Calcul Scientifique en C++ - 2023

Chapter 1 Prérequis

Prérequis

Avant de se lancer dans la lecture du projet, on vous invitera par la suite à vérifier avant de lancer cmake .. et make que :
- doxygen est bien installé : `sudo apt-get install doxygen`
- SFML est bien installé : `sudo apt-get install libsfml-dev`

Nota Bene

Vis-à-vis du rendu intermédiaire :
- peur de ”casser” tout notre code pavec les paramètres en références constante. Cependant on a changé tout le reste qu'il nous manquait :
- respect des consignes. warning: non-void function does not return a value in all control paths [-Wreturn-type] - nom de la bibliothèque générée - analyse et changement de nore indentation - les include ne sont plus relatifs et sont gérés par les CMakeLists.txt - documentation avec doxygen (se trouve dans le dossier GENERATE$_HTML$)

Chapter 2 - Introduction

Objectif de ce lab

Le but de ce Lab est de se familiariser avec les bases d'un environnement C++ pour le calcul scientifique : on utilisera toutes les méthodes que l'on va voir par la suite (débogage de code lldb, profiling de code, etc...)

2.1 Débogage

2.1.0.1 Question 1 et 2

Nous devons identifier les problèmes rencontrés dans la code qui nous est donnée : trace.cxx.
Il nous suffit d'executer notre fichier trace à l'aide du terminal, et nous pouvons voir toutes les erreurs du code. Voici une liste des erreurs les plus importantes qui nous ont amenés à les corriger (nous ne prenons pas en compte ici les erreurs classiques de code : oublie de ;, problème d'indentation, etc...).

- Rajout de using namespace std;.
- Dans ce code, nous avons casté le pointeur retourné par calloc en un pointeur de type double** en utilisant (double**), et chaque élément du tableau matrix est casté en un pointeur de type double* en utilisant (double*).
- On renvoie le pointeur, et non l'adresse du pointeur (ça n'a pas un réel intérêt ici)
- La trace est un réel et non un pointeur, renvoie un double et non un ptr : useless ici.

Pour plus d'informations, on vous invite à voir le nouveau code mis en place dans le fichier trace.cxx.

2.1.0.2 Question 3

On effectue ici un profiling de code, on retrouvera le profiling en entier dans le fichier : XQuestionsTests/Q3analyse.txt.
On a en résumé le profiling suivant :

% time	Cumulative seconds	Self seconds	Calls	ms/call ms/call	Name
50.94	0.27	0.27	1	270.00	print_matrix(double**, int)
39.62	0.48	0.21	10000	0.02	fill_vectors(double*, int)
9.43	0.53	0.05			_init
0.00	0.53	0.00	1	0.00	initialization(int)
0.00	0.53	0.00	1	0.00	__static_initialization_and_destruction_0(int, int)
0.00	0.53	0.00	1	0.00	trace(double**, int)

On remarque ici qu'il va falloir optimiser notre code, on appelle 10000 fois fillvector pour remplir notre matrice, cela pose un réel problème plus la taille de notre matrice sera elevée.

2.1.0.3 Question 4

Au vu des résultats précédents, il nous est venu de repenser entièrement la logique du code qui nous a été fournis. Pour aller plus vite, on pense alors à effectuer les modifications suivantes :

- Enlever le print
- Manipuler autrement le fill vector car il y a dans l'exemple précédent un total de 10000 calloc

Nous avons alors optée pour la solution qui est la suivante : nous allons initialiser une matrice, de telle sorte que ce soit seulement un pointeur en "1D", et non un pointeur de pointeur qui est en "2D",

Optimisation matricielle

Cette méthode de manipulation de mémoire est beaucoup plus pratique et rapide, car il ne faut pas aller à n adresses différentes, ici on bougera juste le pointeur comme il le faut, on passe d'un vecteur "2D" à un vecteur "1D". Cependant, il faut effectuer des manipulations mathématiques pour effectuer des conversions d'indices en "2D" pour avoir nos indices voulus pour le pointeur en "1D".

Après changement de code et optimisation de celui-ci d'après ce qui précède, tout d'abord nous obtenons aucune fuite mémoire, et comparé au profiling effectué dans la question précédente, nous remarquons que notre code est bien plus optimisé :

`Valgrind :` à retrouver dans XQuestionsTests/Q4valgrind.txt

Enter the dimension of a square matrix: 2000
The sum of the diagonal elements is: -1292
Heap Summary: - Memory in use at exit: 0 bytes in 0 blocks - Total heap usage: 5 allocations, 5 deallocations, 32,084,736 bytes allocated
All heap blocks were freed – no memory leaks are possible.
Error Summary: - No errors detected.

`Profiling de code :` à retrouver dans XQuestionsTests/Q4analyse.txt

% time	Cumulative seconds	Self seconds	Calls	ms/call ms/call	Name
100.00	0.11	0.11	1	110.00	fill_vectors(double*, int)
0.00	0.11	0.00	1	0.00	initialization(int)
0.00	0.11	0.00	1	0.00	__static_initialization_and_destruction_0(int, int)
0.00	0.11	0.00	1	0.00	trace(double*, int)

Table 2.1: Statistiques d'exécution

2.2 Résolution d'EDO

Méthodes numériques et mathématiques

Nous allons à chaque fois parler d'une manière rapide et concise des méthodes mathématiques que nous allons mettre en place. Mais en résumé, comme nous sommes en dimension 1, les méthodes utilisés utilisent des approximations de la dérivée.

On rappellera pour la suite que :

La méthode d'Euler est une méthode numérique élémentaire de résolution d'équations différentielles du premier ordre, de la forme

$$\forall x \in I, \quad u'(x) = f(x, u(x))$$

où I est un intervalle de $\mathbb{R}$ et f une fonction réelle sur $I \times \mathbb{R}$.

Étant donné une condition initiale $(a, u(a)) \in I \times \mathbb{R}$, la méthode fournit pour tout point $b \in I$ une suite $(u_n(b))_{n\in\mathbb{N}}$ d'approximations de la valeur $u(b)$ que prend, lorsqu'elle existe, la solution de l'équation qui correspond à cette condition initiale. Divers jeux de conditions sur f peuvent assurer la convergence de cette suite.

La valeur $u_n(b)$ s'obtient en calculant n valeurs intermédiaires $(y_i)_{i\in\{0,n\}}$ de la solution approchée aux points $(x_i)_{i\in\{0,n\}}$ régulièrement répartis entre a et b, donnés par :

$$x_i = a + i \cdot \frac{b-a}{n}$$

Nota Bene

Pour vérifier le bon fonctionnement et l'optimisation de nos codes qui suivent pour résoudre des problèmes d'EDO, nous nous invitons à regarder dans XQuestionsTests les fichiers texte donnant les informations sur le profiling et valgrind associés.

2.2.1 Euler Explicite

On effectue alors la méthode suivante :

En étendant notre notation à $x_0 = a$, $y_0 = u(a)$ et $x_n = b$, $y_n = u_n(b)$, et en utilisant l'approximation de la dérivée :

$$u'(x_i) \approx \frac{u(x_{i+1}) - u(x_i)}{x_{i+1} - x_i}$$

On en déduit la relation suivante :

$$\frac{y_{i+1} - y_i}{x_{i+1} - x_i} = f(x_i, y_i)$$

Les valeurs intermédiaires sont alors données par la relation de récurrence :

$$y_{i+1} = y_i + (x_{i+1} - x_i) \cdot f(x_i, y_i), \quad i \in \{0, n-1\}$$

qui est le schéma d'Euler explicite.

```
for (int i = 1; i <= iteration; i++) {
    liste_y[i] = liste_y[i-1] + (dist/iteration) * f(x_after, liste_y[i-1]);
    x_after += dist/iteration;
}
```

Listing 2.1: Boucle d'itération

On retrouvera la suite du code dans le fichier src/euler explicite.cxx et la démonstration du calcul de la fonction dans le code demo/demo euler explicite.cxx

Optimisation

Pour le calcul des valeurs de la fonction f, on utilise ici et pour la suite le mot clé `inline` pour optimiser notre code.

2.2.2 Euler Implicite

En remarquant que l'on peut aussi approcher la dérivée en x_{i+1} par la même relation :

$$u'(x_{i+1}) \approx \frac{u(x_{i+1}) - u(x_i)}{x_{i+1} - x_i}$$

on en déduit la relation de récurrence :

$$y_{i+1} = y_i + (x_{i+1} - x_i) \cdot f(x_{i+1}, y_{i+1}), \quad i \in \{0, n-1\}$$

qui est le schéma d'Euler implicite.

On notera que dans ce schéma, le terme y_{i+1} apparaît des deux côtés de l'équation, ce qui contraint à utiliser des méthodes de résolution numérique du type de la méthode de Newton-Raphson pour déterminer y_{i+1} à chaque itération si la fonction f est non linéaire. C'est pourquoi nous avons mis en place une fonction newton implicite pour pouvoir mettre en place notre euler implicite.

On retrouvera l'ensemble du code dans le fichier src/euler implicite.cxx et la démonstration du calcul de la fonction dans le code demo/demo euler implicite.cxx

2.2.3 Autres EDO

On utilise les questions précédentes, et on compare nos résultats entre les deux méthodes d'Euler utilisé.

On remarque après analyse que les résultats donnés par la méthode d'Euler implicite sont plus précis que ceux obtenus que par la méthode d'Euler explicite. Mais, la méthode d'Euler implicite demande plus de temps d'exécution que la méthode d'Euler explicite.

Le choix de chacune des deux méthodes dépend alors du cahier des charges que l'on a a respecter.

Piste d'amélioration

Pour encore plus optimiser le code, on pourrait penser à créer notre propre fonction abs avec inline pour peut-être encore plus gagner en rapidité.

Chapter 3 - Particules

Objectif de ce lab

Le but de ce Lab est de nous familiariser avec les notions de classe, de pointeur, de référence en C++ . Il s'agira également de mettre en place les premiers éléments permettant notre simulation à grandes échelles de systèmes particulaires complexes.

3.1 Analyse de performance

L'analyse de performance est très importante, celle-ci nous permettra de mesurer les performances de notre implémentation. Le Lab1 précédent nous a permis de manipuler gprof comme il le fallait et d'avoir une première vision sur son utilisation.

3.2 Mise en place des structures de données

3.2.0.1 Question 1

Pour ce faire, nous avons décidé en plus de notre classe `Particule` de rajouter une classe `Vecteur`. Cette classe `Vecteur` va nous permettre de gérer la position, la vitesse et la force : celles-ci étant caractérisées par des vecteurs en 3 dimensions.

Pour avoir plus d'informations, on vous laisse aller voir dans include/Particule.hpp.

Pourquoi 2D et pas 3D

Nous avons décidé par la suite de tout faire en 2D pour notre Univers, nous expliquerons les changements à effectuer pour passer en 3D par la suite (vector 3D conversion en vector 1D, conversion indice 3D en 1D etc...). On retrouvera toutes les fonctions importantes pour la suite dans le Lab4.

Nota Bene

Pour le Lab2 et Lab3, notre première compréhension du sujet nous avait amené à rajouter une classe `Force` mais finalement juste utiliser une classe `Vecteur` nous est suffisant.

3.2.0.2 Question 2

Piste de réflexion

Avant de créer notre collection de particules, nous devons décider quel type de structure de données nous allons devoir utiliser pour toute la suite de notre projet. Nous avons ici le choix entre set/list/dequeu/vector.

Voici les informations que nous avons pu trouver et que nous avons classés dans un tableau pour faciliter la lecture de celui-ci :

Collection spécifique	Complexité de temps
`std::vector`	
Accès à un élément par indice	O(1)
Insertion/suppression à la fin	O(1)
Insertion/suppression au début ou au milieu	O(n)
Recherche d'un élément	O(n) (O(log n) si trié)
`std::deque`	
Accès à un élément par indice	O(1)
Insertion/suppression à la fin ou au début	O(1)
Insertion/suppression au milieu	O(n)
Recherche d'un élément	O(n)
`std::list`	
Accès à un élément par indice	O(n)
Insertion/suppression à n'importe quel endroit	O(1)
Recherche d'un élément	O(n)
`std::set`	
Accès à un élément par indice	Non applicable
Insertion/suppression	O(log n)
Recherche d'un élément	O(log n)

Table 3.1: Complexités de temps des opérations sur des conteneurs STL

A partir de la lecture de cette Table 2.1, et de la documentation donnée dans l'énoncé du Lab2,on remarque alors que l'on a ici le choix entre deux collections spécifiques : `std::vector` et `std::deque` .

Il faut maintenant faire des recherches supplémentaires pour pouvoir trouver lequel serait le meilleur entre les deux pour notre projet.

Voici les avantages et inconvénients de notre futur choix de garder `std::vector` .

Utiliser un vecteur dans ce contexte a plusieurs avantages et inconvénients. Voici quelques-uns d'entre eux

Avantages

- **Accès rapide aux éléments** : les vecteurs permettent un accès direct à n'importe quel élément à n'importe quel endroit, ce qui rend l'accès rapide aux données très efficace. C'est l'un des principaux avantages par rapport à d'autres structures de données comme on a pu le voir dans la Table 2.1 pour les listes et les set.

- **Flexibilité** : les vecteurs sont dynamiques, ce qui signifie que l'on peut facilement ajouter ou supprimer des éléments.
- **Facilité d'utilisation** : les vecteurs sont très faciles à utiliser et à comprendre, de plus de nombreuses bibliothèques et fonctions en C++ sont conçues pour fonctionner de manière optimale avec les vecteurs.

Inconvénients

- **Ajout/suppression d'éléments au début** : les opérations d'ajout et de suppression au début du vecteur peuvent être coûteuses en termes de temps et de performances. C'est parce que toutes les autres valeurs doivent être déplacées pour faire de la place pour la nouvelle valeur ou pour combler l'espace laissé par la valeur supprimée.
- **Espace de mémoire** : les vecteurs consomment plus d'espace mémoire par rapport à d'autres structures de données. C'est parce qu'ils réservent plus de mémoire qu'ils n'en ont besoin pour éviter les nouvelles allocations lors de l'ajout d'éléments. Cependant, nous allons palier à ce problème en réservant qu'une seule fois de la mémoire dès le début, car on connait le nombre de particules que l'on possède. (on ne créera pas un vecteur vide que l'on va remplir au fur et a mesure que l'on rajoute des particules.)
- **Pas adapté pour les listes liées** : Les éléments ne peuvent pas être facilement insérés ou supprimés au milieu, les vecteurs ne sont pas le meilleur choix dans ce cas là. Pour de telles utilisations, des structures de données comme les listes ou les deques seraient plus appropriées. Cependant, au vu du Lab6, nous n'allons pas supprimer toutes nos particules, ce qui ne nous pose dont aucun soucis, ou ce sera négligeable au vu des autres calculs à effectuer.

Choix final : std::vector

Malgré les problèmes d'insertions evoqués, qui seront finalement négligeable par la suite, std::vector est un bon choix ici pour stocker notre collection de particules en raison de ses propriétés avantageuses. Comme on a pu le voir, son accès direct à n'importe quel élément en temps constant O(1), sera optimal dans notre cas lorsque l'on manipulera chacune de nos particules. De plus, on rajoutera que celui-ci est également très efficace en termes d'utilisation de la mémoire, car il stocke les éléments dans un bloc de mémoire contigu, optimisant ainsi les performances de cache.

Nous pouvons maintenant nous atteler à la création de notre collection de particules.

Voici le code correspondant que l'on retrouvera dans src/collection particules.cpp, pour aller plus loin, on vous invite directement à regarder le Lab3, celui-ci donnant un code plus remplie et en adéquation avec le projet :

```
1  std::vector<Particule> collect_part(int nombre_particules) {
2      std::vector<Particule> collect(nombre_particules);
3
4      // pour la position et la vitesse
5      std::random_device rd;
6      std::mt19937 mt(rd());
7      std::uniform_real_distribution<double> doub(0.0, 1000000.0 + nombre_particules);
8
9      // pour la masse, l'identifiant et la categorie
10     for (int i = 0; i < nombre_particules; i++) {
11         collect[i] = Particule(doub(mt), doub(mt), doub(mt), doub(mt), doub(mt), doub(mt),
               doub(mt), (int)doub(mt), (int)doub(mt));
```

```
12      }
13
14      return collect;
15  }
```

3.2.0.3 Question 3

Nous pouvons maintenant utiliser la bibliothèque `chrono` pour pouvoir regarder les performances entre chacune de nos collections. A l'aide de la fonction particule chrono dans src/collection particules.cpp, on obtient les résultats suivants :

Table 3.2: Temps d'exécution en fonction du nombre de particules

0.5—X—X—

Nombre de particules	Temps d'exécution (s)
2	6.2996e-05
4	3.6596e-05
8	4.1415e-05
16	3.3803e-05
32	2.3466e-05
64	3.8553e-05
128	6.8025e-05
256	0.000125644
512	0.000240392
1024	0.000471985
2048	0.000982453
4096	0.00192517
8192	0.00384733
16384	0.00766072
32768	0.0153907
65536	0.0309322
131072	0.0621859
262144	0.124111
524288	0.248935
1048576	0.497611
2097152	0.994771
4194304	1.98699
8388608	3.97015
16777216	7.94847

On remarque ici d'après les résultats obtenus, que la création de notre collection de particules avec notre std::vector semble avoir une complexité temporelle de O(n). Ce qui est bien cohérent avec notre étude faite précedemment.

Complexité

A chaque fois que l'on multiplie par deux le nombre de particules dans notre collection, on remarque que le temps d'exécution est lui aussi multiplié par deux, ce qui est bien caractéristique d'une complexité O(n).

Nota Bene

On aurait pu faire la même chose pour chacune des collections spécifiques, mais l'étude faites précédemment et les résultats obtenus nous confirment bien notre choix pour la suite.

3.3 Méthode de Störmer-Verlet

Explication rapide de la méthode

La méthode consiste à effectuer des calculs tous les δ_t secondes, ce qui rend le code plutôt coûteux, il va falloir donc maintenant falloir faire encore plus attention aux manipulations que l'on peut effectuer nos codes.

3.3.0.1 Question 4

Nous vous invitons à regarder notre première version de Störmer-Verlet, on remarquera dans les Lab qui suivent que l'on a pu l'optimiser en changeant nos classes, en rajoutant dans `Particule` un vecteur Force et oldForce, ce qui nous permet de ne pas utiliser deux std::vector pour stocker les Forces et OldForces.

Pour avoir la version la plus élégante et optimisée de celles-ci, nous vous invitons à regarder la simulation effectuée dans le Lab4, il faudra alors se rendre dans : src/Univers.cpp simulation

---

3.3.0.2 Question 5

On remarque tout d'abord que nous n'avons aucune erreur vis-à-vis de valgrind. Maintenant nous devons vérifier que nos planètes effectuent un bon trajet, il nous faut alors effectuer une simulation :

3.3.0.3 Question 6

Pour modifier la classe `Particule` afin de protéger les données en écriture, nous sommes amener à mettre en `private` .

L'objectif ici est de restreindre l'accès direct aux attributs de la classe et de fournir des méthodes d'accès pour lire et modifier ces attributs. Pour ce faire nous allons définir les attributs de la classe `Particule` en tant que variables privées. Cela empêchera donc l'accès direct à ces attributs en dehors de la classe! De plus, pour chaque attribut privé nous allons ajouter une méthode publique de type `getter` pour lire la valeur de l'attribut et une méthode publique de type `setter` pour modifier la valeur de l'attribut.

Chapter 4 - Utilisation des opérateurs

Objectif de ce lab

Le but de ce Lab est de remplacer la structure de donnée sous-jacente par une classe propre ayant des propriétés arithmétiques, ce qui permettra de faciliter l'écriture et la lecture du code par la suite.

4.1 Enrichissement des structures de données

4.1.1 Création d'une classe vecteur

4.1.1.1 Question 1 et 2

Voici la classe Vecteur que l'on va garder pour toute la suite, ici on se place en dimension 2, on expliquera par la suite pourquoi on a fait ce choix là.

```
1  class Vecteur{
2
3  public:
4
5      // Initialisation
6      Vecteur(double x = 0, double y = 0) : m_x(x), m_y(y){}
7
8      // Constructeur par copie
9      Vecteur(Vecteur const& autre): m_x(autre.m_x), m_y(autre.m_y){}
10
11     // Oprateurs arithmtiques
12
13     // Addition
14     Vecteur& operator+=(Vecteur const& vec);
15
16     // Soustraction
17     Vecteur& operator-=(Vecteur const& vec);
18
19     // Multiplication par un scalaire
20     Vecteur& operator*=(double scalaire);
21
22     // Division par un scalaire
23     Vecteur& operator/=(double scalaire);
24
25     // Produit scalaire
26     double operator*(Vecteur const& vec);
27
28     // Produit vectoriel
29     Vecteur& operator^=(Vecteur const& vec);
30
31     // Norme
```

```
32      double norme() const;
33
34      // Mthodes utiles
35      double getX() const {return m_x;}
36      double getY() const {return m_y;}
37
38      void setX(double x){m_x = x;}
39      void setY(double y){m_y = y;}
40
41      // Pour effectuer les vrifications
42      void afficheVecteur() const;
43
44  protected:
45      double m_x;
46      double m_y;
47  };
48
49  // Oprateurs externes
50  Vecteur operator+(Vecteur const& vec1, Vecteur const& vec2);
51  Vecteur operator-(Vecteur const& vec1, Vecteur const& vec2);
52  Vecteur operator*(double scalaire, Vecteur vec);
53  Vecteur operator^(Vecteur const& vec1, Vecteur const& vec2);
```

4.1.1.2 Question 3

Il suffit de lancer demo vecteur pour vérifier que tous les tests marchent bien.

4.1.2 Modification de la classe particule

4.1.2.1 Question 4

On retrouvera toutes les informations correspondantes dans la classe `Particule` dans le code du Lab4, cependant, on affirme dès maintenant que les attributs que l'on a gardé sont les suivants :

```
1  protected:
2
3      Vecteur m_position;
4      Vecteur m_vitesse;
5      double m_masse;
6      Vecteur m_force;
7      Vecteur m_oldForce;
8      int m_indiceCell;
9      int m_id;
```

4.1.3 Univers des particules

4.1.3.1 Question 5

La classe `Univers` à observer se trouve dans le Lab4, celle-ci sera la version la plus épurée. On remarque ici que notre classe `Univers` a juste besoin des attributs suivants, et pour effectuer la méthode d'évolution, nous allons ré-utiliser la méthode de Störmer-Verlet. Comme les particules seront uniformes et initialisés de manière aléatoire dans tout l'espace, les attributs de notre classe `Univers` seront seulement :

```
protected:

    int m_nbPart;
    std::vector<Particule> m_collPart;
```

4.1.3.2 Question 6

Nota Bene

La répartition aléatoire de nos particules se fera ici dans notre constructeur de notre Univers.

```
Univers(int nombre_particules = 0, double vitesse = 0) : m_nbPart(nombre_particules),
    m_collPart(nombre_particules){
        //pour la position et la vitesse
        std::random_device rd;
        std::mt19937 mt(rd());
        std::uniform_real_distribution<double> doub(0.0, 1.0);

        for (int i = 0; i < nombre_particules; i++){
            //Suppose toutes les categories et identifiants diffrents!
            m_collPart[i] = Particule(doub(mt), doub(mt), doub(mt), vitesse, vitesse,
                vitesse, doub(mt), i, i);
        }
    }
```

Il nous sufft maintenant de juste choisir le nombre de particules que l'on recherche.

4.1.3.3 Question 7

Pour ce faire, on utilise de nouveau la bibliothèque `chrono`, on trouve alors pour effectuer la création de notre Univers composé de 32768 particules : elapsed time: 0.0419258. Ce qui est extrêment faible! Cependant, comparé à la question 3 de 2.2.0.3, on remarque que l'utilisation d'une classe univers à travers notre constructeur multiplie par 4 ici l'elapsed time qui était de 32768 0.0153907 (test fait sur la même machine, normalement avec le même profil de puissance d'ordinateur).

4.1.3.4 Question 8

Pour tester l'implémentation dans le calcul des interactions, il nous suffit de regarder le temps que prend à effectuer une itération totale dans notre simulation, c'est à dire lors d'un temps δ_t, pour ce faire nous allons effectuer une moyenne du temps pris pour plusieurs δ_t.

k	Temps d'exécution moyen (en secondes) pour un passage en δ_t
3	0.0194553
4	1.22671
5	78.4181

Table 4.1: Temps d'exécution moyen pour différentes valeurs de k

On se rend alors compte que le fait d'effectuer sur un total de n particules des opérations rend pour notre problème le tout très coûteux.

Explications

A chaque passage, une particule dois effectuer n-1 correspondances avec le reste des particules, c'est à dire on aura un total de n(n-1) correspondances, on peut alors au vu des calculs élementaires effectués que notre simulation pour chaque itération δ_t sera en O(n^2). Alors, imagineons que l'on a un pas très bas et que l'on effectue plus de n iterations (ce qui sera très souvent le cas, en réalité même plus de n^2 itérations), on se retrouve à minima avec un code en O(n^3) ce qui est beaucoup trop coûteux. Nous allons devoir optimiser notre code pour la suite.

4.1.3.5 Question 9

On peut découper notre univers en deux ensemble d'indice, on pourrait alors imaginer que notre espace ce découpe en deux, à chaque itération on cherchera quelles sont les particules (représentées par leur indice dans le std::vector) sont dans la partie gauche ou dans la partie droite de notre univers, alors les particules à gauche n'agiront que sur les particules de gauche, et de même les particules à droite n'agiront que les particules de droite, ce qui permettra donc en moyenne comme les particules sont réparties de manières uniformes de diviser le temps de calcul par 2.

Nota Bene

Cette modification très simple pose problème sur les bords de découpe de notre espace, cependant cette méthode reste une méthode non optimisée et naïve au vu de notre problème, mais cela correspond à la modification permettant de diviser le temps de calcul par 2!

Nota Bene

En réalité diviser le temps de calcul par 2 pour une complexité en $O(n), O(n^2), O(n^3)$, ... ne change en réalité rien, on garde les mêmes complexités, nous allons donc voir dans la Question 10 comment nous pouvons être amenés à optimiser notre temps de calcul d'une manière plus significative vis-à-vis de notre problème.

4.1.3.6 Question 10

Nous devons alors réfléchir ici à des simplifications qui auront un impact significatif sur le temps d'execution de notre code, voici les principales idées que nous allons garder :

- On remarque que la force d'interaction est en $\frac{1}{r^2}$, on remarque alors que plus la distance entre deux particules sera grande, plus la force d'interaction sera négligeable, on pourrait alors déterminer des indices de particules à garder pour effectuer les calculs, on regardera alors quelles sont les particules les plus proches que l'on gardera pour effectuer nos calculs de force.
- On peut découper notre univers en cellules avec une certaine taille spécifique et dont on regardera les particules dans notre cellule et celles voisines, on fera en sorte que la taille spécifique soit liée à la distance à partir de la quelle il nous semble judicieux de négliger les interactions.
- On pourrait mettre en place des méthodes de parallélisation des calculs, cependant ce cours correspond au cours suivant : GP-GPU and High Performances Computing ENSIMAG 3A - MMIS — M2 SIAM — M2 MoSiG, cela permettra d'effectuer les calculs encore plus rapidement.

Chapter 5 - Découpage de l'espace

Objectif de ce lab

Le but de ce Lab va être d'optimiser notre Univers, nous allons mettre en place un maillage et des méthodes de calculs optimisés.

5.0.0.1 Question 1

Voici le potentiel de Lennard-Jones pour un système à deux particules en fonction de la distance r :

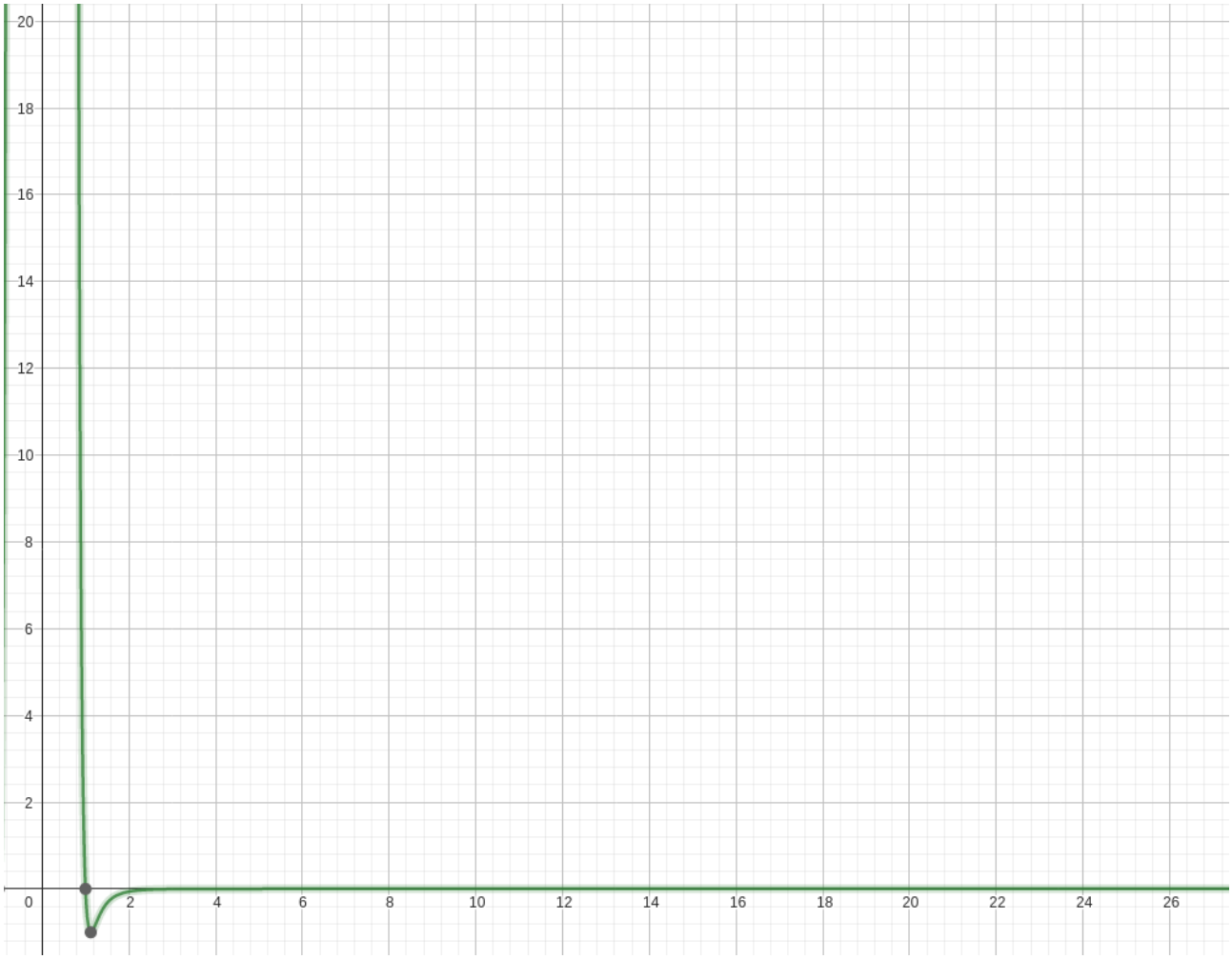

Figure 5.1: Potentiel en fonction de la distance r

Potentiel d'interaction négligé

On remarque alors que l'on peut négliger notre potentiel pour le cas où $r_{cut} > 2$.

Explications

Nous allons maintenant expliquer notre choix de seulement considérer le cas en dimension 2 : (x, y) et allons expliquer la logique de structuration des données et du code que nous aurions adopté pour le cas de la dimension 3 : (x, y, z)

On remarque alors que nous allons devoir manipuler un univers avec des cellules, on serait alors tenter d'utiliser des vecteurs de vecteurs de vecteurs de cellules en dimensions 3 ([][][]) ou des vecteurs de vecteurs de cellules en dimesions 2 ([][])). Mais pour le projet C de première année, et comme on a pu faire dans l'introduction, on sait que le fait d'utiliser des vecteurs de vecteurs peut entraîner une fragmentation de la mémoire et des accès mémoire moins efficaces (se référer à comment cela fonctionne en terme d'allocation mémoire avec les adresses).

C'est pourquoi, nous avons décidé d'effectuer comme on peut le dire depuis le début des conversions 3D vers 1D et des conversions 2D vers 1D, ce qui permettra d'optimiser encore plus notre code.

Au vu des conversions qu'il faut faire pour effectuer ces méthodes, nous avons décidé de nous concentrer sur le problème en 2 dimensions, et non en 3 dimensions, mais cela revient à la même chose, juste avec plus de conversions à effectuer, ce qui rendrait le travail redondant ici.

Mais nous allons quand même expliquer comment bien gérer la répartition spatiale de la 3ème dimension vers une dimension en 1D.

Tout d'abord, concentrons-nous sur la méthode que nous allons effectuer pour la 2ème dimension.

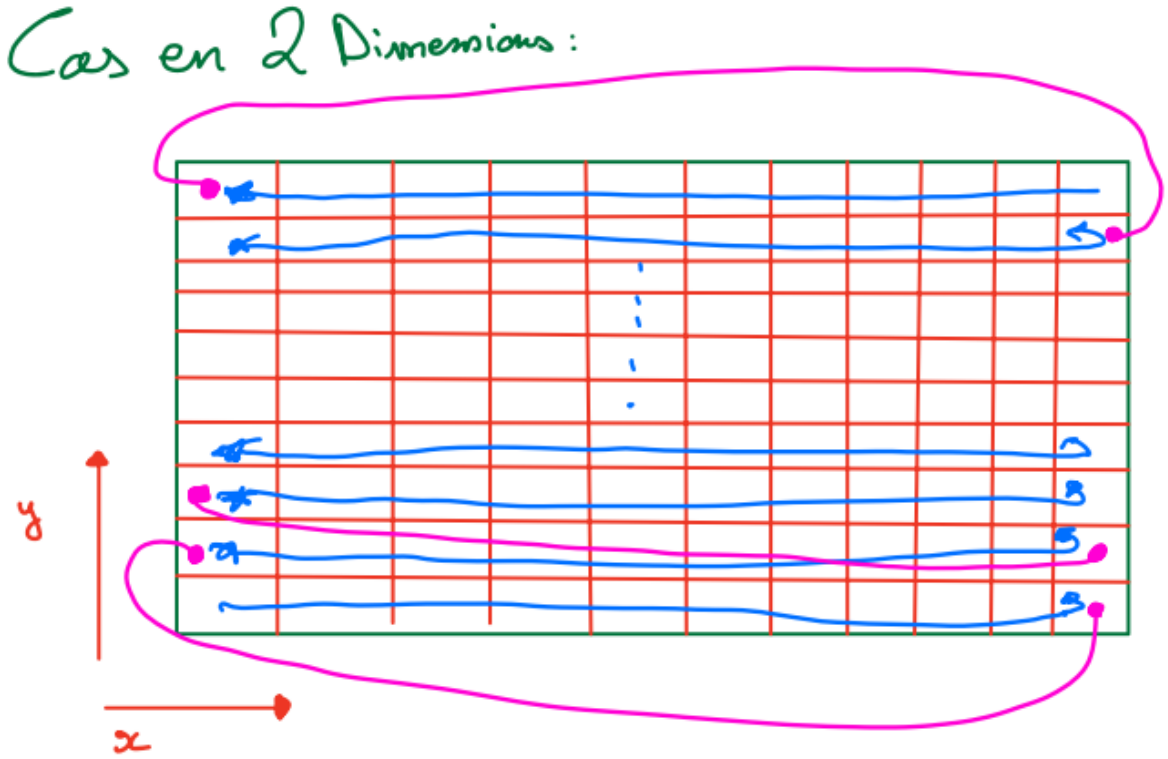

Figure 5.2: Méthode passage 2D to 1D

Pour effectuer la conversion d'indice, nous devons comprendre que lors du passage d'une ligne à une autre, qui aurait du correspondre dans le cas 2D on aurait seulement du passer de [0][] à [1][], cependant ici nous avons un vecteur seulement en 1 dimension. Pour ce faire on se réfère alors au schéma pour comprendre comment nous allons découper nos cellules en 2D en 1D. On remarque lors du passage d'une ligne à une autre cela revient à prendre le nombre de case de la première ligne, puis pour se déplacer sur cette ligne, il suffit de garder le nombre de case de la première ligne puis additionner le nombre de case sur lequel on cherche à se déplacer

Conversions explications

On remarque alors dans ce cas qu'on a : [1][5] correspond à : [nombre de colonnes * 1 + 5].
Plus précisement on utilisera cette méthode de conversion d'indice : int index = i * (longueur / ncLong) + j où longueur/ncLong correspond au nombre de colonnes;

Maintenant que l'on a pu comprendre le fonction en 2D, il suffit d'appliquer la même méthode en 3D mais en comprennant comment cela se passe lors du passage d'un étage à un autre (coordonnées selon z).

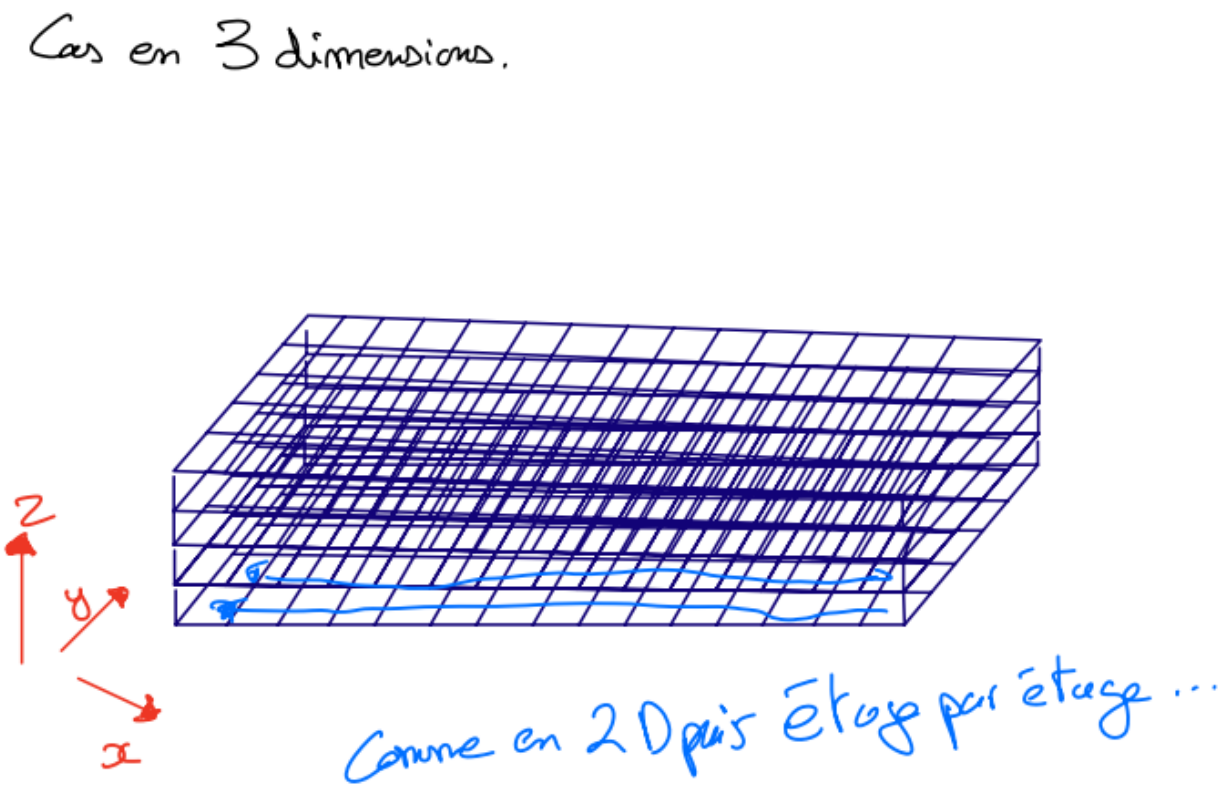

Figure 5.3: Méthode passage 3D to 1D

Si l'on prend le nombre de case d'un étage, si l'on passe à un autre étage, cela correspond à prendre ce nombre où l'on additionne la logique voulue comme si on était en 2D, ainsi l'étage correspondant à z peut se gérer facilement, puis ensuite lorsque l'on se trouve au bon étage on utilise la même méthode que pour la 2D.

Nota Bene

Cette méthode peut se généraliser à tout type de vecteurs de vecteurs de ..., cependant on remarque que la mise en place de celle-ci peut devenir de plus en plus compliqué, et de plus demande une bonne compréhension géométrique.
Pour éviter tout problème que l'on pourrait rencontrer, on effectue tout le travail qui suit en 2D comme on a pu le montrer le travail en 3D se généralise ensuite très rapidement par la suite en effectuant les conversions d'indices correspondants.

5.1 Création d'un Univers

5.1.0.1 Questions 2 et 3

Avant de parler de notre classe `Univers`, il est très important de comprendre tous les changements que l'on a pu effectué au niveau de notre classe `Particule` mais aussi il va falloir parler du rajout qui nous semblait très important d'une classe `Cellule`.

Voici les attributs des classes principales `Particule`, `Cellule` et `Univers`.

Pour la classe `Particule` :

```
protected:

    Vecteur m_position;
    Vecteur m_vitesse;
    double m_masse;
    Vecteur m_force;
    Vecteur m_oldForce;
    int m_indiceCell;
    int m_id;
```

On connaîtra alors l'indice de la cellule où se trouve la particule avec `m indiceCell`, mais aussi l'`id` qui correspond à l'indice dans le std::vector de `Particule`. dans la classe `Cellule`.

Pour la classe `Cellule` :

```
    protected:

    // Identificateur dans le vecteur maillage
    int m_identiVecteur;
    // Position de la cellule dans l'espace (i,j)
    int m_i;
    int m_j;
    // Indices Particules dans la Cellule
    std::vector<int> m_indPart;
    // Savoir la coordonne du centre de la cellule
    Vecteur m_coordonnees;
    // "Coordonnes" indice vecteur 1D des cellules proches :=)
    std::vector<int> m_indiceProche;
```

De la même manière, `m identiVecteur` correspond à l'indice dans le std::vector de `Cellule` dans la classe `Univers`.

Nota Bene

Dans la version v0 que nous avons mis en place, nous avons utilisé des pointeurs et références au lieu d'utiliser des indices. Nous avons décidé de manipuler des indices comme ici ce qui nous a permis d'éviter les segmentations faults, mais aussi d'optimiser le temps d'exécution de notre code.

Passons maintenant à la classe `Univers` :

```
protected:

    // Particules
    std::vector<Particule> m_collParticules;

    // Maillage du problme
    std::vector<Cellule> m_maillage;

    // Caractristiques de l'espace!
    double m_longueur; // dfinit selon l'abscisse
    double m_largeur; // dfinit selon l'ordonne
    double m_rCut;
    int m_ncLong;
    int m_ncLarg;

    // Caractstique  associ    l'espace
    double m_eps;
    double m_sig;
```

De plus, pour initialiser l'Univers, on utilisera alors la fonction : `initialisationCellules()` .

Univers

A l'aide des classes mises en place, nous pouvons dorénavant mettre en place les fonctions qui nous permettront d'effectuer les simulations à venir.

5.2 Calcul des potentiels

5.2.0.1 Question 4

Pour limiter le parcours des particules nous allons alors suivre la logique suivante :

- On se place sur une `Particule` .
- On récupère `m indiceCell` qui nous donne des informations sur la `Cellule` où se trouve la `Particule` .
- Connaissant la `Cellule` , on peut avoir accès aux indices proches grâce à `std::vector¡int¿ m indiceProche` et ensuite on calcul la distance entre la `Particule` et chacune des `Cellule` proches. Selon la distance, on gardera ou non l'indice des `Cellule` .
- On a donc accès aux indices des `Cellule` proches que l'on garde, on calcule alors les forces qui s'appliquent entre notre `Particule` et toutes les autres `Particules` se trouvant dans les `Cellule` proches que l'on a gardé.

Nota Bene

On retrouvera l'utilisation de cette méthode, dans la fonction `forceTot(Particule particule)` .

5.2.0.2 Question 5

La fonction correspondante est `simulation(double t end, double dt)` . Elle utilise toute les fonctions mises à dispositions dans src/Univers.cpp. Cependant, lors de la simulation nous avons eu un problème qui nous a pris beaucoup de temps à résoudre. En réalité même si dans le Lab6 nous avons besoin de mettre en place une gestion des problèmes aux bords pour gérer les conditions aux limites, il faut dès maintenant gérer les `Particule` qui s'évadent de notre `Univers` et qui ainsi créent des problèmes de Segmentation Fault.

Tout d'abord, notre première réflexion avait été qu'à chaque fois que l'on calcule les nouvelles positions de chacun de nos points, on étudie celles-ci et si elles ne se trouvent plus dans la délimitation de notre `Univers` alors on les supprime, on effectuait les manipulations suivantes :

- Regarde si la particule est ou non dans l'Univers
- Si la particule est toujours dedans : on ne fait rien
- Si la particule n'est plus dedans : on récupère son indice que l'on met dans un std::vector
- On supprime ensuite toutes les particules dont l'indice est dans le std::vector
- On remet à jour les particules en leurs donnant un nouvel ID
- On remet à jour les indices des particules dans les cellules

On remarque alors que cette méthode est beaucoup trop coûteuse, et on a décidé d'optimiser celles-ci en rajoutant un attribut dans notre classe `Particule` qui est : `m in` qui correspond à un booléen disant si oui ou non la particule est toujours ou non dans notre `Univers` .

Optimisation

Une simple utilisation d'un if pour effectuer des vérifications :if (verifPosition(part) part.getIn()) permettra de changer le bool à false et d'enlever l'ID de la `Particule` de la `Cellule` . Toutes les `Particule` en false sont de plus supprimées des indices particules des `Cellule` où elles étaient, ce qui simplifie encore plus notre code comme on regarde à chaque itération chacun de nos `Cellule` et non pas chacune de nos `Particule` !

Optimisation

On se rend alors compte de la simplicité et de la rapidité de notre système à travers la mise en place d'un système d'indice pour toutes les manipulations, au lieu d'utiliser des références et pointeurs qui nous auraient complexifier à la fois le code mais aussi augmenté notre temps d'exécution.

5.3 Application : collision de deux objets

5.3.0.1 Question 6

Tout d'abord, le point notable est que nous n'avons aucune segmentation fault. Ce qui est très important pour nous : la simulation se termine bien.

Nous verrons la simulation à travers un affichage dans le Lab qui suit.

On remarque que l'on prend pour un dt de 0.00005s et $t = 19.5$s qu'un passage pour effectuer tous les calculs (pour cette grande itération) nous prend dans le cas de cette simulation :

Paramètre	Valeur
L_1	250
L_2	40
ϵ	5
σ	1
m	1
v	(0, 10)
N_1	1600
N_2	6400
r_{cut}	2.5σ
δt	0.00005

Table 5.1: Paramètres de la simulation

On prend au maximum 0.003 secondes par itération, et on descend à 0.002 secondes au cours du temps (normal car des particules s'échappent de notre univers).

Nota Bene : petit test pour t = 19.5s dt = 0.00005s on a alors :
- elapsed time: 883.911
- elapsed time: 855.134

Conclusion partielle

Finalement, toutes les manières d'optimisations que l'on a pu mettre en place semblent être très concluantes!

Chapter 6 - Test et visualisation

Objectif de ce lab

Le but de ce Lab va être de mettre en place une infrastructre de test pour les classes et méthodes du projet. Mais aussi de mettre en place une infrastructure de sauvegarde de données dans un format standardiser. Cela permettra par la suite d'afficher notre simulation.

6.1 Mise en place de tests

6.1.1 Concept de base

6.1.2 Assertion

6.1.3 Tests

6.1.4 Utilisation de CMAKE

6.1.4.1 Question 1 et 2

On a mis en place notre infrastructure de test, on obtient les résultats suivants :

```
[----------] 6 tests from ParticuleDistance (0 ms total)
[----------] 6 tests from ParticuleCalculeForcePartielle (0 ms total)
[----------] 24 tests from CelluleTest (0 ms total)
[----------] 11 tests from UniversTest (0 ms total)
[----------] Global test environment tear-down
[==========] 47 tests from 4 test suites ran. (0 ms total)
[  PASSED  ] 47 tests.
```

Pour avoir plus d'informations, on vous laisse aller dans test/ et lancer ./GoogleTest

Test

En plus de cela, pour avoir une visualisation de ce qu'il se passe, on vous invite aussi à exécuter testCellule.

6.2 Visualisation

6.2.0.1 Question 3

Nous devons ici intégrer dans notre projet une méthode pour enregistrer les données des particules. Nous avons donc fait différentes recherches.

Reflexion

Notre première piste de reflexion a été de penser qui sont ceux qui optimisent le plus les visualisations pour que celles-ci soient le plus agréable à regarder pour l'utilisateur? Nous nous sommes donc penché sur des réflexions en rapport avec les jeux-vidéos.

Nous avons donc décidé d'utiliser la librairie : SFML .
On obtient pour notre simulation où les particules sont réparties uniformément dans l'espace et dont les vitesses sont aléatoires entre 0 et 1 :

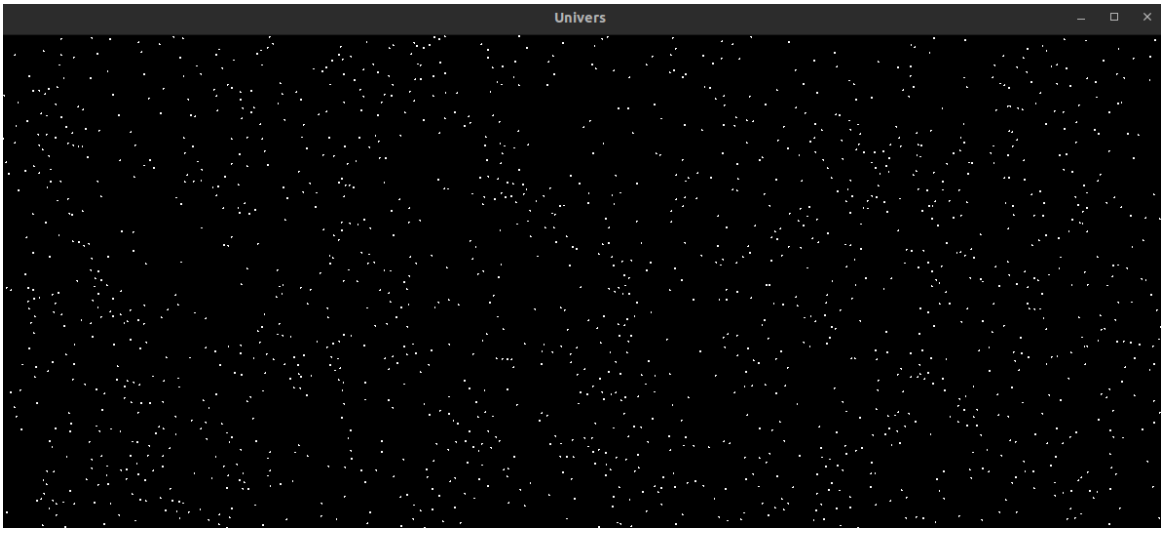

Figure 6.1: Initialisation de l'espace dans le cas d'Absorption

Figure 6.2: Après un certains temps t dans le cas d'Absorption

On remarque alors le bon fonctionnement de notre code, et de plus au vu de la rapidité d'execution de celui-ci on peut alors être content de l'optimisation que l'on a pu fournir tout au cours du projet.

Pour pouvoir regarder la simulation on vous invite à aller dans build : d'écrire dans le terminale cmake.. puis make, d'ensuite vous rendre dans demo et d'executer ./demo univers. Pour avoir plus de choix sur la longueur, nombre de particules, etc... on vous invite à executer ./demo universChoix cependant il faut faire attention au choix des paramètres, sinon il peut y avoir un problème en terme d'espace et donc d'indicage au

vu de la création de notre code (nous supposons dans notre code que l'Univers est bien défini et que toutes les particules sont bien dans l'Univers (pour éviter des seg fault à cause des conversions d'indicage))/

6.3 ACVL

6.3.0.1 Question 4

Le diagramme des cas d'"utilisation illuste les cas d'utilisation d'un seul acteur qui est l'utilisateur qui pourra definir les nouveaux paramètres de la simulation et qui s'occupera de la gestion de cette dernière: lancer et arrêter la simulation.

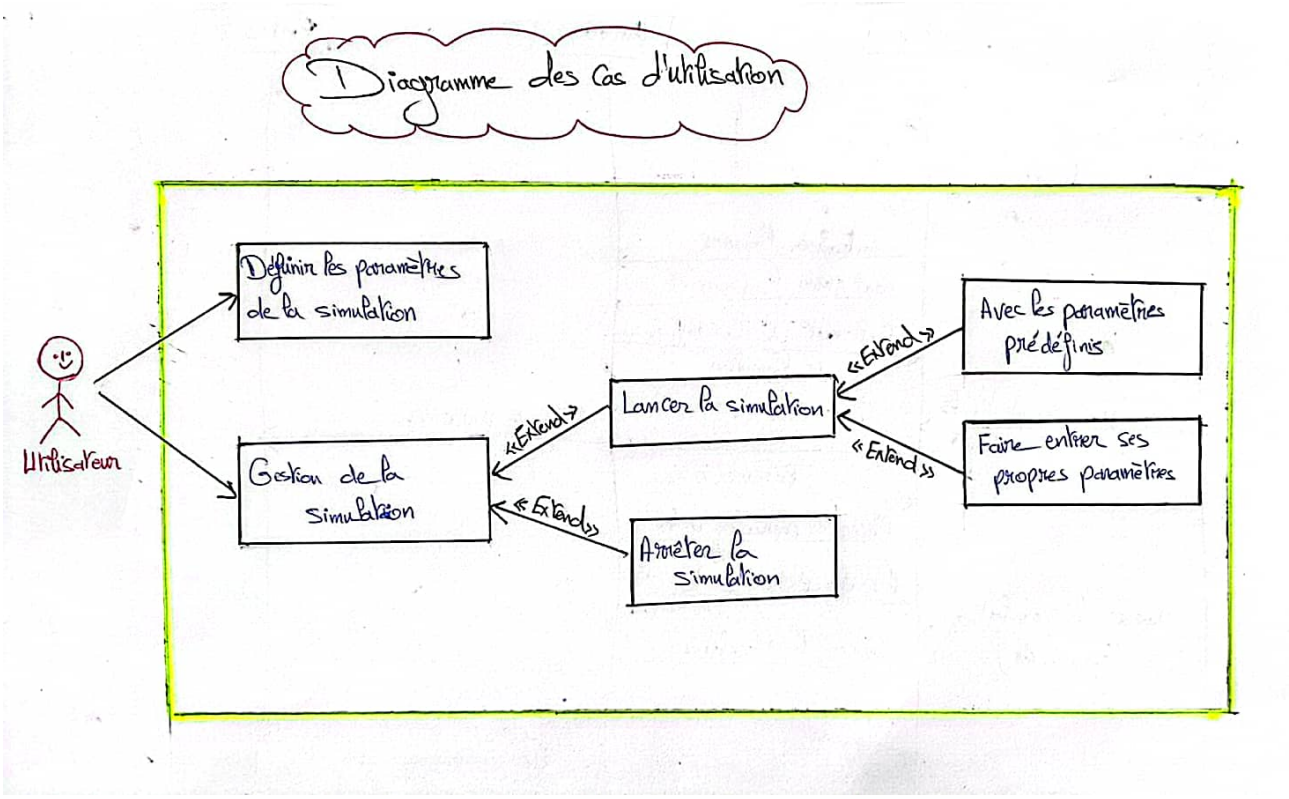

Figure 6.3: Diagramme des cas d'utilisation

6.3.0.2 Question 5

Le diagramme ci-dessous illustre le diagramme de séquence du cas d'utilisation suivant : la configuration de l'univers et des particules, puis le déroulement du lancement de la simulation. On peut regrouper l'Univers, les particules et les Cellules en une seule partie appelée "données du système".

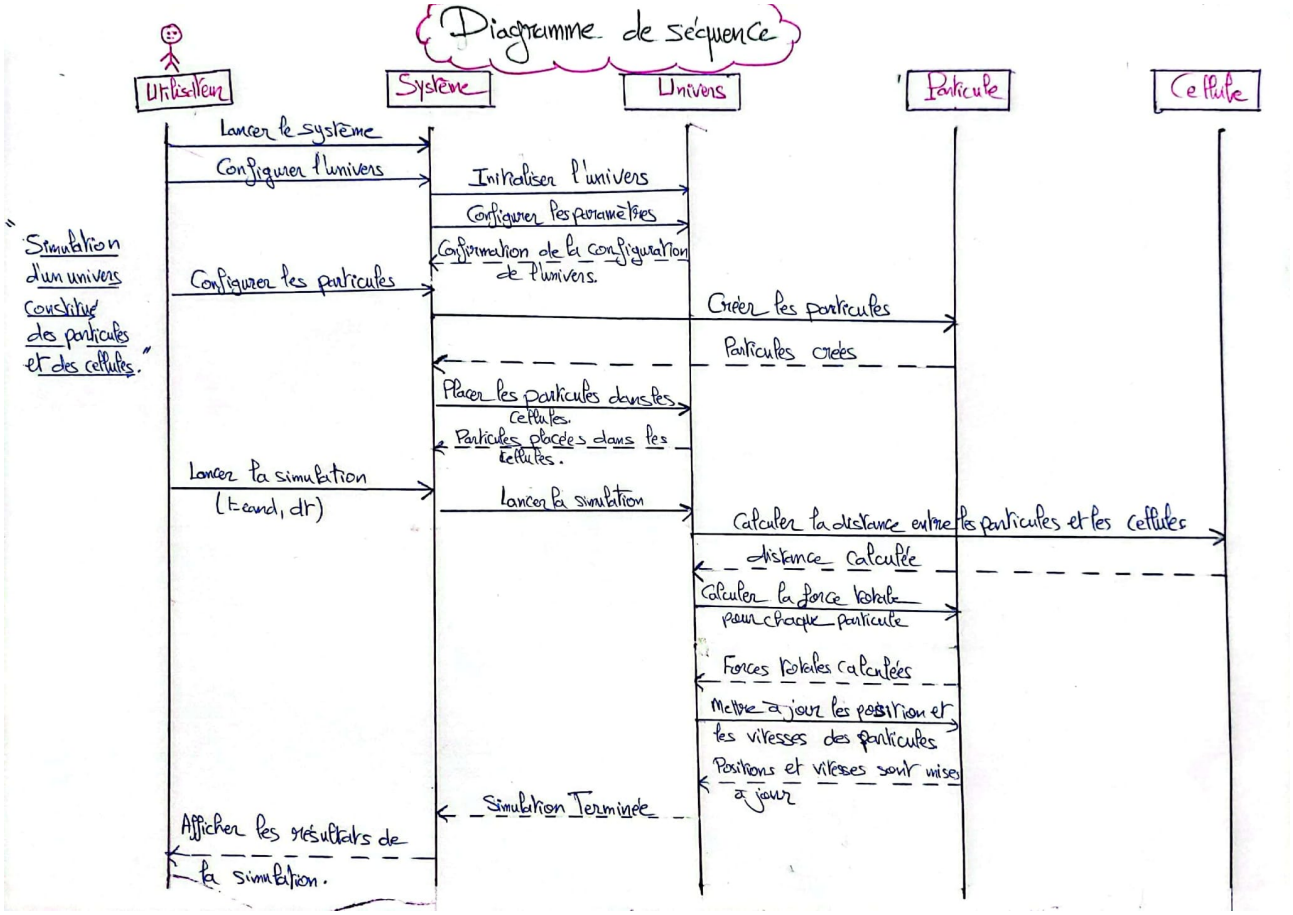

Figure 6.4: Simulation d'un univers constitué des particules et des cellules.

Pré-conditions:

* Les dimensions de l'univers (m longueur et m largeur) doivent être correctement définies.

* La particule doit être valide et avoir une position définie.

* Les cellules du maillage (m maillage) doivent être correctement initialisées.

Post-conditions:

* Vérifie et met à jour la cellule associée à chaque particule si nécessaire en effectuant un changement de cellule (changementCell).

* Les particules et les cellules ont été mises à jour selon les règles spécifiées.

* La simulation de l'univers est terminée.

6.3.0.3 Question 6

Dans ce diagramme ci-dessous, nous illustrons le diagramme d'états et de transitions de notre système de la partie lancement de la simulation.

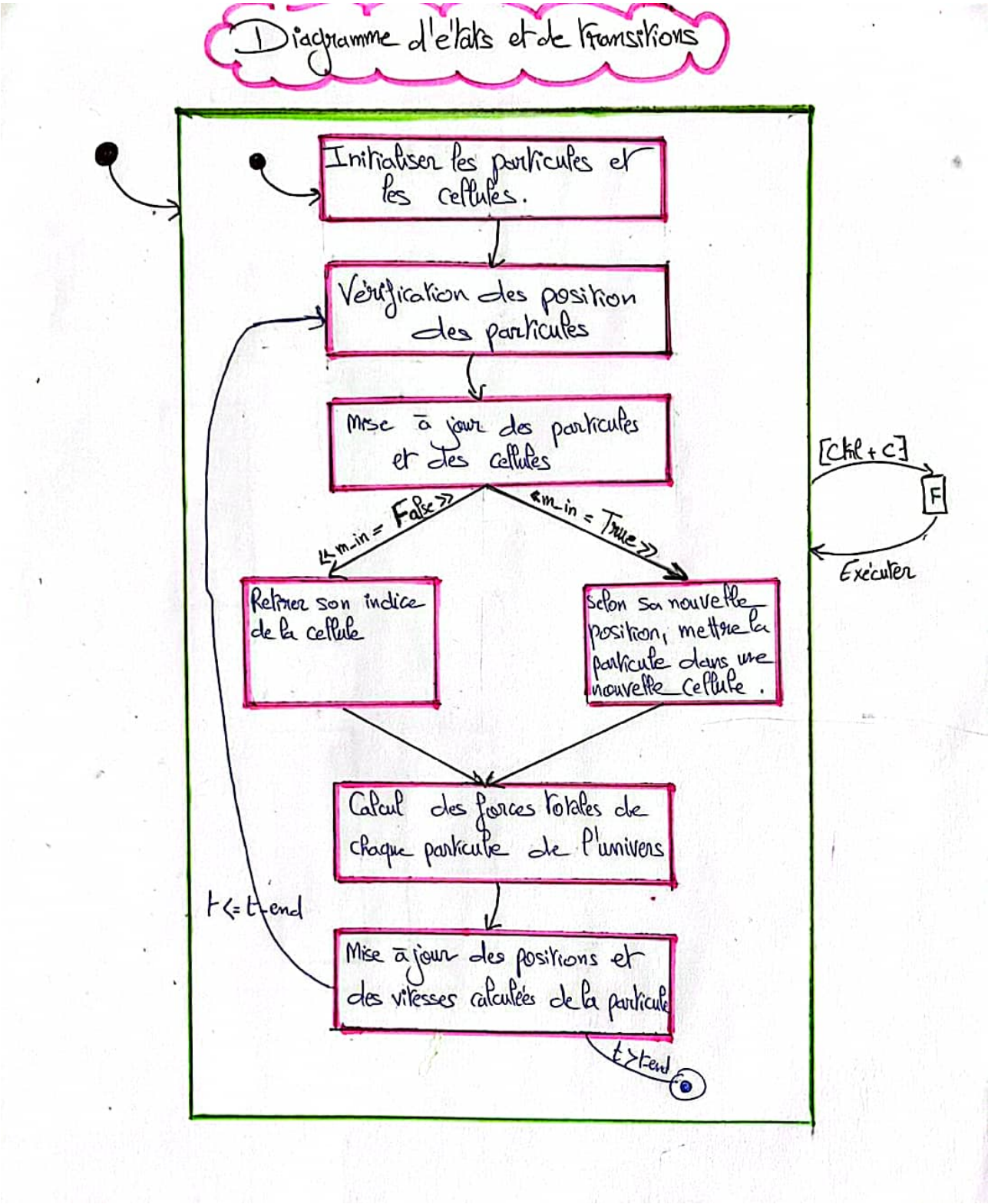

Figure 6.5: Diagramme d'états et de transitions.

Description:

État initial : Initialiser les particules et les cellules

- L'utilisateur initialise les particules et l'univers.
- Passage à l'état suivant : Vérification des positions des particules

État 1 : Vérification des positions des particules

- Le système vérifie les positions des particules et détecte les collisions ou les conditions spécifiques.

- Si [m in = False], passage à l'état suivant : Retirer l'indice de la cellule

- Sinon, passage à l'état suivant : Mettre la particule dans une nouvelle cellule

État 2 : Retirer l'indice de la cellule

- Le système retire l'indice de la cellule actuelle associée à la particule.

- Passage à l'état suivant : Calcul des forces totales de chaque particule de l'univers

État 3 : Mettre la particule dans une nouvelle cellule

- Le système place la particule dans une nouvelle cellule appropriée en fonction de sa position mise à jour.

- Passage à l'état suivant : Calcul des forces totales de chaque particule de l'univers

État 4 : Calcul des forces totales de chaque particule de l'univers

- Le système calcule les forces totales agissant sur chaque particule en prenant en compte les interactions avec les autres particules et l'environnement.

- Passage à l'état suivant : Mise à jour des positions et des vitesses calculées de la particule

État 5 : Mise à jour des positions et des vitesses calculées de la particule

- Le système met à jour les positions et les vitesses des particules en utilisant les forces calculées précédemment.

- Le système vérifie si le temps est supérieur à tend.

- Si oui, passage à l'état final : Fin de l'exécution

- Sinon, retour à l'état : Vérification des positions des particules

État final : Fin de l'exécution

- L'exécution du programme est terminée.

État F : Interruption de l'exécution (via CTRL+C)

- Passage de l'état courant à l'état initial : Initialiser les particules et les cellules

---

6.3.0.4 Question 7

Figure 6.6: Diagramme de classes d'analyse

Chapter 7 - Raffinement du modèle

Objectif de ce lab

Le but de ce Lab va être d'implémenter des conditions aux limites et de mettre en place des configurations différentes de notre modèle.

Nota Bene

En réalité nous avons pris beaucoup de temps à mettre en place tout le Lab4, on y a aussi rajouté la partie Absorption, les autres conditions aux bords sont en réalité assez rapide à mettre en place, mais nous avons préféré passer du temps à effectuer un bon rendu et avoir une bonne hierarchie de rendu pour bien répondre à toutes les instructions demandées.

7.1 Conditions aux limites

7.1.0.1 Questions 1 et 2

Nous avons décidé de nous concentrer sur la partie Absorption : les particules disparaissent lorsqu'elles rencontrent la paroi. Comme vu dans le Nota Bene il est en réalité assez simple de mettre en place les nouvelles conditions aux limites, d'ajouter le potentiel gravitationnel grâce à la gestion de nos classes et méthodes que l'on a pu faire depuis le début.

7.2 Potentiel gravitationnel

7.3 Application : collision de deux objets

7.4 Gestion des erreurs

7.4.0.1 Question 7

Par exemple, nous avons pu mettre en place un mécanisme de gestions des erreurs pour la fonction removeID(), on voulait être sur que l'indice enlevé était bien compris entre 0 et le nombre de particules - 1.

Pour l'initialisation de `Univers` on peut rajouter des vérifications préalables à l'exécution, qui garantiraient que les dimensions ne peuvent pas être négatives, etc...

De même, on peut identifier nos différentes erreurs pendant l'exécution, en nous concentrant de notre côté sur la mise en place d'une gestion d'erreurs vis-à-vis de nos problèmes d'indexation(car tout marche par principe d'indices). Comme nous l'avons souligné, nous avons mis en place des méthodes d'utilisation d'indices pour optimiser au maximum notre code.

Cependant, nous rappelons que les tests ont vérifié (sur des cas simples) que les fonctions fonctionnaient correctement. Il pourrait néanmoins être intéressant d'ajouter une sur-protection en mettant en place différents mécanismes de gestion d'erreurs, comme mentionné précédemment.

7.4.0.2 Question 8

On peut lister différentes forces et faiblesses par le fait de mettre en place un mecanisme de gestion d'erreurs, on rappelle que l'on a pour objectif d'optimiser au maximum notre code et donc de rajouter le moins de "vérifications" possibles qui pourrait rajouter de la complexité...

Pour les forces :

- Rapidité pour débogger : pas de pertes de temps (plus le nombre de fonctions utilisées sera élevées... plus on gagnera du temps pour débogguer)
- Permet de vérifier son code, et d'être sur qu'il n'y aura pas d'erreurs non plus

Pour les faiblesses :

- Rajouter un mécanisme de gestion d'erreurs peut influencer sur le temps d'execution de notre code (mauvais ici pour notre optimisation...)
- Peut prendre du temps à mettre en place
- Si on était amené à faire des erreurs en mettant ces gestions d'erreurs en place il faudra vérifier de nouveau tout notre mecanisme de gestion d'erreurs, et cela pourrait aussi nous amener à changer des fonctions qui pouvaient être bonnes et dont la gestion d'erreur associée était faussée.

Conclusion :

Finalement, même si nous avons pu passer beaucoup de temps à mettre en place une structure de données et de code plutôt fantaisiste ici (passage 2D à 1D,la mise en place d'indice plutôt que d'utiliser des pointeurs, etc...) pour optimiser le plus possible notre code, on obtient quelque chose qui marche bien et qui est bien optimisé.
Malgré le fait qu'il ne nous restait plus qu'à implémenter les dernières conditions aux limites, nous sommes finalement satisfaits des résultats obtenus grâce à tout le travail que nous avons fourni.

Auteur

hadrienmarc - 2023