Victor Defrance
Chloé Balmes

ENSTA Paris
IN104 - 2021

Rapport IN104 : N-Body Problem

Table des matières

Lien du GitHub commun :

Inoxagen/IN104_Chloe_BALMES_et_Victor_DEFRANCE: N-Body problem: projet IN104 (github.com)

Description du problème :

On va chercher à modéliser les interactions et à résoudre les équations du mouvement de N corps quelconques dans un champ gravitationnel de dimension 2. Commençons par poser les notations :

- $b_i \; pour \; i \in [\![1, N]\!]$ correspondant au corps numéro i avec m sa masse,
- $p_i(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}, v_i(t) = \begin{pmatrix} v_x(t) \\ v_y(t) \end{pmatrix} \; et \; a_i(t) = \begin{pmatrix} a_x(t) \\ a_y(t) \end{pmatrix}$ la position, la vitesse et l'accélération du corps i à l'instant $t \in R$.
- Pour deux corps i et j, on note $r_{i,j}(t) = \|p_j(t) - p_i(t)\|$ la distance euclidienne, et $u_{i,j} = \frac{p_j(t) - p_i(t)}{r_{i,j}}$ le vecteur unitaire allant de $p_i(t)$ à $p_j(t)$.
- On note la force gravitationnelle qu'applique un corps j à un corps i de la manière suivante :

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

Ce qui nous donne le Principe Fondamental de la Dynamique suivant :

$$m_i * a_i(t) = \sum_{j \in [\![0,N]\!], j \neq i} F_{i,j}(t)$$

Distribution du travail :

Nous avons décidé de travailler et de compléter les premiers programmes ensemble pour avoir les bases du projet.

Puis une fois que le squelette principal était fait, Victor s'est chargé du problème des collisions et Chloé des tests ainsi que de la compréhension de pygame, pour commencer à implémenter le déplacement avec les flèches. Nous avons ensuite tous les deux travaillé sur l'interface graphique (*screen.py)*. Chloé s'est intéressée au fait de suivre un astre après avoir cliqué dessus, ainsi qu'à la gestion des couleurs. Victor s'est plutôt penché sur l'affichage des orbites et l'utilisation des touches dans l'interface graphique.

Notre implémentation : nos classes

Nous avons travaillé sur plusieurs grandes parties :

- *vector.py* contenant les classes *Vector* et *Vector2*
- *world.py* avec les classes *Body* et *World*
- *engine.py,* le moteur de notre problème physique
- *solver.py* pour la résolution d'équations différentielles
- *simulator.py* pour simuler le problème
- *camera.py* gérant les changements d'échelle entre le solver et l'écran
- *screen.py* pour gérer l'affichage à l'écran

Nous ne redéfinissons pas entièrement les classes à chaque fois : la plupart des classes écrites héritent d'une classe de base dans lesquelles nous redéfinissons les fonctions à changer. Cela nous permet notamment de réduire les doublons de code.

Vector

Le programme *Vector.py* contient deux grandes classes *Vector (classe parente)* et *Vector2 (classe héritière)* dont le but est d'introduire toutes les fonctions nécessaires pour pouvoir faire les calculs simplement avec des vecteurs et non des listes, comme nous le ferions naturellement. En effet, lorsque nous utilisons la sommation sur deux vecteurs, nous nous attendons à une somme sur les coordonnées et non à une concaténation comme ce serait le cas avec des listes.

Par rapport à ce qui était déjà implémenté, nous avons écrit les fonctions réciproques des fonctions multiplication, somme et soustraction pour que ces actions soient réalisables que le premier ou le deuxième terme mis en argument soit un vecteur.

Nous avons aussi rajouté la fonction modulo qui permet de faire le modulo de chaque coordonnée avec un même nombre ou de faire pour chaque paire de coordonnées le modulo d'un vecteur par rapport à un autre.

La fonction *copy*, nous permet quant à elle, de réaliser la copie d'un vecteur mis en argument. Elle a été essentielle pour le reste des programmes pour palier au problème d'étiquettes.

Nous avons de plus, rajouté un argument à la classe *Vector* : un paramètre seuil étant nul si l'on ne le redéfinit pas. Trois fonctions de seuil ont aussi été écrites *get_seuil, estNul, estPresqueNul* pour accéder à la valeur de ce nouveau paramètre, et voir si la norme du vecteur étudié est plus petite ou non que ce seuil : tout cela a été introduit pour l'étude des collisions mais n'a finalement pas été utilisé.

Dans la classe *Vector2*, nous avons seulement implémenter une fonction permettant l'affichage du vecteur étudié.

World

Le fichier contient 2 classes : *Body* et *World*.

La première, *Body*, implémente la description des corps de notre simulation. Ils possèdent :
- une position (*position*) et une vitesse (*velocity*). On stockera dans chacune, un vecteur de classe *Vector2* contentant les 2 cordonnées (respectivement en position et en vitesse);
- une masse (*mass*) qui contiendra la masse du corps sous forme d'un réel (par défaut la masse est 1);
- une couleur (*color*) qui contiendra la couleur au format (R,G,B) : 3 entiers de [0,255] (blanc par défaut);
- un rayon d'affichage à l'écran (*draw_radius*) qui sera donné en nombre de pixel ;
- un nom (*nom*) qui sera stocké sous forme d'une chaine de caractère (le nom par défaut est vide).

La classe *World*, contient un monde entier prêt à être simulé. Elle comprend les variables :
- *self._bodies* = [] : Contiendra la liste des corps.
- *self.total_corps*=0 : Comptabilise le nombre de corps vivants et ayant vécus.
- *self.nom=nom* : Le nom de ce monde (string vide par défaut).
- *self.modeParlant=modeParlant* : Booléen déterminant si on écrit des informations dans la console.
- *self.seuil_collision=seuil_collision* : Distance à laquelle 2 corps entrent en collision dans ce monde.
- *self.bg_color=bg_color* : La couleur du fond affiché pour ce monde.
- *self.time_scale=time_scale* : L'échelle de temps de la simulation idéale pour ce monde.
- *self.camera_scale_initial=camera_scale_initial* : le zoom de caméra initial idéale pour ce monde.

Elle contient aussi des nombreuses fonctions. Nous détaillerons ici seulement celle que nous avons ajouté au squelette fourni.

```
def add_set(self, liste_de_corps):
```
Pour ajouter au monde (*self*) une liste de corps de classe *Body*. On retourne une liste contenant les identifiants respectifs attribués à chaque corps.

```
def add_N_corps_aleat_diff(self,N, borne_pos=[[-1,1],[-1,1]],
borne_vit=[[-2,2],[-2,2]], mass_max=2):
```
Pour ajouter au monde N corps aléatoires de classe *Body*. Les différents paramètres des corps ajoutés varient dans les bornes données en paramètre. On retourne une liste contenant les identifiants respectifs attribués à chaque corps. Ici, on récupère des noms pour nos astres dans la liste des 88 constellations. Pour cela, on prend au hasard un nom dans une liste de noms que nous avons au préalablement écrite dans un autre fichier que nous avons dû importer dans *World*: *corps_*celestes. Ainsi en mode parlant, les collisions donnant dans le terminal des affichages de type :

```
COLLISION entre Indien 6 et Hydre mâle 1
```

```
def clear_all(self,seuil_collision=-1, bg_color=(-1,-1,-1)):
```
Pour supprimer tous les corps du monde. Cette fonction offre la possibilité de redéfinir le seuil et la couleur mais par défaut on garde les anciennes caractéristiques.

```
def pop(self, index):
```
Fonctionne similairement au pop de la classe *List*. Il supprime l'élément d'index *index* du monde et le retourne.

Engine

La classe *Engine* correspond au moteur physique de la simulation.

Elle ressence des fonctions et des classes.

Les fonctions de engine.py

La fonction *gravitational_force* correspond seulement à l'implémentation de la formule écrite en introduction de la force gravitationnelle. Cette fonction est utilisée dans la classe DummyEngine.

On se rend assez vite compte que la première masse donnée en argument de *gravitational_force* ne sert pas car on divisera par celle-ci pour l'appliquer à l'accélération. Pour l'implémentation de l'*engine* plus performante, on calculera une seule fois l'interaction entre 2 corps donnés. On définit donc une nouvelle fonction *acc_grav_massiq_depuis_diff* qui prend en paramètre le vecteur de différence entre les 2 position. Elle renvoie une accélération massique. Il n'y aura plus qu'à multiplier par la masse de l'autre corps pour avoir l'accélération à appliquer. Cette fonction est utilisée dans la classe *SimpleSansCollisonEngine*.

Les classes de engine.py :

Initialement nous avions programmé tout le calcul de la simulation dans *solver.py*. Nous avions commencé par résoudre tous les calculs dans une seule fonction. Nous faisions une boucle sur tous les corps i, puis sur tous les autres éléments exerçant une force sur i. On appliquait le Principe Fondamental de la Dynamique avec chacun de ses corps en sommant toutes les forces. Puis on modifiait les vitesses et les positions. Nous nous sommes rendu compte, grâce aux tests non passés, qu'il fallait en fait diviser les calculs entre *Solver* et *Engine*. Nous avons donc repris ce que nous avions précédemment écrit dans le fichier *Solver* puis l'avons retranscrit dans *gravitational_force*, *derivatives* et *make_solver_state*.

derivatives sera redéfini dans les différentes classes que nous avons écrites dans *engine.py*. Nous avons les classes :

IEngine: C'est le squelette des classes Engine suivantes. Elle contient

- def __init__(self, world): L'initialisation qui ne sera pas retouché.
- def derivatives(self, t0, y0):
- def make_solver_state(self):

DummyEngine(IEngine): Première implémentation de l'Engine. Elle contient 2 fonctions :

- def make_solver_state(self): Crée le vecteur y0 de classe *Vector*. Cette fonction ne sera pas redéfinie.
- def derivatives(self, t0, y0): Cette première version ne gère pas les collisions. Elle est de complexité N^2. Et calcule 2 fois les interactions entre 2 corps. On a mis quelques 'optimisations' : le stockage de la position pour réduire les appels ou encore le passage de la masse à 1 pour éviter une multiplication division par un même nombre.

SimpleSansCollisonEngine(DummyEngine): Hérite de *DummyEngine*.

- On redéfinit *derivative*. On réduit par 2 le nombre de calcul des interactions grâce à la fonction *acc_grav_massiq_depuis_diff*. La complexité est de N^2 et elle ne gère pas les collisions.

SimpleAvecCollisonEngine(DummyEngine): Hérite de *DummyEngine*.

- On redéfinit *derivative*. Par la même astuce on réduit par 2 le nombre de calcul des interactions. La complexité est de N². Le grand ajout est la gestion des collisions. La distance de collision est donnée par la variable *seuil_collision* de *world*. Si on détecte 2 corps a une distance inférieur une collision a lieu ! Effectuer le test de collision dans l'*engine*(.py) et non dans le *main.py* permet de réduire la distance de collision. En effet, on testera beaucoup plus souvent si 2 astres doivent fusionner.
  En cas de collision :
    - On arrête le calcul
    - On fusionne les deux corps collisionnés en un. Les paramètres du nouveau corps sont
        - Position : une pondération des 2 positions par leurs masses
        - Vitesse : une pondération des 2 vitesses par leurs masses
        - Masse : la somme des masses
        - Couleur : le maximum des composantes de chaque couleur RGB
        - Rayon : la racine de la somme des carrés des 2 rayons
        - Nom : on garde le nom du plus lourd
    - On réduit l'autre corps
        - Masse, rayon passés à 0
        - Couleur à la couleur du fond
    - On met à jour la position et la vitesse du nouveau corps créé dans *y0*.
    - On relance le calcul avec le nouveau *y0.*

Un des problèmes est maintenant de vérifier la masse des objets avant chaque calcul. Les objets de masse nulle ne sont pas traités.

Solver

Cette partie correspond à l'intégration de l'équation de la forme $y' = f(y, t)$.

Au début du projet, nous n'avions pris en compte que le pas de temps égal à 1. Nous avons donc changé notre fonction *Integrate* avec une recherche du pas optimal de la bonne longueur. On a choisi de calculer la longueur du pas pour que toutes les intégrations se fassent sur le même temps. Nous utilisons la fonction f pour obtenir y'. A l'aide de la formule d'Euler (à l'ordre 1) nous intégrons ainsi la vitesse et la position. Dans cette fonction *Integrate* nous utilisons la classe *Vector* pour alléger les écritures.

Simulator

Ce fichier contient la classe *Simulator* pour représenter l'état du monde. Voici les deux fonctions qu'elle contient, en plus de l'initialisation :

```
def step(self, h):
```

Fait le lien entre le problème mathématique de *Solver* et le problème physique qui est dans *World.*

```
def re_init(self, world):
```

Nous l'avons écrite car lorsque l'on supprime un corps du monde, il nous faut réintroduire ce nouveau monde dans la classe Engine et créer un y0 de taille réduite de 1.

Camera

Le but de cette classe est de préparer l'affichage à l'écran.

Ses fonctions gèrent le changement de repère depuis les coordonnées utilisées pour les calculs à celles que prend en compte l'affichage à l'écran. Nous avons eu du mal à définir les deux fonctions *to_screen_coords* et *from_screen_coords* en fonction de la position de la caméra. Au départ, nous n'avions mis toujours le même point au milieu de l'interface quel que soit l'emplacement initial des planètes ou la « position de la caméra ». Nous avons ensuite changé nos formules dans ces deux fonctions pour prendre en compte la position de la caméra. Ainsi les tests *CameraTestCase* ont été vérifiés. C'est à ce moment-là que nous avons décidé d'ajouter d'autres tests pour vérifier aussi notre fonction *from_screen_coords* qui, elle, n'était pas testée.

Screen

Screen est la classe qui correspond à toute l'interface graphique : tant les affichages à l'écran que la prise des commandes par l'utilisateur.

Dans l'initialisation de la classe, on récupère la taille de la fenêtre pour la simulation et la couleur du fond. On crée un *Vector2* pour la position de la souris.Les boutons et touchesOn a une liste *buttons* dans laquelle chaque élément sera égale à *True* lorsque la touche ou la fonction associée sera utilisée. Par exemple, lorsque l'utilisateur utilisera le clic gauche de la souris, le premier élément de *buttons* sera égale à True. On a donc :

self._buttons = [False,False,False,False,False,False,False,False,False,False,False,True,False,True,False,-1,False]

correspondant à

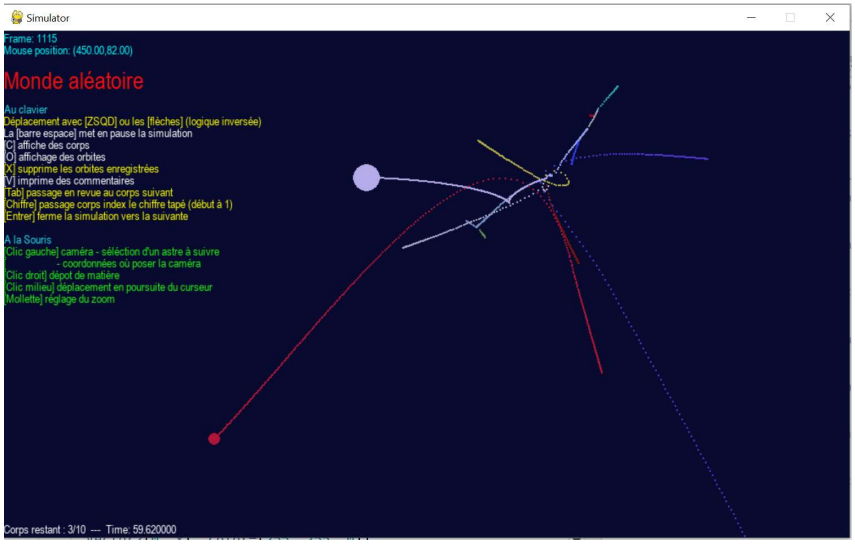
[LeftMouse, MiddleMouse, RightMouse, ScrollWheelUp, ScrollWheelDown, Left_Key, Right_Key, Up_Key, Down_Key, BarreEspace, Touche_O, Touche_C, Touche_X, Touche_V, Touche_TAB, Chiffre, Touche_Entrer]

Celle liste est actualisé dans *get_events*.

On peut distinguer les touches en 2 groupes : celles nécessitant une mémoire et celle n'en ayant pas besoin. En effet, pour certains événements, on demande à ce que la touche reste enfoncée pour agir alors que parfois il suffit d'un clic pour que l'événement fasse effet.

- Les touches sans mémoire sont : le Clic gauche, la Molette, [X], [Tab], [chiffres] et [Entrer]. Elles sont remises à leur valeur initiale à chaque tour (False pour tous sauf Chiffre qui prend -1).
    - A l'impact, les touches du clavier, [X], [Tab], [chiffres] et [Entrer] prennent la valeur True (sauf Chiffre qui prend la valeur du chiffre)
    - Les touches de la souris, le Clic gauche et la Molette passent en True à la pression et redescendent à False lors du relâchement

- Celles nécessitant un mémoire sont : le Clic droit, les [flèches], [ZSQD], [Espace], [O], [C] et [V].
    - Les touches des [flèches], de [Z], de [S], de [Q] et de [D] servent au déplacement. Ils passent à True à la pression et redescendent à False lors du relâchement. Notons que les logiques de déplacement des [Flèches] et du groupe [ZSQD] sont inversés.
    - Le clic droit passe à True à la pression et redescend à False lors du relâchement
    - Les touches [Espace], [O], [C] et [V] inversent leur état à la pression (True→False et False→True)



On a de nombreuse fonctions pour récupérer joliment le contenu de *self._buttons* : *get_left_mouse, get_middle_mouse, get_right_mouse, get_wheel_up, get_wheel_down, get_left_key, get_right_key, get_up_key, get_down_key, get_touche_espace, get_touche_o, get_touche_c, get_touche_x, get_touche_v, get_touche_Tab, get_chiffre get_touche_entrer*

Dessin

La classe *Screen* contient aussi les fonctions d'affichage. La fonction principale est *draw*. Elle affiche dans cet ordre :

- Le fond
- Les orbites à l'aide de *__draw_orbite*.
- Les corps avec *__draw_world*.
- Le texte sur le bord : l'affichage informatif de ce que font les touches. La couleur du texte explicatif de chaque option varie en fonction de la valeur du booléen de *self._buttons* correspondant à cette option. Il atteint le blanc quand l'option est activée.

Nos Programmes

Après avoir défini ces classes et ces fonctions, nous avons implémenter le *main* du projet, ainsi que les différents tests que chacune des parties devaient passer pour être sûr de valider la plupart des cas.

- *main.py* orchestrant le tout
- et les *tests* de ces parties

Main

Le *main.py* est le chef d'orchestre du projet.

Nous avons décidé de présenter 4 simulations pour voir différents aspects du travail : *Système Solaire, Monde pour collisions, Monde aléatoire, Monde aléatoire gigantesque.*

Initialisations

Nous commençons donc par créer les différents corps ainsi que les mondes associés. On les ajoute aux mondes avec les fonctions *add_N_corps_aleat_diff* et *add_set*.

On boucle d'abord sur les mondes (ceux que l'on veut simuler, ajoutés à *mondes_simulés*). On définit la taille de la fenêtre. On initialise la simulation, et on la fait tourner tant que l'écran ne se ferme pas (fenêtre fermée ou [Entrer] pressée).

Dessin et simulation

La première partie de cette boucle de simulation est l'affichage : le dessin des astres ([C] désactive et réactive cette option) et celui des orbites ([O] active et désactive cette option). Si la simulation doit avancer, on calcule le prochain état au travers de l'appel à *simulator*. Tout cela peut ne pas se réaliser si la barre espace a été enfoncée. En effet, cette dernière permet de mettre en pause la simulation.

Nettoyage et enregistrement des orbites

On parcourt les corps pour enlever les objets de masse nulle qui sont le résultat de collisions et on récupère les positions pour les orbites. Une des fonctionnalités décrite plus bas est le suivi d'un astre et cela nous a conduit à faire attention lorsque l'objet que l'on regardait est sujet à une collision. Il faut alors, dans ce cas, faire une recherche du corps le plus proche de la position actuelle de la caméra. Ce corps sera notre nouvelle référence.
Après avoir fait toutes les suppressions sur le monde actuel, on le met à jour avec *re_init*. On entérine ainsi la suppression des corps.

Evénements et conséquences

Commence alors la lecture des différents évènements :

- Déplacement avec [ZQSD] et les [flèches]
- [X] tapée : remet la liste des orbites à 0.
- [V] tapée : active ou désactive le mode Parlant : l'impression ou non des événements (collision, suppression, recherche corps plus proche et l'aboutissement de la recherche) dans le terminal. Le mode Parlant est activé dès le début.
- [Tab] tapée : change le point de vue en mettant l'astre d'indice suivant au milieu.
- [Chiffre] tapé : change le point de vue en mettant l'astre d'indice égal au chiffre tapé au milieu.
- [Entrer] tapée : fermeture de la fenêtre de la simulation et passage à la suivante si elle existe.

- Molette de la souris : zoom et dézoom caméra.
- Clic central de la souris : déplacement de la caméra vers l'endroit cliqué.
- Clic gauche : positionnement de la caméra
    - Si on tombe sur un astre, à un rayon près, on le suit.
    - Sinon on pose la caméra à l'endroit cliqué.
- Clic droit : fait apparaître des ''astéroïdes'' (corps de masse 1) à l'endroit où l'on a cliqué. Ces astéroïdes ont des caractéristiques aléatoires : leur couleur et leur nom. On a eu un problème sur le nom aléatoire. Le nom est donc maintenant un nom de la forme x+l'indice. Soit par exemple x28. Cela fait très nom d'astéroïde !

Les événements prennent alors fin et on met à jour la focalisation de la caméra sur le corps suivi. Quand on suit un corps, on le bloque au milieu de l'écran et on écrit les paramètres du corps en sa couleur en bas à gauche. Une fois qu'une simulation est terminée (la fenêtre est fermée), on libère la mémoire avec *clear_all*. On passe à la suivant si elle existe sinon le programme s'arrête.

Tests

Nous avons décidé de rajouter des tests à ceux déjà écrits pour pouvoir vérifier le bon fonctionnement de nos fonctions dans différents cas :

- *test_copy :* pour tester la fonction *copy* de *Vector* et vérifié que la copie n'a bien pas la même adresse que le vecteur copié : que l'on a bien 2 objets différents, indépendants !
- *test_add_set :* pour vérifier que notre fonction permet bien d'ajouter une liste de corps (de type *bodies*) à *world*.
- *test_add_N_corps_aleat_diff :* pour être sûrs que l'on peut bien ajouter un nombre aléatoire de corps à la liste initiale et qu'en plus ces corps ne doivent pas avoir la même position qu'un corps déjà existant.
- *test_pop :* pour vérifier que lorsque l'on utilise la fonction *pop* le corps est bien complètement supprimé du monde.
- *test_clear_all :* pour tester si le monde est bien complètement effacé après l'utilisation de *clear_all*.
- *test_clear_all_seuil :* pour vérifier que l'on peut bien changer le seuil de collision dans *clear_all.*
- *test_clear_all_bg_color :* pour vérifier que l'on peut bien changer la couleur dans *clear_all.*
- *test_collision :* pour être sûrs que lors de collision, la masse d'un des corps soit mise à zéro et qu'il y ait donc bien collision.
- *test_non_collision :* pour vérifier que notre programme ne fasse pas de collision quand il n'y en a pas lieu d'être.
- *test_position_from_1 :* pour tester si notre formule pour la fonction *from_screen_coords* fonctionnait bien dans tous les cas.
- *test_position_from_2 :* pour tester si notre formule pour la fonction *from_screen_coords* fonctionnait bien dans tous les cas.
- *test_position_from_3 :* pour tester si notre formule pour la fonction *from_screen_coords* fonctionnait bien dans tous les cas.
- *test_position_from_4 :* pour tester si notre formule pour la fonction *from_screen_coords* fonctionnait bien dans tous les cas.