Université de Toulouse III, FSI – Département de Mathématiques
Licence 2, module “Calcul scientifique”, 2019–2020

Projet: “Réalisation d’un petit moteur physique”

Février 2020

Les moteurs physiques servent à modéliser, en temps réel ou non, des interactions entre des objets solides. Ils sont utilisés dans des domaines très variés, qui vont des jeux vidéos à la conception de pièces de machines. Le but de ce projet est d’implémenter un petit moteur physique interactif en Python.

1 Introduction

Les lois de Newton sont connues depuis plus de trois siècles et permettent de modéliser le mouvement d’un corps connaissant les forces qui s’exercent sur lui. On s’intéresse au mouvement d’une balle de masse m, repérée par la position $x\colon \mathbb{R} \to \mathbb{R}^3$ de son centre au cours du temps. La seconde loi de Newton donne :

$$\frac{d^2x}{dt^2} = \frac{1}{m} f(t), \tag{1}$$

où $f\colon \mathbb{R} \to \mathbb{R}^3$ est la somme des forces s’exerçant sur la balle. Dans un premier temps, on prendra pour f la force de gravitation et le frottement du mouvement dans l’air :

$$f = mg + k_1 \frac{dx}{dt} + k_2 \left\| \frac{dx}{dt} \right\| \frac{dx}{dt}, \tag{2}$$

où $g \in \mathbb{R}^3$ est le champ de gravitation supposé constant, k_1 et k_2 sont deux constantes qui dépendent de la balle. L’équation différentielle d’ordre 2 obtenue dans ce cadre n’admet pas de solution analytique, mais sa résolution approchée par les schémas numériques vus en cours ne présente pas de difficulté particulière.

On se ramène dans un premier temps à une EDO d’ordre 1 en posant

$$X = \left(x, \frac{dx}{dt} \right), \tag{3}$$

et l’équation (2) devient

$$\frac{dX}{dt} = F(X), \tag{4}$$

où

$$F\colon \begin{cases} \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3 \times \mathbb{R}^3 \\ (x, v) \mapsto \left(v, g + \left(\frac{k_1}{m} + \frac{k_2}{m} \|v\| \right) v \right). \end{cases} \tag{5}$$

La position et la vitesse de la balle à l’instant 0 étant supposés connues, on peut appliquer un schéma d’Euler explicite pour effectuer un calcul numérique d’une solution approchée :

$$\begin{cases} X_0 = X(0) \\ X_{k+1} = X_k + hF(X_k), \end{cases} \tag{6}$$

ce qui peut se réécrire, en notant v le vecteur vitesse :

$$\begin{cases} x_0 = x(0) \\ v_0 = \dfrac{dx}{dt}(0) \\ x_{k+1} = x_k + hv_k \\ v_{k+1} = v_k + h\left(g + \left(\dfrac{k_1}{m} + \dfrac{k_2}{m}\|v_k\|\right)v_k\right). \end{cases} \tag{7}$$

Ici, h est le pas temporel du schéma.

2 Collisions

On se propose d'implémenter l'interaction de la balle avec divers obstacles (sol, murs, etc.). On modélisera les collisions avec d'autres objets comme des conditions aux limites pour l'équation différentielle régissant le mouvement sans collision. L'un des problèmes à résoudre est de déterminer précisément l'instant des collisions afin d'éviter que la balle ne pénètre dans un obstacle.

Lors de l'utilisation du schéma d'Euler (7), on obtient un ensemble discret de points de la trajectoire de la balle. Pour simplifier, on peut se représenter la trajectoire approchée correspondante comme une fonction continue x_{approx}, affine par morceaux, passant par chacun des points trouvés :

$$\forall t \in [0, h] \colon x_{\text{approx}}(t_k + t) = \frac{1}{h}\Big((h - t)x_k + tx_{k+1}\Big). \tag{8}$$

Avec cette approche, la détection d'une collision entre les instants t_k et t_{k+1} se ramène donc à trouver les éventuels points d'intersection entre la boule dont le centre parcourt le segment $[x_k, x_{k+1}]$ et les obstacles considérés du problème (supposés immobiles).

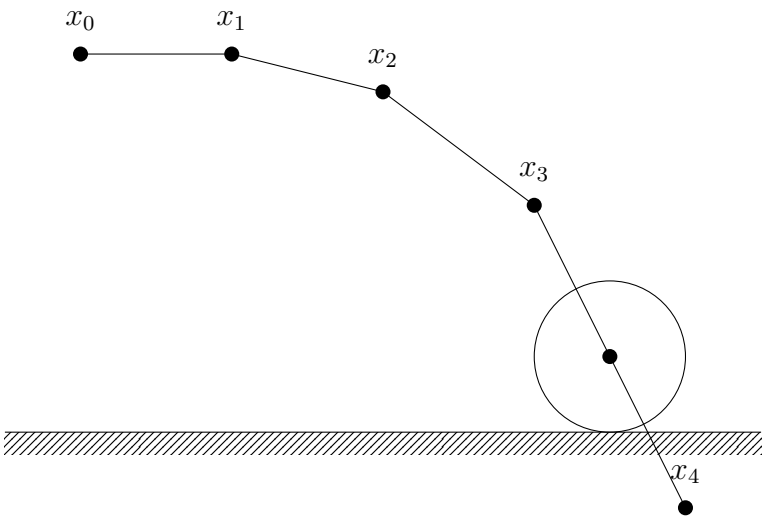

FIGURE 1 – Exemple de détection d'une collision dans l'intervalle $[t_3, t_4]$.

Les obstacles les plus simples à modéliser sont des plans, ou plus précisément des demi-espaces affines, définis par une inéquation du type :

$$\alpha x + \beta y + \gamma z + \delta \geq 0.$$

Le problème de déterminer les intersections entre une boule suivant un parcours rectiligne et un demi-espace se ramène alors à étudier les solutions d'une équation linéaire.

Lorsqu'une collision avec un obstacle est détectée dans l'intervalle $[t_k, t_{k+1}]$, une première approche consiste à :

1. déterminer l'instant t_{col} de la collision,
2. appliquer le schéma d'Euler explicite jusqu'à t_{col} (au lieu d'aller jusqu'à t_{k+1}, ce qui ferait pénétrer la balle à l'intérieur de l'obstacle), de manière à déterminer x_{col} et v_{col},
3. appliquer de nouveau un schéma d'Euler explicite en prenant comme conditions initiales x_{col} et v'_{col}.

Ici, v'_{col} désigne la vitesse de la balle après un rebond. On pourra prendre :

$$v'_{\text{col}} = p(v_{\text{col}}) - \lambda q(v_{\text{col}}), \tag{9}$$

où p désigne le projecteur orthogonal sur l'obstacle, q le projecteur orthogonal associé et $\lambda \in [0, 1]$ une constante dépendant du matériau ($\lambda = 0$ pour un choc mou sans rebond, $\lambda = 1$ pour un rebond parfait).

3 Implémentation en Python

Un moteur graphique temps réel rudimentaire est fourni, capable d'afficher :

- des balles,
- des murs rectilignes.

Il définit deux classes de base (`BaseFlatWall` et `BaseBall`). Le travail des étudiants consistera essentiellement à écrire une classe descendante de `BaseBall` définissant une nouvelle méthode `evolve(self, deltat)` qui sera capable de mettre à jour les vitesse et position de la balle à l'instant $t + \delta t$ les connaissant à l'instant t. Toutes les modifications se feront dans le fichier `ball.py`.

Pour fonctionner, le moteur graphique nécessite l'installation de plusieurs bibliothèques graphiques de Python :

```
pip install PyOpenGL PyGame image --user
```

Listing 2 – Installation des bibliothèques graphiques

L'ensemble des fichiers de travail peut être téléchargé à cette adresse :

https://www.math.univ-toulouse.fr/~vfeuvrie/l2spe/physics.zip

Une fois l'installation effectuée, le moteur se lance de la manière suivante :

```
python engine.py experiment1
```

Listing 3 – Lancement du moteur

Ici, `experiment1` désigne le nom d'un fichier Python situé dans le répertoire courant contenant les paramètres de l'expérience :

- définition des obstacles,
- vitesse et position initiales de la balle.

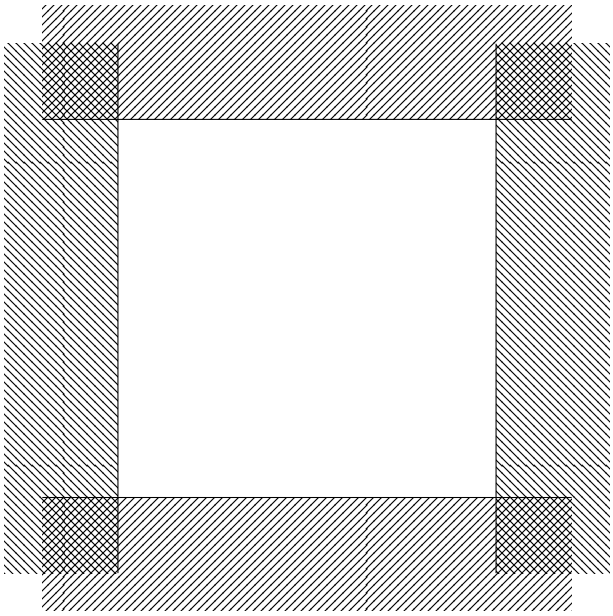

FIGURE 2 – Modélisation des murs d'une « salle » carrée à l'aide de quatre demi-espaces affines

```python
import game, wind

# Une variable globale qui servira à stocker la liste des murs du problème
Bounds = []

class FlatWall(game.BaseFlatWall):
    def intersect(self, p1, p2, r):
        '''
        A faire:
        renvoie l'intersection (si elle existe) entre le mur et une boule de
            rayon r
        se déplaçant sur le segment d'extrémités p1 et p2.

        On utilisera:
        self.P un point qui appartient au plan du mur
        self.N un vecteur normal (sortant) du mur
        '''
        return None

class Ball(game.BaseBall):
    def evolve(self, deltat):
        '''
        A faire:
        mettre à jour P, dP, O, dO (connus à l'instant t) pour déterminer leurs
            valeurs à l'instant t+deltat.
        '''
```

Listing 1 – Fichier `ball.py` à modifier par les étudiants

D'autres fichiers d'expériences sont fournis, et il est possible de définir ses propres paramètres soi-même si on le souhaite. Il est en outre possible d'agir en temps réel sur l'expérience en cours en utilisant les touches de direction du clavier, qui vont modifier le vecteur vitesse de la balle.

Pour simplifier la visualisation des résultats, la modélisation se fera dans $\mathbb{R}^2$. Cela revient simplement à considérer que dans le schéma (7), les vecteurs x_k et v_k ont deux composantes au lieu de trois. La classe `BaseBall` est munie des champs et méthodes suivants :

`R` (flottant) : rayon de la balle ;

`M` (flottant) : masse de la balle ;

`P` (tableau de deux flottants) : position de la balle ;

`dP` (tableau de deux flottants) : vitesse de la balle ;

`O` (flottant) : angle de la balle (à utiliser si on veut modéliser la rotation) ;

`dO` (flottant) : vitesse angulaire de la balle (à utiliser si on veut modéliser la rotation) ;

`evolve(self, deltat)` : une fonction (à réécrire) qui mettra à jour `P` et `dP`. Elle sera appelée fréquemment par le moteur graphique, avant chaque affichage, plusieurs dizaines de fois par seconde. L'affichage se base sur les valeurs de `P` et `R` pour dessiner la balle au cours du temps. Si on ne modifie pas `P` dans la fonction `evolve` alors la balle restera immobile.

La valeur de `deltat` n'est pas fixe : elle dépend de la durée qui s'est écoulée depuis le dernier affichage et s'exprime en secondes. Lors de l'application du schéma (7) pour calculer les nouvelles valeurs de `P` et `dP`, il est important d'utiliser un pas h « petit » pour subdiviser l'intervalle $[t, t+\delta t]$. En effet, dans certaines situations (si la machine « rame », ce qui peut arriver ponctuellement si elle est sollicitée par un autre programme), il se peut que δt soit grand (par exemple, de l'ordre d'une seconde). Si on prenait cette valeur pour le pas h, on pourrait avoir une perte de précision de la modélisation. On s'assurera donc de prendre un pas proche de 10^{-3} par exemple.

Il faudra dans tous les cas ajuster le pas h lors du calcul de manière à :

- ne pas pénétrer dans un obstacle comme expliqué plus haut,
- ne pas dépasser δt.

Il est critique que le code de la fonction `evolve` s'exécute rapidement, pour assurer la fluidité de l'affichage en temps réel. On prendra donc soin de ne pas faire de boucles ou de calculs inutiles dans cette fonction.

La classe `BaseFlatWall` implémente un obstacle plat (un plan) et est munie des champs suivants :

`P` (tableau de deux flottants) : un point qui appartient au plan de l'obstacle ;

`N` (tableau de deux flottants) : un vecteur unitaire normal au plan avec la convention « normale sortante ».

Ainsi, les points M qui ne font pas partie de l'obstacle sont ceux vérifiant :

$$\overrightarrow{PM} \cdot \overrightarrow{N} > 0. \tag{10}$$

Pour gérer les collisions, il faudra écrire un code pour détecter si la balle rencontre l'un des obstacles au cours de l'exécution de la fonction `evolve`, et procéder comme expliqué plus haut.

Voici deux liens vers des vidéos qui permettent de voir ce qu'il est possible d'obtenir avec une gestion complète des collisions :

- https://www.math.univ-toulouse.fr/~vfeuvrie/l2spe/movie1.avi (la vitesse a été divisée par 4 pour laisser le temps de bien observer le comportement lors des collisions)
- https://www.math.univ-toulouse.fr/~vfeuvrie/l2spe/movie2.avi

4 Améliorations possibles

Au cours du travail, plusieurs raffinements pourront être apportés progressivement au modèle physique si l'avancement du projet le permet :

- prise en compte de la rotation de la balle (effet de sol lors des rebonds) ;
- chocs durs, chocs mous, régimes différents lors d'un contact prolongé (lorsque la balle roule sur le sol) ;
- ajout d'une soufflerie pour modéliser l'effet d'un vent (non uniforme) qui agit sur la balle ;
- gestion simultanée de plusieurs balles susceptibles d'entrer en collision les unes avec les autres.