

Thème 4 : Gravité

Dans cette partie, nous allons reprendre le problème traité au semestre précédent sur la gravitation, pour lui appliquer les outils déjà vus dans les précédents thèmes.

L'objectif est ici d'ajouter de nouvelles possibilités au programme simulant la gravitation d'un corps autour d'un autre :

1. remplacer les variables séparées décrivant un corps par des structures plus en rapport avec les représentations formelles habituelles (introduction de *vecteurs*),
2. visualiser graphiquement et en animation les mouvements des corps,
3. prendre en compte les interactions mutuelles,
4. augmenter le nombre de corps en interaction.

On écrira un makefile dans chaque répertoire de travail. Tous les executables devront être compilés par la commande make.

4.1 Rappels théoriques

Les interactions entre corps sont formalisées à partir de quelques équations vectorielle dont la plus importante est la relation fondamentale de la dynamique. Pour des corps assimilés considérés comme ponctuels, cette relation est donnée par :

$$\vec{f} = m\vec{\gamma} \quad (4.1)$$

et (définition de l'accélération comme la dérivée de la vitesse)

$$\vec{\gamma} = \frac{d\vec{v}}{dt} \quad (4.2)$$

Pour résoudre le problème, on utilisera ici la méthode d'Euler où on approxime la définition de la dérivée

$$\vec{\gamma}(t) = \lim_{dt \rightarrow 0} \left(\frac{\vec{v}(t + dt) - \vec{v}(t)}{dt} \right) \quad (4.3)$$

par une équation aux différences finies

$$\vec{\gamma}(t) = \frac{\vec{v}(t + dt) - \vec{v}(t)}{dt} \quad (4.4)$$

d'où l'on tire

$$\vec{v}(t + dt) = \vec{v}(t) + \vec{\gamma}(t)dt \quad (4.5)$$

De la même façon, la définition de la vitesse donne une relation avec le vecteur position $\vec{r} = \vec{OM}$:

$$\vec{r}(t + dt) = \vec{r}(t) + \vec{v}(t)dt \quad (4.6)$$

On complète le problème par l'expression de la force $\vec{f}$ en fonction de la position, de la vitesse et des autres paramètres du système étudié.

4.2 Vecteurs

Pour un scientifique ayant quelques notions de mathématiques niveau bac, il n'est pas nécessaire d'exprimer chaque vecteur des équations précédentes à partir de ses composantes x et y pour comprendre ces équations (au contraire). La méthode employée au semestre précédent, qui n'utilisait que des réels (float ou double) introduit des risques d'erreur supplémentaires, et amène le programmeur à se concentrer sur des détails, alors qu'il doit se concentrer sur le problème à résoudre.

Pour éviter cela, nous allons travailler presque exclusivement avec des vecteurs. Encore faut-il disposer de la bibliothèque permettant de les manipuler.

Exercice 4.1

Écrire un module vecteur (.c et .h) pour manipuler des vecteurs à deux dimensions.

Le module déclarera un type Vecteur et fournira toutes les fonctions utiles pour faire des calculs sur ces vecteurs. Au minimum, on écrira les fonctions permettant d'exécuter le code suivant :

```
Vecteur v1,v2,v3,v4;
double a,b,c,d;
v1=vecteur_creer(12.5,3.4);
v2=vecteur_creer_polaire(20,M_PI*3/4);
v3=vecteur_ajouter(v1,v2);
v4=vecteur_mult_par_scalaire(-4.2,v1);
a=vecteur_produit_scalaire(v2,v3);
b=vecteur_module_carre(v1);
c=vecteur_module(v2);
vecteur_afficher(v1); /* écrit (12.5,3.4)
*/
```

Bien sûr, on testera toutes les fonctions.

A partir de ces briques de base, nous allons pouvoir écrire le programme de simulation de la gravité, en n'utilisant que des vecteurs, et pas les coordonnées (sauf pour l'affichage, mais on pourrait s'en dispenser).

Exercice 4.2

Sur le site <http://francois.marc.emi.u-bordeaux.fr/4TSI402InformatiqueModulaire>, téléchargez le fichier dynamique0.c. Compilez-le avec votre module vecteur.

Si vous avez bien travaillé, il doit compiler du premier coup. Dans le cas contraire, corrigez votre module vecteur.

4.3 Un peu de gravité

Exercice 4.3

A partir du code précédent (que l'on renommera), ajoutez le code pour que le programme :

- simule l'effet de la pesanteur supposée uniforme et constante (et égale à $(0, -g)$,

- arrête la simulation quand le mobile a touché le sol (altitude 0),
- enregistre les positions successives dans un fichier `trajectoire.txt`.
- calcule l'énergie cinétique, l'énergie de potentiel de pesanteur et l'énergie totale (grâce aux fonctions prévues à cet effet) et les enregistre aussi.

Ensuite, faites tracer la trajectoire et les énergies avec `gnuplot`.

On usera et abusera du module `vecteur`.

A ce stade, le programme fait à peu près la même chose que ce que faisait son homonyme au semestre précédent (mais en mieux écrit). Allons maintenant un peu plus loin...

4.4 Graphisme et animation

L'enregistrement des résultats dans un fichier externe permet de tracer de belles courbes pour écrire un rapport ou un article. Pour comprendre le phénomène, il est souvent plus intéressant de le montrer sous forme d'animation. Nous allons progressivement mettre en place cette animation en utilisant les bibliothèques `TPInfo` et `Allegro` vues et mises en œuvre dans le thème précédent. On trouvera au besoin les informations détaillées sur la bibliothèque `Allegro` dans : <http://liballeg.org/stabledocs/en/index.html>.

Exercice 4.4 *Du mouvement*

A partir du code précédent (que l'on renommera `dynamique_anim.c`) et de ce qui a été fait au thème précédent, ajoutez le code nécessaire pour :

- ouvrir une fenêtre graphique 1000x800,
 - y ajuster le système de coordonnées **orthonormé** pour que les points $(-10, -10)$ et $(0, 90)$ soient dans le cadre,
 - tracer les échelles du système de coordonnées,
 - tracer le point représentant le solide.
 - attendre un temps Δt à la fin de chaque itération avant passer au point suivant ¹
-

Ce programme est un début d'animation mais n'est pas encore satisfaisant. En effet, les points tracés ne sont pas effacés. Améliorons cela.

Il existe des techniques perfectionnées pour optimiser l'effaçage de la fenêtre graphique en limitant cette effaçage à la partie animée. Dans le cas présent, nous ferons au plus simple : nous effacerons tout. Pour cela, on peut utiliser l'une des fonctions de la bibliothèque `Allegro` :

```
void clear_bitmap(BITMAP *bitmap);
void clear_to_color(BITMAP *bitmap, int color);
```

Le type `BITMAP *` indique la «surface» sur laquelle on dessine. Dans notre programme, le `BITMAP` est indiqué par la variable globale ² `screen`.

Exercice 4.5 *Nettoyage*

1. Vous pouvez utiliser la fonction `usleep()` de `unistd.h`. La documentation de la fonction est donnée par `man usleep`. Il faut noter que cette méthode est la plus simple mais pas la plus précise : le temps de calcul et de tracé s'ajoute à l'attente. Nous nous en contenterons.

2. `Allegro` utilise une variable globale pour représenter la fenêtre graphique. Cette pratique est déconseillée car elle interdit l'utilisation de deux fenêtre simultanées. Cependant, `Allegro` est à l'origine une bibliothèque pour écrire des jeux vidéos, de préférence en plein écran. On lui pardonne donc.

Modifiez le programme pour avoir une vraie animation, avec un seul point à la fois. L'animation devra cependant garder l'affichage du système de coordonnées. Attention à l'ordre des opérations.

Exercice 4.6 *Un peu d'épaisseur*

Un point pour l'objet, ce n'est pas très lisible. Nous allons remplacer ce point par un cercle. Pour cela, visitez le fichier d'en-tête `TPInfo/graphisme.h` qui se situe dans le répertoire `/usr/local/include` et identifiez la fonction à utiliser pour tracer ce cercle. Utilisez la pour tracer un cercle de rayon 1.

Exercice 4.7 *Frottement*

Nous allons ajouter du frottement à notre système. Pour cela, écrivez une fonction qui calcule la force de frottement $\vec{f}_{\text{frottement}} = (-\mu\vec{v})$ (on suppose la masse égale à 1). Utilisez cette fonction pour appliquer cette force supplémentaire à l'objet et visualiser le résultat.

4.5 Mobiles

Nous allons maintenant simuler l'effet de la gravitation sur plusieurs objets mobiles **simultanément**. La première solution pour faire cela pourrait être de multiplier le nombre de vecteurs position et vitesse, et de multiplier le nombre d'instruction correspondantes dans la fonction `main()`. On pourrait aussi utiliser des tableaux de vecteurs et des boucles (quand c'est possible). Cependant, dans les deux cas, le code deviendrait vite embrouillé alors que ce que l'on veut faire est conceptuellement simple : on veut animer des objets mobiles qui suivent tous les mêmes lois mais peuvent avoir des propriétés différentes (position, vitesse mais aussi couleur, rayon...). Une méthode plus souple consiste à séparer le code associé à chaque mobile. C'est ce que nous allons faire.

Dans cette version, nous allons laisser les fonction du module gérer tout ce qui concerne le mobile et ses propriétés internes.

D'autre part, dans les simulations précédentes, nous avons simplifié les équations pour limiter le nombre de paramètres du système. Cette approche est valide pour simuler un seul objet, mais elle perd son sens pour un système comportant plusieurs mobiles et plusieurs forces. En particulier, la force de frottement ne doit pas être prise en compte directement sur l'accélération, mais doit être ajoutée à la force de la pesanteur, puis la somme doit être divisée par la masse pour calculer l'accélération. Si les mobiles n'ont pas tous la même masse, il faut faire apparaître les masses et les forces dans les équations (ce que nous n'avons pas fait avant).

$$\vec{f}_{\text{totale}} = \vec{f}_{\text{gravite}} + \vec{f}_{\text{frottement}} \quad (4.7)$$

$$= m\vec{g} + (-\mu\vec{v}) \quad (4.8)$$

Dans un deuxième temps, on calcule l'accélération à partir de la force et de la masse

$$\vec{\gamma} = \frac{1}{m} \vec{f}_{\text{totale}} \quad (4.9)$$

Exercice 4.8 *Module mobile*

Construire un module `mobile` qui déclare un type `Mobile` qui comporte les champs `position`, `vitesse`, `acceleration`, `force` et `masse`.

Pour distinguer plus facilement les forces du modèle, on fera une fonction pour chaque force puis on ajoutera les forces successivement afin de calculer $\vec{f}_{totale}$ du mobile. Pour cela, on déclarera 3 fonctions complémentaires :

```
void mobile_annuler_force(Mobile *mobile);
void mobile_affecter_force(Mobile *mobile, Vecteur force);
void mobile_ajouter_force(Mobile *mobile, Vecteur force);
```

représentant respectivement les opérations

$$\begin{aligned}\vec{f}_{totale} &\leftarrow 0 \\ \vec{f}_{totale} &\leftarrow \vec{f} \\ \vec{f}_{totale} &\leftarrow \vec{f}_{totale} + \vec{f}\end{aligned}$$

On notera que dans le module, toutes les fonctions prennent comme premier argument un `Mobile` passé par **référence**.

Ensuite, écrire la fonction

```
void mobile_actualiser_etat(Mobile *mobile, double dt);
```

qui se charge, une fois la force totale exercée sur le mobile calculée, de calculer l'accélération et d'appliquer la méthode d'Euler sur la position et la vitesse.

Enfin, pour finir le travail, on ajoute une fonction pour dessiner :

```
void mobile_dessiner(Mobile *mobile, Ecran *ecran);
```

Le module étant prêt à servir, nous allons maintenant réécrire le programme dynamique que nous allons appeler `dynamique_mobile`.

Exercice 4.9

A partir du fichier précédent, écrire `dynamique_mobile.c`, en remplaçant les vecteurs `r`, `v`, `r_new...` par un `Mobile`.

Les forces seront calculées dans des fonctions prenant un `Mobile` en argument (par référence). Leur expression faisant apparaître les constantes `g` et `mu` définies dans ce fichier, on les laissera dans ce fichier.

Exercice 4.10 *Et de deux...*

Le mobile s'ennuie tout seul. Ajoutez un deuxième mobile. Il partira d'un point différent (au niveau du sol), dans une direction différente.

Exercice 4.11 *Un peu de diversité*

Les deux mobiles sont trop ressemblant. On ne peut pas les distinguer. Ajoutez deux propriétés à la déclaration d'un mobile : un rayon et une couleur et modifiez la façon de dessiner en conséquence.

De plus, modifiez la fonction qui calcule le frottement. Celui-ci sera maintenant donné par : $\vec{f}_{frottement} = (-\mu R \vec{v})$ où R est le rayon du mobile.

Exercice 4.12 *Beaucoup de diversité*

Deux, c'est mieux, mais c'est insuffisant. On veut maintenant avoir NB_MOBILES mobiles qui partent vers le haut (et qui retombent). Pour éviter d'initialiser tout les mobiles à la main, on va générer aléatoirement leurs paramètres initiaux. Pour cela, utilisez les fonctions de la bibliothèque TPIInfo/hasard.h qui permettent de générer un entier ou un réel aléatoire avec un tirage uniforme :

```
int entier_aleatoire(int borne_min, int borne_max);  
double reel_aleatoire(double borne_inf, double borne_sup);
```

La fonction `void initialiser_hasard(void);` doit être appelée au début du programme si on ne veut pas avoir toujours les mêmes valeurs à chaque exécution.

Testez avec NB_MOBILES = 30.
