esiea
ECOLE D'INGENIEURS
DU MONDE NUMERIQUE

LAB1413
Travaux pratiques logiciel
Enseignants
Arnaud Bannier, Nicolas Bodin, Matthieu Le Berre

TP final

Introduction à la simulation physique

Résumé

L'objectif du TP est de simuler la trajectoire d'une balle dans le plan (2 dimensions), en se basant sur le cours de mécanique du point vu en physique et plus précisément sur le principe fondamental de la dynamique. Dans un premier temps, on se servira de la simulation pour générer un fichier CSV, qui permettra de tracer la trajectoire dans un tableur. Dans un deuxième temps, on se servira du PFD pour animer un sprite en temps réel dans une fenêtre.

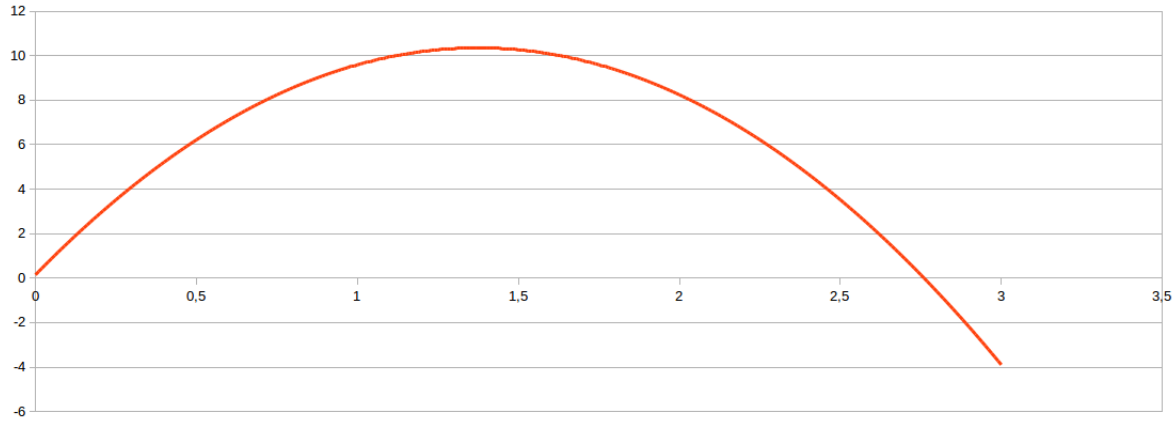

FIGURE 1 – Exemple de trajectoire

1 Rappels du cours de physique

Le principe fondamental de la dynamique (ou 2ème loi de Newton) permet de relier l'accélération $\vec{a}$ d'un solide, sa masse m et l'ensemble des forces $\sum \vec{F}$ qui lui sont appliquées :

$$\sum \vec{F} = m \times \vec{a}$$

Cette relation peut également s'écrire :

$$\vec{a} = \frac{1}{m} \sum \vec{F} \tag{1}$$

La position $\vec{X}$, la vitesse $\vec{v}$ et l'accélération $\vec{a}$ sont reliées par :

$$\vec{v}(t) = \frac{d}{dt}\vec{X}(t)\,,$$
$$\vec{a}(t) = \frac{d}{dt}\vec{v}(t)\,.$$

Et donc,

$$\vec{X}(t) = \int \vec{v}(t)\mathrm{d}t\,,$$
$$\vec{v}(t) = \int \vec{a}(t)\mathrm{d}t\,.$$

Pour l'implémentation, les fonctions d'intégration simples seront utilisées. Elles permettent de déduire les vecteurs vitesse $\vec{v}(t)$ et position $\vec{X}(t)$ à partir de l'accélération :

$$\vec{v}(t+\mathrm{d}t) = \vec{v}(t) + \vec{a}(t)\mathrm{d}t\,, \tag{2}$$
$$\vec{X}(t+\mathrm{d}t) = \vec{X}(t) + \vec{v}(t)\mathrm{d}t\,. \tag{3}$$

C'est quoi $\mathrm{d}t$?

Parce que la physique manque parfois de convivialité, on passe à côté de choses relativement simples à comprendre. Afin de mieux cerner d'où sont issues les formules précédentes, reprenons quelques bases. L'unité internationale permettant de déterminer une distance est le mètre et l'unité de temps est la seconde. Ainsi, naturellement, la vitesse est déterminée en mètres par seconde, noté m/s.

Une accélération peut se définir comme un gain ou une perte de vitesse au cours du temps. Passer de $10m/s$ à $15m/s$ en une seconde correspond donc à une accélération de $+5m/s$ en une seconde, que l'on écrit $+5m/s^2$. Supposons maintenant que notre accélération soit constante à $+5m/s^2$ et que notre vitesse actuelle est de $15m/s$. On cherche à savoir quelle vitesse on aura atteint dans $0,1s$. Bien évidemment, le calcul est simple : $V = 15m/s + 15m/s \times 0,1s = 16,5m/s$. La valeur $0,1s$ correspond au fameux $\mathrm{d}t$.

Forces modélisées

Pour cette première partie, nous ne prendrons en compte que deux forces.

- Le poids $\vec{P} = m \cdot \vec{g}$, avec $\vec{g} = (0, -9.81)^t$.
- Les forces de frottements dans l'air $\vec{f} = -\alpha \cdot \vec{v}$.

2 Calcul et enregistrement de trajectoire

2.1 Fichiers d'entrée et de sortie

Dans cette première partie, le but est d'écrire un programme qui va lire un fichier de configuration décrivant notre balle (masse, coefficient de frottement), son état initial (position initiale et vitesse initiale) et générer un fichier de sortie CSV (comma separated values) décrivant sa trajectoire. Les fichiers csv sont la forme la plus simple de fichier lu par les tableurs comme Excel. Le programme s'exécutera en ligne de commande et prendra 2 paramètres : le nom du fichier d'entrée et le nom du fichier de sortie. Exemple de ligne de commande pour exécuter le programme :

```
1 ./newton balle.txt trajectoire.csv
```

Les noms des fichiers seront récupérés via le paramètre argv de la fonction main. On prendra soin de vérifier le nombre et la validité des arguments (on affichera une aide sur l'utilisation du programme le cas échéant).

Le fichier de configuration devra IMPERATIVEMENT respecter le format suivant :

```
masse <masse en kg>
fCoef <coef de frottement>
position <x> <y>
vitesse <x> <y>
```

Voici un exemple de fichier *balle.txt* :

```
masse 0.5
fCoef 0.9
position 0 0
vitesse 10.0 10.0
```

Le fichier de sortie devra IMPERATIVEMENT respecter le format suivant :

```
<Temps> ; <x> ; <y>[\n]
```

Exemple :

```
0.000000 ; 0.000000 ; 0.000000
0.010000 ; 0.098200 ; 0.097219
0.020000 ; 0.194632 ; 0.191707
0.030000 ; 0.289329 ; 0.283513
0.040000 ; 0.382321 ; 0.372686
0.050000 ; 0.473639 ; 0.459273
0.060000 ; 0.563314 ; 0.543320
0.070000 ; 0.651374 ; 0.624873
...
```

2.2 Implémentation

2.2.a Représentation et opérations sur les vecteurs

Pour effectuer notre simulation, on va avoir besoin de manipuler des vecteurs à 2 dimensions. Pour les représenter on va avoir recours aux structures du langage C :

```
struct Vecteur_decl
{
  float x;
  float y;
};
```

Le mot clef typedef va nous permettre de créer un nouveau type afin de ne pas avoir à taper `struct Vecteur` à chaque fois que l'on voudra déclarer une variable de ce type :

```
typedef struct Vecteur_decl Vecteur;
```

Dans les fichiers *vecteur.h* et *vecteur.c*, implémentez la gestion des vecteurs. Les fonctions suivantes seront nécessaires (respectez les prototypes) :

```
1 // Retourne une structure Vecteur initialisee a (x,y)
2 Vecteur creerVect(float x, float y);
3
4 // Multiplie un scalaire avec un vecteur et retourne le resultat
5 Vecteur multScalVect(float s, Vecteur v);
6
7 // Additionne 2 vecteurs et retourne le resultat
8 Vecteur addVect(Vecteur v1, Vecteur v2);
```

Avant d'aller plus loin, vous pouvez tester votre implémentation en écrivant du code temporaire (`printf()`, ...).

2.2.b Chargement du fichier de configuration et représentation de la balle en mémoire

Notre balle va être également représentée grâce à une structure :

```
1 struct Balle_decl
2 {
3   float masse;            // kg
4   float coeffriction;     // (sans unite)
5   Vecteur position;       // en metres
6   Vecteur vitesse;        // en metres / sec
7   Vecteur acceleration;   // en metres / sec / sec
8 };
9 typedef struct Balle_decl Balle;
```

Dans les fichiers *balle.h* et *balle.c*, déclarez/implémentez la structure et la fonction suivante :

```
1 /* Retourne une structure Balle initialisee
2 avec les valeurs du fichier de configuration */
3 Balle chargerBalle(char * chemin);
```

La fonction `chargerBalle()` prend en paramètre le chemin vers le fichier de configuration de la balle et retournera une variable de type `Balle` correctement complétée. Le chemin sera récupéré grâce au paramètre `argv` de la fonction `main()`. Attention au `fscanf()` avec les `\n` !

2.2.c Calcul de la position selon un pas de temps fixé

On a à présent tout ce qu'il nous faut pour calculer les positions successives de notre balle. Pour cela, on va se fixer un pas d'intégration. Ce pas d'intégration n'est rien d'autre qu'une durée que l'on se donne pour le dt des équations (2) et (3). Plus il sera petit, plus le nombre de positions intermédiaires données par la simulation sera grand. Dans notre cas, 10 ms (0.01 s) donneront un résultat satisfaisant pour le premier programme. La durée de la simulation peut être fixée à 3 secondes pour commencer. Implémentez la fonction suivante qui met à jour la balle passée par adresse en paramètre.

```c
// Met a jour la position de la balle en appliquant le PFD
// retourne -1 si balle est NULL (securite)
int majPosition(Balle * balle, float dt);
```

Algorithme. Pour chaque itération :
- calcul des forces appliquées (poids et forces de frottements visqueux) ;
- calcul de l'accélération courante grâce à l'équation (1) ;
- mise à jour de la vitesse de la balle grâce à l'équation (2) ;
- mise à jour de la position de la balle grâce à l'équation (3).

La nouvelle position de la balle pourra alors être enregistrée dans le fichier après l'appel à la fonction.

Le programme que vous avez à présent constitue la partie CSV des programmes à rendre.

2.2.d Expérimentations

Le fichier de configuration permet de modifier facilement les paramètres de notre simulation. Vous pouvez expérimenter le résultat de l'augmentation/diminution du coefficient de frottement, de même pour la gravité.

3 Simulation temps réel

Dans cette partie, l'objectif est d'écrire trois programmes qui vont permettre de simuler divers comportements physiquement réalistes. La visualisation se fera en temps réel sous forme graphique, grâce à la librairie SDL.

3.1 Introduction basecode

Un code de départ vous est fourni pour chaque programme de cette partie afin que vous n'ayez pas à vous préoccuper des aspects liées à la création d'une fenêtre et à l'affichage dans cette dernière. Si vous compilez le projet avec gcc, n'oubliez pas de linker vers la librairie SDL grâce à l'option `-lSDL` (`-lm` sera certainement nécessaire aussi). Le code source des fichiers *sdl_stuff.h* et *sdl_stuff.c* est fourni, mais il n'est absolument pas nécessaire de comprendre leur contenu.

Dans la partie précédente, la simulation n'était pas en *temps réel*, c'est à dire que la simulation tournait, puis **après** seulement, le résultat pouvait être observé. L'objectif de cette partie est d'observer la balle en temps réel, c'est à dire observer son comportement au fur et à mesure du calcul de son évolution.

Pour cela, la valeur de dt est différente. En effet, une animation correspond à l'affichage successif d'un ensemble d'images (appelées aussi *frames*) comme pour un dessin animé. Le temps séparant deux frames correspond alors à notre dt, puisqu'on cherche quelle est la position de la balle depuis la dernière image. Le temps entre deux frames dépend complètement de la puissance de calcul de votre ordinateur, de sa configuration et des calculs à réaliser.

Ainsi, un ordinateur puissant affichera plus de frames en une seconde que sur un ordinateur lent, donc le dt sera petit. De la même façon, plus il y a de calculs pour mettre à jour les données du jeu et plus il y a d'éléments à afficher, plus le dt sera grand. Ainsi, entre deux ordinateurs, et même sur un ordinateur particulier, le dt n'est pas constant au

cours du programme. C'est pour ces raisons qu'il faut une librairie permettant de calculer l'écoulement du temps de façon précise.

Vous disposez donc des fichiers *fpstimer.c* et *fpstimer.h* qui contiennent des primitives de gestion de temps. Cette librairie s'utilise de la façon suivante. Dans la fonction `main()`, la fonction `fpsInit()` est appelée afin d'initialiser le chronomètre. Après les initialisations SDL, vous trouvez une boucle infinie qui se charge de mettre à jour en continu la fenêtre. Afin de mettre à jour le chronomètre, la fonction `fpsStep()` est appelée au début de cette boucle. Enfin, pour connaitre le temps écoulé depuis la dernière frame, on appelle la fonction `fpsGetDeltaTime()`.

Après la mise à jour de la position de la balle, il ne reste plus qu'à l'afficher. La fonction `sdl_setBallPosition()` vous permettra de modifier la position de la balle à l'intérieur de la fenêtre. Par convention le coin inférieur gauche de la fenêtre correspond aux coordonnées (0,0) et le supérieur droit à (1,1). Cela signifie que l'on observe la balle dans une boite carrée de un mètre de côté. N'oubliez pas de créer la balle à partir du fichier de configuration passé en argument du programme.

3.2 Implémentations

3.2.a Rebonds

Pour que la simulation soit plus attractive, on va à présent simuler les rebonds sur les bords de la fenêtre. On teste pour cela la position de la balle, par exemple si $x < 0$, on modifie la composante x du vecteur vitesse pour qu'il soit dirigé vers les x positifs.

- Gérez le rayon de la balle (donnée par la définition `BALL_RADIUS`) afin que celle-ci ne pénètre pas partiellement dans les « murs », « sol » et « plafond ».
- Simulez de façon simple la perte d'énergie lors des rebonds. Un nouveau coefficient devra être introduit.

Le programme que vous avez à présent constitue la partie Rebonds des programmes à rendre.

3.2.b Forces de gravitation

Le but est de modéliser des forces de gravitation. La force de gravitation appliquée par un corps 1 sur un corps 2 s'exprime par :

$$\overrightarrow{F_{12}} = -G \cdot \frac{m_1.m_2}{d^2} \cdot \overrightarrow{u_{12}},$$

avec

- G la constante gravitationnelle,
- m_1 et m_2 les masses des deux corps,
- d la distance entre les deux corps,
- $\overrightarrow{u_{12}}$ la direction du corps 1 vers le corps 2 (vecteur normalisé).

Dans notre cas, les forces de gravitation s'appliqueront uniquement sur la balle. De plus, nous utiliserons l'approximation suivante :

$$\overrightarrow{F_{12}} = \frac{-0.1}{d^2} \cdot \overrightarrow{u_{12}}.$$

Ces corps sont symbolisés par une planète (Jupiter) et sont au nombre de 5 maximum. L'ajout de ces points de gravitation se fait grâce à un clic gauche de souris sur la fenêtre (ce sera à vous de mettre à jour la liste des attracteurs lors de la détection du clic).

```
typedef struct
{
  Vecteur positionAttracteurs[NB_ATTRACTEURS];
  int utiliseMoi[NB_ATTRACTEURS];
} AttracteurList;
```

La structure `AttracteurList` est définie ci-dessus. Elle est composée de deux tableaux indiquant la position des attracteurs et indiquant si la position correspondante a été initialisée.

Remarque. Pour calculer les forces de gravitation, il faudra compléter notre petite librairie pour manipuler les vecteurs avec quelques fonctions supplémentaires :

```
// Normalise le vecteur v
Vecteur normaliseVect(Vecteur v);

// Calcule la norme du vecteur v
float normVect(Vecteur v);

// Soustrait 2 vecteurs et retourne le resultat
Vecteur subVect(Vecteur v1, Vecteur v2);
```

Vous devez maintenant réaliser les opérations suivantes.

- Compléter les fichiers *balle.c* et *balle.h* avec la structure `AttracteurList` et la fonction `void initAttracteurList(AttracteurList * pAttrList)` qui initialise à zéro les membres.
- Déclarer dans le fichier *main.c* la variable globale `gAttractList` juste au dessus de la fonction `clicSouris(float x, float y)` afin de pouvoir être modifiée par cette dernière. Ce sera la seule variable globale.
- Modifier la fonction `majPosition()` afin de prendre en compte les attracteurs. Le nouveau prototype sera : `int majPosition(Balle * balle, AttracteurList * pAttList, float dt);`
- Compléter la fonction `main()` afin de prendre en compte l'ensemble de ces modifications.

N'oubliez pas que vous pouvez réutiliser de nombreuses fonctions déjà codées (vecteurs...). Afin d'obtenir une animation agréable à visualiser, les paramètres devront être adaptés.

Le programme que vous avez à présent constitue la partie Gravitation des programmes à rendre.

3.2.c Force de rappel d'un ressort

Approche simple. En partant du programme de la section 3.2.a, simulez l'ajout d'un ressort. Sa force de rappel s'exprime :

$$\vec{F} = -k(l - l_0)\,\vec{I}$$

avec k la raideur du ressort (N/m), l l'allongement du ressort (m), l_0 la longueur du ressort au repos et $\overrightarrow{I}$ la direction vers le point d'attache du ressort (vecteur normalisé). On fixera le point d'attache du ressort à (0.5,0.5) par exemple. Attention aux problèmes de stabilité (division par 0 ?).

Chainage. Afin de simuler le comportement d'une corde, on va à présent relier 25 balles entre elle grâce à des ressorts. Un basecode vous est fourni pour l'affichage des 25 balles.

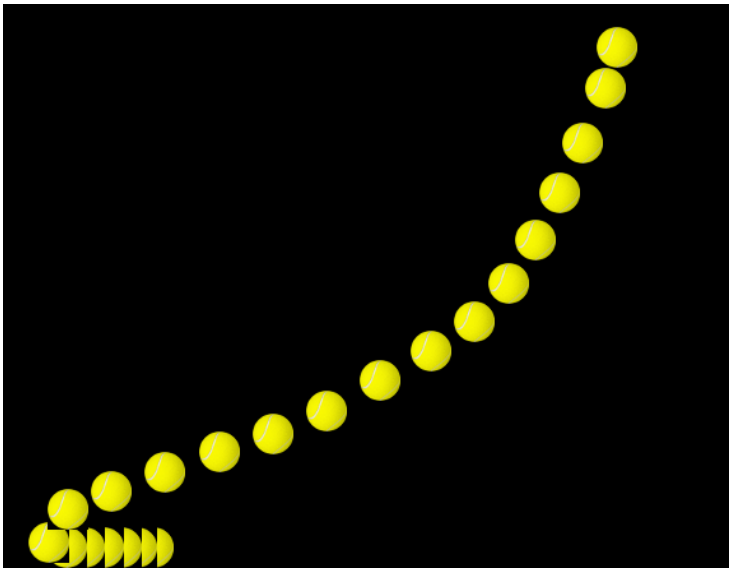

FIGURE 2 – Capture d'écran d'une chaine de balles

Chaque *balle* va donc être soumise à 2 forces de rappels. Pour calculer ces dernières, il va nous falloir la position de la balle précédente dans la chaine ainsi que la position de la balle suivante (pour calculer l'allongement des 2 ressorts). La structure `Balle` doit être modifiée, on va lui ajouter 2 pointeurs :

```
1 struct Balle_decl * ballePrecedente;
2 struct Balle_decl * balleSuivante;
```

N'oubliez pas d'initialiser ces 2 nouveaux membres à `NULL` lors de la création d'une nouvelle balle.

Dans le fichier *main.c*, il nous faut donc :
- Déclarer un tableau de 25 balles (unique variable globale), appelé `gBalleTab`.
- Initialiser les 25 balles.
- Relier les balles entre elles : en mettant `ballePrecedente` et `balleSuivante` à jour (rappel, ce sont des pointeurs).

Puis dans *balle.c* :
- Compléter la fonction `majPosition`. Attention aux extrémités !

La fonction `dragSouris()` du basecode sera utilisée pour définir interactivement la position de la dernière balle. Notre tableau de balles devra dès lors être déclarée global (avant la fonction clic souris) pour être visible à la fois depuis la fonction main et depuis la fonction `dragSouris()`.

Le programme que vous avez à présent constitue la partie Ressorts des programmes à rendre.

3.2.d Expression libre

Les plus rapides pourront rendre un programme supplémentaire optionnel. Vous pouvez par exemple exposer tous les paramètres de la simulation dans le fichier de configuration :

- coefficient pour simuler les pertes d'énergie au rebond,
- les attracteurs,
- le coefficient de raideur du ressort.

Ajouter des fonctionnalités à la simulation :

- Constante en x pour simuler du vent.
- Combinaison de ressorts et attracteurs.
- Interactions différentes avec la souris.
- Toute autre idée qui vous passe par la tête.

Dans le cadre du fil rouge, il est également intéressant de simuler un lancer de fusée. Vous avez déjà les équations grâce à votre cours de physique, il ne reste plus qu'à les implémenter. Vous pouvez également réaliser un petit jeu de simulation physique. Parmi les jeu simples a coder, vous pouvez créer un jeu de raquettes, Flappy bird, Angry brid ou autres... Pensez dans ce cas à réaliser un fichier texte afin d'expliquer le fonctionnement de vos ajouts. Des points bonus seront attribués.

4 Modalités de rendu du TP

Les modalités de rendu sont les suivantes, le non-respect entrainera la perte de points voire 0 :

- Lorsqu'un prototype de fonction, une définition de structure ou un format de fichier vous est précisé, vous NE devez PAS le modifier.
- La deadline est fixée au vendredi 19 janvier à 23h55.
- Le TP se fera par 2 étudiants.
- Le mode de rendu se fera sur Moodle.
- Le TP sera remis sous forme d'archive .zip correctement formée. Le nom de l'archive sera formé par le nom en majuscule de l'étudiant et de son binôme éventuel séparé par le caractère underscore ('_'). Exemple : NORRIS.zip ou NORRIS_VANDAMME.zip. Elle devra créer lors de l'extraction un dossier du même nom (sans l'extension .zip bien entendu).
- L'archive contiendra le code source des 4 programmes à réaliser, dans 4 répertoires distincts, nommés 1_CSV, 2_Rebonds, 3_Gravitation et 4_Ressorts (respectez la casse et sans accent). Le code source devra être commenté et prêt à compiler sous LINUX. Éventuellement un fichier *readme.txt* pourra accompagner chaque partie pour décrire l'usage du programme ou les caractéristiques des fichiers de configuration joints. Un cinquième dossier nommé 5_Bonus pourra être ajouté.