Simulation de Particules en N Dimensions

Mattéo Audigier Mattéo Gautier

21 mai 2025

Résumé

Ce rapport présente une implémentation C++ d'un système de simulation de particules en N dimensions. Le projet utilise de nombreuses optimisations pour simuler efficacement les interactions entre particules. Cette bibliothèques ce calcul scientifique permet de modéliser des phénomènes physiques tels que les forces de Lennard-Jones et la gravitation.

Table des matières

1 Introduction

Dans le contexte du développement logiciel, il est important d'avoir des bibliothèques capables de gérer efficacement des simulations, tout en restant rapides, modulaires et simples d'utilisation. Ce projet propose une architecture générique en C++ permettant de simuler des systèmes de particules dans un espace à N dimensions, où N est un paramètre de template. L'objectif est de fournir une base facilement adaptable à divers cas d'usage, et intégrable dans d'autres projets de calcul dynamique de particules.

1.1 Objectifs du projet

Les objectifs principaux de ce projet sont :
- Concevoir une architecture C++ modulaire et extensible pour la simulation de particules
- Disposer d'un projet complet (documentation, tests unitaires, simple à installer), prêt à être repris et étendu par d'autres développeurs
- Implémenter un modèle physique incluant forces de Lennard-Jones et gravitation
- Optimiser les calculs grâce au partitionnement spatial
- Fournir des outils de visualisation via export au format VTK

2 Installation et utilisation

2.1 Prérequis

Pour compiler et utiliser ce projet, il est nécessaire d'avoir :
- CMake (version 3.16.3 ou supérieure)
- Un compilateur C++ supportant le standard C++17
- Doxygen pour la génération de documentation (optionnel)
- ParaView pour visualiser les fichiers VTK générés (optionnel)

2.2 Installation

Pour installer le projet, suivez ces étapes :

```
# Créer un répertoire de build
mkdir build && cd build

# Configurer avec CMake
cmake ..

```

```
# Compiler le projet
make
```

2.3 Exécution des tests

Pour s'assurer que le programme fonctionne correctement, exécutez les tests :

```
make test  # Exécution de tous les tests unitaires
```

2.4 Exécution des démonstrations

Deux simulations de démonstration sont disponibles :

```
# Simulation de collisions moléculaires
make demo_collision

# Simulation gravitationnelle
make demo_planet

# Afficher l'aide sur les démos disponibles
make demo
```

Les fichiers VTK générés dans le répertoire `demo/nomDeLaDemo` peuvent être visualisés avec ParaView :

```
paraview output_*.vtk
```

2.5 Génération de la documentation

La documentation API du projet peut être générée grâce à Doxygen :

```
make doc
```

La documentation générée est disponible dans le répertoire `doc/html/index.html`.

2.6 Optimisations et performance

Le projet utilise plusieurs niveaux d'optimisations de compilation pour améliorer les performances. Ces options peuvent être modifiées dans le fichier `CMakeLists.txt` si nécessaire.

3 Architecture du logiciel

3.1 Organisation du projet

L'arborescence du projet est organisée comme suit :

```
root/
|---- CMakeLists.txt          # Configuration CMake principale
|---- Doxyfile.in             # Template pour la documentation Doxygen
|---- README.md               # Documentation principale
|---- include/                # Fichiers d'en-tête (.hpp)
|
|---- src/                    # Implémentations (.cpp, .tpp)
|
|---- demo/                   # Exemples de simulations
|
`---- test/                   # Tests unitaires
```

Notre simulation est construite autour de cinq classes principales :

Classe	**Responsabilité**
Vecteur<N>	Opérations vectorielles en N dimensions
Particle<N>	Propriétés et comportements des particules
Cell<N>	Partitionnement spatial pour optimisation
Univers<N>	Gestion globale de la simulation
VTKconverter<N>	Export des données pour visualisation

TABLE 1 – Classes principales du système

3.2 Diagrammes

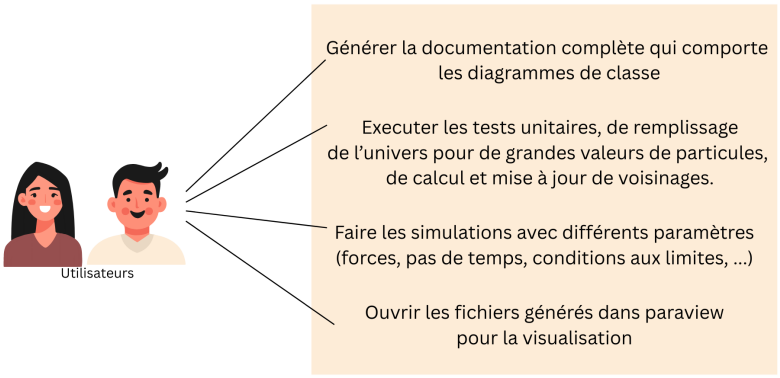

FIGURE 1 – Diagramme de cas d'utilisation

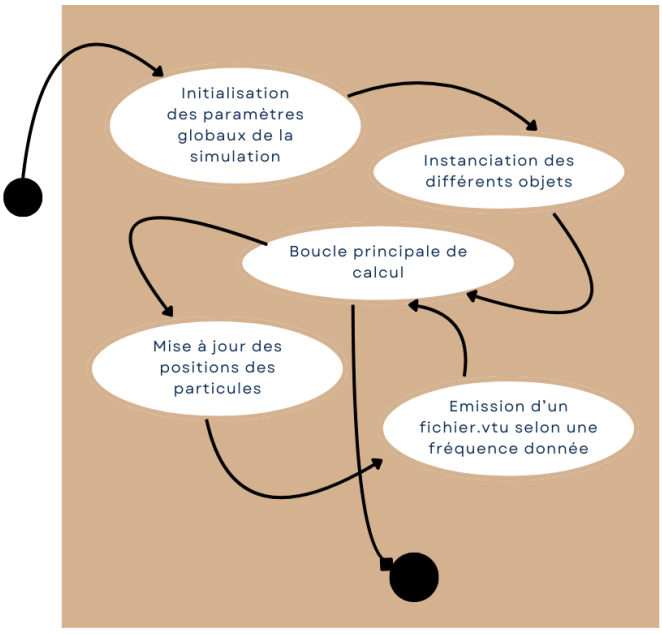

FIGURE 2 – Diagramme d'état de la simulation

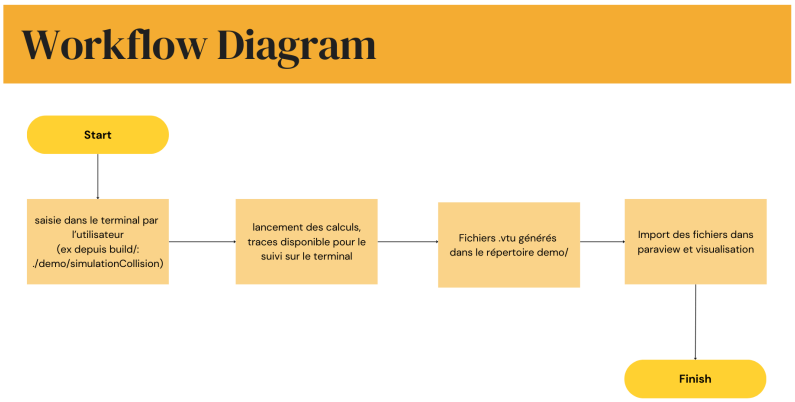

FIGURE 3 – Diagramme de séquence de la simulation

3.3 Diagramme UML des classes

Le diagramme de classes ci-dessous illustre les relations entre les différentes classes du système :

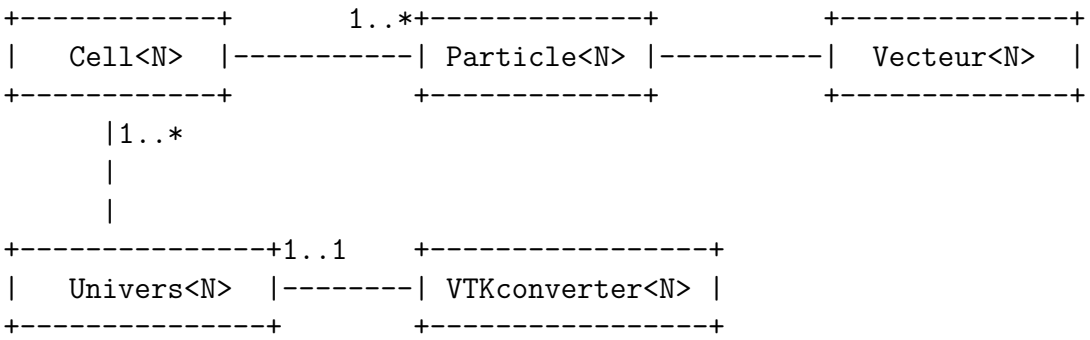

FIGURE 4 – Diagramme de classes UML du système

4 Implémentation

4.1 Classe Vecteur

La classe Vecteur représente un vecteur mathématique en N dimensions et implémente toutes les opérations vectorielles nécessaires pour la simulation :

```cpp
template <std::size_t N>
class Vecteur {
private:
    std::array<double, N> data;

public:
    Vecteur();
    Vecteur(const std::array<double, N>& values);
    double norm() const;

    // Operateurs mathematiques
    friend Vecteur<N> operator+(const Vecteur<N>& v1, const Vecteur<N>& v2);
    friend Vecteur<N> operator-(const Vecteur<N>& v1, const Vecteur<N>& v2);
    friend Vecteur<N> operator*(const Vecteur<N>& v, float f);
    friend Vecteur<N> operator/(const Vecteur<N>& v, double d);
};
```

Listing 1 – Extrait de la classe Vecteur

4.2 Optimisations de la classe Vecteur

Plusieurs micro-optimisations ont été implémentées dans la classe Vecteur pour maximiser les performances :

- **Allocation statique** : Utilisation de `std::array` au lieu de `std::vector` pour un accès plus rapide et une allocation sur la pile
- **Élimination des copies** : Tous les paramètres vectoriels sont passés par référence constante (`const&`)
- **Optimisations à la compilation** : Utilisation de `static_assert` pour vérifier les dimensions à la compilation plutôt qu'à l'exécution
- **Vérification d'auto-affectation** : Test `if (this != &other)` dans l'opérateur d'affectation pour éviter un travail inutile

4.3 Classe Particle

La classe Particle encapsule toutes les propriétés d'une particule et implémente les méthodes pour calculer les forces entre particules :

```cpp
template <std::size_t N>
class Particle {
private:
    int id;
    Vecteur<N> position;
    Vecteur<N> velocity;
    double mass;
    std::string category;
    Vecteur<N> force;
    Vecteur<N> old_force;

public:
    // Getters et setters
    int getId() const;
    const Vecteur<N>& getPosition() const;

    // Methodes de calcul des forces
    Vecteur<N> optimizedGetAllForces(Particle<N>* p,
                                     float epsilon_times_24,
                                     float sigma) const;

    // Methodes pour le partitionnement spatial
    std::array<int, N> getCellIndexofParticle(double
   cellLength) const;
};
```

Listing 2 – Extrait de la classe Particle

4.4 Optimisations de la classe Particle

La classe Particle implémente plusieurs optimisations pour les calculs de forces :

- **Protection contre les forces excessives** : Limitation des forces à des valeurs raisonnables pour éviter les instabilités numériques
- **Précalcul des facteurs constants** : Multiplication préalable d'epsilon par 24 (`epsilon_times_24`) pour éliminer cette multiplication dans chaque calcul
- **Réutilisation des calculs intermédiaires** : Les puissances et distances au carré sont calculées une seule fois puis réutilisées
- **Tolérance numérique** : Utilisation de `constexpr double alpha = 1e-2` pour éviter les divisions par zéro

4.5 Classe Cell

La classe Cell implémente le concept de partitionnement spatial pour optimiser les calculs de voisinage :

```
template <std::size_t N>
class Cell {
private:
    std::vector<Particle<N>*> particles;
    std::vector<std::array<int, N>> neighbourCellsIndex;
    std::array<int, N> cellIndex;
    double length;
    int numberOfParticles = 0;

public:
    // Construction et gestion des particules
    void addParticle(Particle<N>*& particle);
    void removeParticle(Particle<N>*& particle);

    // Acces aux donnees
    std::vector<Particle<N>*> getParticles() const;
    std::vector<std::array<int, N>> getNeighbourCellsIndex()
   const;
};
```

Listing 3 – Extrait de la classe Cell

4.6 Optimisations de la classe Cell

La classe Cell optimise le partitionnement spatial :

— **Précalcul des voisins** : Les indices des cellules voisines sont calculés une seule fois puis stockés

— **Test rapide d'appartenance** : Vérification optimisée avec `isEmpty()` utilisant `particles.empty()`

4.7 Classe Univers

La classe Univers est le cœur du système, gérant l'ensemble de la simulation :

```cpp
template <std::size_t N>
class Univers {
private:
    std::array<double, N> caracteristicLength;
    double cutOffRadius;
    std::array<int, N> numberOfCells;
    std::unordered_map<std::array<int, N>, Cell<N>*, ArrayHash<N>> cells;
    std::vector<Particle<N>*> particles;
    int nbParticles;

public:
    // Construction et configuration
    Univers(std::array<double, N> caracteristicLength, double cutOffRadius);

    // Gestion des particules
    void addParticle(Particle<N>*& particle);
    void updateParticlePositionInCell(Particle<N>* particle,
                                      const Vecteur<N>& newPosition);

    // Calcul des forces et mise a jour
    void computeAllForcesOnParticle(float epsilon, float sigma);
    void update(double dt, float epsilon, float sigma);

    // Gestion des conditions limites
    Vecteur<N> applyReflectiveLimitConditions(Particle<N>* particle,
                                              const Vecteur<N>& newPosition);
};
```

Listing 4 – Extrait de la classe Univers

4.8 Optimisations de la classe Univers

L'Univers, cœur du système, implémente plusieurs optimisations cruciales :

- **Calcul asymétrique des forces** : Comparaison des IDs (`particle->getId() < neighbourParticle->getId()`) pour calculer chaque interaction une seule fois, la force étant symétrique
- **Structures de données adaptées** : Utilisation d'une `unordered_map` avec une fonction de hachage personnalisée pour accéder rapidement aux cellules

5 Optimisations

5.1 Partitionnement spatial

Le partitionnement spatial divise l'espace de simulation en cellules, permettant de limiter les calculs de forces aux particules voisines. Cette optimisation réduit la complexité des calculs de $O(n^2)$ à $O(n)$ dans des systèmes avec une distribution uniforme de particules.

5.2 Flags de compilation

Le projet utilise des flags de compilation avancés pour maximiser les performances. Voici quelques exemples :

```
-O3                     # Optimisation de niveau le plus élevé
-march=native           # Optimisations spécifiques à l'architecture
-ffast-math             # Optimisations mathématiques agressives
-funroll-loops          # Optimisation des boucles
-ftree-vectorize        # Vectorisation automatique
```

6 Résultats et visualisation

Les simulations peuvent être visualisées grâce à l'export au format VTK, permettant l'utilisation d'outils comme ParaView pour analyser les résultats.

6.1 Exemple de simulation

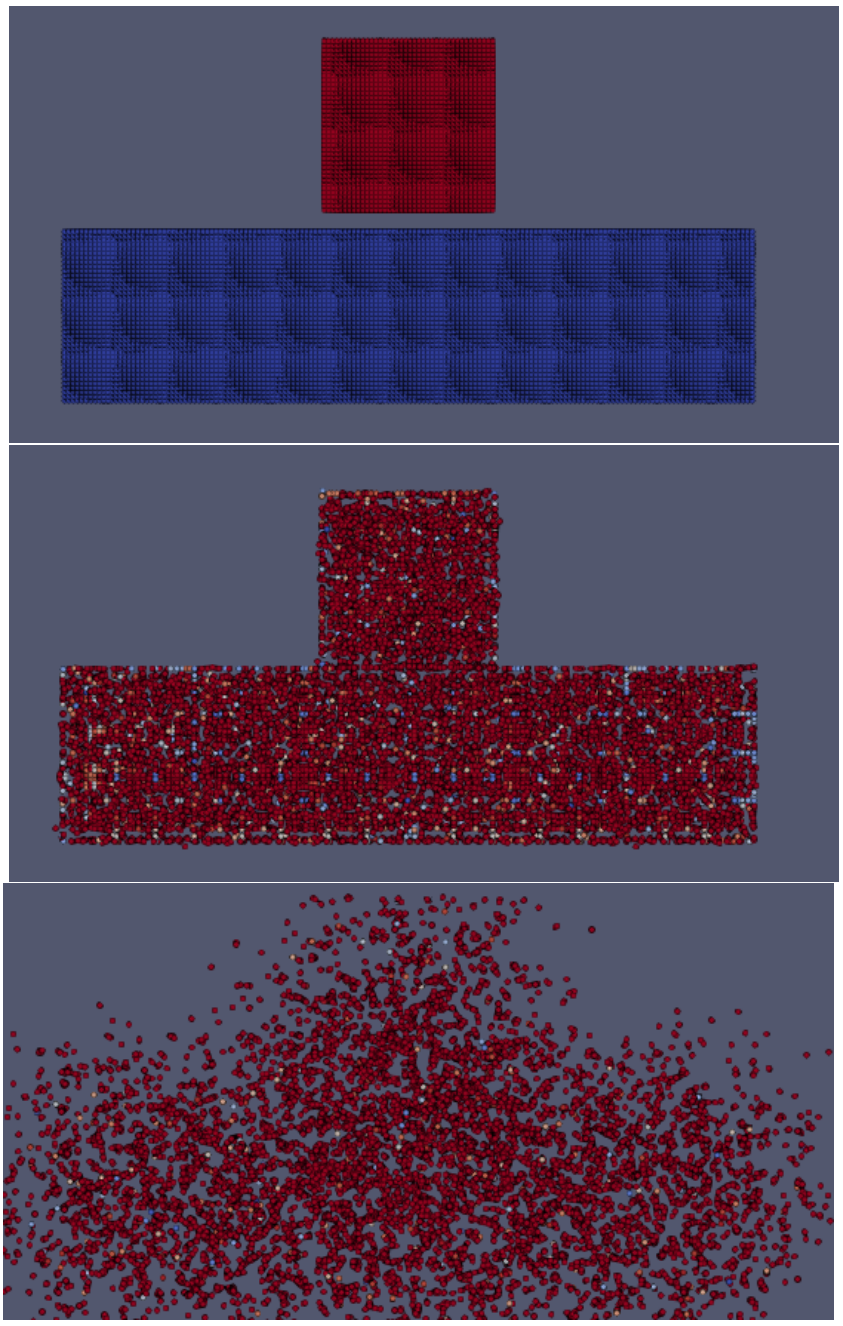

FIGURE 5 – Exemples de visualisation de simulation

7 Performance

7.1 Partitionnement spatial

Le partitionnement spatial permet de réduire le nombre de calculs nécessaires pour déterminer les forces entre particules. En divisant l'espace en cellules, seules les particules dans des cellules voisines sont considérées pour le calcul des forces, ce qui réduit considérablement le temps de calcul.

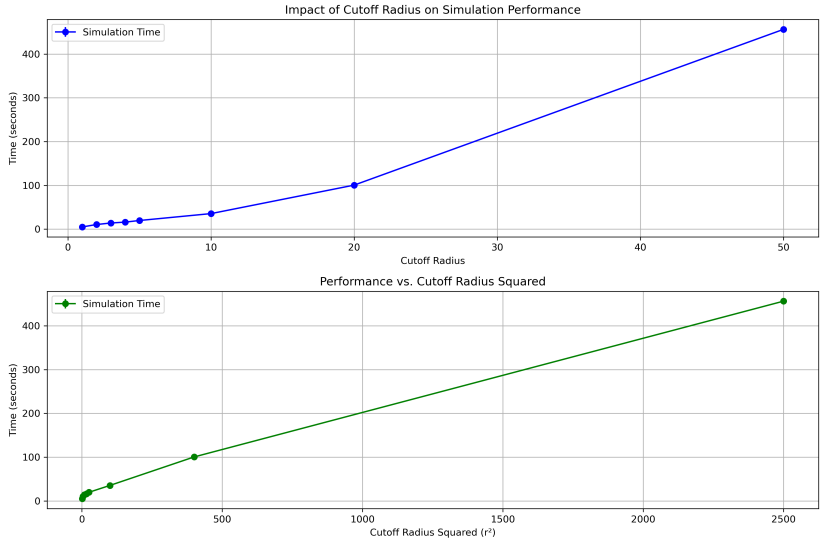

FIGURE 6 – Temps de simulation en fonction du rayon de coupure

```
Performance Summary:
Cutoff      Total Time      Setup Time      Simulation Time
------------------------------------------------------------
1.00        4.95            0.02            4.92
2.00        10.47           0.00            10.46
3.00        13.97           0.00            13.97
4.00        15.97           0.00            15.96
5.00        19.68           0.00            19.68
10.00       35.44           0.00            35.44
20.00       100.51          0.00            100.51
50.00       456.42          0.00            456.41
```

FIGURE 7 – Tableau de performance du partitionnement spatial

7.2 Impact de la génération des VTK

L'exportation des données au format VTK a un impact sur les performances dû au nombre important d'écriture dans des fichiers nécessaires, mais permet une visualisation efficace des résultats. Le temps d'exportation est proportionnel au nombre de particules et à la complexité de la scène.

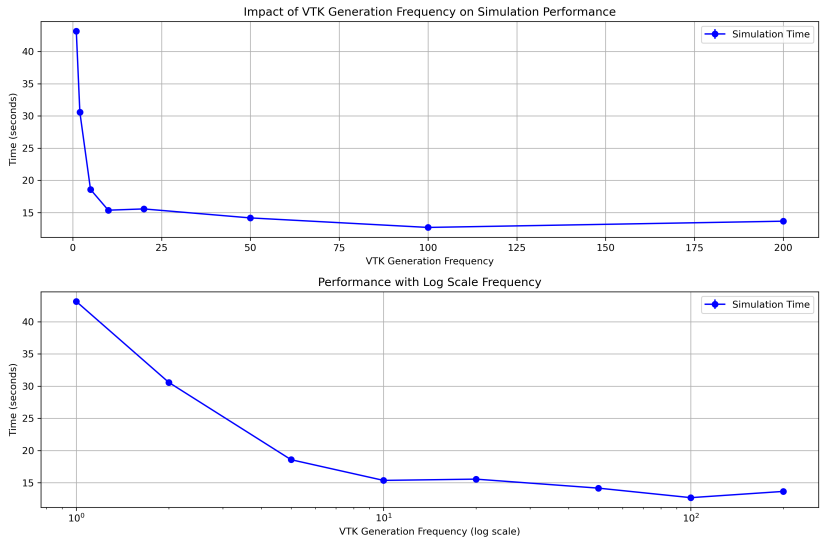

FIGURE 8 – Temps de simulation en fonction de la fréquence d'exportation VTK

```
Performance Summary:
Frequency  Total Time      Setup Time      Simulation Time
--------------------------------------------------------
1          43.15           0.00            43.15
2          30.59           0.00            30.58
5          18.59           0.00            18.58
10         15.37           0.00            15.37
20         15.57           0.00            15.56
50         14.17           0.00            14.17
100        12.69           0.00            12.69
200        13.66           0.00            13.66
```

FIGURE 9 – Tableau de performance de l'exportation VTK

8 Conclusion

Ce projet démontre l'efficacité d'une approche générique basée sur les templates C++ pour la simulation de particules en N dimensions. L'architecture modulaire et les techniques d'optimisation utilisées permettent de simuler efficacement des systèmes complexes avec un grand nombre de particules.

8.1 Perspectives

Plusieurs axes d'amélioration sont envisageables :
— L'utilisation du parallélisme pour accélérer les calculs
— Ajout de nouveaux modèles physiques (forces électrostatiques, etc.)