à chaque pas, il faut chercher si un couple de corps se chevauche (le cas de collisions de plus de 2 corps est plus complexe à traiter et peut être ignoré en première approximation).
Une autre solution est de surveiller les instants t_i et t_{i+1} tels que deux corps se chevauchent à t_{i+1} mais pas à t_i, puis de chercher l'instant précis dans $[t_i, t_{i+1}]$ auquel a eu lieu la collision. Pour ça il faut :

- interpoler les positions entre ces instants : le plus simple est de faire une interpolation linéaire, mais l'on peut aussi se servir des calculs du solveur d'ODE pour obtenir une meilleure interpolation.
- approximer l'instant de la collusion dans l'interpolation : dichotomie, méthode de Newton-Raphson, ...

Résolution Une fois que cette collision à été détectée, il faut la traiter : c'est à dire faire que les objets ne se chevauchent plus, et mettre à jour leurs vitesses respectives pour tenir compte de la collision ; les faire rebondir. On pourrait aussi, dans le cas d'étoiles par exemple, les faire fusionner.

Notez que les algorithmes de détection de collisions peuvent être très couteux : le plus naïf est celui qui teste à chaque pas de simulation les positions des

$\frac{N(N-1)}{2}$ couples de corps possibles. Des méthodes existent pour réduire cette complexité, elles reviennent toutes plus ou moins à distinguer, pour chaque corps, ceux qui sont trop loin, et ceux qui sont suffisamment proches pour être testés exactement. La méthode de simulation de Barnes-Hut introduit un tel partitionnement de l'espace (cf. section 2.1).
Sinon voila un papier traitant spécifiquement de ce problème : Fast Collision Detection among Multiple Moving Spheres.

2.6 Outils

Voici quelques outils qui peuvent vous aider. Je rappelle que vous **devez** implémenter au moins les versions naïves de chaque module du projet, ce n'est que par la suite que vous pouvez remplacer un de vos modules naïf par une interface vers une de ces librairies. J'attends de vous que vous approfondissiez certaines parties du projet, mais vous n'aurez pas le temps de tout faire vous-même, alors n'hésitez pas à utiliser des librairies déjà existantes.

— numpy : librairie de calcul scientifique. Vous ne pourrez pas faire plus efficace que ça pour vos opérations vectorielles et matricielles sur CPU.

— SciPy : librairie d'algorithmes numériques. Vous y trouverez des algorithmes d'intégration déjà fait, des algorithmes d'interpolation, d'optimisation.