Project V: MD - NTV of a Lennard-Jones System

Andrea Belli Contarini

Computing Methods for Physics - MSc Physics

Sommario

Il progetto prevede lo sviluppo di un codice di dinamica molecolare in C++ per simulare un sistema di particelle interagenti nell'*ensemble* canonico (NTV). Utilizzando l'algoritmo reversibile di Tuckerman [**1**], sono state eseguite simulazioni per diversi valori di densità e temperatura. I risultati ottenuti sono stati confrontati con quelli della letteratura [**2**], dimostrando un accordo soddisfacente e confermando l'efficacia del metodo proposto.

1 Introduzione

La dinamica molecolare (MD) è una tecnica computazionale per lo studio dei sistemi molecolari a livello atomico, che fornisce una descrizione dettagliata delle interazioni e dei movimenti delle particelle nel tempo.

Questo progetto si propone di sviluppare un codice di MD in linguaggio C++ per simulare un sistema di particelle interagenti nell'*ensemble* canonico (NTV). In particolare, il codice sviluppato simulerà un numero N_p di particelle in una scatola cubica di volume $L^3 = V$, con interazioni descritte dal potenziale di Lennnard-Jones, troncato con un raggio di *cutoff* pari a $r_c = 2.5\sigma$.

L'algoritmo reversibile proposto da Tuckerman sarà utilizzato per simulare il moto delle particelle, mantenendo costanti il numero di particelle N_p, il volume V e la temperatura T del sistema durante la simulazione. Le simulazioni verranno eseguite per valori specifici di temperatura ridotta $T^* = \frac{k_B T}{\epsilon} = 1.4$, e densità ridotta $\rho^* = \frac{\sigma^3 N_p}{V} = 0.1, 0.2, 0.3, 0.4$, al fine di calcolare la pressione e l'energia del sistema.

Questo studio mira a fornire una comprensione dettagliata del comportamento termodinamico del sistema e a confrontare i risultati ottenuti con quelli riportati nell'articolo di Johnson [**2**].

1.1 Potenziale di Lennnard-Jones

Date particelle interagenti di diametro σ, il potenziale di Lennard-Jones è definito come:

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

in cui ϵ è l'energia di profondità del potenziale e $r \coloneqq r_{ij} = r_i - r_j$. Si noti come esso sia composto da una parte attrattiva (negativa - linea tratteggiata blu nel grafico) ed una repulsiva (positiva, in rosso); il suo andamento è riportato nella figura 1 sottostante.

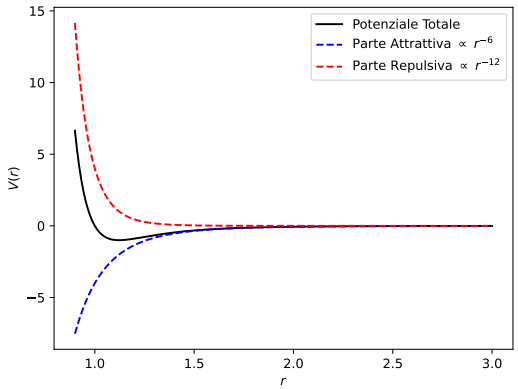

Figura 1: Il potenziale di Lennard-Jones, definito in eq. (1).

Per un potenziale come quello di Lennard-Jones, è possibile definire un raggio di troncamento (*cutoff radius*) $r_c \leq \frac{L}{2}$, al fine di considerare l'interazione di una particella solo con quelle che si trovano a una distanza inferiore a r_c. Oltre r_c il potenziale sarà dunque considerato nullo. Tuttavia, questa procedura introduce un errore nei calcoli di osservabili come energia e pressione; tale errore deve essere corretto tramite le "correzioni di coda".

1.2 Algoritmo di Tuckerman

L'algoritmo introdotto da Tuckerman et al. nel 1992, rappresenta un importante sviluppo nel campo delle simulazioni di dinamica molecolare. Utilizzando la fattorizzazione di Trotter del propagatore di Liouville $\hat{\mathcal{L}}$, gli autori hanno sviluppato integratori di dinamica molecolare reversibili che accelerano significativamente le simulazioni di sistemi con scale temporali multiple o forze a lungo raggio. Questi nuovi algoritmi sono reversibili nel tempo e mantengono tutti i vantaggi degli integratori RESPA (*Reference System Propagator Algorithm*) o NAPA (*Nonlinear Asynchronous Parallel Algorithms*), ma sono risultati più stabili [**1**]. L'approccio inizia con la decomposizione dell'operatore di Liouville in parti che permettono un'integrazione separata delle forze a breve e lungo raggio: $i\hat{\mathcal{L}} = i\hat{\mathcal{L}}_1 + i\hat{\mathcal{L}}_2$. La chiave di questa metodologia è la capacità di trattare separatamente le interazioni nel sistema, consentendo passi temporali maggiori per le forze che variano lentamente senza sacrificare l'accuratezza per quelle che cambiano rapidamente.

L'algoritmo utilizzato in questo progetto (e nel lavoro di Johnson) per integrare le equazioni del moto è il *Velocity Verlet*, derivato come caso particolare dell'algoritmo di Tuckerman.
Questo integratore è espresso in termini di posizioni e velocità delle particelle e garantisce la *time-reversibility* e la simpletticità (e dunque la conservazione dei volumi nello spazio delle fasi).

La procedura generale per generare la soluzione in una simulazione segue tre passaggi principali:

1. Cominciare con lo stato iniziale e generare il moto sotto il propagatore $e^{i\hat{\mathcal{L}}_1 \frac{\Delta t}{2}}$ ottenendo lo stato a metà passo.
2. Usare questo stato intermedio come nuovo stato iniziale e generare il moto con il propagatore $e^{i\hat{\mathcal{L}}_2 \Delta t}$, ottenendo lo stato a tempo pieno.
3. Partire dallo stato ottenuto al passo 2 come condizione iniziale e generare il moto con $e^{i\hat{\mathcal{L}}_1 \frac{\Delta t}{2}}$ per completare il passo temporale.

Difatti la fattorizzazione di Trotter prevede:

$$e^{i\hat{\mathcal{L}}\Delta t} = e^{i\hat{\mathcal{L}}_q \Delta t + i\hat{\mathcal{L}}_p \Delta t} = e^{i\hat{\mathcal{L}}_p \frac{\Delta t}{2}}\ e^{i\hat{\mathcal{L}}_q \Delta t}\ e^{i\hat{\mathcal{L}}_p \frac{\Delta t}{2}} + \mathcal{O}(\Delta t^3)$$

in cui gli operatori $i\hat{\mathcal{L}}_p \equiv i\hat{\mathcal{L}}_1$ e $i\hat{\mathcal{L}}_q \equiv i\hat{\mathcal{L}}_2$ citati sopra sono definiti come:

$$i\hat{\mathcal{L}}_q = \sum_{i=1}^{N_p} \frac{\vec{p_i}}{m_i} \cdot \frac{\partial}{\partial \vec{q_i}}$$

$$i\hat{\mathcal{L}}_p = \sum_{i=1}^{N_p} \vec{f_i}(\vec{q}_1, \ldots, \vec{q}_{N_p}) \cdot \frac{\partial}{\partial \vec{p_i}}$$

L'implementazione di quanto detto è stata effettuata nel *header file* `particle.hpp` all'interno della classe `particleLJ`, come mostrato in *listing* 1 in Appendice. Definite le due funzioni per implementare $e^{i\hat{\mathcal{L}}_p \frac{\Delta t}{2}}$ e $e^{i\hat{\mathcal{L}}_q \Delta t}$, esse verranno poi utilizzate all'interno della funzione `run` nel *header* `sim.hpp` (cfr. *listing* 2).

2 Metodologia

2.1 Approccio alla simulazione

La simulazione è stata configurata con le seguenti specifiche:

- Sistema composto da $N_p = n_x n_y n_z = 8 \cdot 8 \cdot 8 = 512$ particelle[1], di massa $m = 1.0$ e diametro $\sigma = 1.0$, confinate in una scatola cubica di lato[2] L, con condizioni periodiche al bordo.
- Il potenziale L-J è stato troncato a un raggio di *cutoff* $r_c = 2.5\sigma$, come già menzionato.
- Come richiesto, le simulazioni sono state eseguite a diverse densità ridotte $\rho^* = \frac{\sigma^3 N_p}{V}$, specificamente per $\rho^* = 0.1, 0.2, 0.3, 0.4$, al fine di esplorare il comportamento del sistema in vari regimi di densità.
- La temperatura ridotta è stata fissata a $T^* = 1.4$, utilizzando unità ridotte dove $T^* = \frac{k_B T}{\epsilon}$, con $\epsilon = 1$, che rappresenta la "profondità" del potenziale.
- La costante di Boltzmann k_B è presa pari ad 1, così da risultare $\beta = 1/T^*$.
- Un passo temporale di integrazione $dt = 0.004$ è stato utilizzato per l'avanzamento della simulazione, per un totale di *time steps* pari a 20 000 o 30 000, conformemente a quanto riportato nei risultati in letteratura con cui si andrà ad eseguire il confronto.
- Le correzioni di coda relative ad energia e pressione sono state calcolate per ottenere una migliore stima di tali osservabili e confrontare poi i risultati con i dati di riferimento.

[1] n_α rappresenta il numero di particelle lungo la direzione α, con $\alpha = x, y, z$.

[2] Poiché si scelgono a priori ρ^* e N_p, il valore del lato della scatola verrà calcolato di conseguenza: $L = \left(\frac{N_p}{\rho^*}\right)^{\frac{1}{3}}$

2.2 Calcolo degli osservabili

Nel corso della simulazione, sono stati calcolati due osservabili in unità ridotte: l'energia potenziale per particella $u^* = \frac{U^*}{N_p}$ e la pressione P^*. Quest'ultima è calcolata come:

$$P^* = P_{id} + P_{ex} = \rho^* T^* + < \frac{1}{3L^3} \sum_{i<j} \vec{f}_{ij} \cdot \vec{r}_{ij} > \tag{2}$$

dove P_{id} è la pressione del gas ideale e P_{ex} è la cosiddetta "pressione in eccesso", calcolata con il viriale. A questo valore di P^* verrà aggiunto poi un termine di correzione di coda P^*_{tail} definito in eq. (5). Tutto questo avviene tramite la funzione `calc_pressure`, definita nel *listing* 3.

L'energia potenziale per particella si ottiene invece calcolando:

$$u^* = \frac{U^*}{N_p} = \frac{4\epsilon}{N_p} \sum_{i<j} \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{3}$$

Anche qui verrà aggiunto un termine di correzione di coda (u^*_{tail}), definito in formula (4).

Si noti - infine - che eseguire una sommatoria $\sum_{i<j}$ equivale a fare $\frac{1}{2} \sum_{i \neq j}$. Tuttavia, nel primo caso, al livello di codice, si avrà un risparmio computazionale, poiché si andranno ad eseguire esattamente la metà dei calcoli rispetto al secondo caso.

2.3 Correzioni di Coda

Le correzioni di coda sono necessarie nelle simulazioni di dinamica molecolare per compensare gli effetti troncati delle interazioni a lungo raggio, come nel caso del potenziale di Lennard-Jones (troncato proprio a r_c).
Teoricamente, queste correzioni consentono di approssimare l'influenza delle interazioni oltre il raggio di *cutoff* sulle proprietà termodinamiche del sistema, come energia e pressione.

In particolare, l'energia di coda per particella u^*_{tail} e la pressione di coda P^*_{tail} possono essere calcolate - in unità ridotte - come segue (ottenute assumendo la *radial distribution function* $g(\vec{r}) = 1$ per $r > r_c$):

$$u^*_{tail} = \frac{8}{9}\pi\rho^* \left[\left(\frac{\sigma}{r_c} \right)^9 - 3 \left(\frac{\sigma}{r_c} \right)^3 \right] \tag{4}$$

$$P^*_{tail} = \frac{32}{9}\pi(\rho^*)^2 \left[\left(\frac{\sigma}{r_c} \right)^9 - \frac{3}{2} \left(\frac{\sigma}{r_c} \right)^3 \right] \tag{5}$$

Queste correzioni sono state implementate nel codice per garantire che le stime delle proprietà fisiche siano più accurate e confrontabili con i dati sperimentali e teorici disponibili.

3 Implementazione

3.1 Struttura del codice

Compilando il codice `mdsim.cpp` si dà il via alla simulazione. Per il corretto svolgimento di essa è stato necessario scrivere determinati *header files*, da "includere" nel programma C++. Essi sono i seguenti:

- `pvector.hpp`: implementa varie operazioni e funzionalità per i vettori (come prodotto scalare, somma di vettori, ecc.), rendendole più immediate durante la scrittura del codice.
- `params.hpp`: contiene una classe *template* chiamata `simparsLJ`, che definisce i parametri (come r_c, n_i, ρ^*, ecc...) per configurare la simulazione (MC o MD) delle particelle con potenziale di Lennard-Jones, eseguibile sia nell'ensemble NTV che nel NPT. La classe fornisce valori predefiniti per tutti i parametri, che possono essere modificati secondo necessità.
- `particle.hpp`: definisce la classe *template* `particleLJ`, che fornisce metodi per muovere la particella, calcolare l'interazione tra due particelle e aggiornare la posizione e la velocità utilizzando gli opportuni operatori.
- `sim.hpp`: definisce le classi `simLJ`, `mcsimLJ`, e `mdsimLJ`, che sono utilizzate nella simulazione. Le classi forniscono metodi per inizializzare il sistema, eseguire le simulazioni e calcolare le grandezze fisiche di interesse come energia, pressione e altre misure.

Tutti i *file* sono depositati nel *Google Drive folder* (linkato).

3.2 Preliminare verifica dell'efficacia dell'algoritmo

Innanzitutto si verifica che l'energia totale fluttui attorno ad un valore medio (cioè che si conservi), come mostrato in figura 2.

Inoltre vogliamo verificare che l'algoritmo utilizzato sia effettivamente del II ordine. Si studia quindi la dipendenza lineare della deviazione standard dell'energia (σ_E) dal valore del passo d'integrazione utilizzato per la simulazione, elevato al quadrato (dt^2).

Si eseguono dunque diverse simulazioni, ogni volta scegliendo un passo di integrazione dt differente e modificando appositamente anche il numero totale di *steps*, al fine di avere una durata totale di esperimento sempre uguale: $T = tot\_steps \cdot dt = 40$ *unità di tempo*. I valori di cui ci si è avvalsi per eseguire tale studio grafico - mostrato in figura 3 - sono riportati nella tabella 1.

dt	σ_E	tot_steps
0.001	0.00422	40 000
0.002	0.00792	20 000
0.004	0.02442	10 000
0.006	0.05007	6666
0.008	0.08469	5000
0.010	0.12977	4000

Tabella 1: Passo di integrazione utilizzato, deviazione standard dell'energia ottenuta e numero totale di *steps*.

Di seguito si riportano i due *plot* citati.

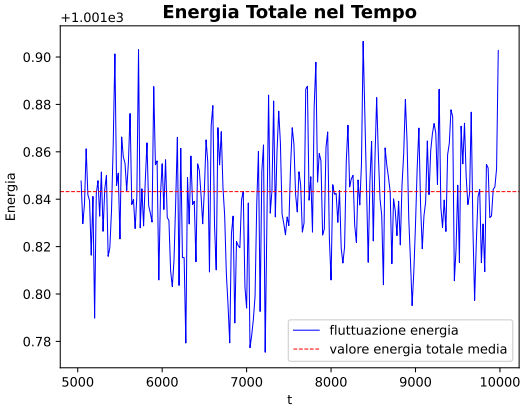

Figura 2: Fluttuazione di E_{tot}.

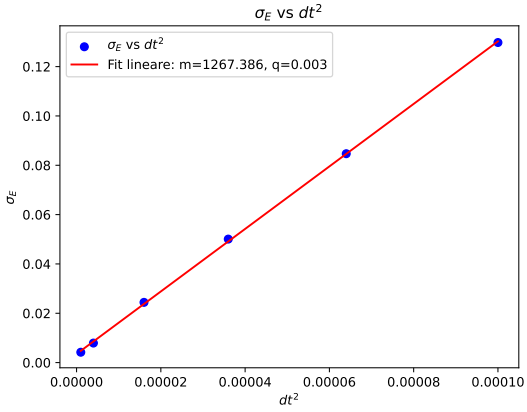

Figura 3: Relazione lineare tra σ_E e dt^2.

Il grafico a destra mostra proprio la relazione lineare tra σ_E e dt^2, con parametri di *fit*: coefficiente angolare $m = 1267.4 \pm 1.9$ ed intercetta $q = 0.0030 \pm 0.0035$. Il *fit* lineare evidenzia come l'errore sull'energia vari effettivamente in maniera proporzionale al quadrato del passo d'integrazione.

Si è inoltre notato, come anche nell'articolo di Johnson, che il sistema "equilibra" dopo un tempo $t = equilibration\_time \simeq 5000$.

3.3 Controllo della temperatura

3.3.1 Velocity Rescaling

Il metodo di *Velocity Rescaling* è una tecnica semplice ma efficace utilizzata nelle fasi iniziali di una simulazione di dinamica molecolare per portare il sistema a una temperatura target. Questo metodo aggiusta le velocità delle particelle moltiplicandole per un fattore di scala λ calcolato in modo tale che l'energia cinetica totale del sistema corrisponda a quella desiderata. Tuttavia, l'uso del *Velocity Rescaling* presenta alcune limitazioni significative. Anzitutto, non permette le fluttuazioni di temperatura proprie dell'*ensemble* canonico, poiché mantiene la temperatura forzatamente costante al valore target. In secondo luogo, poi, il processo di riscaldamento non è reversibile nel tempo, il che può influenzare la corretta rappresentazione della dinamica microscopica del sistema.

Nonostante queste limitazioni, il *velocity rescaling* presenta alcuni vantaggi. Intanto è abbastanza semplice da implementare, si veda *listing* 4 in Appendice. Inoltre, è particolarmente utile nella fase di riscaldamento o inizializzazione della simulazione, dove è importante portare rapidamente il sistema vicino allo stato di equilibrio

desiderato; difatti il metodo verrà implementato per $t < equilibration\_time$.
In sintesi, il *Velocity Rescaling*, sebbene non sia l'ideale per tutte le fasi di una simulazione di dinamica molecolare, può essere un utile strumento durante la fase iniziale per stabilizzare la temperatura del sistema.

Matematicamente la sua implementazione è la seguente: se la temperatura al tempo t è $T(t)$ e le velocità sono "riscalate" di un fattore λ, allora il cambiamento di temperatura associato può essere calcolato come:

$$\Delta T = \sum_i \frac{m_i(\lambda v_i)^2}{N_{dof}k_B} - \sum_i \frac{m_i v_i^2}{N_{dof}k_B} = (\lambda^2 - 1)T(t) \tag{6}$$

dove N_{dof} è il numero d i gradi di libertà ed il parametro λ è definito come:

$$\lambda = \sqrt{\frac{T_0}{T(t)}}$$

dove $T(t)$ è la temperatura corrente calcolata dall'energia cinetica e T_0 è la temperatura target [**3**].
Il modo più semplice per controllare la temperatura è quindi moltiplicare le velocità a ogni passo temporale per il fattore λ:

$$\vec{v}_i^{new} = \lambda\ \vec{v}_i^{old} \tag{7}$$

3.3.2 Il termostato di Nosè-Hoover

L'utilizzo del termostato è un'alternativa al metodo di *rescaling* delle velocità per il controllo della temperatura. Mentre il *velocity rescaling* modifica direttamente le velocità delle particelle per raggiungere e mantenere una temperatura target, un termostato agisce in modo più fisicamente realistico per simulare lo scambio di calore tra il sistema e un bagno termico. Il termostato di Nosè-Hoover (N-H) è progettato per permettere fluttuazioni naturali, essenziali per rispecchiare correttamente le proprietà dell'*ensemble* canonico.

Questo metodo consiste nell'introdurre una variabile ausiliaria ξ, che regola l'energia del sistema per farla corrispondere alla temperatura desiderata, agendo come un meccanismo di *feedback* che evolve secondo un'equazione di moto specifica (si leggano equazioni 6.1.25 e 6.1.26 di [**4**]), la quale regola l'interazione termica tra il sistema e il termostato stesso:

$$\dot{\xi} = \frac{1}{Q}\left(\sum_i \frac{p_i^2}{m_i} - gk_BT_0\right) \tag{8}$$

dove T_0 è la temperatura target che si desidera mantenere nel sistema, $g = 3N_p + 1$, mentre Q è un parametro (talvolta chiamato "massa fittizia") che influenza la dinamica del termostato.

Particolare attenzione deve essere dedicata alla scelta del valore della massa fittizia. Da un lato, valori troppo grandi di Q (*loose coupling*) possono causare un controllo della temperatura non ottimale (il termostato di Nosè-Hoover con $Q \longrightarrow \infty$ è una MD che genera un *ensemble* microcanonico[3]). D'altra parte, valori troppo piccoli (*tight coupling*) possono causare oscillazioni di temperatura ad alta frequenza.

Per impostare Q a un valore che permettesse un controllo stabile di T (fluttuazioni non troppo eccessive), sono state eseguite alcuni simulazioni di prova. Alla fine, si è scelto di impostare $Q = 10$, osservando dei risultati coerenti con quelli riportati in letteratura.

Implementando questo metodo, l'equazione del moto del momento, per ogni particella, diventa [**4**]:

$$\dot{\vec{p}}_i = \vec{f}_i - \xi\vec{p}_i \tag{9}$$

dove il termine $-\xi\vec{p}_i$ rappresenta una sorta di frizione, con cui il pistone termico controlla la temperatura.

In conclusione, l'hamiltoniana finale del sistema ("hamiltoniana di Nosè") sarà data da:

$$\mathcal{H}_N = \sum_i \frac{p_i^2}{2m_i} + V(\vec{q}_1, ..., \vec{q}_N) + \frac{\xi^2 Q}{2} + gk_BT\xi \tag{10}$$

Tipicamente l'energia totale definita da $\mathcal{H}_N$ è conservata.

L'evoluzione temporale della variabile ausiliaria ξ può essere calcolata utilizzando un algoritmo numerico, per garantire appunto che il sistema raggiunga l'equilibrio con la temperatura desiderata (*listing* 5).

[3] Quando Q è molto grande, il termine di controllo della temperatura diventa trascurabile rispetto alla dinamica delle particelle. Questo significa che il sistema è praticamente isolato rispetto al termostato, e il suo comportamento è determinato quasi esclusivamente dalle equazioni del moto delle particelle.

4 Analisi dei risultati

4.1 Configurazione spaziale

Innanzitutto, a scopo illustrativo, si presentano in figura 4 e 5 i *plot* della configurazione iniziale delle particelle, limitate dentro al cubo per la simulazione. Il grafico è stato eseguito per $\rho^* = 0.4$, dunque $L = (\frac{512}{0.4})^{\frac{1}{3}} \simeq 10.86$.

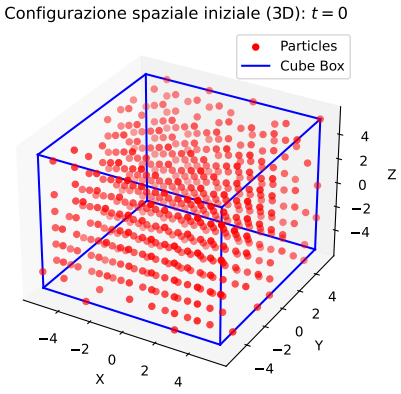

Figura 4: Configurazione spaziale iniziale, in 3D.

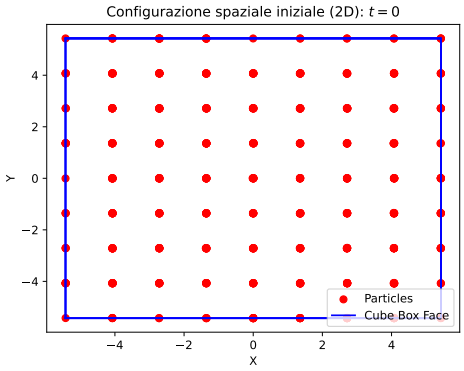

Figura 5: Configurazione in 2D iniziale, vista da una faccia del cubo (piano XY).

Per questi stessi parametri di configurazione, si forniscono simili grafici (fig. 6 e 7), al tempo $t = 20\,000$, ossia a fine simulazione.

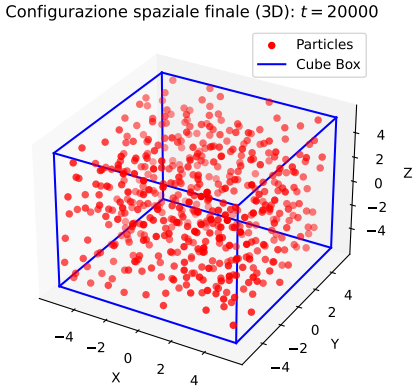

Figura 6: Configurazione spaziale finale, in 3D.

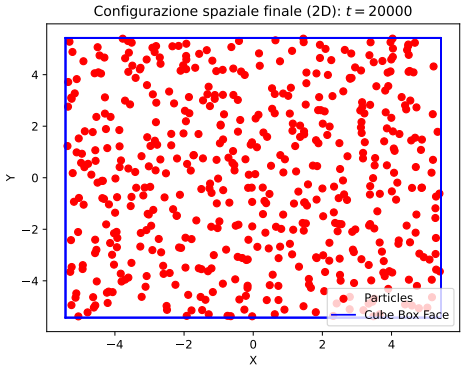

Figura 7: Configurazione in 2D finale, vista da una faccia del cubo (piano XY).

Questi quattro grafici sono stati effettuati utilizzando l'interfaccia di *Jupyter Notebook*, fornita da *Google Colab*, dopo aver salvato in un apposito `file.dat` le coordinate di tutte le particelle al tempo t desiderato. Gli *script* scritti in linguaggio *Python* per eseguire i grafici sono mostrati nei *listing* 6 e 7.

4.2 Gli osservabili

Nella tabella 2 che segue si mostrano i valori (medi) di riferimento degli osservabili, riportati nell'articolo di Johnson **[2]**. Si noti che il numero tra parentesi costituisce l'incertezza, ad esempio $P^* = 0.1035(3) = 0.1035 \pm 0.0003$.

ρ^*	P^*	u^*	*Total steps*
0.1	0.1035(3)	−0.766(3)	30 000
0.2	0.1524(4)	−1.483(4)	30 000
0.3	0.1721(1)	−2.136(6)	20 000
0.4	0.1962(2)	−2.745(5)	20 000

Tabella 2: Risultati di riferimento, da confrontare con quelli della nostra simulazione.

Di seguito i risultati ottenuti nelle simulazioni utilizzando - per il controllo della temperatura - prima il *velocity rescaling* (tab. 3, P_1^* e u_1^*) e in seconda battuta il termostato N-H (tab. 4, P_2^* e u_2^*).

ρ^*	P_1^*	u_1^*	*Total steps*
0.1	0.1041(9)	−0.739(9)	30 000
0.2	0.148(3)	−1.468(4)	30 000
0.3	0.175(3)	−2.097(8)	20 000
0.4	0.188(8)	−2.736(3)	20 000

Tabella 3: Risultati usando *velocity rescaling*.

ρ^*	P_2^*	u_2^*	*Total steps*
0.1	0.1035(6)	−0.759(3)	30 000
0.2	0.153(2)	−1.481(4)	30 000
0.3	0.172(1)	−2.090(8)	20 000
0.4	0.194(4)	−2.727(4)	20 000

Tabella 4: Risultati usando il termostato.

Eseguendo un test di compatibilità (ad esempio quello di Gauss) si può verificare come, in entrambi i nostri casi, si osserva una buona compatibilità tra i valori ottenuti e quelli di riferimento, sia per pressione sia per energia.

Inoltre, di seguito (figura 8) si mostrano le isoterme nel piano di Clapeyron relative alle pressioni calcolate nei tre diversi casi: valori di letteratura (in rosso), utilizzando *velocity rescaling* (in blu), utilizzando termostato (in verde). Analogamente, in figura 9, si riporta il *plot* dei valori di u^* in funzione di ρ^*.

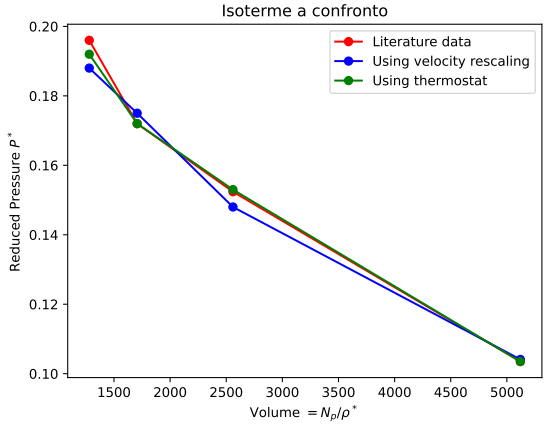

Figura 8: Confronto tra le curve isoterme.

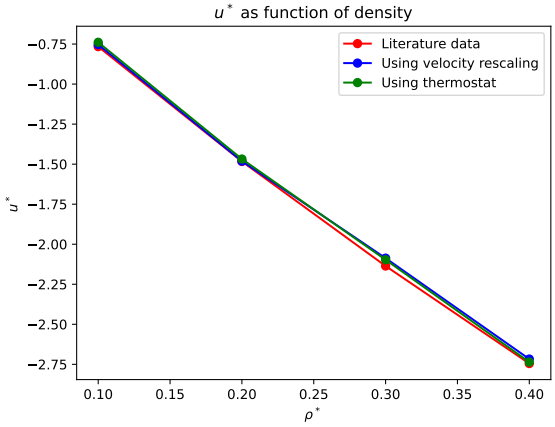

Figura 9: Energia potenziale in funzione di ρ^*.

Si nota, per quanto riguarda il grafico a sinistra, come le tre curve siano molto prossime tra loro. In particolare, nel caso dell'utilizzo del termostato, notiamo una maggiore sovrapposizione tra le isoterme.
Analogamente, osservando il grafico a destra, si nota che anche i valori di u^* sono molto prossimi tra loro.

5 Conclusioni

Grazie all'implementazione accurata dei metodi di simulazione e all'utilizzo del termostato di Nosè-Hoover, i risultati ottenuti si sono avvicinati notevolmente ai valori riportati nella letteratura, confermando l'efficacia del nostro approccio. Anche con la più semplice tecnica del *velocity rescaling* comunque si sono ottenuti valori compatibili con quelli mostrati nell'articolo di Johnson. In particolar modo, però, l'introduzione del termostato N-H ha permesso una simulazione che meglio riflette le condizioni reali di un *ensemble* canonico, fornendo una regolazione più precisa della temperatura e permettendo al sistema di raggiungere uno stato di equilibrio più rappresentativo. Questo successo evidenzia l'importanza di scegliere strategie di simulazione appropriate per studiare sistemi fisici complessi.

6 Appendice: frammenti di codice

```
void updLp(ntype dt) {
    //Action of operator exp(iLp*dt) to update velocities:
    v += dt * f / m;
}
void updLq(ntype dt) {
    //Action of operator exp(iLq*dt) to update positions:
    r += dt * v;
}
```

Listing 1: Operatori $e^{i\hat{\mathcal{L}}_p}$ ed $e^{i\hat{\mathcal{L}}_q}$ nella classe `particleLJ`.

```
void run(void) {
    calc_forces(Us, U);
    calc_pressure(P);
    ntype energiaPotTotalePerParticella = U/pars.Np;
    std::fstream fo;
    fo.open("totalenergy.dat", std::fstream::out);
    for (long long int t=0; t < pars.timesteps; t++)
    {
        // 1) Apply exp(iLp*dt/2) to all particles
        for(auto &part : parts) {
            part.updLp(pars.dt / 2);
        }

        // 2) Apply exp(iLq*dt) to all particles and PBC
        for(auto &part : parts) {
            part.updLq(pars.dt);
            pbc(part); // Adjusting positions according to PBC
        }

        // Recalculating forces after position update
        calc_forces(Us, U);

        // 3) Applying exp(iLp*dt/2) to all particles here:
        for(auto &part : parts) {
            part.updLp(pars.dt / 2);
        }
    }
}
```

Listing 2: Implementazione della funzione `run`.

```
void calc_pressure(ntype& P)
  {
    ntype wij_sum = 0.0; //viriale
    ntype vij, vijs, wij;

    // Computing sum of wij for all couples:
    for (size_t i = 0; i < parts.size(); ++i) {
      for (size_t j = i + 1; j < parts.size(); ++j)
        {
          pvector<ntype, 3> fijv = parts[i].fij(parts[j], pars.L,
              ↪ vij, vijs, wij);
          wij_sum += wij;
        }
    }
    // Tail corrections:
    ntype P_tail = (32.0 / 9.0) * M_PI * std::pow(pars.rho, 2) * (
        ↪ std::pow(pars.sigma / pars.rc, 9) - 1.5 * std::pow(pars
        ↪ .sigma / pars.rc, 3));
    // Computing total pressure:
    ntype volume = pars.L[0] * pars.L[1] * pars.L[2];
    P = (pars.Np * pars.T + wij_sum / 3.0) / volume + P_tail;
    // NOTE: dividing virial by 3 because, as for U, the summation
        ↪  is for j>i (computational saving)
  }
```

Listing 3: Funzione per calcolare la pressione.

```
void velocityRescaling() {
    ntype currentTemperature = calcCurrentTemperature();
    ntype targetTemperature = pars.T;
    ntype rescalingFactor = sqrt(targetTemperature /
        ↪ currentTemperature);

    for (auto &part : parts) {
        part.v *= rescalingFactor; // Rescale velocities
    }
}

ntype calcCurrentTemperature() {
    ntype kineticEnergy = 0.0;
    for (const auto &particle : parts) {
        kineticEnergy += 0.5 * particle.m * (particle.v * particle
            ↪ .v);
    }

    ntype degreesOfFreedom = 3 * parts.size();
    ntype currentTemperature = (2.0 / degreesOfFreedom) *
        ↪ kineticEnergy; // from equipartition theorem

    return currentTemperature;
}
```

Listing 4: Implementazione della funzione `VelocityRescaling`.

```
void NoseHooverThermostat(ntype& xi, ntype Q) {
    ntype kineticEnergy = calcK();
    // System's target temperature:
    ntype targetTemperature = pars.T;

    // Computing g*kB*T, with kB=1 and g=3N+1:
    ntype gkT = (3 * parts.size() + 1) * targetTemperature;

    // Updating xi:
    xi += pars.dt * ((2.0 * kineticEnergy - gkT) / Q);
    // Updating velocities based on xi:
  for (auto &part : parts) {
    part.v += -xi * pars.dt * part.v;
    }
}
```

Listing 5: Implementazione del termostato di Nosè-Hoover.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

# Import data from file 'config-0.dat'
data = np.loadtxt("conf-0.dat", delimiter="␣")

# Def variables for coordinates
x = data[:, 0]
y = data[:, 1]
z = data[:, 2]

# Creating figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot particeles as scatter plot
ax.scatter(x, y, z, c='r', marker='o', label='Particles
    ↪ ')

ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.title("Configurazione␣spaziale␣iniziale␣(3D):␣$t␣=␣
    ↪ 0$")

#Cube's side:
L = (512 / 0.4) ** (1/3)
# Cube in the origin
cube = np.array([[-L/2, -L/2, -L/2],
                 [-L/2, -L/2, L/2],
                 [-L/2, L/2, -L/2],
                 [-L/2, L/2, L/2],
                 [L/2, -L/2, -L/2],
                 [L/2, -L/2, L/2],
                 [L/2, L/2, -L/2],
                 [L/2, L/2, L/2]])

# Connections for cube's vertices
connections = [
    (0, 1), (1, 3), (3, 2), (2, 0),
    (4, 5), (5, 7), (7, 6), (6, 4),
    (0, 4), (1, 5), (2, 6), (3, 7)
]

cube_label_added = False
for connection in connections:
    start = cube[connection[0]]
    end = cube[connection[1]]
    # Adding label only 1st time
    if not cube_label_added:
        ax.plot([start[0], end[0]], [start[1], end[1]],
            ↪  [start[2], end[2]], color='b', label='
            ↪ Cube␣Box')
        cube_label_added = True
    else:
        ax.plot([start[0], end[0]], [start[1], end[1]],
            ↪  [start[2], end[2]], color='b')

ax.grid(False)
ax.legend()
plt.savefig("3D_initial_config.pdf", format='pdf')
plt.show()
```

Listing 6: Script *Python* per grafico 3D.

```
import matplotlib.pyplot as plt
import numpy as np

data = np.loadtxt("config-0.dat", delimiter="␣")

x = data[:, 0]
y = data[:, 1]

fig = plt.figure()
ax = fig.add_subplot(111)

p1 = ax.scatter(x, y, c='r', marker='o', label="
    ↪ Particelle")

L = (512 / 0.4) ** (1 / 3)

X = np.array([-L/2, L/2, L/2, -L/2, -L/2, L/2, L/2, -L
    ↪ /2])
Y = np.array([-L/2, -L/2, L/2, L/2, -L/2, -L/2, L/2, L
    ↪ /2])

faces = np.array([
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [0, 4, 5, 1],
    [3, 2, 6, 7],
    [0, 3, 7, 4],
    [1, 5, 6, 2],
])

# Plotting the cube:
for face in faces:
    ax.plot(X[face], Y[face], color='b')

ax.set_xlabel("X")
ax.set_ylabel("Y")
plt.title("Configurazione␣spaziale␣iniziale␣(2D):␣$t␣=␣
    ↪ 0$")
ax.legend(["Particles", "Cube␣Box␣Face"], loc='lower␣
    ↪ right', prop={'size': 10})
plt.grid(False)
plt.savefig("2D_initial_config.pdf", format='pdf')
plt.show()
```

Listing 7: Script *Python* per grafico 2D.

Riferimenti bibliografici

[1] Tuckerman *et al.* in Journal of Chemical Physics, 97 1990–2001 (1992).

[2] J. K. Johnson *et al.*, Molecular Physics, 78, 591–618 (1993).

[3] Andrienko, D. (2007). A comparative study of several thermostats. Retrieved from https://www2.mpip-mainz.mpg.de/~andrienk/journal_club/thermostats.pdf

[4] Frenkel, D., & Smit, B. (2002). Understanding Molecular Simulation: From Algorithms to Applications. Academic Press.