Introduction

Le but de cette session est de se familiariser avec les notions de classe, de pointeur, de référence du `C++`.

Il s'agit également de mettre en place les premiers éléments permettant la simulation à grandes échelles de systèmes particulaires complexes.

Vous êtes libres d'utiliser l'environnement de développement de votre choix. Je recommande d'utiliser `Visual Studio Code`.

1 Analyse de performance

Vous serez amener à analyser les performances de votre implémentation à différentes étapes.

Il existe plusieurs applications permettant de faire de mesurer les performances d'une implémentation. L'application la plus utilisée est `gprof`.

Afin d'utiliser `gprof`, le programme doit être compilé avec l'option `-pg`. L'exécution du programme générera un fichier de trace `gmon.out`.

En exécutant `gprof` sur le programme, vous obtiendrez un rapport d'analyse, verbeux, de l'exécution.

Le rapport est composé de deux parties :

- le profil plat, i.e., le détail des appels de chacune des fonctions : le temps, le nombre d'appel ;
- le profil d'appel, i.e, la hiérarchie d'appel des fonctions.

Pour rendre le rapport plus digeste, en enlevant les explications, il faut exécuter `gprof` avec l'option `-b`.

2 Mise en place des structures de données

Dans ce premier exercice, nous allons décrire la structure de donnée élémentaire utilisée.

Nous allons étudier le comportement de plusieurs dizaines, plusieurs centaines, voire plusieurs milliers de particules.

Dans un premier temps, on définira une particule comme un objet tri-dimensionnel caractérisé par une position, une vitesse, une masse, un identifiant, un catégorie (un type) et une force.

Question 1.
Créer une classe `Particule` qui correspondent à ces caractéristiques.

La simulation se compose d'une collection de particules. Il va donc être nécessaire de les garder en mémoire.

On pourrait créer une liste chaînée écrite à la main. Cependant, la bibliothèque standard offre une grande variété de collections. Elles ont toutes des propriétés différentes. Vous pouvez vous référer à la documentation pour les détails. On reviendra sur le sujet plus tard.

Pour utiliser une collection spécifique, vous devez inclure l'entête correspondante :

```
1 #include<set>
2 #include<list>
3 #include<deque>
4 #include<vector>
```

La collection est `template`, c'est-à-dire qu'elle s'adapte au type pour lequel on l'initialise (la notion de `template` sera abordée plus tard).

On instancie une collection, par exemple, de la manière suivante

```
std::list<Particle> particleList;
```

Toutes les collections ont des méthodes communes :

- insertion : `insert(val)` ;
- effacement : `erase(pos)` ;
- comptage : `size()` ;
- et bien d'autre.

Une fois construite, la collection se parcours à l'aide d'un itérateur :

```
auto it = particleList.begin();
```

`it` se comporte comme un pointeur. On peut l'utiliser dans les boucles pour parcourir la collection entre ses bornes :

```
for (it = particleList.begin(); it != particleList.end(); ++it++) {};
for (auto it : particleList){} ;
```

Etant donner que nous souhaitons créer une variété différente de particules, nous allons les initialiser de manière aléatoire

```
#include <random>
#include <iostream>

int main() {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    for (int i=0; i<16; ++i)
        std::cout « dist(mt) « "\n";
}
```

Question 2.
Créer une collection de particules.

Question 3.
Comparer les performances de chacune des collections pour $64, 128, \ldots, 2048, \ldots$ particules. A partir de quelle taille est-ce que les différences s'observent ?

Pour mesurer les performances, nous allons utiliser la bibliothèque `chrono` disponible ici.
Il suffit ensuite d'entourer les éléments de code à mesurer avec des compteurs

```cpp
auto start = std::chrono::steady_clock::now();
/* Code à évaluer */
auto end = std::chrono::steady_clock::now();
std::chrono::duration<double> elapsed_seconds = end-start;
std::cout « "elapsed time: " « elapsed_seconds.count() « "s\n";
```

3 Méthode de Störmer-Verlet

Il s'agit ensuite de déplacer les particules en les soumettant à différentes forces d'interaction.

Les forces d'interaction entre les particules sont définies explicitement. Dans un premier temps, nous allons utiliser la force d'interaction gravitationnelle :

$$F_{ij} = \frac{m_i m_j}{r_{ij}^3} \mathbf{r}_{ij} \tag{1}$$

La force qui s'applique à une particule est la somme des forces d'interactions.

$$F_i = \sum_{i \neq j} F_{ij} \tag{2}$$

L'évolution du système particulaire se calcule à l'aide de l'algorithme de Störmer-Verlet :

Algorithme 1 : algorithme de Strömer-Verlet

Data : particles initiales
Data : vecteur de force $\mathbf{F}^{old}$
Initialisation : calcul des forces $\mathbf{F}$;
while $t < t_{end}$ **do**
  $t = t + \delta_t$;
  for *toutes les particles* i **do**
    $x_i = x_i + \delta_t * (v_i + 0.5/m_i * \mathbf{F}_i * \delta_t)$;
    $F_i^{old} = F_i$;
  end
  Calculer les forces $\mathbf{F}$;
  for *toutes les particles* i **do**
    $v_i = v_i + \delta_t * 0.5/m_i * (\mathbf{F}_i + \mathbf{F}_i^{old})$;
  end
  Calculer les quantités dérivées ;
  Afficher les quantités t, x, y ;
end

Dans un premier temps, nous allons nous intéressés au systèmes gravitationnel composé du soleil, de Jupiter, de la Terre et de la comète de Halley.

Corps	Masse	Position	Vitesse
Soleil	1	(0,0)	(0,0)
Terre	$3.0e^{-6}$	(0,1)	(-1,0)
Jupiter	$9.55e^{-4}$	(0, 5.36)	(-0.425,0)
Haley	$1.e^{-14}$	(34.75,0)	(0, 0.0296)

On choisira $\delta_t = 0.015$ et $t_{end} = 468.5$.

Question 4.
Implémenter l'algorithme de Strömer-Verlet.

Question 5.
Vérifier votre implémentation pour le système gravitationnel proposé.

Question 6.
Modifier la classe `Paritcule` afin de protéger les données en écriture. Pour cela, vous devrez utiliser des méthodes d'accès.

Vous pourrez sauvegarder les résultats dans un fichier texte. Ce fichier devra contenir la position de chaque astre à chaque itération sur la même ligne.