# Rapport du Code C++.

Abdellah Ouadi
Abdellah Belaid

**Group** 2

May 18, 2022


# Contents

**1 Mise en place des structures de données** **3**
1.1 Comparaison des performances des différents collections . . 3

**2 Algorithme de Strömer-Verlet** **4**

**3 Univers de particules** **4**
3.1 architecture . . . . . 4
3.2 performance . . . . . 5
3.3 Résultats . . . . . 5

**4 Découpage de l'espace** **6**
4.1 Architecture . . . . . 6
4.2 performance . . . . . 7
4.3 Résultats . . . . . 7

**5 Test et visualisation** **8**
5.1 Implémentation des tests à l'aide de l'outil gtest . . . . . 8
5.2 Résultats de Paraview . . . . . 8

**6 ACVL** **10**
6.1 Diagramme des cas d'utilisations . . . . . 10
6.2 Diagramme de séquence . . . . . 11
6.3 Diagramme de transitions . . . . . 11
6.4 Diagramme de classes d'analyse . . . . . 12

**7 Raffinement du modèle** **13**
7.1 Conditions aux limites . . . . . 13
7.2 Conditions aux limites réflexives avec potentiel . . . . . 13
7.3 Potentiel gravitationnel . . . . . 13
7.4 Application : collision de deux objets . . . . . 13

# 1 Mise en place des structures de données

## 1.1 Comparaison des performances des différents collections

Pour une particule p, on utilise la méthode d'insertion suivante:

**particleList.insert(particleList.end(), p);**

Pour un nombre croissant de particules, on évalue les performances des insertions pour les structures de données **vector**, **deque**, et **list**, on trouve les résultats suivants:

* Elapsed time for 4096 particles:
vector : 0.00435621s
deque : 0.00291876s
list : 0.00322025s
* Elapsed time for 2048 particles:
vector : 0.00350553s
deque : 0.00190853s
list : 0.00293957s
* Elapsed time for 1024 particles:
vector : 0.00164309s
deque : 0.00133216s
list : 0.00168998s
* Elapsed time for 128 particles:
vector : 0.000265469s
deque : 0.000144511s
list : 0.000208552s
* Elapsed time for 64 particles:
vector : 0.000123876s
deque : 0.000110709s
list : 0.00011727s

On remarque alors qu'à partir de 128 particules, la structure de donnée ayant le meilleur temps d'exécution est la structure **deque**.

# 2 Algorithme de Strömer-Verlet

Une Première implementation est fait dans le deuxième TP, pour 2 dimensions. un teste sur l'ensemble des particules : Soleil, Terre, Jupiter et Haley a été effectuer, et grace a un progamme python **grav.py**, on a pu visualiser les resultats de deplacements au cours du temps.
La figure suivante represente le resultats obtenue:

Et Pour diviser le temps d'exécution en deux, l'approche était: au lieu de mettre a jour la force de chaque particule toute seule , on bénéficie de la troisième loi de **Newton** qui montre que :

$$F_{ij} = -F_{ji}$$

Et alors on met a jour mutuellement les deux particules.

Autre proposition pour réduire la complexité est de définir un rayon d'influence pour prendre en compte juste les particules existante dans ce rayon et négliger les autres.

Figures:

```
$ ./test
```

# 3 Univers de particules

## 3.1 architecture

Dans cette étape nous avons implémenter notre classe **Vecteur** tester de façon correcte avec l'exécutable **testVecteur**

Ainsi que notre classe **Univers** qui généralise nos implémentions précédentes, et on a pu revalider les résultats de la section précédente a l'aide le l'exécutable **testStromer**

## 3.2 performance

Pour l'insertion des éléments, on a remarquer qu'elle se fait facilement même pour un $k > 10$.

Or cela n'est pas le cas pour les interactions, celles ci commence a prendre beaucoup de temps a partir de $k = 5$.

Et même après le recours a notre solution implémenter précédemment de division de temps de calcul par deux, le calcul reste assez lourd, et alors il faut une optimisation plus efficace, qui va être présenter par la suite.

## 3.3 Résultats

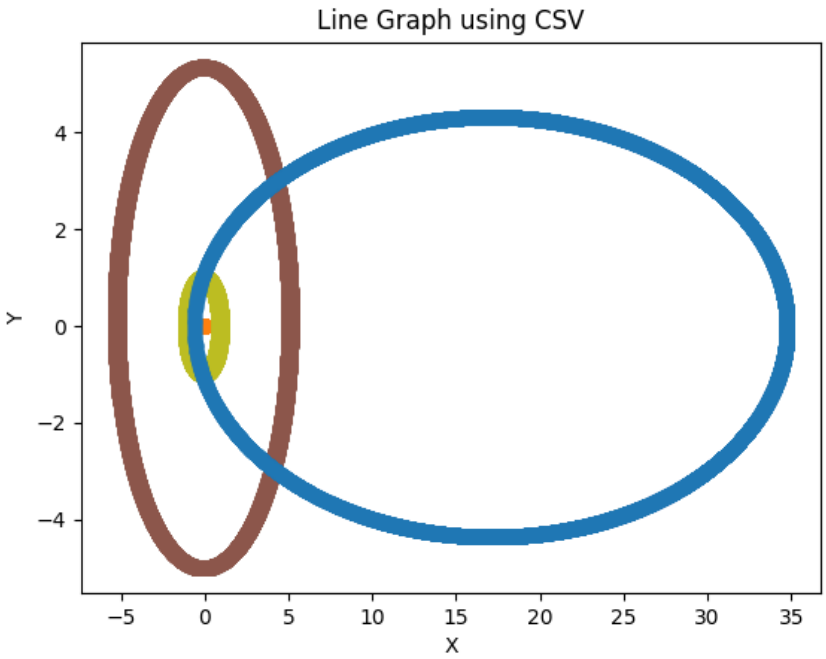


Figure 1: Système Solaire

# 4 Découpage de l'espace

## 4.1 Architecture

Afin d'améliorer la performance de notre programme nous allons découper notre espace en un ensemble de cellule.

Pour ce fait on crée d'abord un **Univers** qui contient maintenant des **cellules** qui en elle même contient des **particules** et on associe a chaque **Cellule** l'ensemble de ses voisins.

Et de même que pour le potentiel gravitationnel, nous avons suivi la même démarche pour calculer les forces de Lennard-Jones.

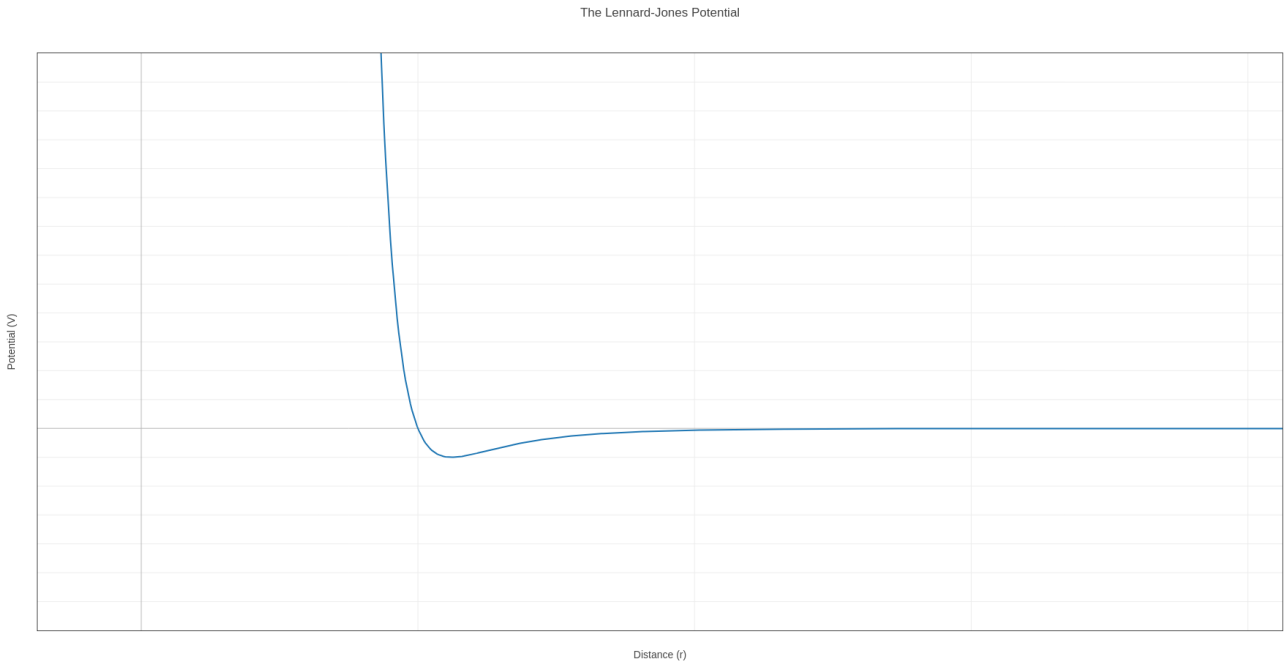


Figure 2: potentiel de Lennard-Jones.

cette figure montre l'utilite de cette approche d'optimisation, puisque le potentiel de Lennard-Jones **s'annule** a partie d'un certain $r_c$

Pour mettre a jour la position de chaque particule dans l'algorithme de **Stromer-Verlet**, on parcours l'ensemble des particules de l'univers puis on essaie d'insérer cette particule dans la cellule correcte et la supprimer de l'ancienne.

## 4.2 performance

La complexité de notre algorithme peut encore s'améliorer surtout au niveau de l'insertion de particule qui est avec une complexité de $O(n)$, une idée est de conserver un ordre pour les cellules telle que l'indice correspondant d'une cellule dans l'univers peut être obtenu a partir de la position d'une particule de cette cellule et alors passer a une complexité $O(1)$.

Nous avons décider de changer le paramètre $L_2 = 40$ donne en sujet a $L_2 = 140$, pour visualiser les résultats et le pas $dt = 0.05$ car sinon le programme devient assez lent a exécuter.

## 4.3 Résultats

Le résultat afficher en cette figure est celui pour $t = 10$.

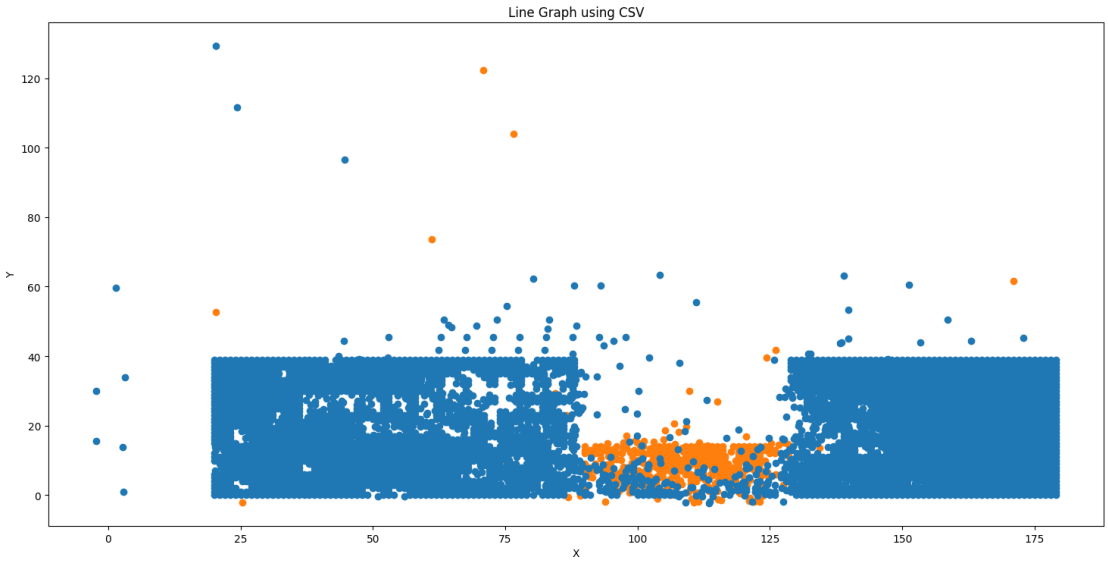


Figure 3: collusion entre deux objets.

# 5 Test et visualisation

## 5.1 Implémentation des tests à l'aide de l'outil gtest

On a mis en place une base de tests des deux classes principales qui sont la classe **Vecteur** et la classe **Particle** à l'aide de l'outil **gtest**, les tests comportent l'utilisation des différents constructeurs et opérateurs. Le code des tests est implémenté dans le fichier **gtest.cc** dans le répertoire **/test**.

## 5.2 Résultats de Paraview

On visualise les résultats de collision entre deux objets à l'aide de paraview, en utilisant les conditions initiales et les paramètres suivants :

$$\begin{array}{llll} L_1 & = 250, & L_2 & = 40, \\ \varepsilon & = 5, & \sigma & = 1, \\ m & = 1, & v & = (0, 10), \\ N_1 & = 1600, & N_2 & = 6400, \\ r_{cut} & = 2.5\sigma, & \delta t & = 0.00005 \end{array}$$

Figure 4: paramètres.

avec un temps de calcul $t_{max} = 19.5$.

La fonction responsable du calcul des vitesses et des positions est la fonction **evolution_vis** de la class Univers, elle est implémentée dans le fichier **univers.cxx**.

La fonction utilise les méthodes suivantes:
La méthode **compute_forces** de la classe Univers qui calcul les forces appliquées sur les particules.
La méthode **advance_position** de la classe Particle qui calcul les nouvelles positions des particules.
La méthode **advance_vitesse** de la classe Particle qui calcul les nouvelles

vitesses des particules.
La méthode visualisation qui génère les fichiers ayant les valeurs des positions, vitesses et masses à chaque instant afin de faire la visualisation par paraview.

On visualise les résultats suivants:

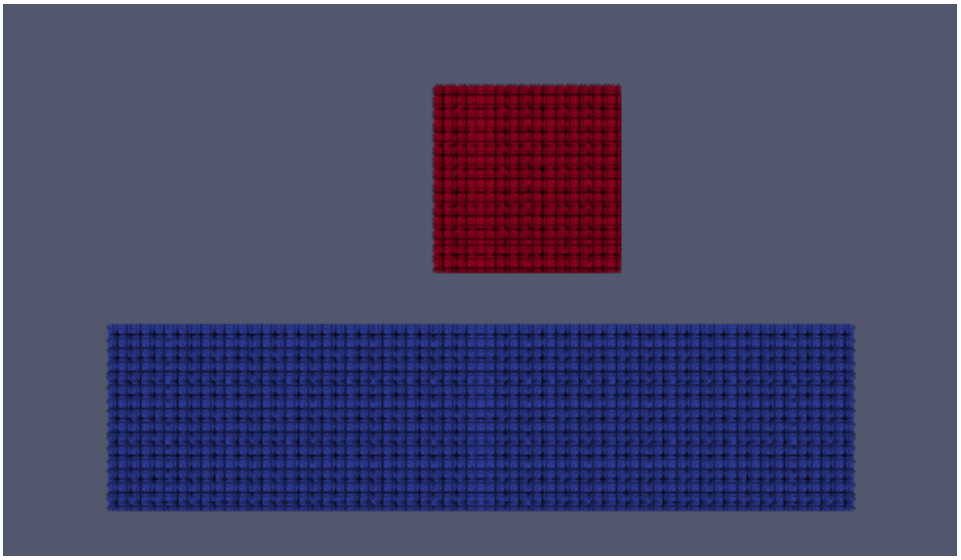

Figure 5: collusion entre deux objets $t = 0$.

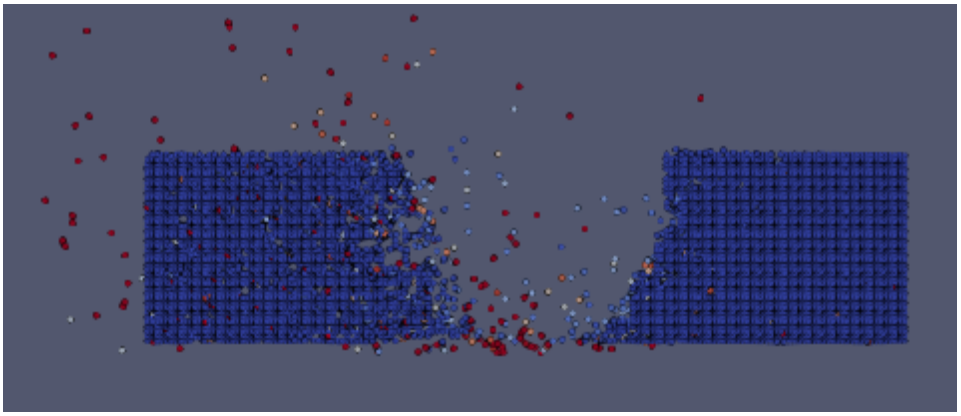

Figure 6: collusion entre deux objets$t = 19.5$.

On remarque que L'objet cubique écrase une partie de l'objet parallélépipédique et se détruit au cours de la collision.

# 6 ACVL

## 6.1 Diagramme des cas d'utilisations

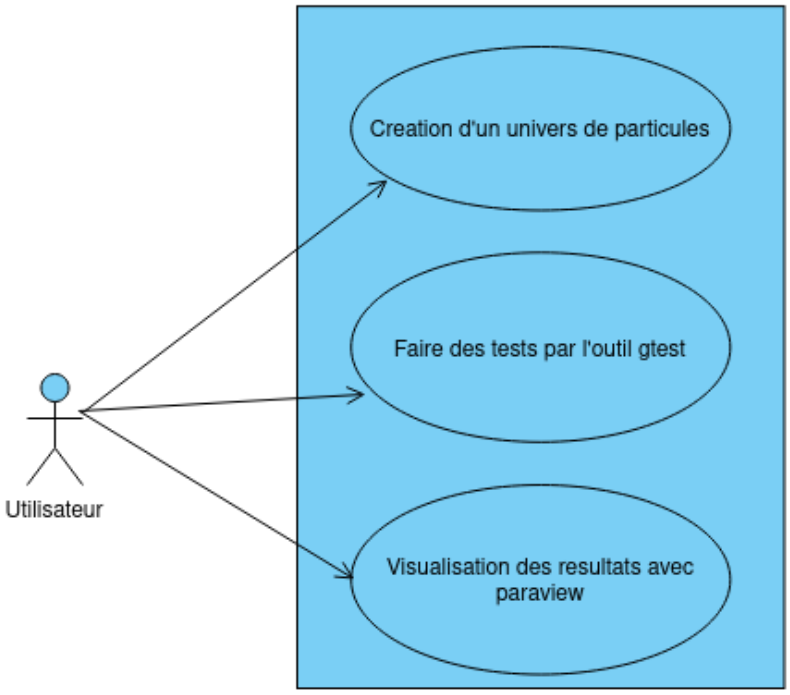


Figure 7: cas d'utilisations.

## 6.2 Diagramme de séquence

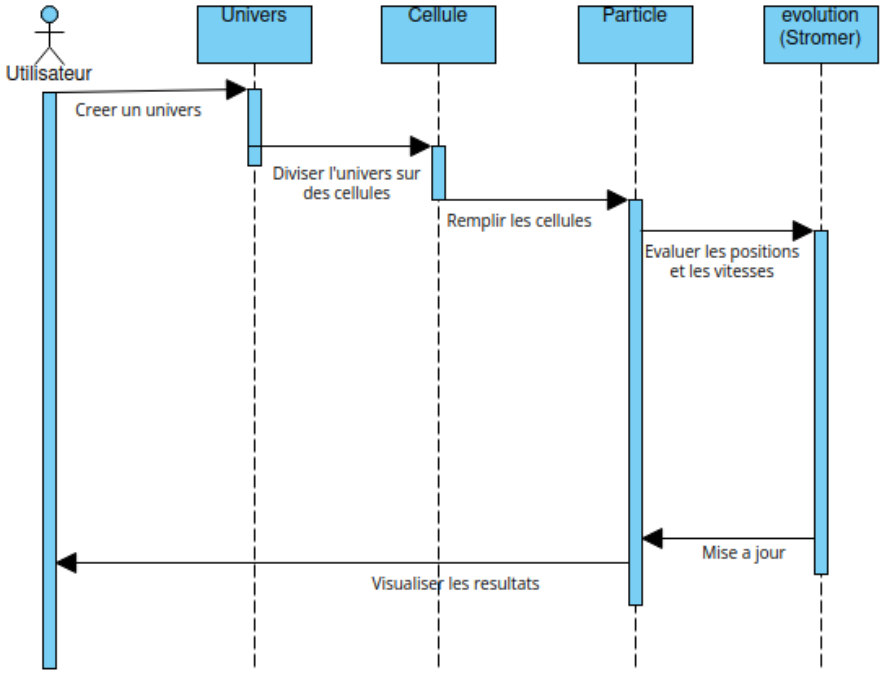


Figure 8: collusion entre deux objets$t = 19.5$.

## 6.3 Diagramme de transitions

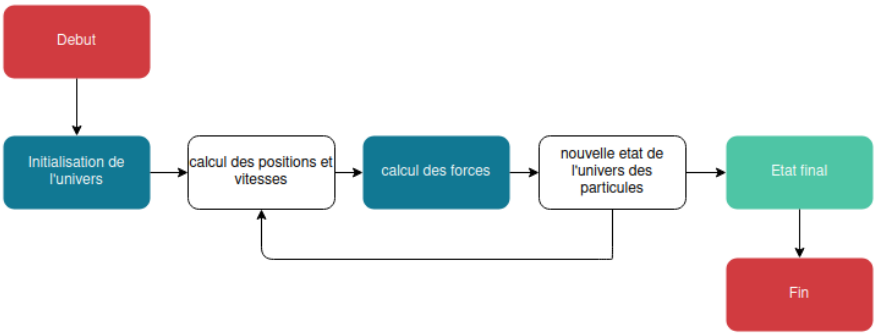


Figure 9: collusion entre deux objets$t = 19.5$.

## 6.4 Diagramme de classes d'analyse

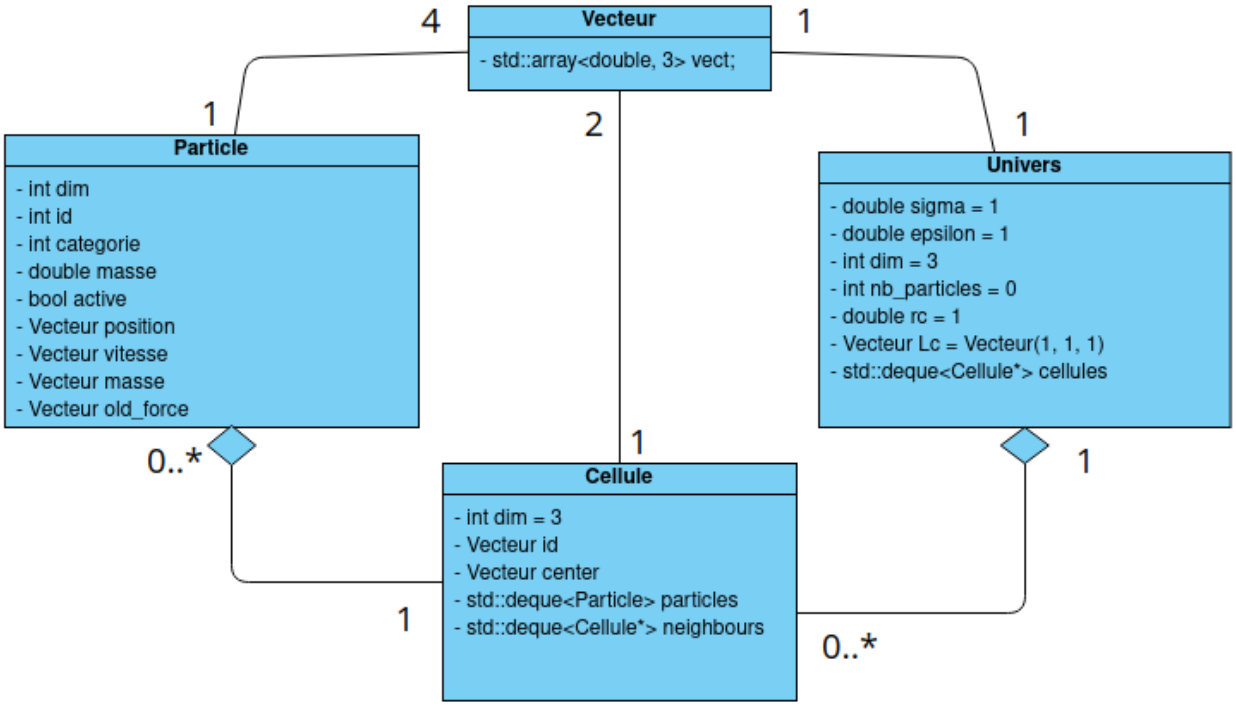


Figure 10: collusion entre deux objets$t = 19.5$.

# 7 Raffinement du modèle

## 7.1 Conditions aux limites

**Réflexion:** La fonction qui permet de d'appliquer une réflexion sur les particules après collision avec les parois est la fonction réflexion qui est implémentée dans le fichier **univers.cxx**.

**Absorption :** La fonction qui permet de faire disparaître les particules qui dépassent les parois est la fonction absorption qui est implémentée dans le fichier **univers.cxx**.

**Périodique :** La fonction qui permet de donner une position périodique aux particules qui dépassent les parois est la fonction périodicité qui est implémentée dans le fichier **univers.cxx**.

## 7.2 Conditions aux limites réflexives avec potentiel

La fonction qui permet de calculer le potentiel aux bords de l'univers est la fonction **reflexion_potentiel** qui est implémentée dans le fichier **particule.cxx**. Cette fonction retourne un vecteur de force qui doit être ajouté à l'attribut **force** de la classe Particle.

## 7.3 Potentiel gravitationnel

La fonction qui permet de calculer le potentiel gravitationnel des particules est la fonction **gravitational_potentiel** qui est implémentée dans le fichier **particule.cxx**. Cette fonction retourne un double qui doit être ajouté à la composante horizontale de l'attribut force de la classe Particle.

## 7.4 Application : collision de deux objets

La fonction qui permet la modification des vitesses par le calcul de l'énergie cinétique est la fonction **energieCinetique** qui est implémentée dans le fichier **univers.cxx**