# Compte-rendu DM classique

CHOPIN Antonio

## Table des matières

# 1 Introduction

L'objectif de la manipulation est d'utiliser un code de dynamique moléculaire écrit en c ou en Fortran 90 (*md.c* ou *md.f90*) afin de **simuler une collision entre deux nano-particules d'Argon** nommées **A** et **B**.

Chacune de ces particules est composée de $\mathbf{N = 1000}$ atomes d'Argon chacun de **masse m**. La particule A se trouve à l'instant initial à la position 0 du repère. La particule B se trouve à une certaine distance $d_0$ de A et s'en rapproche à la vitesse initiale $v_0$.

La simulation se fait en **3D**. La boîte de simulation est un cube d'arête égale à 100000.

Pour cette simulation, on utilise les **unités réduites** $\sigma$, $\epsilon$ et $\sqrt{\frac{m\sigma^2}{\epsilon}}$ **de longueur, d'énergie et de temps.**
On a $\sigma = 3.4 \times 10^{-10}$ m, $\epsilon = 1.6 \times 10^{-21}$ J et $m = 6.7 \times 10^{-26}$

Les interactions entre particules sont modélisées via une version tronquée et décalée $V_{tr}(r_{ij})$ du potentiel de **Lennard-Jones** :

$$\boxed{V(r_{ij}) = 4\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^{6} \right]} \tag{1}$$

$$\boxed{=> V_{tr}(r_{ij}) = \begin{cases} V(r_{ij}) - V(R_C) & \text{si } r \leq R_C \\ 0 & \text{sinon} \end{cases}} \tag{2}$$

Où $r_{ij}$ représente la distance entre deux particules nommées i et j et $R_c$ est le rayon de troncature de $V(r_{ij})$

Ce document se décompose en trois parties : une première sur la prise en main de code et quelques ajouts de fonctionnalités sur la version *md.c*, une seconde axée sur des optimisations envisagées pour le script *md.f90* et leur implémentation. Enfin, une troisième sur les tests effectués et les résultats obtenus.

On fournit le dossier **Prise en main** contenant les codes .c relatifs à la première partie :

— *md_ part1.c*, contenant le code modifié pour la particule A seule
— *md_ part2.c*, contenant le code modifié pour la collision de A et B

Et le dossier **Optimisations** avec les code .f90 relatifs à la seconde partie :

— *md_ calcul_ distances.f90*, optimisé pour le calcul des distances
— *md_ PBC.f90*, contenant l'optimisation du calcul de hbox
— *md_ uc.f90*, contenant l'optimisation du calcul de uc
— *md_ Verlet.f90*, contenant l'optimisation avec la liste de Verlet

# 2 Collision de nano-particules

## 2.1 Prise en main du code

Dans cette section, on vérifie que le code fonctionne correctement puis on présente les résultats obtenus suite à des modifications précises. On se limite dans un premier temps à l'étude d'une seule nano-particule.

On travaille sur la version du code **md.c**
On utilise d'abord le fichier **nano_A.xyz** contenant les conditions initiales pour l'étude d'une seule particule.
On compile via la commande : **gcc md.c -lm**

### 2.1.1 Test préliminaires

Tout d'abord, on exécute le code fourni et brut afin de vérifier son bon fonctionnement. En particulier, on vérifie que la conservation de l'énergie a bien lieu au cours de la simulation.

Il n'y a pas d'erreur lors de la compilation. On exécute et on trace l'énergie totale en fonction du temps. On obtient la figure 1

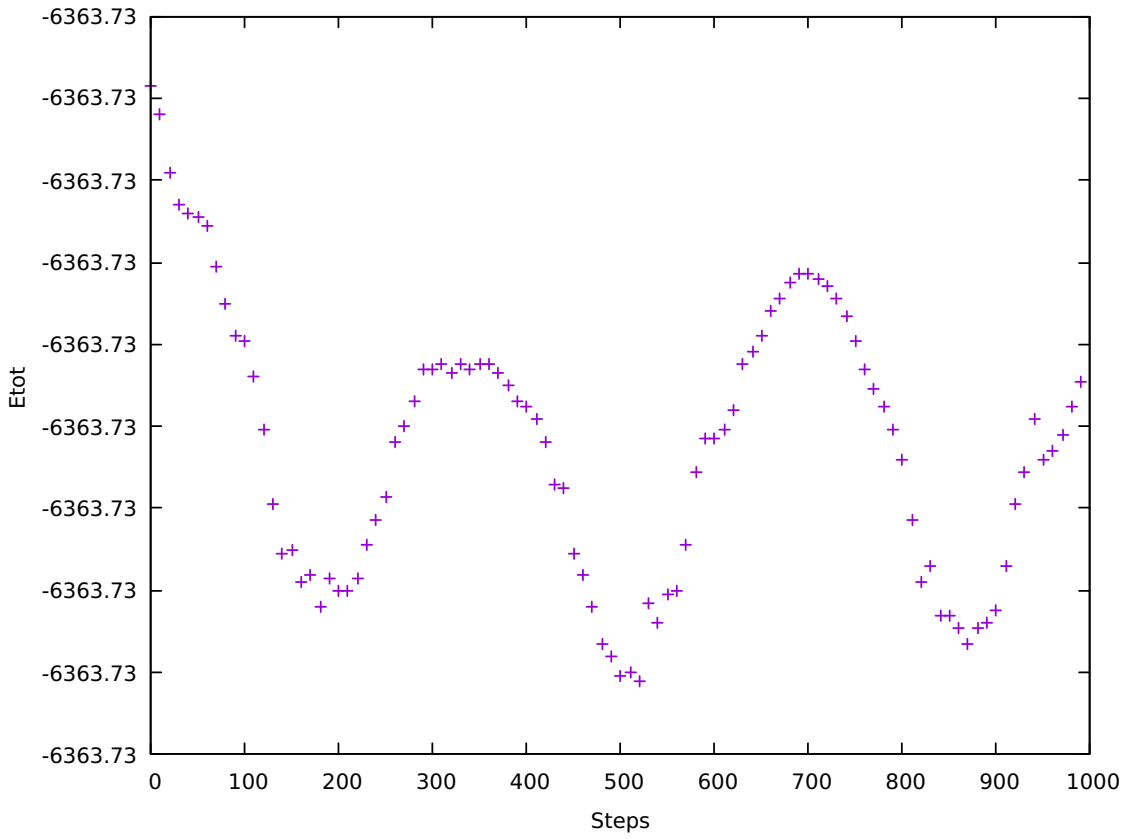


FIGURE 1 – Évolution de l'énergie totale au cours de a simulation brute

On observe que les variations de l'énergie totale sont négligeables (gnuplot n'affiche pas assez de décimales). Ces légères variations sont normales dans une simulation numérique.

Le code fonctionne donc correctement.

Dans le script du fichier, une fonction/routine nommée **pbc** (*Periodic Boundary Conditions*) permet d'apporter les corrections d'énergie liée aux conditions périodiques imposées aux bords. On voudrait constater l'importance de ces corrections sur l'énergie totale. Pour ce faire, on supprime tous les appels de cette fonction dans le code et on relance.

Si on retrace l'évolution de l'énergie totale, on obtient la figure 2

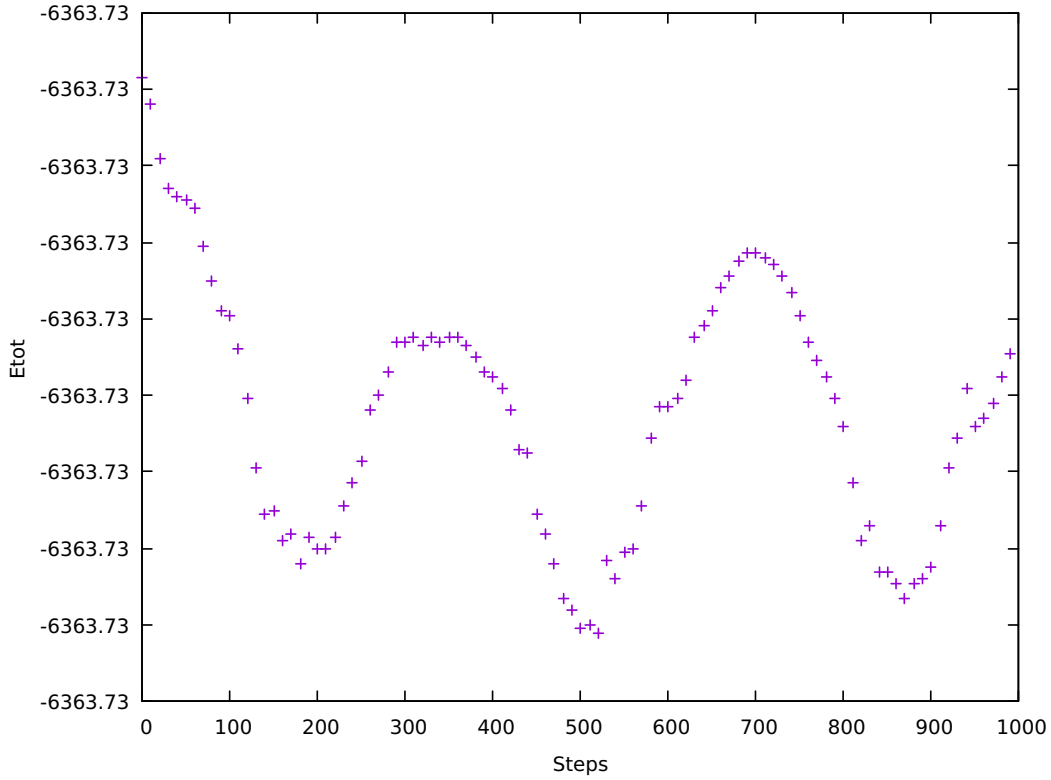


FIGURE 2 – Évolution de l'énergie totale sans les termes correctionnels liés aux conditions périodiques aux bords

On constate que la figure est identique à la première (1). S'il y a des différences, alors elles sont trop faibles pour être détectées. L'influence de ces corrections est donc négligeable.

Cela peut s'expliquer de différentes façons :

* La particule peut se trouver au centre de la boite de simulation et donc trop loin des bords pour que les corrections aient lieu, d'autant plus que le potentiel utilisé est tronqué et non pas infini.

* Si on regarde le fichiers **nano_A.xyz** contenant les conditions initiales du système, on voit que les vitesses initiales des atomes constituant la nano-particule sont nulles. Si la particule se trouve effectivement loin des bords, aucun mouvement d'ensemble ne peut l'en rapprocher.

## 2.2 Centre de masse

### 2.2.1 Théorie

Une grandeur permettant de facilement localiser la position d'un objet dans l'espace est le centre de masse.

Si on applique ce concept au cas étudié, on peut calculer le centre de masse d'une nano-particule à partir des postions des N atomes la constituant :

$$\boxed{\vec{R}_{CM} = \frac{1}{M} \sum_{i=1}^{N} \vec{r}_i} \tag{3}$$

$\vec{R}_{CM}$ est un grandeur vectorielle en 3 dimensions.

Ici, on ne s'intéresse qu'à la composante selon $\vec{e}_x$ du centre de masse d'une particule. Si on suit l'évolution de cette composante à chaque instant, on sera capables d'estimer la façon dont se déplace la nano-particule étudiée.

### 2.2.2 Implémentation

On veut donc étudier le mouvement de la particule étudiée au travers de la composante selon $\vec{e}_x$ de son centre de masse.

Pour ce faire, on implémente la fonction :

**Center_of_mass (int ndim, int n, double r[])**

Cette fonction prend 3 paramètres :

* La dimension du problème *ndim*
* Le nombre de particules *n*
* Le tableau contenant les positions des particules *r[]*

Le code de cette fonction est présenté en annexes B.1

On modifie ensuite la fonction **main()** et **report()** pour calculer cette grandeur à chaque itération et stocker la valeur obtenue dans le fichier de sortie.

Modifications effectuées, on exécute le code pour 1500 itérations. On trace alors la valeur de la composante selon x du centre de masse. On obtient la figure 3

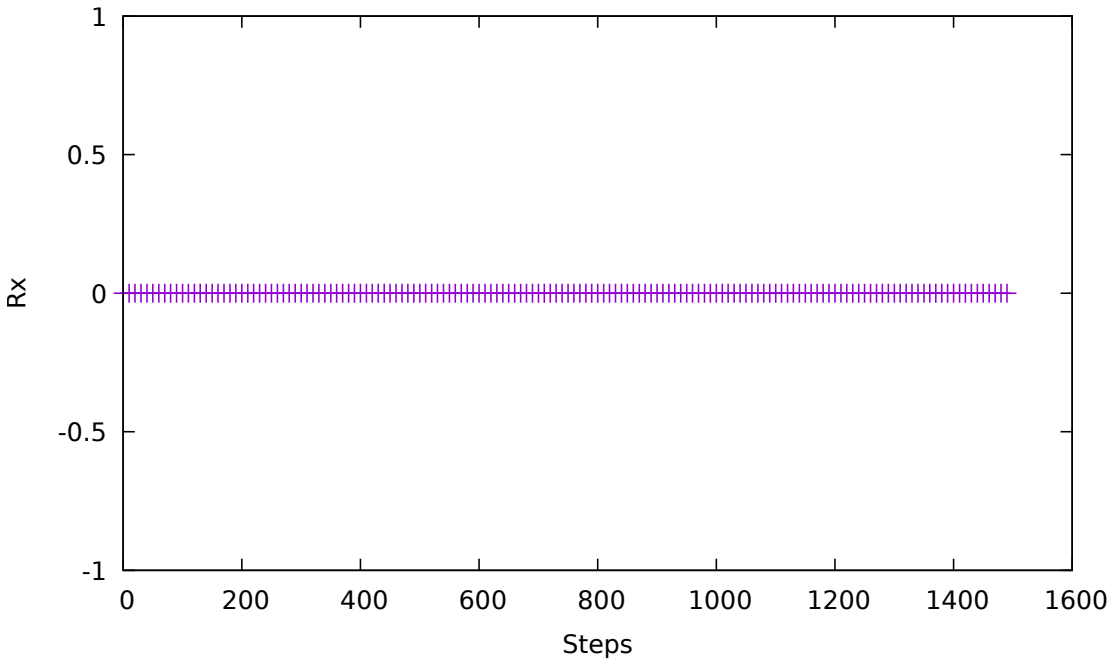


FIGURE 3 – Centre de masse de la particule A au cours du temps

On voit que la particule se trouve en 0 et ne bouge pas. Les vitesses initiales des atomes étant nulles, il n'y a pas de mouvement d'ensemble. La particule n'a donc pas de mouvement. Le comportement observé est donc correct.

### 2.2.3 Conclusion

Ces manipulations préliminaires ont permis d'une part de vérifier que le code fonctionne bien et d'autre part d'établir les concepts physiques de base pour étudier le mouvement d'une nano-particule.

On a désormais les capacités d'étudier le phénomène de collision entre deux nano-particules similaires à celles étudiée.

## 2.3 Rayon quadratique moyen

### 2.3.1 Théorie

Afin de d'estimer le rayon de la nano-particule, on peut calculer le rayon quadratique moyen. Il se calcul comme suit :

$$R_g = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\vec{r_i} - \vec{R}_{CM})^2} \tag{4}$$

Où $\vec{r_i}$ est la position de l'atome i et $\vec{R}_{CM}$ est le centre de masse de la particule formée par les N atomes.

$R_g$ est une grandeur **scalaire.**

On peut calculer $R_g$ avec les composantes selon $\vec{e}_x$ des positions et du centre de masse.

### 2.3.2 Implémentation

On crée la fonction **quadratic_radius()**

Elle prend 4 paramètres :

* *ndim*, la dimension
* *n*, le nombre de particules
* *r[]*, tableau contenant les positions des atomes
* *Rx*, le centre de masse selon x

Le code est donné en annexes (section B.2)

On adapte le code afin de conserver les valeur du rayon quadratique à chaque itération et pouvoir ensuite le visualiser.

On obtient la figure 4

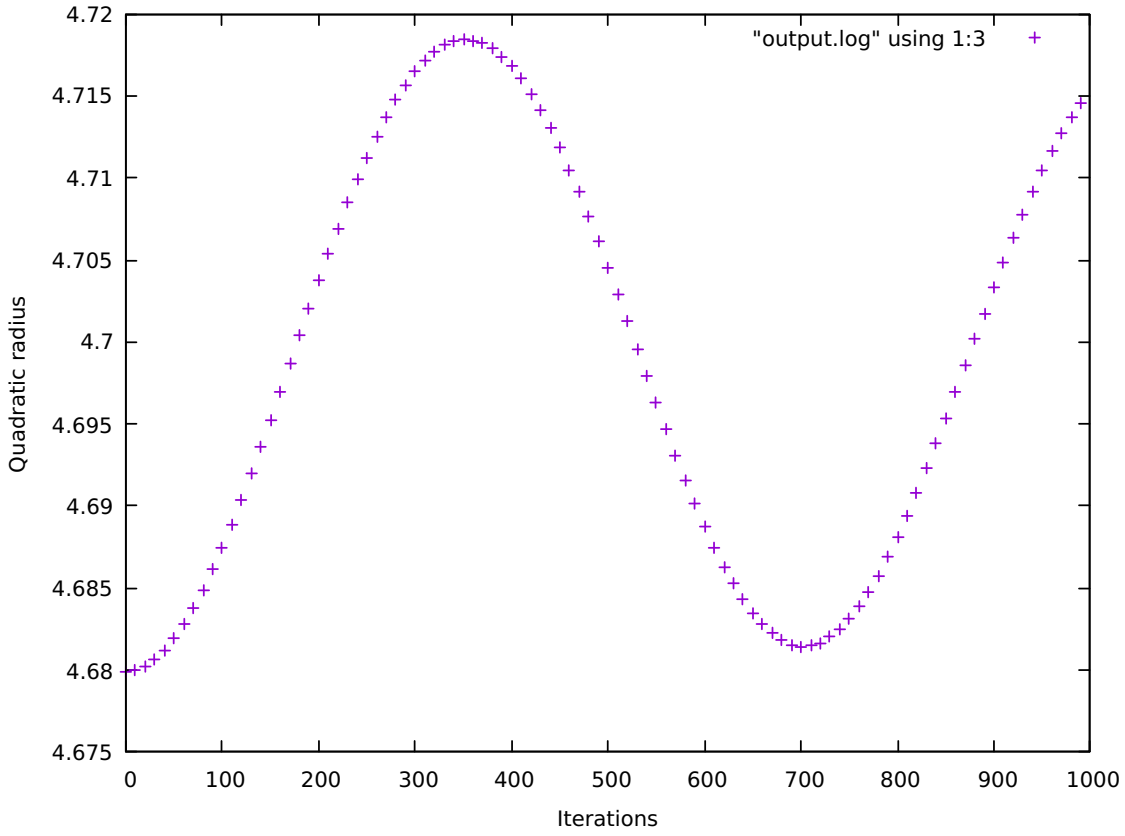


FIGURE 4 – Evolution du rayon quadratique moyen au cours de la simulation

On observe des oscillations de la valeur du rayon moyen autour de 4.7. On en conclut que la nano-particule a un rayon d'environ $4.7\sigma$, soit $4.7 \times 3.4 \times 10^{-10} \approx 10^{-9}$ m.

L'appellation "nano" est donc bien justifiée. On comprend également que les forces entre atomes provoquent des déformations régulières de la nano-particule qui n'a pas un rayon strictement constant.

## 2.4 Collision de 2 nano-particules

On veut désormais étudier la collision de deux nano-particules d'Argon et en particulier comprendre le comportement d'après choc.

On ré-utilise le code **md.c**
Les conditions initiales du système sont récupérées du fichier **nano_A_B.xyz**

### 2.4.1 Adaptations du code

On étudie le mouvement des deux nano-particules grâce à la composante selon $\vec{e_x}$ de leurs centres de masse $Rx_A$ et $Rx_B$. On adapte la fonction **Center_of_mass()** et le code afin de calculer ces deux grandeurs à chaque itération et stocker leurs valeurs dans le fichier de sortie (voir annexes B.3).

La nouvelles fonction prend en paramètres :

* La dimension du problème *ndim*
* Le début des itérations $n_start$
* La fin des itérations $n_end$
* Le tableau contenant les positions des particules *r[]*

Ces adaptations permettent d'envoyer en plusieurs fois le tableau *r[]* contenant les positions et vitesses des particules tout en adaptant la zone à parcourir.

### 2.4.2 Conditions initiales et résultats

Le fichier des conditions initiales permet de voir que la norme de la vitesse initiale de la particule B est $v_0 = 0.6$.

Si on calcule la valeurs absolue de la différence entre les deux centres de masse $|Rx_A - Rx_B|$ à l'instant 0, on voit que la particule B se trouve à une distance algébrique $d_0 = 20$ de la particule A (elle-même située à l'origine du repère).

On lance la simulation de la particule B entrant en collision avec A. Afin que la collision ait lieu, on calcule sur $N = 17770$ itérations.

On obtient la figure 5

On voit que :

* La particule A est originalement à la position 0 selon x et n'a pas de mouvement.

* La particule B se trouve à -20 selon x et est animée d'une vitesse $v_0$ initiale la faisant se rapprocher de A.

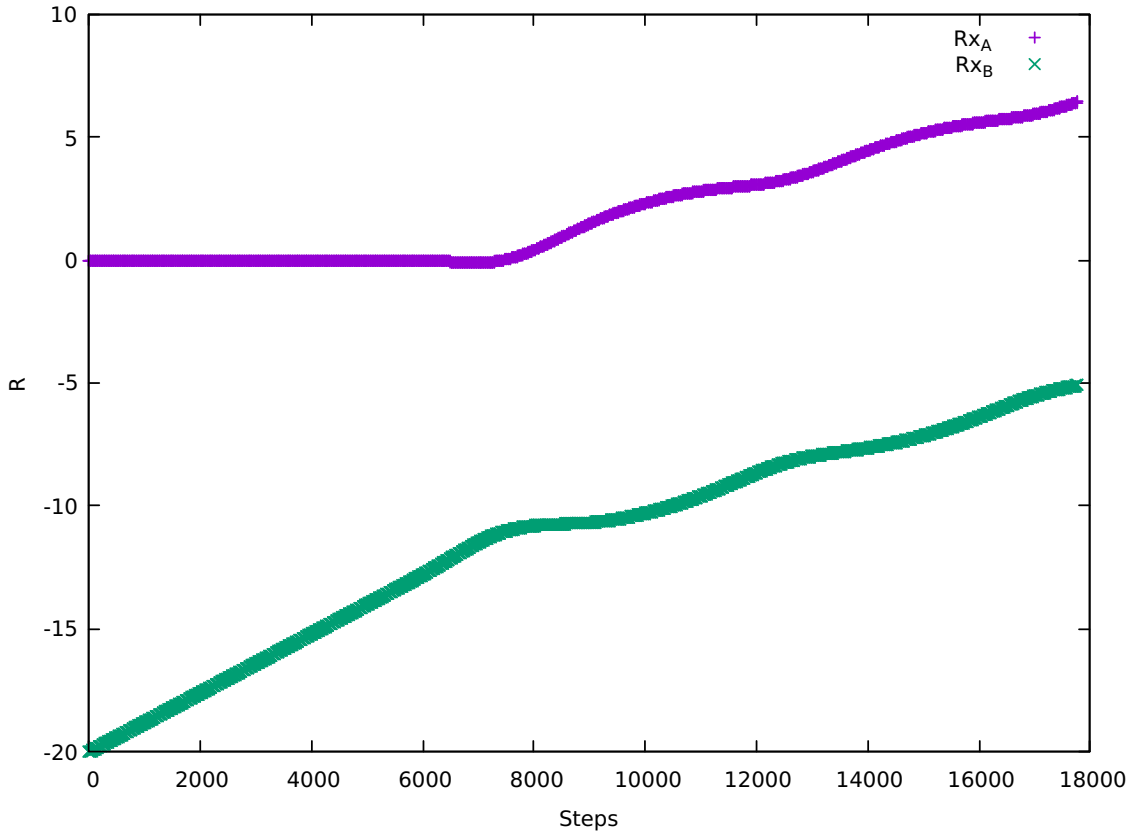


FIGURE 5 – Évolution des centres de masse de A et B, collision et évolution post-collision

* La collision a lieu environ à la 6600ème itération.

* La distance séparant les particules au moment et suite à la collision est égale au double du rayon d'une nano-particule.

* La particule B a transféré un mouvement à la particule A.

* Suite à la collision, les deux centres de masse semblent se coordonner et avoir un mouvement global commun.

### 2.4.3 Conclusion

Les résultats traduisent bien le déplacement de B et sa collision avec A.

La figure 5 permet de supposer que les deux particules se sont peut-être liées entre elles. Cela expliquerait le mouvement d'ensemble et similaire des deux centres de masse suite à la collision. Ou alors elles se sont désintégrées lors du choc mais de façon assez régulière pour que les centres de masses restent à peu près stabkes.

La figure 5 permet d'estimer le rayon des nano-particules à environ $5\sigma$. C'est cohérent avec la valeur moyenne trouvée pour le rayon quadratique moyen.

# 3 Optimisation

Cette section décrit l'implémentation de plusieurs types d'optimisations pouvant être apportées au code de simulation.

On en présente 4 :

* Optimisation du calcul des distances
* Optimisation de l'invariant $\frac{box}{2}$
* Optimisation de l'invariant $uc$
* Optimisation par la liste de Verlet

On travaille avec le code **md.f90**
On lance avec les fichiers **input.xyz** ou **input_N4096.xyz**
Les codes sont compilés grâce à **gfortran**

On étudie deux cas : **le cas de 512 particules** et celui d**e 4096 particules**

## 3.1 Code fourni brut

Pour commencer, on effectue quelques manipulations sur le code fourni dans afin d'avoir des valeurs de référence.

A l'exception de la liste de Verlet, les optimisations apportées ne modifieront pas les conditions physiques et donc n'impacteront pas les résultats obtenus. Elles n'auront d'influence que sur le temps d'exécution du script. Leur efficacité (ou non) sera basée sur ce critère. On pourra vérifier leur bon fonctionnement en s'assurant qu'on obtient bien les mêmes résultats après chaque modification apportée.

Pour Verlet, on s'assurera que l'énergie totale reste à peu près constante autour d'une valeur moyenne proche de celle d'avant modification.

On utilise la commande **time** lors de l'exécution des fichiers afin d'obtenir le temps total $t_{real}$ et le temps de calcul (par un seul processeur) $t_{user}$.

L'unité des temps mesurés est la **seconde**.

### 3.1.1 Cas 512 particules

On exécute le fichier **dm.f90** pour $\boxed{N = 2000}$ itérations.

Les temps d'exécution et de calcul moyens en secondes sur 5 exécutions sont :

$$\boxed{t_{real} = 10.068} \quad \boxed{t_{user} = 9.996}$$

De plus, on vérifie que l'énergie totale du système est conservée. on obtient la figure 6 (annexes section A)

On peut donc voir que la simulation fonctionne correctement.

### 3.1.2 Cas 4096 particules

On exécute le fichier **dm.f90** pour $\boxed{N = 100}$ itérations.

Les temps d'exécution et de calcul moyens en secondes sur 3 exécutions sont :

$$\boxed{t_{real} = 28.171} \quad \boxed{t_{user} = 27.996}$$

De plus, on vérifie que l'énergie totale du système est conservée. on obtient la figure 7 (annexes section A)

On peut donc voir que la simulation fonctionne correctement.

## 3.2 Optimisations envisagées et implémentation

On explique ici le principe des optimisations apportées.

On précise qu'à chaque fois, l'évolution de l'énergie totale au cours du temps a été tracée pour le cas de 502 et 4096 particules. On retrouve bien les figures 6 et 7.

Pour la liste de Verlet, la vérification de la conservation et de la cohérence des valeurs pour l'énergie totale sont données en annexes (A).

### 3.2.1 Optimisation du calcul des distances

On exploite la localité temporelle des données en conservant la position d'une particule i dans un tableau temporaire afin d'éviter un temps de recherche superflu dans la boucle parcourant les autres particules.

On modifie la routine **forces** en y ajoutant un tableau local permettant d'enregistrer les positions i. Le code se trouve en annexes section C.1

### 3.2.2 Optimisation des PBC

Dans l'application des conditions aux limites périodiques, il y a un invariant. Autrement dit, une valeur qui ne change pas. Cependant, elle est re-calculée à chaque tour, ce qui est peu efficace.

On la calcule une seule fois dans la fonction main puis on la passe en paramètre à la routine nommée **pbc**. Le code se trouve en annexes section C.2

### 3.2.3 Optimisation du calcul du potentiel

Comme pour l'optimisation PBC, il y a un invariant : uc. C'est l'énergie permettant de décaler le potentiel de Lennard-Jones tronqué.

On calcule donc uc dans main une unique fois et on le passe ensuite en paramètres de la routine **potential**. Le code se trouve en annexes section C.3

### 3.2.4 Liste de Verlet

L'intérêt principale de cette optimisation est de réduire le coût de calcul. On associe à chaque particule i des N particules du système une liste de particules qui seront les seules à être testées lors du calcul des forces (Au lieu de tester la totalité des N-1 particules restantes).

Ces particules se trouvent dans un rayon compris entre le rayon de coupure du potentiel $r_{cut}$ et un rayon maximal $r_{max}$ tels que :

$$\boxed{r_{max} = r_{cut} + r_{skin}.} \tag{5}$$

Avec $r_{skin}$ donné.

On choisit pour les simulations menées $\boxed{r_{max} = 3}$

La fonction **main** est adapté afin de prendre en compte cette amélioration. Le code modifié se trouve en annexes section C.4

# 4 Tests et comparaisons des optimisations

Les différentes versions du code optimisé prêtes, on peut désormais les exécuter et comparer les temps d'exécution et de calcul aux valeurs de référence trouvées précédemment (sections 3.1.1 et 3.1.2).

On teste les effets pour 502 et 4096 particules.

## 4.1 Effet des optimisations sur le cas de 512 particules

On exécute le fichier **dm.f90** avec le fichier **input.xyz**.

On fait à chaque fois la moyenne sur 5 exécutions.

### 4.1.1 Optimisation du calcul des distances

$t_{real} = \frac{10.074+10.158+10.078+9.778+10.095}{5} = 10.036$ s

$t_{user} = \frac{10.009+10.099+10.006+9.714+9.996}{5} = 9.965$ s

Soit des temps similaires à plus de 99% aux temps de référence.

### 4.1.2 Optimisation des PBC

$t_{real} = \frac{9.558+9.726+9.583+9.744+9.908}{5} = 9.704$ s

$t_{user} = \frac{9.497+9.653+9.510+9.664+9.830}{5} = 9.631$ s

Soit des temps similaires à plus de 96% aux temps de référence.

### 4.1.3 Optimisation calcul de uc

$t_{real} = \frac{9.890+9.837+9.727+9.732+9.779}{5} = 9.793$ s

$t_{user} = \frac{9.814+9.757+9.645+9.650+9.666}{5} = 9.706$ s

Soit des temps similaires à plus de 97% aux temps de référence.

### 4.1.4 Liste de Verlet

$t_{real} = \frac{3.28+3.600+3.197+3.440+3.402}{5} = 3.383$ s

$t_{user} = \frac{3.204+3.571+3.169+3.410+3.354}{5} = 3.347$ s

Soit un gain de près de 66% par rapport aux temps de référence.

## 4.2 Effet des optimisations sur le cas de 4096 particules

On exécute le fichier **dm.f90** avec le fichier **input_N4096.xyz** pour 100 itérations.

On fait ç chaque fois la moyenne sur 3 exécutions :

### 4.2.1 Optimisation du calcul des distances

$t_{real} = \frac{27.691+27.486+27.797}{3} = 27.658$ s

$t_{user} = \frac{27.581+27.389+27.686}{3} = 27.552$ s

Soit des temps similaires à plus de 98% aux temps de référence.

### 4.2.2 Optimisation des PBC

$t_{real} = \frac{27.469+27.362+27.482}{3} = 27.438$ s

$t_{user} = \frac{27.360+27.237+27.299}{3} = 27.299$ s

Soit des temps similaires à plus de 96% aux temps de référence.

### 4.2.3 Optimisation calcul de uc

$t_{real} = \frac{27.878+27.986+28.619}{3} = 28.610$ s

$t_{user} = \frac{27.789+27.882+28.527}{3} = 28.066$ s

Soit des temps similaires supérieurs aux temps de référence.

### 4.2.4 Liste de Verlet

$t_{real} = \frac{3.256+3.197+3.178}{3} = 3.210$ s

$t_{user} = \frac{3.210+3.145+3.133}{3} = 3.163$ s

Soit un gain en temps de près de 88% par rapport aux temps de référence.

### 4.3 Conclusion

Pour le cas de 502 et de 4096 particules, l'optimisation du calcul des distances et des invariants, les résultats ne sont pas significatifs. Le gain de temps est de quelques centièmes de secondes au mieux, voire même contre-productifs dans certains cas. On ne peut pas conclure que ces modifications fassent une réelle différence. Et si elle existe, alors elle est négligeable.

La liste de Verlet, en revanche, apporte un gain très significatif. C'est de loin très efficace par rapport aux autres optimisations testées. Les gains en temps de calcul et d'exécution sont toujours supérieurs à 65%. De plus, les modifications impactent peu les résultats qui restent assez similaires entre avant et après modifications. La liste de Verlet est donc très efficace.

## 5 Conclusion générale

Les travaux effectués ont permis de prendre en main un code de dynamique moléculaire et d'étudier superficiellement la collision de deux nano-particules d'Argon. On a vu qu'à partir de l'une des composantes des centres de masse il est possible de visualiser le mouvement des particules et même d'estimer leur rayon. Le rayon quadratique a permis de valider le rayon estimé graphiquement avec le centre de masse et de constater un comportement de déformations périodiques de la molécule indétectable avec le seul centre de masse.

La plupart des optimisations apportées n'ont pas permis de démontrer leur réelle efficacité. La différence entre avant et après est faible, voire négligeable. On peut cependant dire qu'elle sont à priori bien implémentées car on retrouve les mêmes résultats après modification des codes. Peut-être que la machine utilisée pour mesurer les temps n'était pas optimale.

Le code incorporant la liste de Verlet est beaucoup plus concluant. En effet, les gains de temps sont de l'ordre de la seconde, voire de la dizaine de secondes. Cette optimisation est donc efficace. Elle dépend du rayon maximum choisi cependant. Plus ce rayon est grand, plus les résultats obtenus se rapprocheront de ceux d'avant modifications. Enfin, on peut dire que l'optimisation de la liste de Verlet est à priori bien implémentée car l'énergie totale varie autour de valeurs proches de celles d'avant modifications (voir annexes section A).

# A Figures

## A.1 Énergie totale - 502 particules

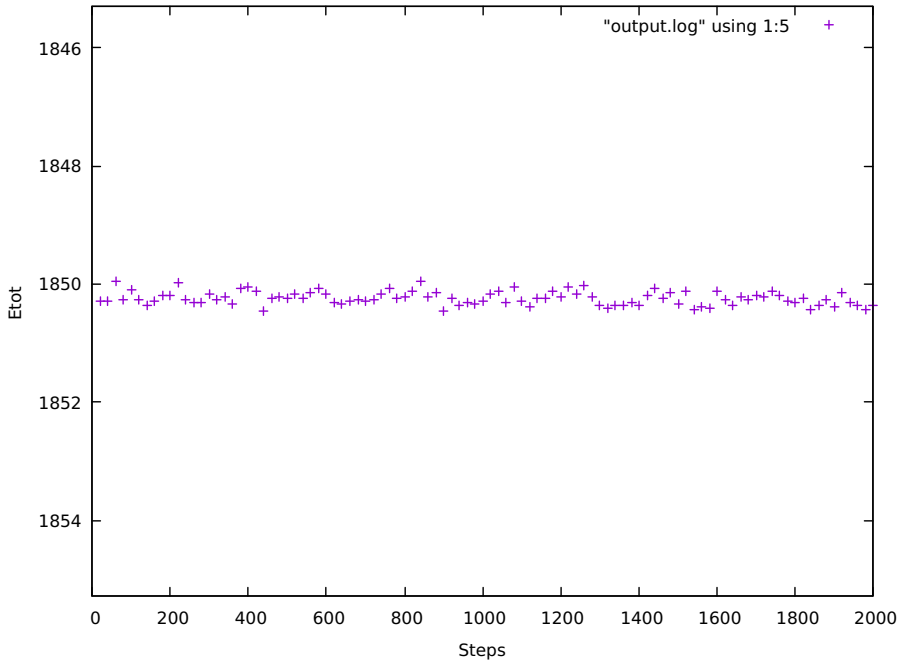


FIGURE 6 – Énergie totale du fichier d'origine (512 particules)

## A.2 Énergie totale - 4096 particules

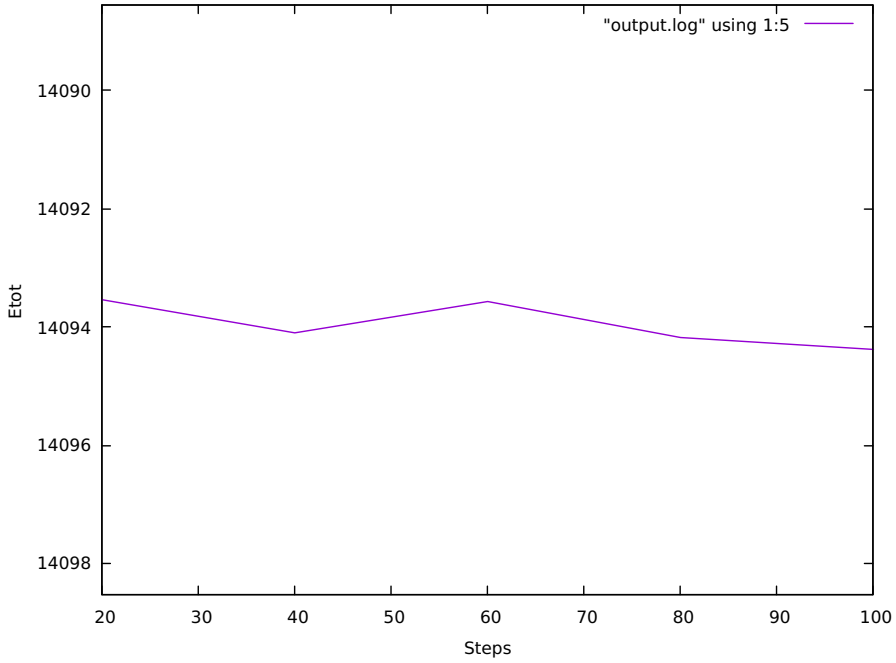


FIGURE 7 – Énergie totale du fichier d'origine (4096 particules)

## A.3 Énergie totale - 502 particules avec Verlet

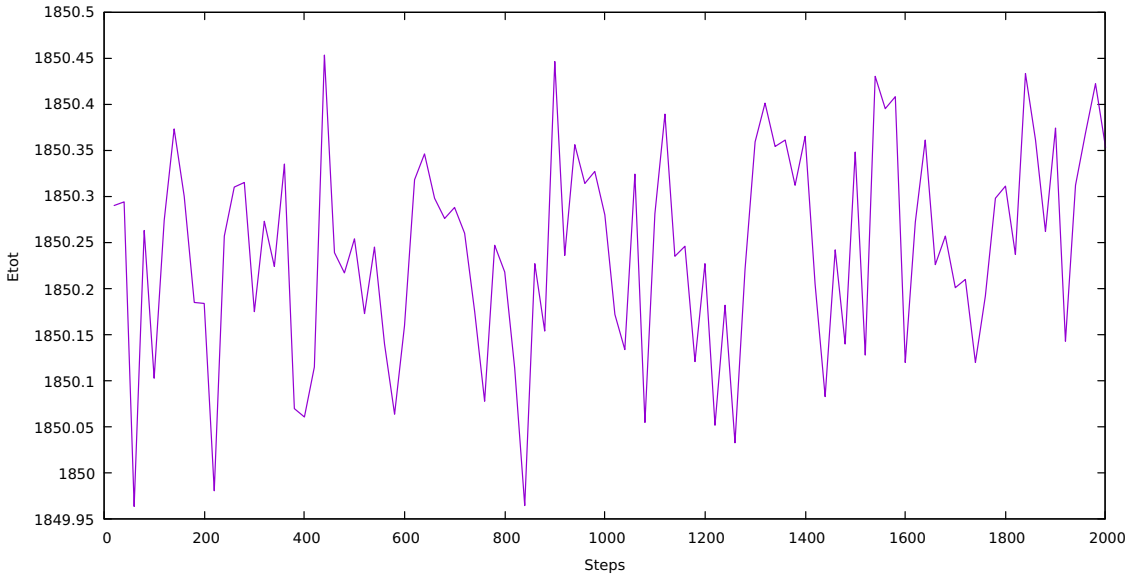


FIGURE 8 – Énergie totale du fichier d'origine (512 particules)

## A.4 Énergie totale - 4096 particules avec Verlet

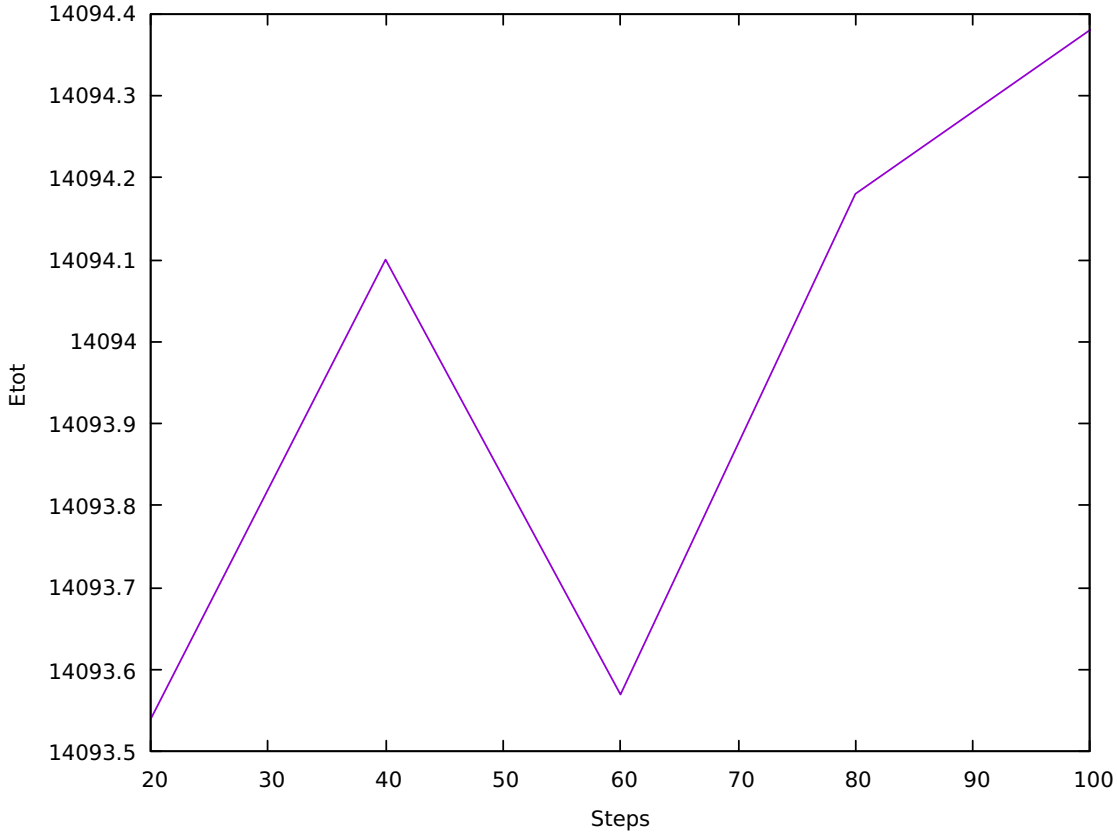


FIGURE 9 – Énergie totale du fichier d'origine (512 particules)

# B Fonctions utilisées dans les codes c modifiés

## B.1 Centre de masse - Particule unique

```c
/* Evaluates the center of mass of the nano-particle */
double Center_of_mass (int ndim, int n, double r[]) {

  int i, j;
  double Rx = 0.0;

  for (j=0; j<n; j++) {
    for (i=0; i<ndim; i++) {
      Rx += r[j*ndim+i];
    }
  }

  return Rx/n;

}
```

## B.2 Rayon quadratique moyen - Particule unique

```c
/*Evaluates the quadratic radius*/
double quadratic_radius(int ndim, int n, double r[], double Rx)
{
  int i, j;
  double Rg = 0;

  for (j=0; j<n; j++) {
    for (i=0; i<ndim; i++) {
      Rg += (r[j*ndim+i] - Rx)*(r[j*ndim+i] - Rx);
    }
  }

  return sqrt(Rg/n);
}
```

## B.3 Centre de masse - Deux particules

```c
/* Evaluates the center of mass of the nano-particle */
double Center_of_mass (int ndim, int n_start, int n_end, double r[]) {

  int i, j;
  double Rx = 0.0;

  for (j=n_start; j<n_end; j++) {
    for (i=0; i<ndim; i++) {
      Rx += r[j*ndim+i];
    }
  }

  return Rx/(n_end-n_start);
}
```

# C Codes d'optimisations pour les codes f90

## C.1 Optimisation du calcul des distances

```
! Evaluate the potential energy and the forces between particles
  subroutine forces(rcut,box,pos,for,epot)
    real(DP), intent(in)  :: box(:), rcut
    real(DP), intent(in)  :: pos(:,:)
    real(DP), intent(out) :: for(:,:)
    real(DP), intent(out) :: epot
    real(DP)              :: rij(size(pos,1)), rijsq, uij, wij
    integer               :: i, j

    ! Déclaration tableau temporaire
    real(dp), allocatable :: ri(:)

    for = 0.0_DP
    epot = 0.0_DP

    do i = 1,size(pos,2)

       ! Optimisation calcul des distances avec tableau temporaire
       ri = pos(:,i)

       do j = i+1,size(pos,2)
          rij = ri - pos(:,j)
          call pbc(rij,box)
          rijsq = dot_product(rij,rij)
          if (rijsq < rcut**2) then
             call potential(rcut,rijsq,uij,wij) ! wij = -(du/dr)/r
             epot = epot + uij
             for(:,i) = for(:,i) + wij * rij
             for(:,j) = for(:,j) - wij * rij
          end if
       end do
    end do
  end subroutine forces
```

## C.2 Invariant PBC

```
! Apply minimum image convention to a distance vector r.
  ! This subroutine can also be used to fold a particle back
  ! in the central cell during a simulation. However, it won't
  ! correctly fold back a particle from an arbitrary position.
  subroutine pbc(r,box, hbox)
    real(DP), intent(inout) :: r(:)
    real(DP), intent(in)    :: box(:), hbox(:)
    where (abs(r) > hbox)
       r = r - sign(box,r)
    end where
  end subroutine pbc
```

## C.3 Invariant uc

```
! Evaluate the potential energy and the virial contribution
  ! using the cut and shifted Lennard-Jones potential
  subroutine potential(rcut,rsq,uc,u,w)
    real(DP), intent(in)  :: rsq, rcut, uc
    real(DP), intent(out) :: u, w

    u = 4 * (1/rsq**6 - 1/rsq**3) - uc
    w = 24 * (2/rsq**6 - 1/rsq**3 ) / rsq
```

```
  end subroutine potential
```

## C.4 Fonction main modifiée pour liste de Verlet

```
program main

  use kernels
  USE neighbors_module

  implicit none

  ! Declaration variables
  real(DP),allocatable :: pos(:,:), vel(:,:), for(:,:), box(:)
  integer, allocatable  :: neighbors_number(:), neighbors_list(:,:)
  integer              :: unit_inp=100, unit_log=101, max_neighbors,
   period_neighbors = 15;
  integer              :: i, nsteps=1000, period_log=10, nv
  character(256)       :: file_inp="", file_log="output.dat"
  real(DP)             :: dt=0.002_DP, epot, rcut=2.5_DP, dum, rmax =
   3.0_DP

  call input(file_inp,file_log,nsteps,dt,period_log)
  open(unit=unit_inp,file=file_inp,status="old")
  open(unit=unit_log,file=file_log,status="unknown")

  ! Read input configuration and allocate arrays
  call read(unit_inp,box,pos,vel)
  allocate(for(size(pos,1),size(pos,2)))

  ! Dump to log file
  write(unit_log,"('# Number of particles =',i15)") size(pos,2)
  write(unit_log,"('# Box side            =',f15.3)") box(1)
  write(unit_log,"('# Time step           =',es15.3)") dt
  write(unit_log,"('#')")
  call report("header",unit_log,i,box,pos,vel,epot)

  ! Main MD loop
  call compute_neighbors(rmax,box,pos,neighbors_number,neighbors_list)
  call forces_with_neighbors(rcut,box,pos,neighbors_number,
   neighbors_list,for,epot)
  do i = 1,nsteps
     call evolve_with_neighbors(dt,rcut,box,pos,neighbors_number,
   neighbors_list,vel,for,epot)
     if (mod(i,period_neighbors)==0) call compute_neighbors(rmax,box,pos
   ,neighbors_number,neighbors_list)
     if (mod(i,period_log)==0) call report("log",unit_log,i,box,pos,vel,
   epot)
  end do

  close(unit_inp)
  close(unit_log)

end program main
```