TP2 : La programmation orientée objet (1)

Préambule

Qu'est ce que la POO ?

Qu'est ce qu'un objet ? Il s'agit d'un mélange de plusieurs variables et fonctions. Imaginez que vous avez créé un programme qui permet de résoudre des équations différentielles ordinaires : vous pouvez afficher vos solutions, calculer des ordres de convergence, comparer deux méthodes ... Le code est complexe : il aura besoin de plusieurs fonctions qui s'appellent entre elles, ainsi que de variables pour mémoriser la solution au cours du temps, la méthode utilisée, le pas de temps choisi ... Au final, votre code est composé de plusieurs fonctions et variables. Votre code sera difficilement accessible par quelqu'un qui n'est pas un expert du sujet : Quelle fonction il faut appeler en premier ? Quelles valeurs doit-on envoyer à quelle fonction pour afficher la solution ? etc ... Votre solution est de concevoir votre code de *manière orientée objet*. Ce qui signifie que vous placerez tout votre code dans une grande boîte. Cette boîte c'est ce qu'on appelle un **objet**. L'objet contient toutes les fonctions et variables mais elles sont masquées pour l'utilisateur. Seulement quelques outils sont proposés à l'utilisateur comme par exemple : définir mon pas de temps, mon intervalle de calcul et ma méthode, calculer l'ordre de la méthode, afficher la solution ...

En quelques lignes :

- La programmation orientée objet est une façon de concevoir son code dans laquelle on manipule des objets.
- Les objets peuvent être complexes mais leur utilisation est simplifiée. C'est un des avantages de la programmation orientée objet.
- Un objet est constitué d'attributs et de méthodes, c'est-à-dire de variables et de fonctions membres.
- On appelle les méthodes de ces objets pour les modifier ou obtenir des informations.

Qu'est ce qu'une classe ? Pour créer un objet, il faut d'abord créer une classe. Créer une classe consiste à définir les plans de l'objet. Une fois que la classe est faite (le plan), il est possible de créer autant d'objets du même type. Vocabulaire : on dit qu'un objet est une **instance** d'une classe.

Objectifs

- Comprendre le principe de la programmation objet à l'aide de l'objet *string* (qui n'est pas un type comme les autres ...)
- Implémenter sa première classe : constructeur, attributs et méthodes.

Exercice 1 - Lancer de poids (sans POO)

L'objectif ici est de modéliser un petit concours de lancer de poids entre ami.e.s.

Tout le TP sera basé sur l'équation de la balistique. La figure 1 illustre le lancer d'un poids. Pour simplifier le problème, on va faire les hypothèses de modélisation suivantes :

- la direction de la largeur est négligée : le problème est donc 2D : longueur x et hauteur y,
- la position initiale d'un poids (x_0, y_0) est proche de $(0, s)$ où s correspond à la taille de la personne qui lance le poids,
- des frottements sont considérés.

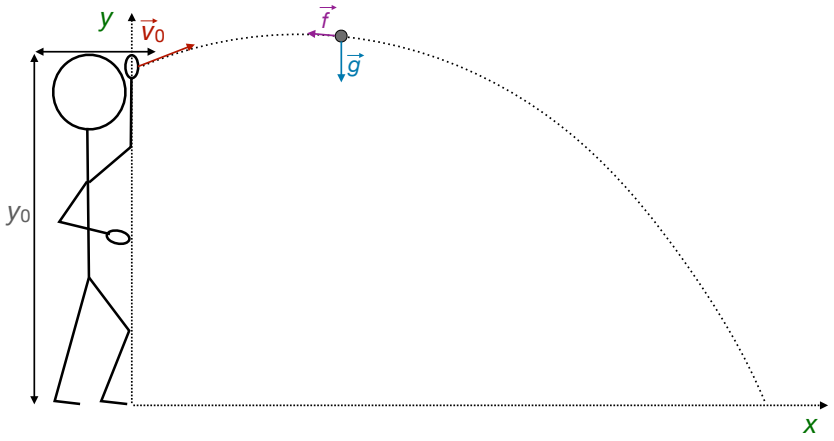

FIGURE 1 – Lancer d'un poids.

Le principe fondamental de la dynamique donne :

$$m\vec{a} = m\vec{g} + \vec{f}, \tag{1}$$

où $\vec{a}$ est l'accélération, m est la masse du poids, $\vec{g}$ est la force gravitationnelle : $\vec{g} = -g\,\vec{y}$, avec g la constante de gravité (égale à 9.80665 m.s^{-2}) et $\vec{f}$ la force de frottement. On suppose les frottements suivants :

$$\vec{f} = -k\|\vec{v}\|\vec{v} \quad \text{où} \quad \vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix},$$

où $\vec{k}$ est la vitesse et k est le coefficient de frottement en g.s^{-1}.
Le problème (1) se réécrit donc sous la forme d'un système d'ordre 1 (gauche) que l'on résoud avec la discrétisation d'Euler explicite (droite) :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{v_x} \\ \dot{v_y} \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ -\dfrac{k}{m}\sqrt{v_x^2 + v_y^2}\, v_x \\ -\dfrac{k}{m}\sqrt{v_x^2 + v_y^2}\, v_y - g \end{pmatrix}, \qquad \begin{cases} x^{n+1} &= x^n + \Delta t\, v_x^n, \\ y^{n+1} &= y^n + \Delta t\, v_y^n, \\ v_x^{n+1} &= v_x^n - \Delta t\, \frac{k}{m}\sqrt{(v_x^n)^2 + (v_y^n)^2} v_x^n, \\ v_y^{n+1} &= v_y^n - \Delta t\, \frac{k}{m}\sqrt{(v_x^n)^2 + (v_y^n)^2} v_y^n - \Delta t\, g. \end{cases}$$

Prénom	Taille (m)	Amplitude $\|\vec{v}_0\|$ (m.s^{-1})	Angle accos$\left(\frac{\vec{v}_0^x}{\|\vec{v}_0\|}\right)$ (rad)
Ines	1.63	7	0.5
Maël	1.75	6.5	0.7
Abdel	1.83	7.2	0.6
Jeanne	1.58	5.7	0.65

TABLE 1 – Caractéristique des 4 joueurs/joueuses

1. Créer un code permettant de générer le lancer d'un poids. Pour valider votre code, la position finale estimée d'un poids est 6.26331 m avec les paramètres :

```
double g = 9.80665; // Gravity (m.s^{-2})
double dt = 0.005; // Step time
double x0(0), y0(1.8), normv0(7.2), anglev0(0.6); // Initial conditions
double coeff = 100; // Turbulent friction (g.s-1)
double mass = 4000; // Mass (g)
```

2. Créer un code permettant de générer 3 lancers de poids de 4 joueurs/joueuses (dont les infos sont données dans le tableau 1). Une illustration est donnée dans la figure 2. On suppose qu'à chaque lancer d'un joueur, la position initiale et la vitesse suivent les lois suivantes :

$$y_0 \sim \mathcal{N}(\text{Taille}, 0.05^2),\ \|\vec{v}_0\| \sim \mathcal{N}(\text{Amplitude}, 0.01^2) \text{ et } \text{accos}\left(\frac{\vec{v}_0^x}{\|\vec{v}_0\|}\right) \sim \mathcal{N}(\text{Angle}, 0.03^2).$$

N'hésitez pas à utiliser l'exercice 8 du TP 1.

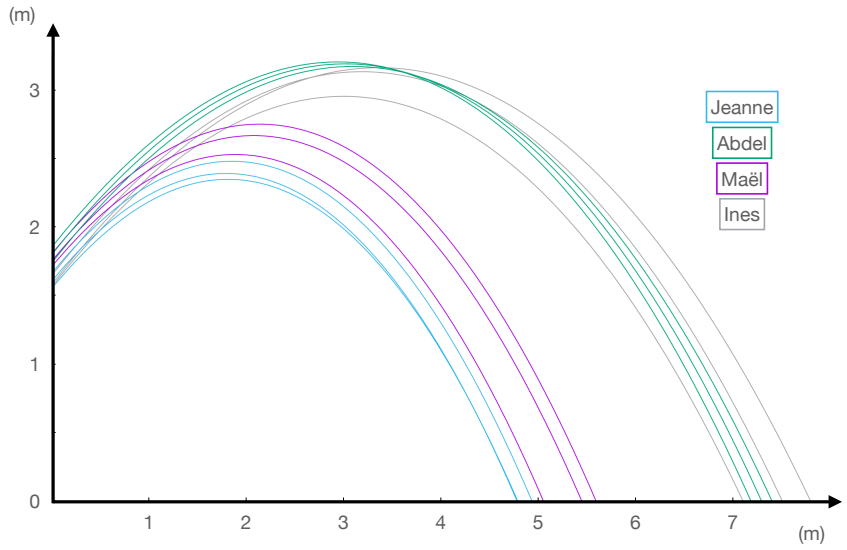

FIGURE 2 – Type de courbes à obtenir

Remarque : On peut tracer les solutions avec des boucles sur `Gnuplot` (avec guillemet droit) :

```
plot for [p=0:3] for [l=0:2] 'Results/sol_np_'.p.'_nl_'.l.'.txt' using 2:3 notitle w lines lc p
```

3. Qui est la gagnante/le gagnant ?

Exercice 2 - Création d'une première classe

Le **vrai objectif** de ce TP est d'introduire la Programmation Orientée Objet et de créer vos premières classes !
Rappel (voir le préambule 1) :

- une **classe** ce sont des fonctions/méthodes,
- les **variables/attributs** et les **objets** ce sont des instanciations de la classe.

Le code que vous venez de créer ne serait pas facile à suivre pour un novice ... Comprendre les enchaînements de boucle etc ... Un des objectifs de la POO est de pouvoir séparer le code qui s'adresse aux **utilisateurs** (quelqu'un qui voudrait juste *jouer avec le code* et donc juste dire combien d'amis ont joué, leurs tailles et leurs capacités à lancer des poids) et celui qui s'adresse aux **développeurs** (quelqu'un qui voudrait implémenter une autre fonctionnalité du jeu comme par exemple différents types de lancer : à la cuillère, bras levé etc ...). En C++, la partie de code pour l'utilisateur c'est la fonction *main* (c'est TOUJOURS le point d'entrée quand on découvre dans un nouveau code). On aimerait pouvoir avoir comme fonction *main* un code très accessible comme celui ci :

```
int main()
{
  // Nombre de joueurs
  int nbplayers(4);
  // Leurs caractéristiques
  std::vector<double> size(nplayer);
  std::vector<double> normv0(nplayer);
  std::vector<double> anglev0(nplayer);
  // Ines, taille moyenne, lanceuse de poids
  size[0] = 1.63; normv0[0] = 7; anglev0[0] = 0.5;
  // Mael, taille moyenne, pas de sport connu
  size[1] = 1.75; normv0[1] = 6.5; anglev0[1] = 0.7;
  // Abdel, taille supérieure à la moyenne, nageur
  size[2] = 1.83; normv0[2] = 7.3; anglev0[2] = 0.6;
  // Jeanne, taille inférieure à la moyenne, pas de sport connu
  size[3] = 1.58; normv0[3] = 5.7; anglev0[3] = 0.65;
  // Nombre de lancers
  int nbLaunch(3);
  // Boucle sur les joueurs
  for (int nf = 0 ; nf < nbplayers ; nf++)
  {
    // Définir le player
    Player player(nf, size[nf], normv0[nf], anglev0[nf], nbLaunch);
    // Le faire jouer
    player.multThrow();
  }
  return 0;
}
```

Pour réussir à faire ça il faut définir un objet *Player* (comme on définit par exemple un personnage dans un jeu vidéo).

1. Pour commencer, nous allons tout d'abord créer une classe *Shot* définissant un objet correspondant à un poids. L'idée est de pouvoir lancer le code suivant (le copier dans un fichier intitulé *main_class_shot.cc*) :

```
#include <fstream>
#include "Shot.h"
using namespace std;
int main()
{
  // Initial conditions
  double x0 = 0;
  double y0 = 1.8;
  double normv0 = 7.2;
  double anglev0 = 0.6;

  // Build shot
  double mass = 4000; // g
  Shot shot(mass, y0, normv0, anglev0);
  // Throw shot
  shot.computeTrajectory();
  // Print shot after
  cout << shot << endl;
  // Write in a file
  string name_file("Results/solution_1_shot.txt");
  shot.saveSolution(name_file);

  return 0;
}
```

Si vous compilez ce code, vous obtenez une erreur puisque la classe *Shot* n'est pas définie :

```
[shot] $ g++ -std=c++11 -o run main_class_shot.cc
main_class_shot.cc:5:2: error: unknown type name 'Shot'
        Shot shot(9.3);
```

2. Dans un fichier *Shot.h* placer le prototype/la déclaration de la classe *Shot* :

```
// Les includes dont vous avez besoin dans le fichier .h
// Attention à ne pas mettre de using namespace dans le .h car c'est ce
// fichier qui va être inclu et il peut donc contaminer le reste du code ...
#include <iostream>
#include <vector>
// Déclaration de votre classe
class Shot {
  private: // Les attributs de la classe
    // La masse du poids
    const double _mass;
    // La position initiale du poids
    const double _y0;
    // L'amplitude moyenne de la vitesse initiale
    const double _normv0;
```

```
    // L'angle moyen de la vitesse initiale
    const double _anglev0;
    // L'angle moyen de la vitesse initiale
    std::vector<double> _v0;
    // Temps, position x, position y
    std::vector<std::vector<double> > _t_x_y;

  public: // Méthodes et opérateurs de la classe
    // Constructeur qui permet de construire un objet de la classe
    // Il peut y avoir différents constructeurs
    // Il doit porter le même nom que la classe !
    Shot(const double mass, const double y0, const double normv0, const double anglev0);
    // Un deuxième constructeur (dit par copie)
    Shot(const Shot& shot);
    // Lancer un poids
    void computeTrajectory();
    // Sauvegarder la solution
    void saveSolution(std::string nameSol);

    // Récupérer la masse du poids à l'extérieur de la classe
    // (sans pouvoir la modifier !!)
    const double& getMass() const {return _mass;};
    // Récupérer la position y0
    const double& gety0() const {return _y0;};
    // Récupérer la vitesse v0
    const double& getnormv0() const {return _normv0;};
    const double& getanglev0() const {return _anglev0;};
    // Récupérer le temps, la position et la vitesse du poids
    const std::vector<std::vector<double> >& getdata() const {return _t_x_y;};
};
// Permet d'afficher les propriétés du poids
std::ostream& operator<<(std::ostream& out, Shot const& shot);
```

Par convention, on met un underscore devant les variables privées de la classe. Dans un fichier *Shot.cpp* il faut définir les fonctions. Pour savoir que les fonctions dépendent de la classe il faut rajouter le nom de la classe devant :

```
type NomClasse::nomFonction(arguments)
```

Ce qui devient dans notre cas :

```
#include <fstream>
#include "Shot.h" // Inclure le fichier .h

// Un premier constructeur qui détermine la masse
Shot::Shot(const double val_mass, const double val_y0, const double val_normv0,
                                  const double val_anglev0)
{
}
// Un deuxième constructeur (dit par copie)
Shot::Shot(const Shot& shot)
```

```
{
}
// Calculer la trajectoire d'un poids
void Shot::computeTrajectory()
{
}
// Sauvegarder la solution dans un fichier
void Shot::saveSolution(std::string nameSol)
{
}
```

Le fichier .h est déjà intégré dans le *main_class_shot.cc* (**Vérifier que c'est le cas !**) et il ne faut surtout pas ajouter le .cpp.

3. Implémenter les fonctions dans le .cpp.
Quelques conseils/astuces :

a) *Constructeur 1 :* Le constructeur a pour rôle d'initialiser les attributs de votre classe (ici *_mass*, *_y0*, *_normv0* et *_anglev0*). L'underscore devant permet de ne pas oublier que ce n'est pas une variable classique. Il y a 2 manières de faire qui sont données dans le code ci-dessous :

```
// Dans cet ex, la valeur du 1er attribut est donnée à travers le constructeur :
// Il est supposé constant comme _mass par ex.
// Le 2ème est fixé directement (on voit qu'il s'agit d'un string)
// Le 3ème est un vecteur vector<type3> dont la taille est fixée à 10.
// 1ère stratégie
MaClasse::MaClasse(type1 val_attribut1) :
_attribut1(val_attribut1)
{
   this->_attribut2 = "mon_resultat";
   this->_attribut3.resize(10);
}
// 2ème stratégie (A PRIVILÉGIER AU MAXIMUM !)
MaClasse::MaClasse(type1 val_attribut1) :
_attribut1(val_attribut1), _attribut2("mon_resultat"), _attribut3(10)
{
}
```

Ces variables sont accessibles ***PARTOUT*** dans la classe. Pour bien le signaler on les appelle en ajoutant

```
this->_mass
this->_y0
this->_normv0
this->_v0
etc...
```

b) *Constructeur 2 :* Utiliser les fonctions *getMass*, *gety0*, *getnormv0* et *getanglev0*.

Une fois les constructeurs implémentés, compiler et exécuter le code. À partir de maintenant, compiler très très régulièrement pour valider les différentes étapes de votre implémentation !

c) *Fonction get** :* Elles ont déjà été implémentées, directement dans le fichier *Shot.h* comme c'est classique de faire pour les fonctions *get*.

d) *Fonction computeTrajectory :* Implémenter le lancer d'un poids (modèle balistique). Mettre les résultats dans le vecteur de vecteurs $\_t\_x\_y$ (1ère colonne : temps ; 2ème : x ; 3ème : y).

e) *Fonction saveSolution :* Sauvegarder la solution (1ère colonne : temps ; 2ème : x ; 3ème : y).

4. Il y a encore deux notions très importantes sur les classes qui n'ont pas été abordées. Tout d'abord il y a la notion de **template**. Jusqu'ici nous avons passé des variables en paramètre des fonctions. Grâce au concept de template, il est possible de passer en paramètre des types et ainsi de définir des fonctions génériques. Mais le concept de template ne s'arrête pas aux fonctions, on peut aussi l'utiliser pour les classes et les structures. La notion n'a pas de sens dans le contexte actuel mais vous avez déjà utilisé des classes templatées comme par exemple la classe vector :

```
std::vector<int> mon_vecteur_d_entiers;
std::vector<double> mon_vecteur_d_entiers;
```

L'objectif du template de la librairie *vector* est d'utiliser le même code pour un vecteur d'entiers, de réels, de complexes ... La force des templates (reconnaissable aux chevrons) est d'autoriser une fonction/une classe à utiliser des types différents.

5. La deuxième notion importante concerne la **surcharge d'opérateurs**. C'est une technique qui permet de réaliser des opérations mathématiques intelligentes entre vos objets, lorsque vous utilisez dans votre code des symboles tels que +, -, *, ==, <, etc. Encore une fois vous l'avez déjà utilisé par exemple pour additionner ou comparer 2 strings :

```
string mot1("Ciao"), mot2("Salut"), mot3, mot4;
mot3 = "Vous préférez dire " + mot1 + " ou " + mot2 + " ?";
cout << mot3 << endl; //Concaténation
if (mot1 == mot2) //Comparaison de 2 mots
    cout << "Les deux mots sont identiques." << endl;
else
    cout << "Les deux mots sont differents." << endl;
```

L'addition de deux poids n'a pas vraiment de sens mais nous pouvons par contre comparer 2 poids. Une définition raisonnable est de dire que les poids doivent avoir la même masse, la même position et vitesse initiales. Nous pouvons aussi vouloir afficher les propriétés du poids comme par exemple sa masse, sa position et sa vitesse initiales et sa position finale (s'il a été lancé). On peut aussi vouloir comparer deux poids en comparant leur position finale (> si le poids est arrivé plus loin).

6. Les surcharges d'opérateur se font à l'extérieur de la classe. Les prototypes des surchages d'opérateur dans le fichier .h se mettent après la fermeture de l'accolade et du point virgule :

```
// Surchage d'opérateur
// Permet d'afficher les propriétés du poids
std::ostream& operator<<(std::ostream& out, Shot const& st);
// Permet de comparer deux poids entre eux
bool operator==(Shot const& st1, Shot const& st2);
bool operator>(Shot const& st1, Shot const& st2);
```

et les définitions dans le .cpp se mettent à la fin du fichier pour plus de clarté :

```
// Permet d'afficher les propriétés du poids
```

```cpp
std::ostream& operator<<(std::ostream& out, Shot const& st)
{
  out << [...] << std::endl;
  return out;
}
// Permet de dire si deux poids sont les mêmes
bool operator==(Shot const& st1, Shot const& st2)
{
  if [...] // Effectuer la comparaison
     return true;
  else
     return false;
}
// Permet de comparer deux poids entre eux
bool operator>(Shot const& st1, Shot const& st2)
{
  if [...] // Effectuer la comparaison
     return true;
  else
     return false;
}
```

Exercice 3 - Lancer de poids (avec la POO)

1. Maintenant nous allons pouvoir créer la classe *Player* permettant d'utiliser la fonction *main* introduite page 4. Voici le prototype de la classe *Player* (à placer dans un fichier Player.h).

```cpp
class Player {
  private: // Les attributs de la classe
    // Dépendant du joueur
    // L'identité du joueur (ici un numéro mais pourrait être un nom)
    const int _id;
    // La taille du joueur
    const double _size;
    // L'amplitude moyenne de la vitesse initiale
    const double _normv0;
    // L'angle moyen de la vitesse initiale
    const double _anglev0;
    // Le nombre de poids lancés à chaque fois;
    const int _nbLaunch;
    // Les distances de lancer des poids
    std::vector<double> _positions;
    // Masse moyenne d'un poids
    const double _massShot;
    // Pour générer de l'aléatoire
    std::default_random_engine _generator;
    std::normal_distribution<double> _distribution;

  public: // Méthodes et opérateurs de la classe
```

```
23      // Constructeur qui permet de construire un objet de la classe
24      Player(int id, double size, double normv0, double anglev0, int nbLaunch);
25      // L'action principale du joueur
26      void multThrow();
27      // Distance atteinte par le joueur
28      const std::vector<double>& getFinalPos() const {return _positions;};
29  };
```

Implémenter *Player.cpp* par mimétisme avec la classe *Shot* (sans le constructeur par copie).

2. Augmenter le nombre de lancers. Quelle est la moyenne des lancers de chaque joueur ? Qui a le plus de chance de gagner si on considère le meilleur lancer ? Et si on considère la moyenne ?