

# Rapport Projet C++

Mohammed Reda Belfaida & Mehdi El Oudghiri

## 1 Rapport du TP1 ( Calcul Scientifique en C++ )

**Objectif :** Ce rapport présente le travail réalisé lors de la première séance de travaux pratiques sur les bases d'un environnement C++ pour le calcul scientifique. L'objectif était de se familiariser avec les outils de développement, le débogage, le profiling et la résolution d'équations différentielles ordinaires (EDO).

### 1.1 Débogage du code de calcul de trace

### 1.2 Problèmes identifiés

Le code fourni pour le calcul de la trace d'une matrice présentait plusieurs problèmes :

- Utilisation de `calloc` (C) au lieu de `new` (C++) pour l'allocation mémoire
- Mauvaise gestion de la mémoire (pas de libération)
- Types incorrects dans les `sizeof` lors des allocations
- Fonction `fill_vectors` modifie son paramètre d'entrée alors qu'elle pourrait travailler directement sur la matrice

### 1.3 Corrections apportées

Les corrections suivantes ont été implémentées :

```
// Allocation avec new au lieu de calloc
double** initialization(int n) {
    double** matrix = new double*[n];
    for(int i = 0; i < n; i++) {
        matrix[i] = new double[n]{0}; // Initialisation à zéro
    }
    return matrix;
}

// Ajout d'une fonction de libération
void free_matrix(double** matrix, int n) {
    for(int i = 0; i < n; i++) {
        delete[] matrix[i];
    }
    delete[] matrix;
}
```

### 1.4 Profiling et optimisation

Le profiling a été réalisé avec `gprof`, révélant que :

- La fonction `fill_vectors` était un point chaud
- L'allocation mémoire représentait une part significative du temps

Pour améliorer les performances, on pourrait :

- Utiliser des conteneurs STL (`std::vector`)
- Pré-allouer toute la mémoire en une seule opération
- Paralléliser le remplissage des vecteurs

## 1.5 Résolution d'équations différentielles

### 1.5.1 Méthode d'Euler explicite

La méthode d'Euler explicite a été implémentée sous forme récursive :

```
double euler_explícite(int n, double step, double init, double a,
                       double (*phi)(double x, double y)) {
    if(n == 0) return init;
    double val = euler_explícite(n-1, step, init, a, phi);
    return step * phi(val, a + (n-1)*step) + val;
}
```

Pour  $\varphi(x, u(x)) = 2xu(x)$  sur  $[0, 1]$  avec  $u(0) = 1$ , la solution théorique est  $u(x) = e^{x^2}$ . Les résultats numériques montrent une bonne convergence pour des pas petits.

### 1.5.2 Méthode d'Euler implicite

La version implicite a été implémentée avec un schéma itératif :

```
double euler_implicite(int n, double step, double init, double a,
                       double (*phi)(double x, double y)) {
    if(n == 0) return init;
    double start = euler_implicite(n-1, step, init, a, phi);
    double curr = start;
    double eps = 1e-3;
    for(int i = 0; i < 100; i++) {
        double prev = curr;
        curr = step * phi(curr, a + n*step) + start;
        if(abs(curr - prev) < eps) break;
    }
    return curr;
}
```

### 1.5.3 Application à une autre EDO

L'équation  $\frac{du}{dx} = 50(u \cos(x))$  avec  $u(0) = 0$  a été résolue avec les deux méthodes. Les résultats montrent que :

- La méthode explicite est plus simple à implémenter
- La méthode implicite est plus stable pour les grands pas
- Les deux méthodes convergent vers la solution théorique

## 1.6 Résultats numériques

### 1.6.1 Résolution avec $\varphi(x, u(x)) = 2xu(x)$

Pour  $n = 100$  itérations avec un pas  $h = 10^{-3}$ , nous obtenons les résultats suivants pour l'équation  $\frac{du}{dx} = 2xu(x)$  avec  $u(0) = 1$  :

On observe que :

- Les deux méthodes fournissent une excellente approximation de la solution théorique
- Les erreurs sont du même ordre de grandeur (environ  $10^{-4}$ )
- La méthode explicite sous-estime légèrement la solution
- La méthode implicite surestime légèrement la solution

Méthode	Valeur théorique	Valeur numérique	Erreur
Euler explicite	1.01005	1.00995	0.000101663
Euler implicite	1.01005	1.01015	0.000100327

TABLE 1 – Résultats pour  $\varphi(x, u(x)) = 2xu(x)$

### 1.6.2 Résolution avec $\varphi(x, u(x)) = 50u(x) \cos(x)$

Pour la seconde équation  $\frac{du}{dx} = 50u(x) \cos(x)$  avec  $u(0) = 0$ , les résultats sont :

Méthode	Valeur théorique	Valeur numérique	Erreur
Euler explicite	0	0	0
Euler implicite	0	0	0

TABLE 2 – Résultats pour  $\varphi(x, u(x)) = 50.u(x). \cos(x)$

Les résultats montrent que :

- Pour la condition initiale  $u(0) = 0$ , les deux méthodes donnent la solution exacte
- Ce résultat était prévisible car la solution triviale  $u(x) = 0$  satisfait à la fois l'équation différentielle et la condition initiale
- La linéarité de l'équation et la condition initiale nulle expliquent cette convergence parfaite

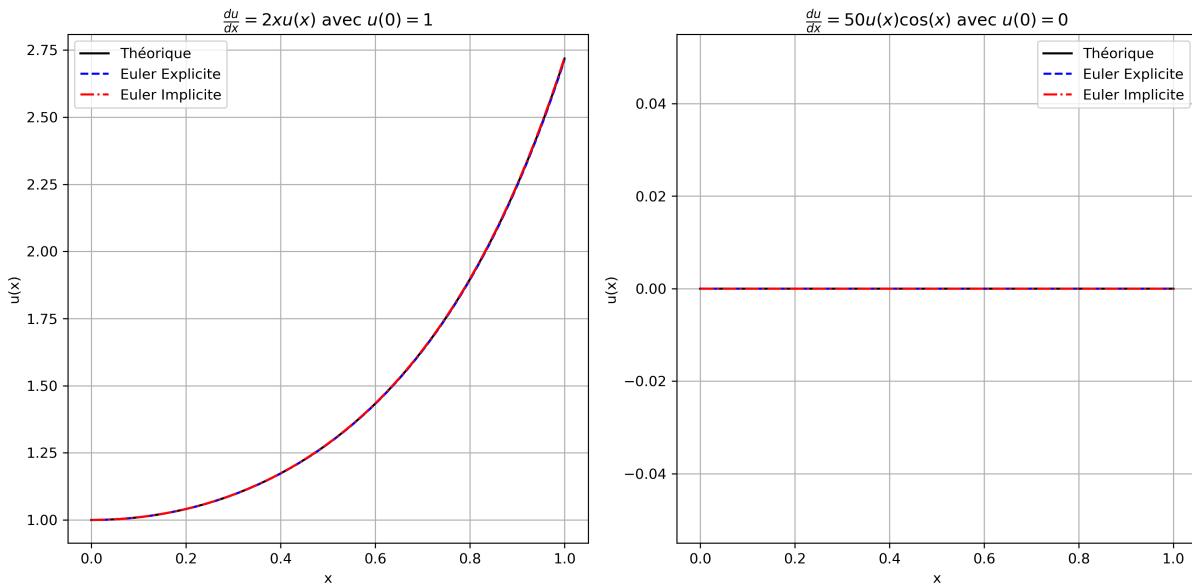


FIGURE 1 – Comparaison des solutions numériques

## 2 Transition vers le travail principal

Pour la suite, notre travail portera sur le développement d'un simulateur numérique modélisant la dynamique de particules dans un espace discréte. L'approche consiste en :

### 1. Modélisation spatiale :

- L'univers de simulation est partitionné en cellules discrètes
- Chaque cellule contient un ensemble de particules en interaction
- Les contraintes spatiales sont définies par des longueurs caractéristiques

### 2. Paramétrisation :

- Dimensions configurables (1D, 2D ou 3D)
- Nombre de particules personnalisable
- États initiaux paramétrables (positions, vitesses, forces)
- Contrôle temporel (pas de temps  $\Delta t$  et durée totale  $t_{\text{end}}$ )

### 3. Mécanismes de simulation :

- Interactions interparticulaires
- Dynamique de mouvement contraint par les limites cellulaires
- Intégration temporelle des équations du mouvement

### 4. Visualisation scientifique :

- Génération automatisée de séries temporelles au format VTK
- Compatibilité avec ParaView pour la représentation graphique
- Archivage des états (position, vitesse, masse) à chaque itération

Ce système permet d'étudier numériquement les phénomènes d'auto-organisation particulaire dans des environnements contraints, avec applications potentielles en :

- Physique statistique
- Dynamique des fluides numérique
- Science des matériaux

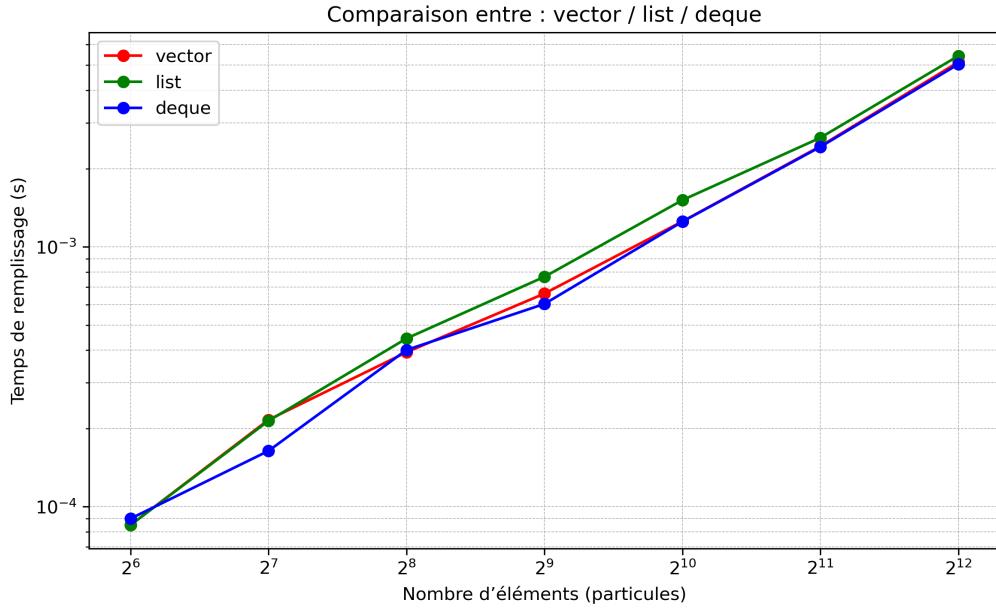
## 3 Rapport du TP2

### Q1 – Classe Particule

La classe `Particule` est entièrement définie dans `src/Particule.cxx`. Elle regroupe les caractéristiques physiques (id, catégorie, masse, position, vitesse, forces) ainsi que les méthodes de base nécessaires aux calculs gravitationnels et aux mises à jour.

### Q2 & Q3 – Comparaison des conteneurs STL

Le programme `src/comparaisonPerf.cxx` génère des lots de `Particules` aléatoires (de 64 à 4096 éléments) et mesure le temps d'insertion dans trois conteneurs : `vector`, `deque`, `list`.



Le graphique montre une progression linéaire du temps lorsque la taille du lot augmente : doubler  $N$  double grossièrement la durée. Pour toute la plage testée, `vector` s'avère le plus rapide ( 5 ms à 4096 éléments), suivi de près par `deque`, tandis que `list` reste 10 % plus lente. On choisira donc : `vector` pour un remplissage séquentiel classique, `deque` si l'on souhaite aussi des ajouts en tête, `list` pour des insertions / suppressions fréquentes au milieu.

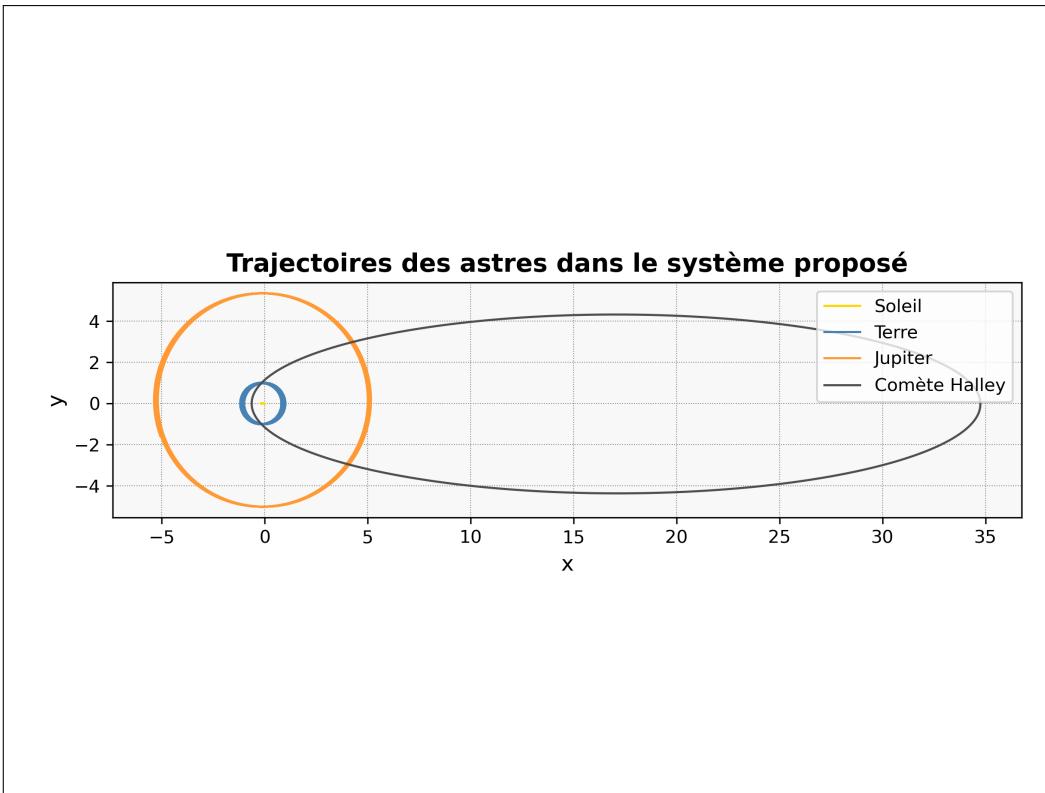
#### Q4 – Intégrateur Strömer–Verlet

La méthode `Univers::applyStormerVerlet` réalise le schéma Strömer–Verlet en quatre appels successifs, très lisibles :

1. `updatePositions` : avance les positions  $x$  ;
2. `sauvegarderAnciennesForces` : conserve les forces  $F_{\text{old}}$  ;
3. `updateForcesCellulaire` : recalcule les nouvelles forces  $F$  ;
4. `updateVelocities` : met à jour les vitesses  $v$  à partir de  $F$  et  $F_{\text{old}}$ .

Chaque étape est déléguée à une fonction dédiée, ce qui garde la boucle principale compacte et facilement vérifiable.

#### Q5 – Vérification de l'implémentation



Cette image représente la trajectoire de chaque astre lors de la simulation du système proposé avec les données fournies.

Le plot de l'image s'est fait à partir des données `trajectory.txt` disponibles dans le dossier `out`.

#### Observations :

- On remarque d'abord que Jupiter et la planète Terre ont une orbite circulaire autour du Soleil, ce qui est cohérent avec les données, vu que la vitesse initiale est tangentielle.
- Ensuite, le Soleil reste quasi immobile, ce qui est cohérent physiquement vu la masse du Soleil.
- Enfin, la comète de Halley suit une trajectoire fortement excentrique avec une orbite très allongée.

La simulation est donc très cohérente avec les données. On conclut que l'implémentation de l'algorithme s'est faite avec succès.

#### Q6 – Protection des données

Les attributs internes de `Particule` sont déclarés `private`. La lecture passe par des accesseurs qui retournent une copie (`getPosition`, `getMass`, ...), l'écriture par des mutateurs (`setPosition`, `addForce`, ...). Le contenu de la particule ne peut donc pas être modifié directement de l'extérieur, garantissant l'encapsulation demandée.

## 4 Rapport du TP3

### 4.1 Q1 & Q2 - Classe Vecteur

La classe **Vecteur** est entièrement définie dans `src/Particule.cxx`. La classe **Vecteur** implémente les opérations fondamentales d'un vecteur 3D en C++ avec :

- **Attributs** : Composantes  $x, y, z$  (`c_x, c_y, c_z`)
- **Opérations de base** :
  - Addition/soustraction (`add, sub`, opérateurs `+`, `-`)
  - Multiplication scalaire (`mul`, opérateur `*`)
  - Négation (opérateur `-` `unaire`)
- **Fonctionnalités avancées** :
  - Calcul de norme (`norm()`)
  - Normalisation (`normalized()`)
  - Accès sécurisé aux composantes (`operator[]`)
- **Utilitaires** :
  - Affichage (`toString()`, opérateur `<>`)
  - Copie (`set()`)

### 4.2 Q3 - Testing de la classe Vecteur

Les tests unitaires de la classe **Vecteur** ont été implémentés dans le fichier `test/testVecteur.cxx` en utilisant le framework Google Test (gtest). Ces tests vérifient toutes les fonctionnalités de la classe, incluant :

- Les constructeurs et accesseurs (getters)
- Les opérations mathématiques de base (addition, soustraction, multiplication scalaire)
- Les surcharges d'opérateurs
- Le calcul de norme et normalisation
- La gestion des erreurs (vecteur nul)

**Exemple de test :**

Listing 1 – Extrait des tests de la classe Vecteur

```
TEST(VecteurTest, NormAndNormalize) {
    Vecteur v(3.0, 4.0, 0.0);
    EXPECT_DOUBLE_EQ(v.norm(), 5.0); // Verification norme

    Vecteur normed = v.normalized();
    EXPECT_NEAR(normed.getxCoordinate(), 0.6, 1e-6); // Normalisation
}
```

Catégorie de test	Nombre de cas
Constructeurs/Accessseurs	3
Opérations mathématiques	6
Surcharges d'opérateurs	5
Normalisation	2
Gestion d'erreurs	1
<b>Total</b>	<b>17</b>

TABLE 3 – Récapitulatif des tests unitaires pour la classe Vecteur

#### 4.3 Q4 - Modification de la classe Particule

La classe `Particule` a été restructurée pour utiliser la classe `Vecteur` comme type fondamental pour toutes les grandeurs vectorielles :

- **Attributs modifiés :**
  - Position, vitesse, force et ancienne force sont désormais des objets `Vecteur`
  - Suppression des composantes individuelles (x,y,z) au profit du type vectoriel
- **Méthodes mises à jour :**
  - Les accesseurs (`get/set`) manipulent directement des `Vecteur`
  - `calculateGravitationalForce` utilise les opérations vectorielles
  - `toString` affiche les vecteurs via l'opérateur « surchargé »
- **Avantages :**
  - Code plus lisible et maintenable
  - Sécurité accrue (encapsulation des opérations vectorielles)
  - Meilleure précision numérique

Listing 2 – Exemple d'utilisation

```
// Calcul de force gravitationnelle entre deux particules
Vecteur delta = p1.getPosition() - p2.getPosition();
double distance = delta.norm();
Vecteur force = delta.normalized() * (G * m1 * m2 / (distance*distance));
```

#### 4.4 Q5 - Classe Univers

La classe `Univers`, implémentée dans le fichier `src/Univers.cxx`, constitue le composant central de notre simulation. Elle intègre :

- La gestion des particules et de leurs interactions
- Le contrôle temporel de la simulation (pas de temps, durée totale)
- Les mécanismes d'évolution du système physique

L'architecture complète de cette classe est fournie en annexe (section ??).

#### 4.5 Q6 & Q7

Le programme `comparaisonPerf.cxx` a été conçu pour évaluer les performances d'insertion de particules dans un univers cubique. La configuration testée comprend :

- Génération de  $(2^5)^3 = 32,768$  particules
- Répartition uniforme dans le domaine  $[0, 1]^3$
- Conteneur de type `std::vector` (STL)
- Mesures effectuées sur architecture x86-64 (Ryzen 7 6800H)

#### 4.5.1 Résultats

Nombre de Particules	Temps Total (ms)	Temps/particule ( $\mu$ s)
$(2^5)^3 = 32,768$	$14.0 \pm 0.3$	$0.427 \pm 0.009$

TABLE 4 – Performances d’insertion pour un univers dense

Plus généralement :

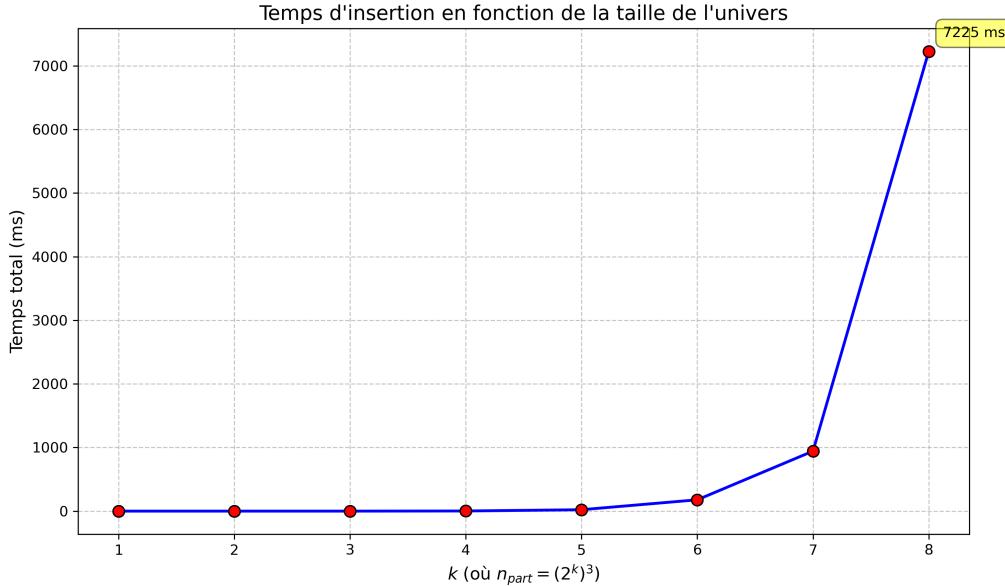


FIGURE 2 – Évolution du temps total d’insertion en fonction de la valeur  $k$  (où le nombre de particules est  $(2^k)^3$ )

#### 4.5.2 Interprétation

Les données révèlent que :

- L’insertion moyenne prend 427 ns par particule.
- Le temps total de 14 ms montre une bonne complexité.
- L’overhead mémoire est négligeable grâce à l’allocation contiguë du **vector**

### 4.6 Q8 & Q9 & Q10

Le programme `src/InteractionPerf.cxx` évalue la performance des calculs d’interaction entre particules dans notre simulation. Les résultats mettent en évidence l’impact de l’optimisation sur le temps d’exécution.

#### 4.6.1 Résultats Bruts : Complexité Quadratique $\mathcal{O}(N^2)$

La méthode naïve montre une explosion du temps de calcul dès que  $N$  augmente, en raison de sa complexité quadratique.

**Observation :** le temps estimé devient prohibitif (plusieurs heures) pour  $k$  dépassant 6.

k	Nombre de Particules	Temps Calculé (ms)
$k = 1$	8	0
$k = 2$	64	0
$k = 3$	510	14
$k = 4$	4096	919
$k = 5$	32768	57805

TABLE 5 – Performance de l'approche naïve

#### 4.6.2 Première Optimisation : Pré-allocation Mémoire

Une simple pré-allocation dans le constructeur de `Univers` réduit les surcoûts de gestion dynamique de la mémoire :

Listing 3 – Constructeur optimisé

```
Univers::Univers(int d, int n)
    : dim(d), number(n) {
    particules.reserve(n); // Pré-allocation initiale
}
```

#### 4.6.3 Résultats après optimisation :

k	Nombre de Particules (N)	Temps (ms)	Gain
3	512	11 (-21%)	3 ms
4	4,096	623 (-32%)	296 ms
5	32,768	38,816 (-33%)	18,989 ms

TABLE 6 – Performance après optimisation

**Explication** : La pré-allocation évite les réallocations fréquentes du vecteur, réduisant les temps d'exécution de **20 à 33%**.

#### 4.6.4 Pistes d'Optimisation Avancée

Pour mieux diminuer la complexité de nos calculs, deux axes majeurs sont envisagés :

##### Parallélisation (Loi d'Amdahl)

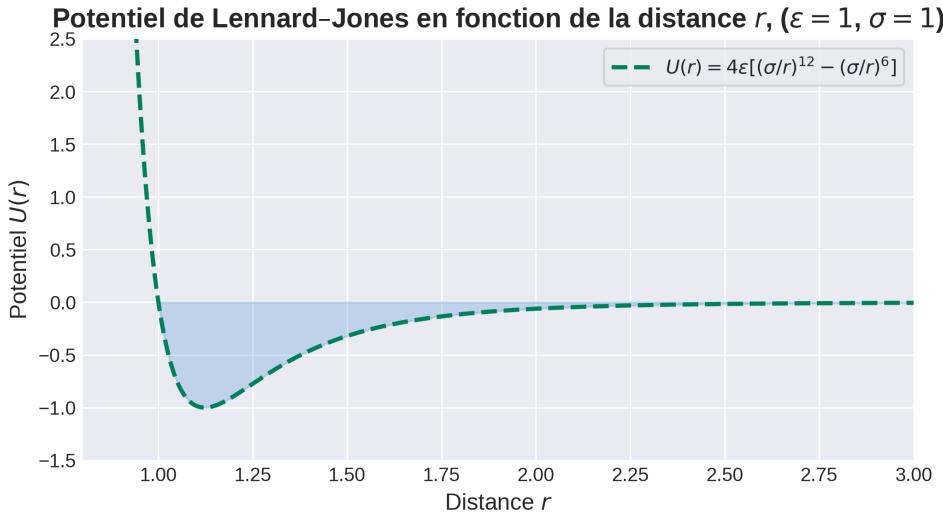
- **Cible** : Calcul des forces d'interaction (embarassingly parallel).
- **Gain attendu** : Jusqu'à **3.1×** sur un CPU multicœur (e.g., 8 cœurs).
- **Outils** : OpenMP, `std::thread`, ou GPU avec CUDA/OpenCL.

##### Réduction des Accès Mémoire

- **Stratégie** :
  - Partitionnement spatial (grilles, arbres octaux).
  - Optimisation du cache (blocage mémoire, **array of structures AOS vs Structure of arrays SOA**).
- **Impact potentiel** : Réduction de la complexité effective de  $O(N^2)$  vers  $O(N \log N)$ .

## 5 Rapport du TP4

### 5.1 Q1 - Représentation de $U(r)$



Comme c'est mentionné au début de l'énoncé du TP4, on a pris  $\varepsilon = 1$  et  $\sigma = 1$  et on a tracé la courbe représentant le potentiel  $U(r)$  en fonction de la distance  $r$  pour un système de deux particules.

- Pour  $r < \sigma = 1$ , le potentiel est nettement positif ce qui a un effet repulsif entre les deux particules.
- Pour  $r > \sigma = 1$ , le potentiel devient négatif ce qui a un effet attractif entre les deux particules.
- Pour un  $r$  grand  $U(r)$  tend rapidement vers zéro : L'interaction devient donc négligeable.  
**Remarque :** le choix de  $r_{\text{cut}}$  dans la simulation de la question 6 de ce TP est justifié. On remarque directement sur le graphe que pour  $r > r_{\text{cut}} = 2,5\sigma$ ,  $U(r)$  est quasi nulle.

### 5.2 Questions 2 & 3 – Implémentation de la classe Univers et initialisation

La classe **Univers** modélise l'espace de simulation. Elle contient les attributs suivants :

- **int dim** : la dimension de l'espace (1, 2 ou 3),
- **int number** : le nombre de particules initial,
- **std::vector<Particule> particules** : liste des particules,
- **std::vector<Cellule> cellules** : grille de cellules,
- **int taillesGrilles[3]** : nombre de cellules dans chaque direction,
- **double tailleCellule** : taille d'une cellule,
- **double rayonDeCoupure** : distance limite pour l'interaction.

Les tailles des grilles sont calculées selon la formule :

$$\text{taillesGrilles}[i] = \left\lfloor \frac{\text{Ld}[i]}{\text{rcut}} \right\rfloor$$

L'initialisation se fait via la méthode **initialiserGrille()**, qui :

- crée les cellules (**cellules.resize()**),
- répartit les particules dans les cellules via **indexCellule()**,
- et appelle **initialiserVoisines()** pour lister les voisines selon la dimension.

### 5.3 Question 4 – Calcul de la force via le maillage

Pour optimiser les performances du calcul des forces, nous avons modifié l'algorithme de parcours classique  $\mathcal{O}(N^2)$  en exploitant la structure de maillage par cellules.

Le code implémenté est le suivant :

Listing 4 – Calcul des forces avec maillage

```
void Univers::updateForces(double eps, double sigma) {
    for (auto& p : particules) {
        p.setForce(Vecteur(0, 0, 0));
    }

    // Calcul des interactions entre les particules des cellules voisines de
    // chaque cellule
    for (Cellule& cellule : cellules) {
        for (Particule* p1 : cellule.getParticules()) {
            for (Cellule* voisine : cellule.getVoisines()) {
                for (Particule* p2 : voisine->getParticules()) {
                    if (p1 < p2) {
                        double distance = (p1->getPosition() -
                            p2->getPosition()).norm();
                        if (distance < rayonDeCoupure) {
                            calculateLennardJonesForce(*p1, *p2, eps, sigma);
                        }
                    }
                }
            }
        }
    }
}
```

**Choix algorithmique** : chaque cellule stocke directement ses cellules voisines (la cellule elle-même inclue), ce qui réduit le nombre de paires à évaluer à un sous-ensemble local. Cela permet d'éviter un parcours quadratique global. Par ailleurs, le potentiel de Lennard-Jones décroît rapidement avec la distance : au-delà de  $r_{cut}$ , la force est négligeable, ce qui justifie le test `distance < rayonDeCoupure` pour ignorer les interactions lointaines.

**Remarque** : à la fin de chaque itération, on appelle `mettreAJourCellules()` pour repositionner les particules dans la bonne cellule, assurant la cohérence du maillage dans le temps.

### 5.4 Question 5 – Mise à jour des cellules

La fonction `mettreAJourCellules()`, définie dans `Univers.cxx`, permet d'actualiser la répartition des particules dans la grille. Elle repose sur la méthode `indexCellule()` pour déterminer la cellule logique associée à chaque particule. À chaque itération, les anciennes listes sont vidées, puis les particules sont replacées dans leur cellule correspondante.

### 5.5 Question 6 — Collision carré/rectangle (Lennard–Jones)

La figure 3 retrace l'évolution du système à six instants clés<sup>1</sup>. On y observe successivement :

1. configuration initiale ( $t = 0$ ) ;
2. premier impact et onde de compression ( $t \simeq 1$ ) ;
3. mélange et gerbe centrale ( $t \simeq 2$ ) ;
4. fragmentation de l'ensemble ( $t \simeq 5$ ) ;
5. dispersion avancée ( $t \simeq 10$ ) ;
6. nuage dilué, forces LJ négligeables ( $t = 19,5$ ).

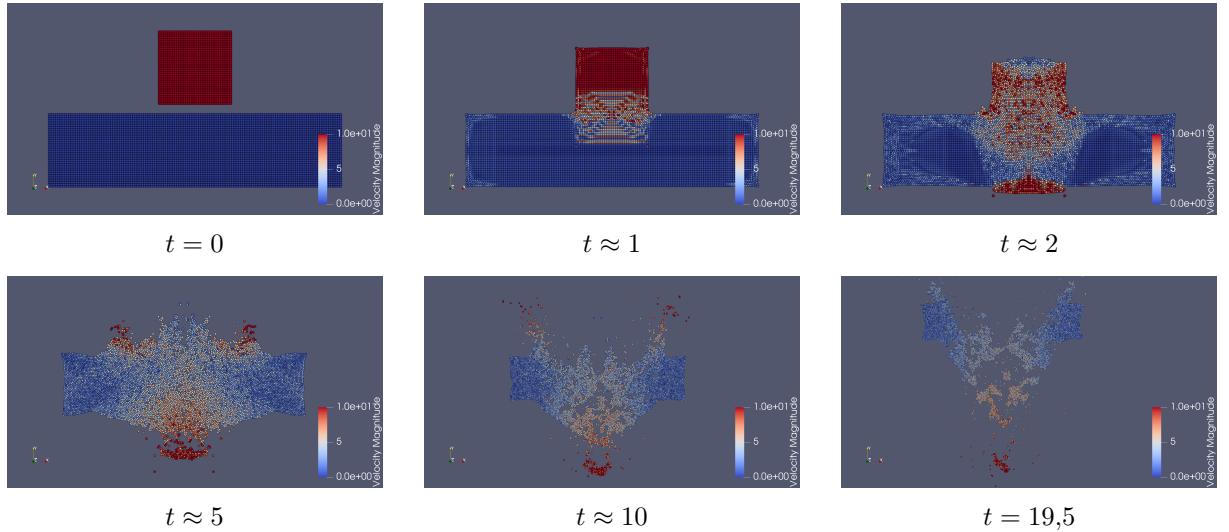


FIGURE 3 – Chronologie succincte de la collision ; chaque vignette utilise la même échelle de couleurs (bleu :  $v \simeq 0$ , rouge :  $v \simeq 10$ ).

**Remarque :** pour des raisons de mémoire, le pas de temps utilisé dans la simulation a été porté à  $\Delta t = 5 \times 10^{-4}$  au lieu de  $5 \times 10^{-5}$ , ce qui a probablement altéré le résultat final.

---

1. Visualisations ParaView : particules colorées par norme de vitesse.

## 6 Rapport du TP5

### 6.0.1 Q1 & Q2 - Logique de test et intégration

La logique de test et son intégration sont détaillées dans l'annexe (Section ??).

### 6.0.2 Q3 - Génération des fichiers VTK

La méthode `genererVTK` de la classe `Univers` permet de générer des fichiers au format `.vtu`. Elle prend en paramètres :

- `step` (entier) : représente l'instant  $t$  de la simulation pour lequel la génération est effectuée
- `directoryName` (chaîne de caractères) : spécifie le nom du répertoire de destination des fichiers générés

Par défaut, les fichiers produits sont stockés dans le sous-répertoire `out` du répertoire d'exécution.

### 6.1 Q4 & Q5 & Q6

Pour clarifier l'architecture et le comportement dynamique de notre simulateur, nous présentons trois vues complémentaires :

- Le **diagramme de cas d'utilisation** (Fig. 4) identifie les interactions entre acteurs et fonctionnalités
- Le **diagramme de séquence** (Fig. 5, 6) détaille les flux temporels
- Le **diagramme de transitions** (Fig. 7) modélise les états du système

#### 6.1.1 Diagramme de cas d'utilisation

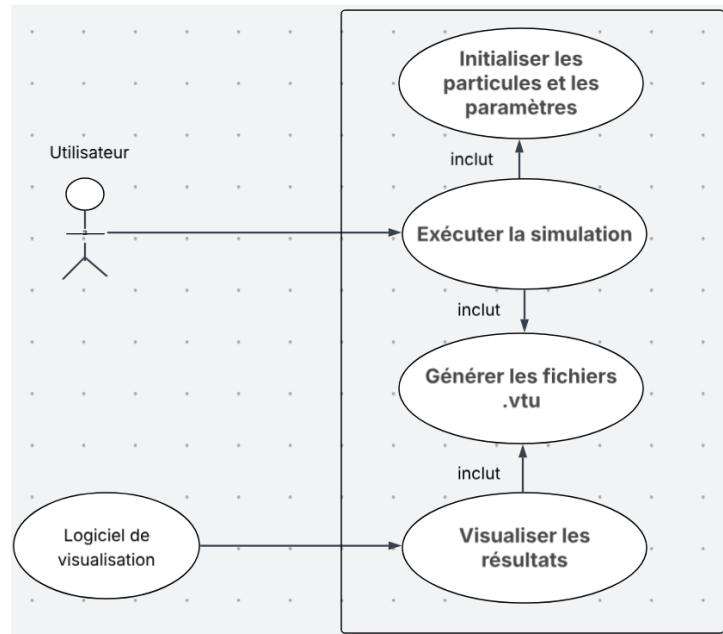


FIGURE 4 – Diagramme de cas d'utilisation du simulateur de particules

### 6.1.2 Diagramme de séquence

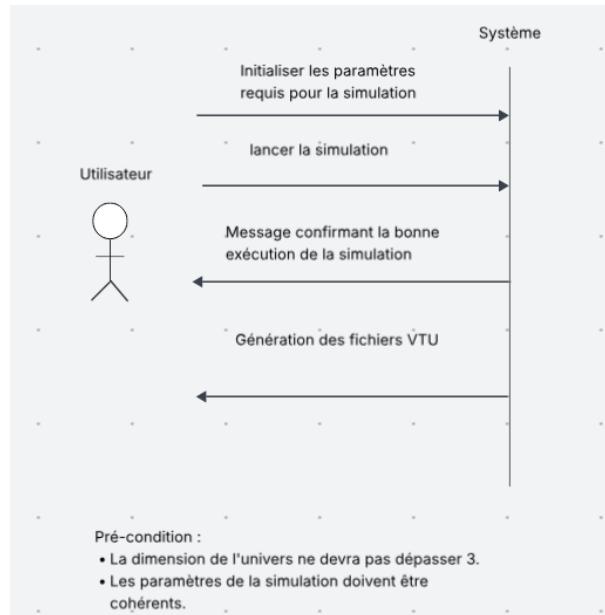


FIGURE 5 – Sequence principale de la simulation

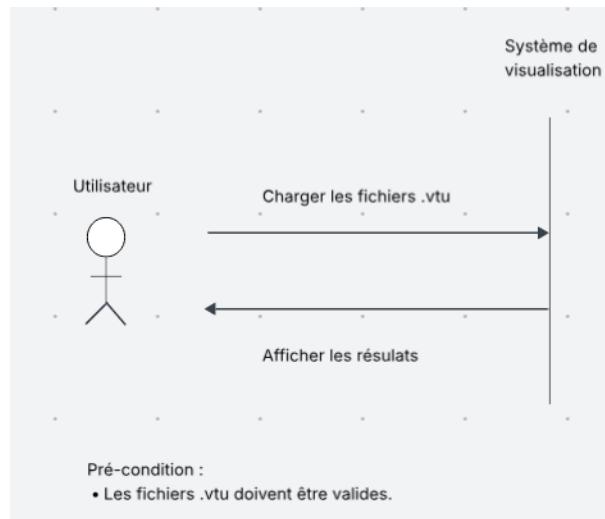


FIGURE 6 – Sequence détaillée du calcul des forces

### 6.1.3 Diagramme de transitions

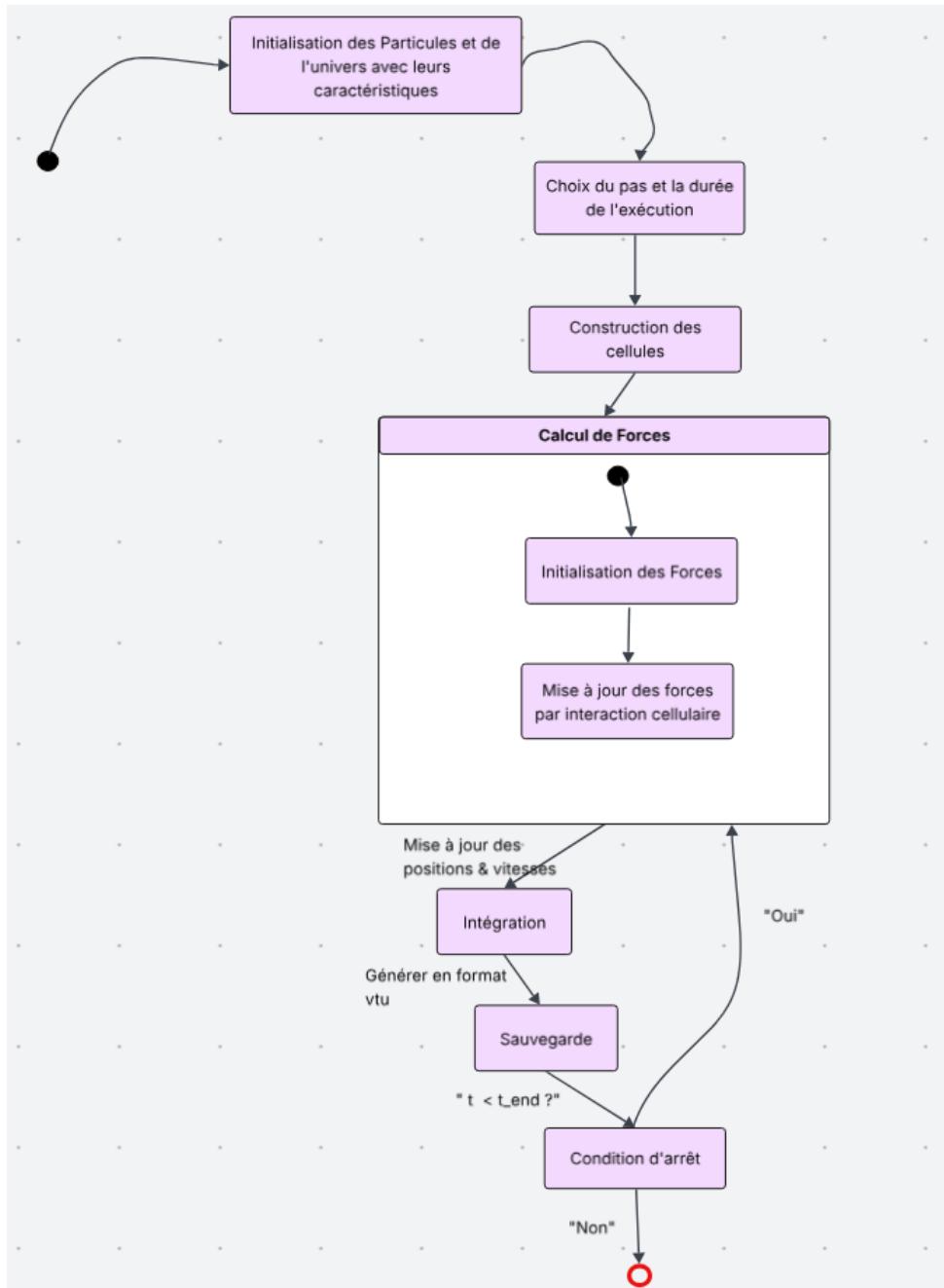


FIGURE 7 – Diagramme d'états du système

## 6.2 Q7

Le diagramme de classes d'analyse est donné par :

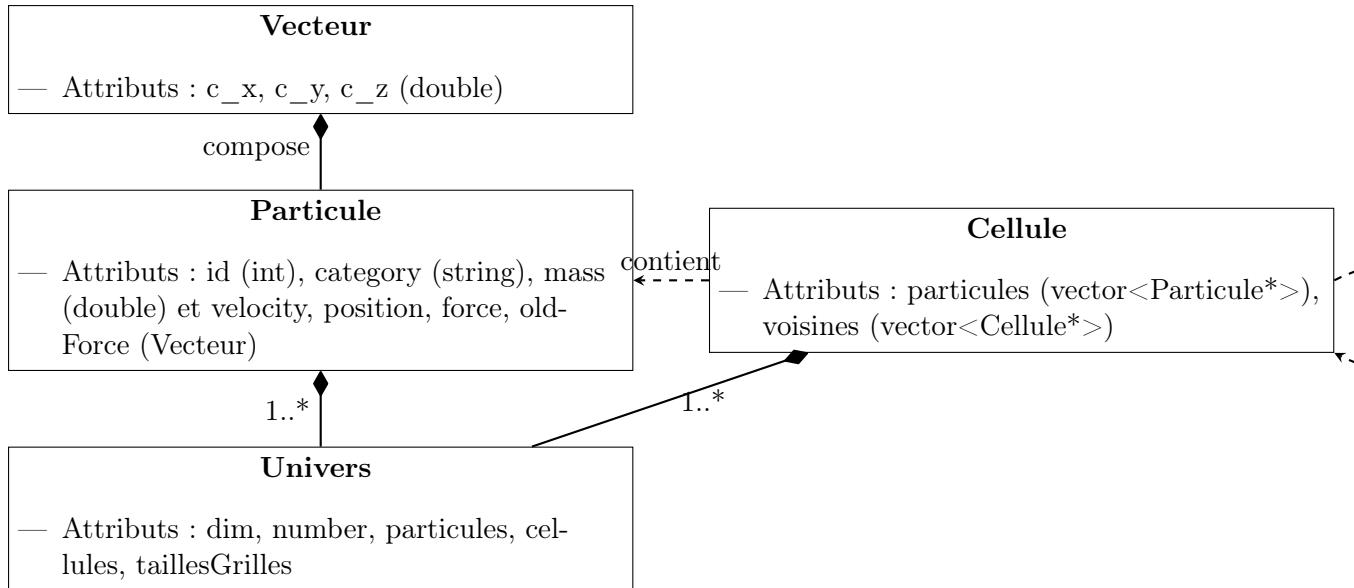


FIGURE 8 – Diagramme de classes du système de simulation

### 6.2.1 Description des relations

- **Composition** (losange plein) :
  - Une **Particule** contient 4 **Vecteur** (position, vitesse, etc.)
  - L'**Univers** gère le cycle de vie des **Particule** et **Cellule**
- **Agrégation** (losange vide) :
  - Une **Cellule** référence des **Particule** (mais ne les possède pas)
- **Association** (flèche pointillée) :
  - Les **Cellule** sont connectées entre elles (voisinage)

La documentation détaillée des classes est disponible grâce à **Doxxygen**.

## 7 Rapport du TP6

**Q1 – Conditions aux limites** nous avons isolé la gestion des frontières dans une hiérarchie dédiée (`conditionsLimites`). Le principe retenu relève du *pattern Stratégie* : une classe abstraite `ConditionLimite` définit l’interface `appliquer(Particule&, Univers&)`, ainsi qu’un test `estHorsLimites(...)`. Trois réalisations concrètes en héritent :

- **Reflexion** : si la coordonnée sort du domaine, on la *replie* ( $x \leftarrow 2L - x$ ) et on inverse la composante de vitesse correspondante ; la conservation de l’énergie cinétique est ainsi respectée.
- **Absorption** : la particule est simplement retirée via `Univers::supprimerParticule`, ce qui coûte  $\mathcal{O}(1)$  grâce au stockage contigu.
- **Periodique** : on applique un décalage modulo la longueur  $L$  ( $x \leftarrow x \pm L$ ), sans toucher à la vitesse, afin d’imiter un espace infini tuilé.

Chaque stratégie ne réalise qu’un test  $\forall i, 0 \leq x_i < L_i$  ; la complexité reste donc linéaire en nombre de particules et indépendante du type de condition. L’approche polymorphe permet enfin de sélectionner la condition voulue au lancement de la simulation sans modifier le reste du code.

**Q2 - tests** Les tests pour les conditions aux limites sont dans `./testConditionsLimites.cxx`

**Q3 – Réflexion potentielle** L’objet `ReflexionPotentiel` est instancié dans le constructeur de `Univers`. À chaque itération, on applique cette condition sur toutes les particules proches des bords. Si la distance à une paroi est inférieure à `rCut`, une force de répulsion est ajoutée

**Q5 - champ gravitationnel** L’effet du champ gravitationnel est ajouté à `updateForces` de la classe `univers`.