Moteur physique

PELLISIER Matthieu, LAFORTUNE Tristan, TECHER Florian

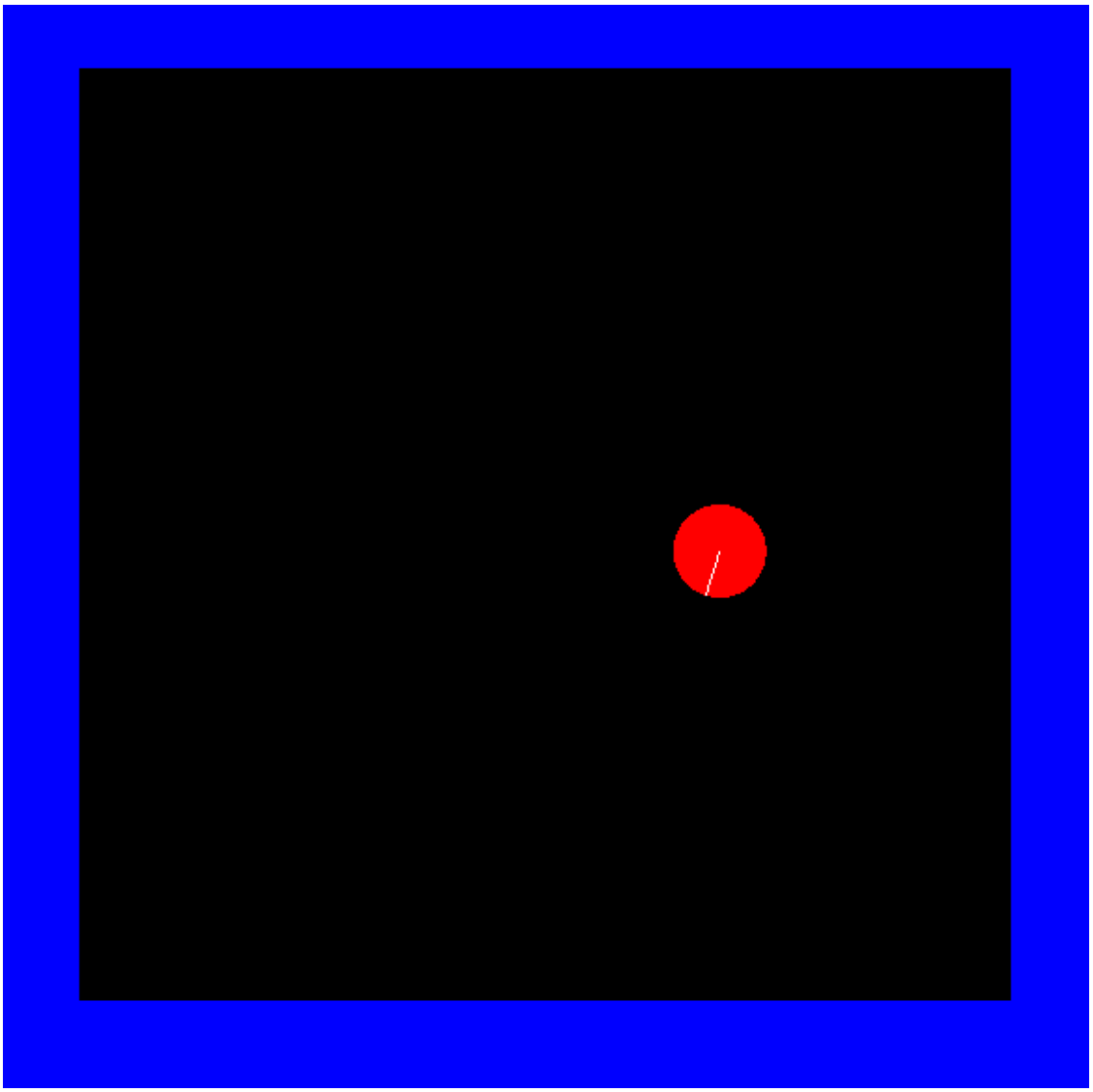


Table des matières

1 Introduction.

Le projet moteur physique, un moyen pour nous, étudiants, d'être initiés à la modélisation informatique, si utile aux physiciens, ou encore à la programmation de jeux vidéo.

L'objectif de ce projet consiste à modéliser le mouvement - en deux dimensions - d'une balle de rayon r soumise à son poids et à une force de frottement due à l'air, confinée dans des espaces de différentes formes (carrée, pentagonale, ou circulaire).

D'abord cette modélisation débute par la création d'une fonction *intersect* permettant de détecter l'intersection entre la balle et les parois de l'espace dans lequel elle est enfermée.

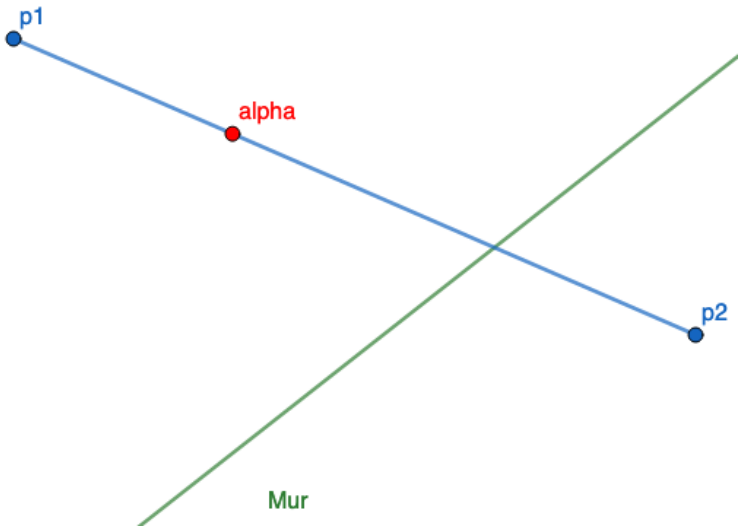
Ensuite, elle se poursuit avec l'implémentation d'une fonction *evolve* qui met à jour la position et la vitesse de la balle, appelle *intersect* pour détecter les collisions et dans ce cas construit la vitesse post-collision.

Commentaire : *nous avons choisi de reprendre le projet depuis le début, et de ne pas seulement nous intéresser aux améliorations possibles.*

2 Construction de la fontion *intersect.*

2.1 Prémisses mathématiques.

Afin de savoir si la balle va rencontrer un obstacle sur sa trajectoire nous allons découper cette trajectoire de sorte à étudier le mouvement de la balle entre deux instants.
Nous avons dans un premier temps approximé la balle à un point pour simplifier les calculs. Nous allons considérer deux points de sa trajectoire : $p1$ et $p2$ de coordonnées respectives $(x1, y1)$, $(x2, y2)$. Entre deux points assez proches, la trajectoire de la balle peut être approchée par un segment.



$$[p1, p2] = \{x \in \mathbb{R}^2 | x = \alpha p1 + (1 - \alpha)p2, \alpha \in [0, 1]\}$$

Nous savons que la balle rentre en contact avec le mur dès lors que $\langle \overrightarrow{PM}, \overrightarrow{N} \rangle = 0$.
Avec P un point du plan de l'obstacle, M un point de $[p1, p2]$ et $\overrightarrow{N}$ un vecteur normal au mur.

$$\begin{aligned} \langle \overrightarrow{PM}, \overrightarrow{N} \rangle &= 0 \\ (\alpha x1 + (1 - \alpha)x2 - x_P, \alpha y1 + (1 - \alpha y2 - y_P).(x_N.y_N) &= 0 \\ \alpha[(x1 - x2).x_N + (y1 - y2).y_N] &= (-x2 + x_P).x_N + (-y2 + y_P).y_N \end{aligned}$$

Finalement :

$$\alpha = \frac{(x_P - x2).x_N + (y_P - y2).y_N}{(x2 - x1).x_N + (y2 - y1).y_N}$$

Si α est entre 0 et 1, il nous donne le moment exacte de la collision.
Enfin pour prendre en compte le rayon de la balle il suffit de le rajouter au numérateur.

2.2 La fontion *intersect.*

On commence par initialiser une liste L qui sera la liste que renverra notre fonction.

```
L = [False]
```

A l'aide de notre analyse mathématiques, nous pouvons décomposer alpha en son dénominateur et en son numérateur :

```
ProdDenom = (p2[0]-p1[0])*self.N[0] + (p2[1]-p1[1])*self.N[1]
ProdNum = (self.P[0]-p1[0])*self.N[0] + (self.P[1]-p1[1])*self.N[1] + r
```

La première chose à faire maintenant est de vérifier si son dénominateur est non nul. On ne voudrait pas diviser par zéro. Une fois cette condition vérifiée on construit alors *alpha*, on assure que sa valeur est cohérente ($0 < alpha < 1$), puis on actualise L qui devient la liste contenant un indicateur de collision (= 1 si collision), *alpha*, le vecteur normal au mur et celui tangent.

```
if ProdDenom != 0 :
            alpha = (ProdNum / ProdDenom)
            if alpha <1 and alpha > 0:
                L = [True, alpha, self.N, self.U]
```

Finalement on renvoit L :

```
Return L
```

3 La fonction *evolve.*

3.1 Actualisation de la position et de la vitesse.

Grâce au principe fondamental de la dynamique, il est facile d'obtenir une équation différentielle pour la position d'un point matériel de masse m, soumis à son poids et à la force de frottement visqueux due à l'air. On peut alors, avec le schéma d'Euler explicite, se ramener à une solution approchée tel que présentée dans le sujet.
Il nous a donc fallu la réécrire sous Python : les lignes suivantes retranscrivent les expressions de la position (*self.P*) et de la vitesse (*self.dP*) de la balle - en l'absence de toute collision - obtenues avec Euler explicite :

```
self.P += h * self.dP
self.dP += h*(g + (k1/self.M + (k2/self.M)*linalg.norm(self.dP))*self.dP)
```

3.2 Faire appel à *intersect.*

Pour éviter que la balle ne traverse tout simplement les murs, nous devons utiliser la fonction *intersect* afin de modifier le mouvement de la balle lors d'une collision. Cette fonction nous renvoie (comme expliqué plus haut) *True* en cas de contact avec un mur (ainsi que les vecteurs $\vec{N}$ et $\vec{U}$), et *False* dans la situation inverse.
La boucle "for W in Bounds :" nous permet de tester la fonction *intersect* pour chaque murs de l'expérience avant tout déplacement de la balle. La fonction *intersect* étudiant la possible intersection entre un instant t et t+1, nous nous plaçons à l'emplacement de la balle au moment du test (position $p1$), et à celui suivant dans le cas où il n'y aurait pas de murs (position $p2$).

```
p1 = self.P
p2 = self.P + h * self.dP
```

Nous pouvons ainsi appeler la fonction *intersect*, laquelle implique une modification qu'en cas de L[0] = True

```
L = W.intersect(p1,p2,r)
if L[0]:
```

Ici, la première ligne nous permet de stocker le résultat de la fonction intersect pour un mur, avant de tester son résultat lors de la seconde ligne de code.

3.3 Vitesses "post-collision" et angle.

3.3.1 Prémisses physiques.

Dans le cas où $L[0] == True$, le mouvement de la balle doit être modifié puisque celle-ci entre en collision avec un mur. Pour obtenir sa vitesse post-collision, on peut l'étudier dans un repère formé de la normale sortante au mur $\vec{N}$, ainsi que sa tangente $\vec{U}$.

Considérons maintenant la balle avant collision avec l'une des parois. Cette dernière va subir - lors du choc - des effets de sol, correspondant à des frottements secs, qui modifierons non seulement la vitesse de translation mais également la vitesse de rotation de la balle.

Nous construisons la vitesse de rotation post-collision "atrificiellement", considérant que la vitesse de rotation devrait être atténuée proportionnellement à la vitesse de translation de la balle. Les frottements exercés par le mur sur la balle, ralentiront le rotation de celle-ci lors de chaque choc, d'où l'apparition du coefficient β. Ainsi, à l'aide des professeurs encadrant le projet, nous avons obtenu la formule :

$$\dot{\theta}_{\oplus} = \beta\dot{\theta}_{\ominus} - \tfrac{1-\beta}{r}\langle\overrightarrow{v_{\ominus}}, \overrightarrow{U}\rangle$$

Il y aura, avant et après le choc, conservation de la composante tangentielle de la quantité de mouvement de la balle :

$$Q^t = m\langle\overrightarrow{v}, \overrightarrow{U}\rangle - I\dot{\theta}$$

Cette dernière nous permet de déterminer la composante tangentielle de la vitesse post-collision ($\overrightarrow{v_{\oplus}}$) :

$$\begin{aligned}
Q^t_{\oplus} &= Q^t_{\ominus} \\
m\langle\overrightarrow{v_{\oplus}}, \overrightarrow{U}\rangle - I\dot{\theta}_{\oplus} &= m\langle\overrightarrow{v_{\ominus}}, \overrightarrow{U}\rangle - I\dot{\theta}_{\ominus} \\
m\langle\overrightarrow{v_{\oplus}}, \overrightarrow{U}\rangle &= I\dot{\theta}_{\oplus} + m\langle\overrightarrow{v_{\ominus}}, \overrightarrow{U}\rangle - I\dot{\theta}_{\ominus} \\
m\langle\overrightarrow{v_{\oplus}}, \overrightarrow{U}\rangle &= I\dot{\theta}_{\oplus} + Q \\
\langle\overrightarrow{v_{\oplus}}, \overrightarrow{U}\rangle &= \frac{I\dot{\theta}_{\oplus} + Q}{m}
\end{aligned}$$

Concernant la composante normale de la vitesse après le choc, nous supposerons seulement qu'elle est de sens opposé à la composante normale avant le choc et qu'elle à été diminué, en la multipliant par un facteur $-\beta$ avec $\beta < 1$:

$$\langle \overrightarrow{v_\oplus}, \overrightarrow{N} \rangle = -\beta \langle \overrightarrow{v_\ominus}, \overrightarrow{N} \rangle$$

On obtient enfin :

$$\begin{aligned} \overrightarrow{v_\oplus} &= \langle \overrightarrow{v_\oplus}, \overrightarrow{N} \rangle \overrightarrow{N} + \langle \overrightarrow{v_\oplus}, \overrightarrow{U} \rangle \overrightarrow{U} \\ &= -\beta \langle \overrightarrow{v_\ominus}, \overrightarrow{N} \rangle \overrightarrow{N} + \frac{I\dot{\theta}_\oplus + Q}{m} \overrightarrow{U} \end{aligned}$$

3.3.2 Mise sous forme de code.

Dans le code il nous suffit simplement, d'entrer - dans la boucle *if* vérifiant qu'il y a collision - les expressions de Q^t, $\dot{\theta}_\oplus$ et $\overrightarrow{v_\oplus}$ (ici nous avons pris $\beta = 0.7$) :

```
if L[0] :
        Q = self.M * dot(self.dP,L[3]) - (self.I * self.dO)
        self.dO = 0.7 * self.dO - ((1 - 0.7)/r) * dot(self.dP,L[3])h = L[1] * h
        self.dP = -0.7*dot(self.dP,L[2])*L[2] + ((Q + self.I * self.dO)/self.M)*L[3]
```

Si il n'y a pas collision, le code exécute le schéma d'Euler du **3.1** pour la vitesse et la position ; pour l'angle, on se contente de modifier simplement la rotation de la balle avec :

```
self.O += h * self.dO
```

3.4 Exécution du programme.

Le programme devant pouvoir être exécuté, et conforme aux lois physiques, depuis différents ordinateurs, nous avons fait en sorte que la simulation ne repose pas sur des variables dépendantes de l'ordinateur telles que la fréquence d'affichage par exemple. On souhaite que la position de la balle soit modifiée tous les "*deltat*".

```
while a < deltat :
                h = 0.01
```

"a" représente l'avancement total dans le temps entre deux instants t séparés de *deltat*. Cette boucle nous permet d'avancer dans le temps via "a" jusqu'à ce qu'il soit égal à *deltat*, et donc de trouver progressivement la position de la balle en *deltat*.
On actualise virtuellement la position de la balle, tout en faisant augmenter a jusqu'à ce qu'il soit exactement égal à *deltat*. De même nous déterminons une potentielle nouvelle position, avec "$a += h$".
La condition suivante nous permet d'éviter les erreurs d'arrondies de Python au niveau des décimales :

```
if a + h >= deltat :
h = deltat - a
a = deltat
```

En cas de collision il se doit de diminuer la valeur de h, afin d'éviter que la balle entre légèrement dans le mur, à l'aide de "$h = L[1] * h$".
Cela revient à effectuer une subdivision de deltat à l'aide d'un pas variable h. On sort de la boucle lorsque $a = deltat$, et pouvons donc modifier la position de notre balle à l'aide des nouvelles vitesses obtenues en *deltat*.

4 Bilan et améliorations possibles

Le code obtenu est capable de simuler le mouvement d'une balle tel qu'il serait dans la réalité. On peut agir directement sur certain paramètres tels que la gravité (le poids), la rotation, etc... On pourrait ainsi avoir une bonne idée de ce à quoi ressemblerai le mouvement d'une balle sur Mars.

Une amélioration possible aurait été la modélisation des chocs entre plusieurs balles. La fonction pour détecter les chocs serait analogue à $intersect$, à la différence près qu'on aurait à considérer l'intersection entre deux segments de la trajectoire de chaque balle.

5 Annexe

5.1 Code de *ball.py* final.

```
import game, wind
from numpy import *

Bounds = []

Wind = None

class FlatWall(game.BaseFlatWall):
        def intersect(self, p1, p2, r):

            L = [False]

            ProdDenom=(p2[0]-p1[0])*self.N[0]+(p2[1]-p1[1])*self.N[1]
            ProdNum=(self.P[0]-p1[0])*self.N[0]+(self.P[1]-p1[1])*self.N[1] + r

            if ProdDenom!=0 :
                alpha = (ProdNum / ProdDenom)
                if alpha <1 and alpha > 0:
                    return [True,alpha,self.N,self.U]

            return L

class Ball(game.BaseBall):
        def __init__(self, R = 1.0, M = 1.0, P0 = [0, 0], dP0 = [0, 0], O0 = 0.0, dO0 = 0.0):

            super().__init__(R, P0, dP0, O0, dO0)
            self.M = M
            self.I = 2 * self.M * self.R ** 2 / 5

        def evolve(self, deltat):
            g=[0,-9.81]
            a = 0
            r=self.R
            k1 = 0
            k2 = 0

            while a < deltat:
```

```
h = 0.001

if a + h >= deltat :
    h = deltat - a
    a = deltat

for W in Bounds :
    p1=self.P
    p2=self.P+h*self.dP
    L = W.intersect(p1,p2,r)

    if L[0] == True:
        Q = self.M * dot(self.dP,L[3]) - (self.I * self.dO)
        self.dO = 0.7 * self.dO - ((1 - 0.7)/r) * dot(self.dP,L[3])
        h = L[1] * h
        self.dP = -0.7*dot(self.dP,L[2])*L[2]+((Q + self.I*self.dO)/self.M)*L[3]

self.P += h * self.dP
self.O += h * self.dO
self.dP += h*(g + (k1/self.M + (k2/self.M)*linalg.norm(self.dP))*self.dP)

a += h
```