POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Chimica e dei Processi Sostenibili

Tesi di Laurea Magistrale

Sviluppo di un metodo per calcolo automatizzato della viscosità con Dissipative Particle Dynamics

Relatore/i

prof. Daniele Marchisio

prof. Antonio Buffo

prof. Gianluca Boccardo

Candidato

Tommaso Maria Ungetti

Anno Accademico 2021-2022

Indice

Introduzione

Con il rapido sviluppo delle tecnologie in ambito computazionale, l'importanza della simulazione come tecnica di studio dei fenomeni chimici e fisici in ambito industriale è sempre più evidente. Il lavoro di tesi si concentra su una di queste tecniche, la *Dissipative Particle Dynamics*, impiegata per lo studio reologico dei fluidi semplici e strutturati. I fluidi strutturati sono ampiamente impiegati e diffusi in numerosi processi industriali, ad esempio in ambito farmaceutico, biologico, cosmetico ed alimentare; lo studio e la definizione delle proprietà reologiche dei fluidi strutturati si dimostra quindi essenziale quando si devono valutare progettazioni impiantistiche e scale-up delle apparecchiature e dei processi in cui essi sono presenti. I fluidi complessi/strutturati sono miscele di più fasi, ad esempio sospensioni proteiche o soluzioni polimeriche; questo tipo di fluidi presentano proprietà reologiche peculiari e complesse da studiare a causa della formazione di microstrutture nel fluido. Esse sono osservabili solo in scale temporali e spaziali molto maggiori rispetto a quella atomica. La *Dissipative Particle Dynamics* permette l'analisi di questi fenomeni. Infatti sarebbe estremamente onerosa la descrizione di questi fenomeni per mezzo di altre tecniche di simulazione come la *Molecular Dynamics* MD, che simula i fenomeni alla scala atomica. Oltre alla descrizione del comportamento reologico di un fluido, sono stati svolti diversi studi con l'obiettivo di riuscire a predire quantitativamente le proprietà di trasporto dei fluidi per mezzo della DPD. Si sono riscontrate numerose difficoltà, tra cui la definizione dei parametri di input alla simulazione e la conversione dei risultati da unità DPD ad unità reali. Molte di queste difficoltà sono direttamente correlabili a come è definito il metodo DPD. Esso è infatti un metodo detto di *coarse-graining*, cioè un tipo di metodo che prevede il raggruppamento di molecole in entità dette *beads*, il numero di molecole all'interno di una singola *bead* è pari al numero di *coarse-graining Nm*. Questo metodo permette un risparmio computazionale notevole rispetto al metodo MD grazie alla possibilità di limitare lo sforzo dovuto alla simulazione di ogni particella del sistema. Il *coarse-graining* comporta la rimozione di gradi di libertà del sistema. Essi vengono reintrodotti grazie all'aggiunta di due contributi che compaiono nella definizione delle equazioni delle forze agenti sulle *beads*: il contributo fluttuante/stocastico, che introduce l'effetto di rumore termico del sistema presente a livello molecolare, ed il contributo dissipativo, dovuto all'attrito agente tra le particelle DPD. Il lavoro di tesi consiste nello sviluppare un metodo per il calcolo della viscosità per mezzo del sistema DPD che sia il meno computazionalmente costoso possibile Il metodo è stato sviluppato utilizzando il software LAMMPS per le simulazioni DPD e la generazione dei risultati, ed il linguaggio di programmazione ad oggetti Python per la definizione delle operazioni da eseguire e l'analisi dei dati.

Il lavoro compiuto durante questi mesi è incentrato sullo sviluppo di un metodo automatizzato per il calcolo della viscosità. Il metodo non deve solo garantire che il risultato sia sufficientemente accurato per un determinato sistema simulato, ma deve anche riuscire a raggiungere il risultato con il minor costo computazionale. Un punto nevralgico infatti delle simulazioni DPD è la definizione dei parametri di *post-processing* adatti all'elaborazione dei dati e, quindi, al calcolo delle proprietà di trasporto del fluido. Il metodo quindi deve essere in grado di compiere autonomamente delle scelte e dei controlli sugli output della simulazione, e deve essere in grado di funzionare su tutti i supporti disponibili al calcolo, che siano macchine singole con un PC o il cluster di ateneo fornito dal servizio HPC@POLITO.

Il testo della tesi è quindi diviso in due macro sezioni, nella Parte I verrà esposta la teoria che è stata sviluppata per la Dissipative Particle Dynamics, con cenni di meccanica statistica e la spiegazione sul perché questa tecnica è arrivata ad essere formulata nella forma come la conosciamo oggi. La Parte II invece si occuperà di spiegare come il metodo di calcolo della viscosità è stato effettivamente sviluppato, a partire dalle osservazioni preliminari che hanno portato alla formulazione, a finire con l'implementazione in codice e quindi dello *script* in sé, soffermandosi sui singoli aspetti che ne compongono il processo di decisionale e di calcolo.

Parte I

Capitolo 1

Introduzione alla Dissipative Particle Dynamics

Gli ultimi sviluppi dell'industria moderna comporta una serie di nuove sfide nell'ambito dell'Ingegneria Chimica. In particolare, il progredire tecnologico e la possibilità di fare affidamento su potenze di calcolo sempre maggiori, ha portato allo sviluppo di metodi di simulazione sempre più raffinati e precisi in grado di fungere da sostegno per i tradizionali metodi sperimentali. Per la stessa natura dei processi di cui si occupa l'industria chimica ha a che fare con fluidi complessi. Essi si possono trovare sia in ambito farmacologico/biologico che in ambito chimico/polimerico. In generale i fluidi complessi possono essere definiti come una miscela di due fasi, alcuni esempi possono essere le sospensioni solido-liquido o le emulsioni liquido-liquido, essi presentano delle caratteristiche meccaniche e reologiche peculiari che non sono riscontrabili nei fluidi semplici. Queste proprietà sono dovute alla presenza ed allo sviluppo di microstrutture all'interno della miscela, responsabili delle sopracitate proprietà peculiari. Essendo i fluidi complessi così diffusi in ambito industriale, si rende necessario sviluppare un sistema efficiente per simularne il comportamento e valutarne le proprietà reologiche e fluidodinamiche. Un metodo possibile per la modellazione di tali fluidi può essere la *Molecular Dynamics* (MD), metodo che prevede la risoluzione numerica delle equazioni di Newton. Nonostante questo metodo sia in grado di produrre modelli molto precisi e dettagliati dei fluidi analizzati, richiede un costo computazionale estremamente alto per poter essere utilizzato nella maggior parte dei casi di interesse industriale. Un altro approccio possibile è quello di considerare il fluido come un continuo, un esempio di questo tipo di modelli può per esempio essere il modello dei fluidi complessi di Navier-Stokes-Fourier. La differenza concettuale rispetto ad MD consiste nel fatto che in questo caso non si considera il fluido dal punto di vista dei singoli atomi, ma dal punto di vista di volumi finiti in cui gli atomi sono racchiusi. Questi modelli si basano su determinati concetti chiave:

- I volumi devono contenere un numero molto alto di atomi (*limite del continuo*),
- I volumi devono essere sufficientemente grandi da poter riprodurre l'equilibrio termodinamico del sistema (*assunzione dell'equilibrio locale*)

Questi metodi permettono di descrivere il sistema da un punto di vista macroscopico vedendo il fluido come un continuo invece che come singoli atomi come in MD. Nel nostro caso noi siamo interessati ai fenomeni che avvengono a scale spaziali comprese tra 10-10^4 nm e scale temporali comprese tra 1-10^6 ns [1]: questi intervalli sono definibili come *mesoscala.* La modellazione della materia in forma di raggruppamenti, o volumi, di atomi non permette di descrivere i fenomeni che avvengono alla mesoscala, e quindi tutti i fenomeni legati all'evoluzione ed allo sviluppo delle microstrutture presenti nei fluidi complessi. Dalle seguenti osservazioni nasce quindi la necessità di ipotizzare un sistema che sia in grado di modellizzare il comportamento dei fluidi in una scala compresa tra quella del continuo, troppo grossolana per i nostri interessi, e quella di MD, che richiede una potenza computazionale troppo elevata. Una prima soluzione è stata proposta accoppiando alla simulazione CFD (tipica per la modellazione di fluidi nella scala del continuo) con dei termini fluttuanti, le equazioni risultanti da questo approccio sono dette LLNS da Landau-Lifshitz-Navier-Stokes. Nonostante l'introduzione dei termini fluttuanti nella descrizione dei fenomeni idrodinamici siano effettivamente adatti a descrivere i fenomeni alla mesoscala, ci sono numerosi casi in cui il modello idrodinamico del continuo non è applicabile o non è noto. Casi del genere possono essere per esempio sistemi polimerici, miscele proteiche o membrane. Un strategia per permettere di conservare una certa specificità chimica delle componenti della miscela del fluido complesso durante la simulazione mesoscala, è stata ipotizzata essere l'utilizzo di modelli di *coarse-graining,* cioè modelli in cui si considerano gruppi di atomi come se fossero un'unica entità definite *beads.* Una nota procedura per descrivere un sistema fluidodinamico per mezzo di questo modello *coarse-grained* prende il nome di meccanica statistica del non-equilibrio, teoria di Mori-Zwanzing, o teoria del *coarse-graining* [2]. Il vantaggio di utilizzare un metodo del genere consiste diminuire sensibilmente il costo computazionale della simulazione senza andare ad inficiare sulla qualità del modello. Ad esempio, in una simulazione di una sospensione acquosa si potrebbe condurre considerando le molecole di solvente come raggruppate in *beads,* abbassando il costo computazionale sensibilmente, ma allo stesso tempo garantendo specificità chimica al soluto grazie ad una modellazione molecolare più dettagliata. Un particolare metodo per la simulazione di fluidi complessi basato sul concetto di *coarse-graining* è stato ipotizzato alla fine del 20esimo secolo e prende il nome di *Dissipative Particle Dynamics* (DPD). Questo metodo permette la simulazione di particelle di fluidi complessi a livello meso-scala, considerando agenti sulla particella sia effetti dovuti all'idrodinamica, e sia effetti dovuti alla fluttuazione termica. L'algoritmo è estremamente versatile, in quanto permette la modellazione di fluidi complessi variando l'intensità delle forze conservative agenti tra le *beads.* La prima versione del modello DPD di Hoogerbrugge & Koelman [3] venne ipotizzata nel 1992 e si basava sulla capacità

di particelle puntiformi capaci di muoversi *off-lattice* di interagire tra loro per mezzo di tre tipi di forze:

- Forza Conservativa,
- Forza Dissipativa,
- Forza Stocastica.

Il modello originale DPD è stato successivamente ripreso ed ampliato [4] in modo tale da poter essere applicata ad numerosi *case studies.* Ad oggi il modello DPD costituisce uno dei metodi promettenti ed utili per simulare fluidi complessi e materia soffice, in condizioni dinamiche e statiche, grazie alla possibilità del modello di esplorare scale spaziali e temporali che in realtà sarebbero inaccessibili per mezzo di altre tecniche di simulazioni.

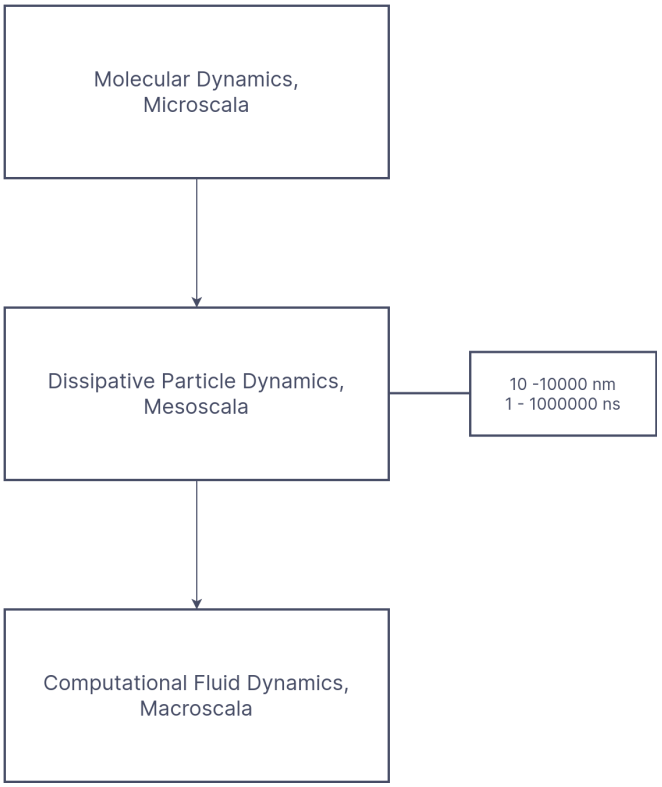


Fig.1.1 - Schematizzazione delle principali tecniche di simulazione computazionale con scale temporali e spaziali specificate per *Dissipative Particle Dynamics*

1.1 - Modello DPD & *Coarse-Graining*

Come detto in precedenza, il modello DPD consiste in un metodo *coarse-graining,* cioè un modello in cui si vanno a considerare dei *clusters* di particelle interagenti tra loro con quello che viene definito come *soft potential* di tipo repulsivo. Lo scopo ed il motivo per cui questa tecnica di simulazione è impiegata consiste nella possibilità di simulare fluidi complessi avendo il vantaggio di un costo computazionale molto ridotto rispetto ad MD, ma garantendo allo stesso tempo una modellizzazione sufficientemente accurata dei fenomeni che avvengono alla mesoscala [5]. L'effettivo vantaggio computazionale rispetto ad una simulazione MD è stato riportato in [6] usando come confronto una simulazione di acqua pura. Si è riscontrato un incremento della velocità pari a 1000*Nm^5/3 [5], con Nm che rappresenta il grado di *coarse-graining* della simulazione, ovvero il numero di particelle che si è deciso di clusterizzare. Il parametro *Nm* è estremamente importante da valutare correttamente, incide infatti non solo sulla velocità di simulazione ma anche sull'accuratezza dei risultati ottenuti. In generale questo parametro non è universale ma va valutato in base alla simulazione che si sta effettuando. Una volta impostato il valore di Nm abbiamo quindi definito le *beads* del sistema. Come già detto in precedenza, queste entità andranno ad interagire tra loro con una diversi tipi di forze, in particolare avremo una forza dipendente da quello che è un potenziale repulsivo conservativo soffice, o *soft potential.* Questo parametro garantisce che le *beads* non siano entità solide impenetrabili come potrebbero essere gli atomi, ma garantisce la compenetrabilità delle diverse particelle DPD del sistema in modo tale da meglio approssimare un *bead*. Esso non è costante in tutto lo spazio ma tende ad annullarsi all'aumentare della distanza tra le particelle. Oltre a questa forza repulsiva sono anche presenti due altre forze, una forza dissipativa che dipende dal *viscous drag* del fluido (forza dissipativa dovuta alla resistenza opposta al moto delle particelle dal fluido stesso), ed una forza randomica fluttuante dovuta al *thermal noise,* cioè dovuta al rumore termico associato al moto delle particelle dovuto ad una certa temperatura. L'azione combinata di queste due ultime forze funge da termostato in quanto l'energia sottratta al sistema per mezzo dell'effetto dissipativo viene bilanciata dall'aggiunta al sistema di energia grazie alla forza randomica che reintroduce nel sistema energia vibrazionale e termica, che si perdono nel processo di coarse graining in quanto associate alle singole molecole elementari e non alle *beads.*

1.2 - Equazioni caratteristiche

La posizione ed la quantità di moto delle particelle DPD, o *beads*, sono descritti dalle stesse equazioni della dinamica classica, ovvero le equazioni del moto di Newton:

$$\frac{d\boldsymbol{r}_i}{dt} = \boldsymbol{v}_i, \quad \frac{d\boldsymbol{v}_i}{dt} = \frac{\boldsymbol{f}_i}{m_i}, \tag{1.1}$$

dove $\boldsymbol{r}_i$ e $\boldsymbol{v}_i$ sono rispettivamente la posizione della *bead* i-esima di massa m_i, il termine $\boldsymbol{f}_i$ comprende le forze agenti sulla *bead* i-esima ed è costituito dalla somma delle singole forze agenti sulla particella DPD descritte in precedenza:

$$\boldsymbol{f}_i = \sum_{j \neq i} \left(\boldsymbol{F}_{ij}^{C} + \boldsymbol{F}_{ij}^{D} + \boldsymbol{F}_{ij}^{R} \right). \tag{1.2}$$

I tre termini che costituiscono $\boldsymbol{f}_i$ dipendono dalle interazioni reciproche delle beads *i* e *j*, le interazioni avvengono entro una certa distanza definita come raggio di cutoff. La forza conservativa definita da $\boldsymbol{F}_{ij}^{\mathbf{C}}$ va a descrivere il termine repulsivo-conservativo agente tra le *beads,* cioè il *soft potential* agente tra due beads i & j, esso può essere descritto matematicamente come:

$$\boldsymbol{F}_{ij}^{\mathbf{C}} = \begin{cases} a_{ij}\left(1 - \dfrac{r_{ij}}{r_c^{C}}\right)\widehat{\mathbf{r}_{ij}}, & r_{ij} < r_c^{C}, \\ 0, & r_{ij} < r_c^{C}, \end{cases} \tag{1.3}$$

dove a_{ij} indica la massima repulsione possibile tra le *beads i* e *j*, indica la distanza che $r_{ij} = |\boldsymbol{r}_{\mathbf{ij}}| = |\boldsymbol{r}_{\mathbf{i}} - \boldsymbol{r}_j|$ intercorre tra due *beads i* e *j*, $\widehat{\boldsymbol{r}_{ij}} = \boldsymbol{r}_{ij} / r_{ij}$ rappresenta il versore che indica la direzione del collegamento tra due *beads*, mentre r_c^{C} è il raggio di *cutoff* delle forze conservative. Intuitivamente quest'ultimo rappresenta *entro quanta distanza* tra due beads viene avvertito l'effetto delle forze repulsive conservative. Oltre alle forze conservative, nella somma delle forze agenti su ogni *bead*, vengono anche considerati i contributi dovuti alle forze dissipative $\boldsymbol{F}_{ij}^{D}$, dovute all'effetto di *drag* del fluido, e le forze stocastiche $\boldsymbol{F}_{ij}^{R}$, dovute all'effetto del rumore termico. Entrambi questi ultimi effetti agiscono da termostato garantendo il controllo termico della simulazione. Le forze dissipative e stocastiche possono essere descritte matematicamente dalle seguenti equazioni:

$$\boldsymbol{F}_{\mathbf{ij}}^{D} = -\gamma w^{D}(r_{ij})\left(\widehat{\boldsymbol{r}_{ij}} \cdot \boldsymbol{v}_{ij}\right)\widehat{\boldsymbol{r}_{ij}}, \tag{1.4}$$

$$\boldsymbol{F}_{ij}^{R} = \sigma w^{R}(r_{ij}) \frac{\zeta_{ij}}{\Delta t^{1/2}} \widehat{\boldsymbol{r}}_{ij}, \tag{1.5}$$

dove r_{ij} e $\widehat{\boldsymbol{r}}_{ij}$ sono già stati definiti precedentemente, $\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$ è la differenza di velocità tra due *beads* *i* e *j*, i termini $w^{D}(r_{ij})$ e $w^{R}(r_{ij})$ sono funzioni peso. Analogamente al caso delle forze conservative, infatti, abbiamo una distanza entro la quale le forze dissipative e stocastiche esercitano il loro effetto. Questi termini sono valutati considerando il fatto che se la distanza tra due *beads* tende ad infinito, allora le forze devono necessariamente tendere ad annullarsi:

$$w(r_{ij}) = \begin{cases} \left(1 - \frac{r_{ij}}{r_c^{D}}\right)\widehat{\boldsymbol{r}}_{ij}, & r_{ij} < r_c^{D} \\ 0, & r_{ij} \geq r_c^{D} \end{cases} \tag{1.6}$$

una volta definita la funzione peso generica, associando ad essa un raggio di r_c^{D}, che identifica il raggio d'azione delle forze dissipative, possiamo anche definire $w^{D}(r_{ij})$ e $w^{R}(r_{ij})$ come:

$$w^{D}(r_{ij}) = \left[w^{R}(r_{ij})\right]^2 = \begin{cases} \left(1 - \frac{r_{ij}}{r_c^{D}}\right)^2 \widehat{\boldsymbol{r}}_{ij}, & r_{ij} < r_c^{D} \\ 0, & r_{ij} < r_c^{D} \end{cases} \tag{1.7}$$

Gli altri termini che compaiono nelle equazioni (1.4) e (1.5) sono i parametri γ e σ. Essi corrispondono rispettivamente al coefficiente di attrito legato all'effetto della forza dissipativa, ed al coefficiente di intensità del rumore legato alla forza stocastica. Questi due fattori sono legati tra loro da una relazione chiamata *Fluctuation - Dissipation Theorem* (FDT). Questo teorema ci permette di definire:

$$\sigma = \sqrt{2k_B T\gamma} \tag{1.8}$$

in cui k_B è la costante di Boltzmann, mentre T è la temperatura del sistema. Nell'equazione (1.5), compare Δt che si può definire come il *sample rate* temporale della simulazione. In pratica possiamo definire intuitivamente questa grandezza temporale come un misura di *quanto vicini* nel tempo sono effettuati i calcoli durante la simulazioni. In ultimo definiamo ζ_{ij} come una variabile aleatoria con distribuzione Gaussiana il cui valore è indipendente per ciascuna coppia di beads interagenti e indipendente dalla scelta di Δt. La conservazione del moto è garantita dalla condizione $\zeta_{ij} = \zeta_{ji}$.

1.3 - Variazioni sul modello originale: DPD Estesa

La descrizione dinamica di un fluido rappresenta un limite del metodo DPD convenzionale, questo a causa della differenza tra le estensioni temporali in cui si verificano i fenomeni idrodinamici ed i fenomeni di diffusività. Un parametro fondamentale da tenere in considerazione durante questo tipo di simulazioni è il numero di Schmidt, definito come $Sc = \nu/D$. Esso valuta il rapporto tra viscosità cinematica e diffusività di materia: rappresenta il rapporto tra la diffusività di quantità di moto e la diffusività di materia di un dato fluido. Il valore tipico di Schmidt per l'acqua è $Sc = 1000$; le simulazioni DPD convenzionali invece restituiscono un risultato circa unitario. La causa di questa discrepanza tra i valori è da ricercare nel modo in cui vengono definite le interazioni tra le *beads*. Queste infatti essendo delle interazioni tra *sfere soffici* non riescono propriamente a descrivere gli effetti di ingabbiamento delle particelle singole soggette a potenziali diversi. Ciò permette quindi una maggiore mobilità delle *beads,* che ne enfatizzano la diffusività di materia, abbassando quindi il numero di Schmidt. La seconda causa che concorre ad abbassare il numero di Schmidt è la mancanza di effetti dovuti alla *shear dissipation,* la dissipazione di taglio dovuta a componenti di forza ortogonali alla direzione della congiungente tra due *beads*. Come si può notare dalle Eq (1.3), (1.4) e (1.5), infatti, nella definizione delle forze che intercorrono tra le *beads* non si hanno componenti tangenziali, ma solo normali alle *beads* stesse. In un primo momento si è deciso di valutare l'effetto che si ha sul valore di Schmidt al variare del parametro di simulazione legato alla forza dissipativa γ, ed il raggio di *cutoff* globale r_c. I risultati riassunti in Fig. 1.2 evidenziano che il numero di Schmidt aumenta sensibilmente in funzione del raggio di *cutoff* seconda la relazione $Sc \propto r_C^8$.

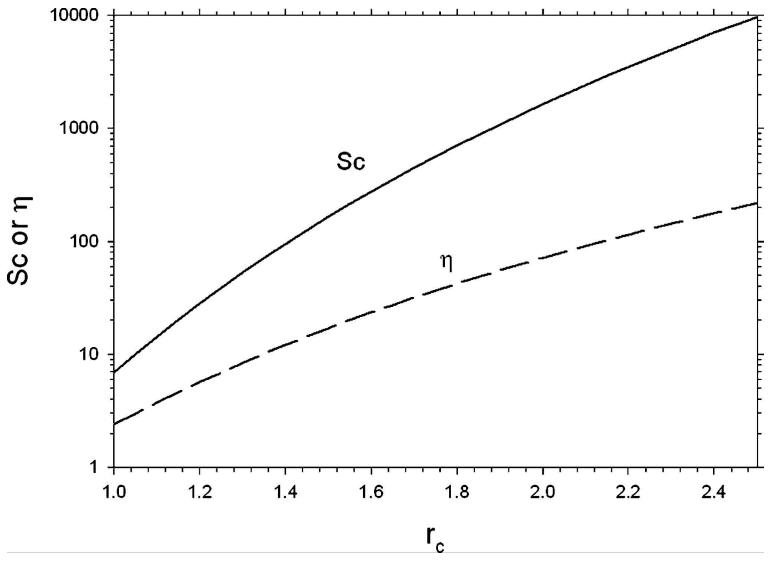


Fig. 1.2 - Andamento del numero di Schmidt in funzione del raggio di cutoff, [7]

In conseguenza all'aumento di r_c si ha un aumento significativo dello sforzo computazionale, e che per sistemi complessi si avranno tempi di calcolo estremamente lunghi. In [7] viene anche studiata la dipendenza di Schmidt dal parametro dissipativo. Anche in questo caso si può notare una diretta proporzionalità descritta dalla relazione $Sc \propto \gamma^2$. Anche in questo caso però si hanno delle conseguenze: l'aumento di γ infatti comporta una minore capacità di controllo termico del sistema, con conseguente abbassamento dello valore di Δt ed aumento del costo computazionale. Per sopperire al problema, vengono aggiunte due componenti alle equazioni delle forze del modello DPD convenzionale. Queste due componenti vengono incorporate nelle equazioni delle forze dissipative $\boldsymbol{F}_{ij}^{D}$ e stocastiche $\boldsymbol{F}_{ij}^{R}$. Le componenti che estendono le equazioni del modello originale agiscono ortogonalmente rispetto alla direzione della congiungente tra *beads*, includendo un ulteriore attrito agente in direzione perpendicolare alla velocità relativa fra le *beads*. Le equazioni risultanti del modello esteso risultano quindi essere:

$$\boldsymbol{F}_{ij}^{\mathrm{D}} = -\gamma_{\parallel} w_{\parallel}^{2}\left(\widehat{r_{ij}} \cdot v_{ij}\right)\widehat{r_{ij}} - \gamma_{\perp} w_{\perp}^{2}\left(r_{ij}\right)\left(\boldsymbol{I} - \widehat{r_{ij}}\widehat{r_{ij}}^{T}\right)v_{ij}, \tag{1.9}$$

$$\boldsymbol{F}_{ij}^{R} = \sigma_{\parallel} w_{\parallel}\left(r_{ij}\right)\frac{\xi_{ij}}{\sqrt{\Delta t}}\widehat{r_{ij}} + \sigma_{\perp} w_{\perp}\left(r_{ij}\right)\left(\boldsymbol{I} - \widehat{r_{ij}}\widehat{r_{ij}}^{T}\right)\frac{\xi_{ij}}{\sqrt{\Delta t}}. \tag{1.10}$$

dove $\gamma_{\parallel}$ e $\gamma_{\perp}$ corrispondono ai coefficienti associati alla forza dissipativa agente in direzione parallela e perpendicolare alla direzione congiungente le *beads*, ed analogamente $\sigma_{\parallel}$ e $\sigma_{\perp}$ corrispondono ai coefficienti associati all'ampiezza del rumore termico parallelo e perpendicolare. Tra gli altri termini troviamo le funzioni peso $w_{\parallel}\left(r_{ij}\right)$ e $w_{\perp}\left(r_{ij}\right)$ delle componenti parallele e perpendicolari delle forze, il vettore $v_{ij} = v_{\mathrm{i}} - v_{j}$, i parametri randomici ξ_{ij} scalare e $\boldsymbol{\xi}_{ij}$ vettoriale con distribuzione Gaussiana necessari a prevedere i fenomeni stocastici del processo, la matrice identità di secondo rango $\boldsymbol{I}$, ed il *time step* associato alla simulazione Δt. A differenza dei casi analizzati precedentemente, con l'implementazione di queste nuove componenti nel modello non si ha un aumento significativo del costo computazionale, garantendo comunque una sufficiente accuratezza delle proprietà dinamiche del sistema. Come nella formulazione originale della DPD, anche in questo caso i coefficienti legati alla forza dissipativa $\boldsymbol{F}_{ij}^{D}$ e stocastica $\boldsymbol{F}_{ij}^{R}$, sono legate tra loro grazie a FDT per mezzo della seguente relazione:

$$\sigma_{\alpha} = \sqrt{2k_{B}T\gamma_{\alpha}}, \qquad \alpha \in \{\parallel, \perp\}. \tag{1.11}$$

Il modello fornisce anche un'ulteriore modifica rispetto alla DPD originale nella definizione delle funzioni peso. Le funzioni peso sono dei termini estremamente importanti. Da esse dipende la distanza tra le beads tra cui agiscono le interazioni: nella DPD convenzionale vengono definite dalle Eq. (1.7) nel caso dei parametri dissipativi e stocastici. Nel caso invece di DPD estesa le forze peso vengono definite dalle relazioni:

$$w_{\alpha}(r_{ij}) = \left(1 - \frac{r_{ij}}{r_c D}\right)^{s_\alpha}, \qquad \alpha \in \{\parallel, \perp\} \tag{1.12}$$

dove $s_\alpha \in (0, 1]$, l'esponente è considerato uguale nei casi di funzione peso perpendicolare e parallela, quindi $s_{\parallel} = s_{\perp} = s$ che comporta come conseguenza che $w_{\parallel}(r_{ij}) = w_{\perp}(r_{ij}) = w(r_{ij})$. Uno studio condotto da Groot e Warren [8] si è occupato di derivare una relazione tra i valori della viscosità dissipativa η^D e dell'esponente *s*. Per derivare tali relazioni sono state fatte delle assunzioni semplificative, tra cui assumere una funzione di distribuzione radiale unitaria, $g(r) = 1$, imponendo come conseguenza che la densità del sistema simulato sia uniforme. Le assunzioni sono accettabili in quanto i risultati derivati sono in accordo con risultati derivati con sistemi più completi. Si definisce la viscosità dinamica, in un sistema DPD, come somma di due contributi, un termine cinetico ed un termine dissipativo. Il valore finale della viscosità dipenderà soprattutto dal termine dissipativo, ed è quindi possibile considerare il termine cinetico trascurabile. Definite le assunzioni del modello, si definisce la viscosità dinamica con la relazione:

$$\eta^D = \frac{2\pi\gamma\rho^2 r_C^5}{15}\left(\frac{1}{s+1} - \frac{4}{s+2} + \frac{6}{s+3} - \frac{4}{s+4} + \frac{1}{s+5}\right), \tag{1.13}$$

dove i termini presenti sono legati ai parametri DPD di input, troviamo il coefficiente dissipativo γ, la densità ρ, ed il raggio di *cutoff* globale r_C.
I risultati degli andamenti dei parametri fluidodinamici in funzione di *s* sono riprodotti in Fig. 1.3, se ne trae che all'aumentare del valore di *s* un aumento della viscosità dissipativa, con conseguente diminuzione della diffusività.

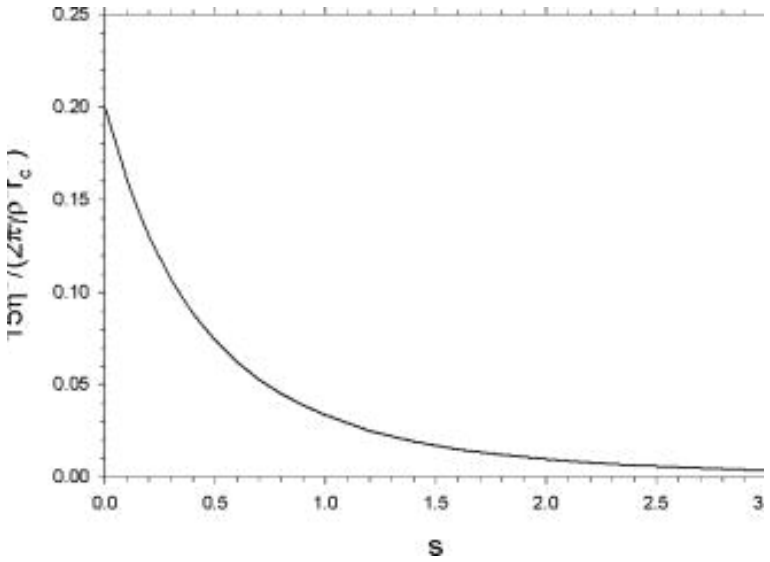


Fig. 1.3 - Andamento della viscosità dissipativa in funzione dell'esponente *s,* []

Capitolo 2

Aspetti Teorici: Meccanica Statistica

La validità della *Dissipative Particle Dynamics* introdotta ha Hoogerbrugge e Koelman [3] deve necessariamente basarsi su uno studio legato alla meccanica statistica. Questo è quindi l'obiettivo di questo capitolo che introdurrà i concetti fondamentali di questa disciplina e proporrà una descrizione accurata di quelli che sono i principali aspetti teorici di questo metodo di simulazione. Il metodo DPD, come descritto precedentemente, è concepito come un miglioramento delle tecniche di simulazioni come la *Molecular Dynamics* (MD). La complessità del comportamento idrodinamico di sistemi come i fluidi complessi, se venisse descritto per mezzo di MD, imporrebbe un costo computazionale estremamente elevato. Da qui la necessità di trovare un'alternativa che, pur fornendo dei risultati attendibili, garantisca anche un'elevata efficienza computazionale. Il metodo DPD soddisfa queste due necessità: essendo infatti un metodo *coarse-grained*, permette di mantenere una buona approssimazione di quello che è il comportamento idrodinamico del fluido, mantenendo allo stesso tempo basso il costo computazionale della simulazione. Il comportamento macroscopico di un set di particelle simulato è idrodinamico, questa caratteristica importantissima della DPD è garantita dall'imposizione della conservazione del numero di particelle del sistema (come nel caso della *dinamica Browniana* convenzionale) ed anche la conservazione della quantità di moto totale. Di conseguenza si avrà l'equazione di trasporto della quantità di moto associata all'equazione del trasporto di materia. L'energia viene conservata, di conseguenza non si rende necessario implementare l'equazione di trasporto di energia nella DPD. Tuttavia il sistema è anche isotermo, e ci sono delle relazioni che legano direttamente la temperatura ai coefficienti del modello γ e sigma. Un fondamento teorico del modello DPD si può verificare formulando l'equazione differenziale stocastica continua dell'algoritmo originale. Questo procedimento ci permette di derivare l'*equazione di Fokker-Planck* associata e studiare la sua soluzione all'equilibrio [4], da qui possiamo derivare una relazione legata alla temperatura del sistema.

2.1 - Derivazione dell'Equazione di Fokker-Planck

L'equazione di Fokker-Planck descrive l'evoluzione temporale della funzione di densità di probabilità della posizione di una particella. L'utilizzo di questa equazione permette una derivazione rigorosa delle equazioni idrodinamiche della DPD. La derivazione segue il procedimento illustrato in [4]: esso comincia con la definizione della forza totale agente su una data particella DPD, essa è definita come:

$$\dot{\boldsymbol{p}}_i = \sum_{j \neq i} \boldsymbol{F}_{ij}^{\mathrm{C}} + \sum_{j \neq i} \boldsymbol{F}_{ij}^{D} + \sum_{j \neq i} \boldsymbol{F}_{ij}^{R}, \tag{2.1}$$

dove i singoli termini delle forze sono gli stessi descritti precedentemente. Data la relatività galileiana del sistema, diciamo che le forze $\boldsymbol{F}_{\mathrm{ij}}^{D}$ e $\boldsymbol{F}_{\mathrm{ij}}^{R}$ dipendono esclusivamente da $\boldsymbol{r}_{ij} \equiv \boldsymbol{r}_i - \boldsymbol{r}_j$, termine relativo alle posizioni delle particelle, e $\boldsymbol{v}_{ij} \equiv \boldsymbol{v}_i - \boldsymbol{v}_j$, termine relativo invece alle velocità. L'assunzione dell'isotropia impone inoltre che le forze si trasformino in base alla rotazione come vettori. Infine si considera la forza dissipativa lineare in funzione della quantità di moto ed invece la forza randomica indipendente da quest'ultima. L'equazione di Fokker-Planck avrà una soluzione all'equilibrio di tipo gaussiano. Andando a sostituire dentro Eq. (2.1) le espressioni delle forze Eq. (1.3), Eq. (1.4), Eq. (1.5), possiamo trovare le equazioni di Langevin nella forma di un sistema di equazioni differenziali stocastiche (SDE):

$$\begin{cases} d\boldsymbol{r}_i = \dfrac{\boldsymbol{p}_i}{m_i} dt, \\ d\boldsymbol{p}_i = \left[\displaystyle\sum_{j \neq i} F_{ij}^{\mathrm{C}}(\mathrm{r_{ij}}) + \sum_{j \neq i} \left(-\gamma w_D(r_{ij}) \left(\boldsymbol{e}_{ij} \cdot \boldsymbol{v}_{ij} \right) \boldsymbol{e}_{ij} \right) \right] dt \\ \qquad\qquad + \displaystyle\sum_{j \neq i} \sigma w_R(r_{ij}) \, \boldsymbol{e}_{ij} d\boldsymbol{W}_{ij}, \end{cases} \tag{2.2}$$

dove m_i è la massa della particella i mentre $d\boldsymbol{W}_{ij} = d\boldsymbol{W}_{ji}$ sono incrementi del processo di Wiener. Le SDE trovate sono simili a quelle che descrivono il metodo proposto da Hoogerbrugge e Koelman. Va comunque considerato il fatto che si sono considerate delle funzioni peso non equivalenti invece di imporre $w_D(r) = w_R(r)$ come nel lavoro originale [8]. Possiamo quindi passare alla derivazione dell'equazione di Fokker-Planck che corrisponda ad Eq. (2.3). Viene prima di tutto definito il differenziale df di una generica funzione f del secondo ordine che sostituisca la SDE. Per applicazione del lemma di Ito troviamo che dW è un infinitesimale di ordine ½: questo ci consente di ricavare $\langle df / dt \rangle$ e quindi formulare l'equazione di Fokker-Planck che governa l'evoluzione temporale della

funzione distributiva delle posizioni e delle quantità di moto delle particelle del sistema, $\rho(r, p; t)$. L'equazione di Fokker-Planck quindi risulta essere:

$$\partial_t \rho(r, p; t) = L_C \rho(r, p; t) + L_D \rho(r, p; t), \tag{2.3}$$

definendo gli operatori L_C e L_D come:

$$\begin{cases} L_C \rho(r, p; t) \equiv -\left[\sum_i \frac{p_i}{m} \frac{\partial}{\partial r_i} + \sum_{i, j \neq i} F_{ij}^{C} \frac{\partial}{\partial \mathbf{p_i}}\right] \rho(r, p; t), \\ L_D \rho(r, p; t) \equiv \sum_{i, j \neq i} e_{ij} \frac{\partial}{\partial \mathbf{p_i}} \left[\gamma w_D(\mathbf{r_{ij}}) (\mathbf{e_{ij}} \cdot \mathbf{v_{ij}}) + \frac{\sigma^2}{2} w_R{}^2(\mathbf{r_{ij}}) \mathbf{e_{ij}} \left(\frac{\partial}{\partial \mathbf{p_i}} - \frac{\partial}{\partial \mathbf{p_i}}\right)\right] \boldsymbol{\rho(\mathbf{r}, \mathbf{p}; \mathbf{t})} \end{cases}$$

L'operatore L_C è l'operatore di Liouville per un sistema Hamiltoniano interagente per mezzo della forza conservativa $\boldsymbol{F}^C$, mentre l'operatore L_D serve a considerare le forze dissipative e randomiche del sistema. La soluzione dell'equazione di Fokker-Planck allo stazionario può essere trovata imponendo $\partial_t \rho(r, p; t) = 0$. Nei sistemi Hamiltoniani, la distribuzione finale del sistema allo stazionario è una funzione delle proprietà fisse del sistema, assumendo il sistema ergodico. Da qui si intuisce che la scelta dell'*ensemble* di equilibrio del sistema va valutata in funzione di quelle che sono le condizioni iniziali. Per Eq. (2.3) invece si ha che la distribuzione all'equilibrio è unica e indipendentemente dalla distribuzione iniziale il sistema tenderà sempre alla stessa distribuzione allo stazionario. Troviamo ora una soluzione a Eq. (2.3) imponendo come distribuzione all'equilibrio quella di Gibbs-Boltzmann:

$$\rho^{eq}(r, p) = \frac{1}{Z} exp\left[-\frac{\left(\sum_i \frac{p_i{}^2}{2m_i} + V(r)\right)}{k_B T}\right], \tag{2.5}$$

dove H è l'Hamiltoniano del sistema, V è la funzione potenziale associata alle forze conservative, k_B è la costante di Boltzmann, T è la temperatura del sistema all'equilibrio e Z è la funzione di partizione. La soluzione all'equilibrio per il sistema conservativo è il *canonical ensemble* (NVT), cioè per $L_C \rho^{eq} = 0$. Possiamo anche imporre $L_D \rho^{eq} = 0$ trovando come uniche soluzioni possibili le seguenti relazioni:

$$w_R(r) = \left[w_D(r)\right]^{1/2}, \tag{2.6}$$

$$\sigma = \left(2 k_B T \gamma\right)^{1/2}. \tag{2.7}$$

Queste due relazioni le abbiamo già incontrate nel capitolo 1 in cui si introducevano le relazioni della DPD secondo la trattazione originale [8]. Questa relazione prende il nome di *fluctuation-dissipation theorem* (FDT) per il metodo DPD, concettualmente questa relazione va ad imporre che ci sia un bilanciamento

tra le forze dissipative e forze stocastiche del sistema, dalla relazione $L_D \rho^{eq} = 0$. Eq. (2.6) ed Eq. (2.7) sono quindi le relazioni fondamentali necessarie per dimostrare che il sistema tende ad una distribuzione di equilibrio, e che una soluzione stazionaria dell'equazione di Fokker-Planck è la distribuzione di Gibbs-Boltzmann [5], i risultati trovati dimostrano che la soluzione quindi è stabile, e che il sistema iniziale tende sempre all'equilibrio indipendentemente dalle condizioni iniziali.

2.2 - Criteri per la scelta dei parametri di simulazione

2.2.1 - Scelta di Δt, γ e σ

Il parametro temporale Δt per la simulazione è scelto considerando un compromesso tra la velocità di simulazione - più è basso il Δt e più è lenta la simulazione - ed il rispetto delle condizioni di equilibrio. Questo controllo sulle condizioni di equilibrio viene effettuato monitorando la temperatura del sistema. Nel caso il parametro temporale non fosse sufficientemente piccolo si avrà quindi una differenza del valore della media delle temperature calcolate dalla simulazione ed il valore della temperatura impostata come input. La temperatura è misurata in funzione della velocità media delle particelle:

$$k_B T = \frac{\langle v^2 \rangle}{3}, \tag{2.8}$$

mentre il valore di temperatura nelle simulazioni viene impostato $k_B T = 1$. I parametri legati alle forze stocastiche ed alle forze dissipative vengono decisi sulla base di quelle che sono le relazioni in Eq. (1.4) ed Eq. (1.5), queste equazioni come detto nel paragrafo 1.1 sono valide, in quanto è dimostrato dalla meccanica statistica che garantiscono che le condizioni di equilibrio vengano rispettate.

2.2.2 - Scelta del parametro repulsivo *a*

Nel modello DPD una fase cruciale della definizione dei parametri di simulazione consiste nel parametro di repulsione *a* tra *beads*. Questo parametro repulsivo compare nella definizione delle forze conservative, e corrisponde ad un potenziale che deve simulare il comportamento delle beads di particelle interagenti tra loro. Di conseguenza non è simile al potenziale che descrive le interazioni tra atomi ma descrive un potenziale che viene detto *a sfere soffici.* Per descrivere e valutare correttamente questo parametro repulsivo, ed in generale la termodinamica di un fluido che descriviamo con il modello delle *sfere soffici,* si devono descrivere in modo coerente le fluttuazioni presenti a livello di particelle del fluido. Le fluttuazioni del liquido sono definite dalla compressibilità del sistema, di conseguenza si definisce il parametro adimensionale:

$$\kappa^{-1} = \frac{1}{\rho k_B T \kappa_T} = \frac{1}{k_B T}\left(\frac{\partial p}{\partial \rho}\right)_T \tag{2.9}$$

dove il parametro κ_T corrisponde al parametro di compressibilità isoterma del fluido DPD, definita dalla relazione:

$$\kappa_T = -\frac{1}{V}\left(\frac{\partial V}{\partial p}\right)_T = \frac{1}{\rho}\left(\frac{\partial p}{\partial \rho}\right)_T . \tag{2.10}$$

Basandosi sulla teoria della perturbazione dei fluidi Weeks-Chandler-Anderson [7], si è andata a definire la seguente relazione di scaling:

$$\kappa^{-1} = \frac{1}{k_B T}\left(\frac{\partial p}{\partial \rho}\right)_{sim} = \frac{1}{k_B T}\left(\frac{\partial p}{\partial n}\right)_{exp} \tag{2.11}$$

dove *n* è la funzione di densità numerica per il sistema considerato. Noi utilizzeremo una generalizzazione di queste equazioni che considera un numero di *coarse-graining* generico *Nm,* avremo quindi che:

$$\frac{1}{k_B T}\left(\frac{\partial p}{\partial \rho}\right)_{sim} = \frac{Nm}{k_B T}\left(\frac{\partial p}{\partial n}\right)_{exp} \tag{2.12}$$

dove *Nm* è il numero di *coarse-graining*, cioè il numero di molecole che andremo ad includere dentro la singola *bead* nella nostra simulazione DPD. Per mezzo di simulazioni numeriche in funzione della densità, e per diversi valori del parametro repulsivo, Groot e Warren [9] propongono anche un'equazione di stato viriale nella forma:

$$p = \rho k_B T + \frac{1}{3V}\left\langle \sum_{j>i} \left(\boldsymbol{r}_i - \boldsymbol{r}_j \right) \cdot \boldsymbol{f}_i \right\rangle \tag{2.13}$$

dove si impone che $\boldsymbol{f}_i = \boldsymbol{F}_{ij}^{\mathrm{C}}$ in quanto si è dimostrato che all'equilibrio il nostro sistema tende alla distribuzione di Gibbs-Boltzmann. L'Eq. (2.13) è stata ricavata per un sistema monocomponente e può essere anche riscritta in forma integrale:

$$p = \rho k_B T + \frac{2\pi}{3}\rho^2 \int_0^1 r f(r)\, g(r)\, r^2\, dr, \tag{2.14}$$

dove *g(r)* è la funzione di distribuzione radiale. I risultati delle simulazioni riportano che tutti i risultati per *a* diverse ed alte densità ρ ricadono sulla stessa curva di valori, di conseguenza si può estrapolare una relazione per calcolare la pressione per $\rho > 2$, la nostra equazione di stato è quindi:

$$p = \rho k_B T + \alpha a \rho^2 \tag{2.15}$$

Questa relazione ci consente di derivare, dalla sostituzione in Eq. (2.10), l'espressione della compressibilità adimensionale:

$$\kappa^{-1} = 1 + \frac{2\alpha a \rho}{k_B T} \approx 1 + \frac{0.2\, a\rho}{k_B T}. \tag{2.16}$$

Il valore di α è quello riportato in [9], ed è $\alpha = 0.101 \pm 0.001$. Nota Eq. (2.16), è facile a questo punto ricavare l'espressione di *a* per un sistema DPD:

$$a = k_B T \left(\frac{\kappa^{-1} - 1}{0.2\rho} \right). \tag{2.17}$$

L'Eq. (2.17) è stata ricavata per Nm unitario, la relazione generalizzata per trovare *a* diventa quindi:

$$a = k_B T \left(\frac{\kappa^{-1} Nm - 1}{0.2\rho} \right). \tag{2.18}$$

2.3 - Integrazione temporale nel metodo DPD

Come visto in Sez. 1, il modello DPD si basa sulla definizione di quelle che sono equazioni di tipo stocastico. A differenza del caso di dinamica molecolare convenzionale, la DPD presenta maggiori difficoltà per quanto riguarda l'aspetto dell'integrazione temporale proprio a causa della presenza di questo tipo di equazioni. Una prima soluzione al problema viene trovata utilizzando uno schema di integrazione temporale basato sull'algoritmo di Eulero:

$$\boldsymbol{r}_i(t + \Delta t) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t)\, \Delta t, \tag{2.19}$$

$$\boldsymbol{v}_i(t + \Delta t) = \boldsymbol{v}_i(t) + \boldsymbol{f}_i(t)\, \Delta t, \tag{2.20}$$

$$\boldsymbol{f}_i(t + \Delta t) = \boldsymbol{f}_i\left(\boldsymbol{r}_i(t + \Delta t),\ \boldsymbol{v}_i(t + \Delta t) \right). \tag{2.21}$$

Questo sistema numerico consente di calcolare le velocità e le posizioni delle particelle ad un certo step temporale semplicemente conoscendo lo step precedente. Nonostante la facilità di implementazione, questo schema numerico non consente la reversibilità temporale delle traiettorie calcolate e, siccome può portare ad un accumulo di energia nel sistema, può anche limitare quello che è il controllo termico del processo. Questo problema in particolare è problematico dal punto di vista del costo computazionale della simulazione. Una soluzione differente che si è proposta per l'integrazione temporale per il metodo DPD è l'algoritmo di Verlet: questo algoritmo permette un buon compromesso tra costo

computazionale ed accuratezza del risultato, inoltre corregge quei problemi che si verificano con l'utilizzo dell'algoritmo di Eulero. Nel metodo DPD, sviluppato da Groot e Warren si è implementata una versione modificata del *velocity*-Verlet [9], che può essere descritta per mezzo delle seguenti equazioni:

$$\boldsymbol{r}_i(t+\Delta t) = \boldsymbol{r}_i(t) + \Delta t \boldsymbol{v}_i(t) + \frac{1}{2}(\Delta t)^2 \boldsymbol{f}_i(t), \tag{2.22}$$

$$\tilde{\mathbf{v}}_i(t+\Delta t) = \boldsymbol{v}_i(t) + \lambda \Delta t \boldsymbol{f}_i(t), \tag{2.23}$$

$$\boldsymbol{f}_i(t+\Delta t) = \boldsymbol{f}_i\left[\boldsymbol{r}_i(t+\Delta t), \tilde{\mathbf{v}}_i(t+\lambda\Delta t)\right], \tag{2.24}$$

$$v_i(t+\Delta t) = v_i(t) + \frac{1}{2}\Delta t\left[f_i(t) + f_i(t+\Delta t)\right]. \tag{2.25}$$

dove oltre ai termini già noti, quelli delle forza, velocità e posizione, viene introdotto un termine legato alle fluttuazioni stocastiche del sistema, il termine λ. L'algoritmo appena presentato è definito *predictor-corrector*, cioè un algoritmo che iterativamente va a predire la velocità al tempo $t + \Delta t$ a partire da quella calcolata al tempo t, il valore *predetto* di velocità $\tilde{\boldsymbol{v}}_i$ servirà poi per calcolare il valore della forza. L'ultima equazione, quella della velocità $\boldsymbol{v}_i$, calcola il valore *corretto* della velocità. Questa versione di integrazione temporale di Verlet non aumenta il costo computazionale, avendo tuttavia il vantaggio di questa doppia iterazione che rende più accurato il calcolo finale. Molti altri integratori sono stati sviluppati, altri esempi possono essere per esempio gli algoritmi *leap-frog,* basato sui metodi alle differenze finite, Verlet *self-consistent* [12]*,* definito così in quanto ripete l'operazione di iterazione fino ad ottenere un valore stabile di velocità, ed il metodo di *splitting* di Shardlow [13], che si basa sul concetto di *dividere* il calcolo delle forze come somma di due termini, uno conservativo ed uno dissipativo-stocastico. Il metodo di Shardlow funziona implementando due metodi diversi per l'integrazione dei due termini, il termine conservativo si calcola utilizzando il sopracitato metodo *velocity*-Verlet modificato, il secondo termine invece dissipativo-stocastico viene integrato implicitamente in modo tale da permettere la conservazione della quantità di moto. Quest'ultimo metodo è quello scelto per il lavoro di tesi.

2.3.1 - Algoritmo di Shardlow

Il metodo di integrazione temporale di Shardlow [13] è debolmente convergente. Come già detto questo integratore opera una scomposizione della forza in due termini: un termine conservativo, che viene integrato con lo schema *velocity*-Verlet, ed un termine dissipativo-stocastico, che viene integrato implicitamente per conservare la quantità di moto. Le equazioni del metodo di Shardlow del primo ordine per le *beads i* e *j* sono:

$$v_{i,\,n+1/2} = v_{i,\,n} - \frac{1}{2m}\gamma_{ij}\left[w_{ij}^{D}\left(r_{ij,\,n}\right)\right]\left[v_{ij,\,n}\cdot\hat{\mathbf{r}}_{ij,\,n}\right]\hat{\mathbf{r}}_{ij,\,n}dt + \frac{1}{2m}\sqrt{2\gamma_{ij}k_BT}\left[w_{ij}^{R}\left(r_{ij,\,n}\right)\right]\psi_{ij}\hat{r}_{ij,\,n}dt^{1/2} \quad (2.26)$$

$$v_{j,\,n+1/2} = v_{j,\,n} + \frac{1}{2m}\gamma_{ij}\left[w_{ij}^{D}\left(r_{i,j,\,n}\right)\right]\left[v_{ij,\,n}\cdot\hat{\mathbf{r}}_{ij,\,n}\right]\hat{\mathbf{r}}_{ij,\,n}dt - \frac{1}{2m}\sqrt{2\gamma_{ij}k_BT}\left[w_{ij}^{R}\left(r_{ij,\,n}\right)\right]\psi_{ij}\hat{\mathbf{r}}_{ij,\,n}dt^{1/2} \quad (2.27)$$

$$v_{i,\,n+1} = v_{i,\,n+1/2} + \frac{1}{2m}\sqrt{2\gamma_{ij}k_BT}\left[w_{ij}^{R}\left(r_{ij,\,n}\right)\right]\psi_{ij}\hat{r}_{ij,\,n}dt^{1/2} - \frac{1}{2m}\,\frac{\gamma_{ij}\left[w_{ij}^{D}\left(r_{ij,\,n}\right)\right]dt}{1+\left[w_{ij}^{D}\left(r_{ij,\,n}\right)\right]dt}\left\{\left[\mathbf{v}_{\mathrm{ij,\,n+1/2}}\cdot\hat{\mathbf{r}}_{\mathbf{ij,\,n}}\right]\hat{r}_{ij,\,n} + \sqrt{2\gamma_{ij}k_BT}\left[w_{ij}^{R}\left(r_{ij,\,n}\right)\right]\psi_{ij}\hat{\mathbf{r}}_{\mathbf{ij,\,n}}dt^{1/2}\right\}$$

(2.28)

$$v_{j,\,n+1} = v_{j,\,n+1/2} - \frac{1}{2m}\sqrt{2\gamma_{ij}k_BT}\left[w_{ij}^{R}\left(r_{ij,\,n}\right)\right]\psi_{ij}\hat{r}_{ij,\,n}dt^{1/2} + \frac{1}{2m}\,\frac{\gamma_{ij}\left[w_{ij}^{D}\left(r_{ij,\,n}\right)\right]dt}{1+\left[w_{ij}^{D}\left(r_{ij,\,n}\right)\right]dt}\left\{\left[v_{ij,\,n+1/2}\cdot\hat{\mathbf{r}}_{ij,\,n}\right]\hat{\mathbf{r}}_{ij,\,n} + \sqrt{2\gamma_{ij}k_BT}\left[w_{ij}^{R}\left(r_{ij,\,n}\right)\right]\psi_{ij}\hat{r}_{ij,\,n}dt^{1/2}\right\}$$

(2.29)

Successivamente è necessario calcolare i valori delle velocità e delle forze con il metodo *velocity*-Verlet modificato descritto precedentemente.

2.3.2 - Scelta dell'integratore temporale

Come detto precedentemente, lo schema di integrazione temporale scelto è il metodo si *splitting* Shardlow. I parametri da valutare quando si sceglie un algoritmo da implementare per una simulazione sono spesso molto simili, in generale si deve scegliere quello che meglio funge da compromesso tra stabilità, costo computazione, velocità ed accuratezza dei risultati. In questo caso si sono considerati il controllo termico, necessario per garantire che il risultato sia affidabile, ed il controllo della funzione di distribuzione radiale delle particelle simulate. Uno studio di Chaundhri e Lukes [35] si occupa di verificare gli effetti di diversi integratori temporali per dei set di simulazioni DPD a parametri fissi. I parametri dei set sono illustrati in [35], insieme agli schemi di integrazione testati sono l'integratore GW, cioè il metodo *velocity*-Verlet già citato, il metodo *self-consistent* SCPHF, il *self-consistent velocity*-Verlet SCVV, precedentemente citato, ed il metodo di Shardlow del primo e del secondo ordine.

Una volta portate a termine le simulazioni, si sono analizzati prima gli effetti dei vari schemi ottenuti da Chaundhri e Lukes [35] sul controllo termico e sulla funzione di distribuzione radiale RDF. I risultati indicano che per i set di parametri fino a $\Delta t = 0.04$ e $\sigma = 3, 6$ gli integratori GW e SCVV garantiscono un ottimo controllo termico, mentre se si incrementa ad 8 il valore stocastico a $\Delta t = 0.04$, il metodo GW comincia a deviare, che è un risultato aspettato. Se si osserva invece la RDF, si nota che per i valori bassi del parametro temporale si hanno meno fluttuazioni e rumore che invece con valori più alti. Lo schema di integrazione SCPHF valutato per gli stessi parametri mostra un buon controllo termico, con una deviazione di solo l'1-2 % sulla deviazione standard tra il valore calcolato della T e quello di input della simulazione, con anche delle RDF più stabili rispetto al metodo GW. In ultimo si sono testati i metodi di *splitting*-Shardlow del primo e del secondo ordine, i quali risultati finali dimostrano che questi schemi di integrazione garantiscono la migliore conservazione termica del sistema tra tutti quelli testati, l'errore rispetto alla temperatura impostata per la simulazione è minore dell'1 %. Queste conclusioni sono state anche verificate in altri studi indipendenti da parte dello stesso Shardlow [13].

Come detto precedentemente, non basta che un algoritmo sia il più accurato, ma deve anche garantire il minor costo computazionale possibile. Nello stesso studio di Chaundhri e Lukes [35] viene anche analizzata l'efficienza computazionale dei differenti schemi: la velocità viene calcolata come il tempo di completamento della stessa simulazione sulla stessa macchina, l'unica variazione consiste nei differenti metodi impostati. Il metodo più veloce risulta essere GW, questo risultato è confermato anche da altri studi comparativi, il metodo SCVV risulta essere invece di circa 1.6 volte meno veloce di GW, i metodi di *splitting*-Shardlow del primo e del secondo ordine risultano essere rispettivamente 1.9 e 2.8 volte meno veloci di GW, mentre lo schema più lento di tutti è SCPHF che è 5.1 volte meno veloce di GW.

Gli studi di Espanol e Serrano [36] hanno dimostrato, per mezzo di MCT (*Mode Coupling Theory*), che la funzione di autocorrelazione della velocità VACF decade esponenzialmente a bassi valori del parametro di attrito adimensionale $\bar{\Lambda} = \bar{\gamma}\bar{r}_C \big/ d\bar{v}_T$, e fattore di sovrapposizione $\bar{s} = \bar{r}_C \big/ n^{-1/d}$, il parametro d definisce le dimensioni del sistema, $\bar{v}_T$ è la velocità delle particelle dovuta all'agitazione termica. Sono state quindi calcolate le VACF per 4 casi specifici - variando il valore di $\bar{\Lambda}$, per $\bar{s} = 1.4422$ e d. Per ognuno di questi casi vengono quindi eseguite un set di simulazioni in cui, a parametri costanti e timestep di 0.04, vengono implementati i diversi schemi di integrazione temporale GW, SCPHF, SCVV, S1, S2. I risultati finali, riportati in [35], mostrano che gli integratori GW e SCVV mostrano differenze al tempo zero della VACF, dove invece il valore dovrebbe essere di 3 siccome $\mathrm{VACF} \to 3\overline{k_B T}$ per $t_{DPD} \to 0$

posto che nelle simulazioni effettuate $\overline{k_B T} = 1$. Come invece ci si aspettava, il migliore controllo di temperatura risulta essere garantito da S1, S2, SCPHF. Questo risultato vale per tutti i valori di $\overline{\Lambda}$. Si è verificato anche l'effetto dell'integratore temporale sull'andamento della funzione di autocorrelazione degli stress SACF, essa è calcolata utilizzando la relazione di Irving-Kirkwood:

$$\overline{\Gamma}_{pq} = -\frac{1}{\overline{\Omega}}\left\{\sum_i \overline{m}_i\left[\bar{v}_{i,p}\bar{v}_{i,q}\right] + \frac{1}{2}\sum_i \sum_{j \neq i} \bar{r}_{ij,p}\overline{F}_{ij,q}{}^{D}\right\}, \tag{2.30}$$

dove $\overline{\Gamma}_{pq}$ è il *pq*-esimo termine del tensore degli stress. In Eq.() non vengono considerate le forze stocastiche e conservative, il motivo è che solo le forze dissipative contribuiscono a quelli che sono i valori calcolati finali dei coefficienti delle proprietà di trasporto. La SACF viene calcolata usando l'espressione $\left\langle \overline{\Gamma}_{pq}(t)\,\overline{\Gamma}_{pq}(0) \right\rangle$, il cui valore al tempo iniziale sarà dato dall'espressione $\left\langle \overline{\Gamma}_{xx}(t)\,\overline{\Gamma}_{xx}(0) \right\rangle_t$, valendo $\overline{\Gamma}_{xx}$ circa 3 - perchè è un termine che compare nel tensore della pressione e per cui quindi vale il principio di equipartizione - si ha quindi che $\left\langle \overline{\Gamma}_{xx}(t)\,\overline{\Gamma}_{xx}(0) \right\rangle_t = 9$. Anche in questo caso, osservando le relazioni graficate in fig.(), si può notare che l'andamento della SACF è fortemente influenzato dal tipo di integratore scelto per la simulazione e da $\overline{\Lambda}$. Si può notare che all'aumentare di $\overline{\Lambda}$ si un aumento della velocità di decadimento. Questo è un risultato atteso in quanto la maggiore dissipazione introdotta dal sistema tenderà a far decadere più velocemente lo *shear-stress* del sistema. A bassi valori di attrito $\overline{\Lambda}$ gli integratori GW ed SCPHF mostrano un una deviazione maggiore degli integratori S1 ed S2 anche se comparabile. All'aumentare del valore del parametro di attrito adimensionale si ha anche un incremento dell'errore, in particolare si ha che gli schemi GW ed SCPHF restituiscono valori non sono accurati. Si ha una tendenza invertita invece per quanto riguarda l'errore associato a SCVV, infatti all'aumentare di $\overline{\Lambda}$, SCVV tenderà a compiere più iterazioni per stabilizzare la temperatura iterando il termine dissipativo. Alla luce dei risultati riscontrati si è quindi deciso di scegliere di adoperare il metodo *splitting*-Shardlow, nonostante la sua debole convergenza infatti si è rivelato il metodo di integrazione temporale che garantisce la migliore stabilità dell'algoritmo ed accuratezza dei risultati.

Capitolo 3

Proprietà di Trasporto nella Dissipative Particle Dynamics

Il seguente lavoro di tesi consiste nel calcolo e nel monitoraggio delle proprietà di trasporto del sistema fluido simulato mediante *Dissipative Particle Dynamics.* Per valutare correttamente queste proprietà è quindi opportuno avere delle relazioni come termine di paragone per i risultati ottenuti dalle simulazioni che leghino quest'ultime a quelli che sono i parametri della simulazione. Di queste relazioni ne sono state derivate diverse. Il modello matematico più completo di queste relazioni è stato sviluppato da Marsh *et al.* [37]: esso si basa sulla teoria cinetica che lega la viscosità, il coefficiente di diffusione ed, in generale, i coefficienti di trasporto ai parametri iniziali impostati per la simulazione.

3.1 - Relazioni tra proprietà di trasporto e parametri DPD

Come detto precedentemente, le relazioni che legano i parametri DPD che vengono impostati in input e i valori delle proprietà di trasporto vengono ipotizzate da Groot e Warren [9] con una procedura semplificata. La diffusività viene calcolata con la formula:

$$D \approx {45k_BT}\Big/{2\pi\gamma\rho r_C^3}. \tag{3.1}$$

Il termine di diffusività compare anche nell'espressione della viscosità ν:

$$\nu \approx {D}\Big/{2} + {2\pi\gamma\rho r_C^5}\Big/{1575}. \tag{3.2}$$

La viscosità quindi viene calcolata per mezzo della somma di due termini, il primo dovuto alla diffusività, quindi dovuto al flusso della quantità di moto nel sistema, ed il secondo dovuto alle forze dissipative, che è legato alle forze di attrito che intercorrono tra le *beads.* Infine, la relazione analitica per descrivere il numero di Schmidt è:

$$Sc \approx 1/2 + \left(2\pi\gamma\rho r_C^4\right)^2 / 70875 k_B T. \tag{3.3}$$

Sono state utilizzate queste relazioni per un set di parametri, e confrontati con i valori calcolati per mezzo della simulazione: si ha che le stime rientrano in un range di errore $\approx$ 10-30%.

3.2 - Calcolo delle proprietà di trasporto nella DPD

3.2.1 - Viscosità

Il lavoro di tesi, ed in generale la caratterizzazione reologica di un fluido, si basa sulla caratterizzazione della viscosità. Il metodo DPD ha come obiettivo l'investigazione di scale temporali sufficienti a descrivere il comportamento idrodinamico del fluido, e la viscosità viene calcolata numericamente a causa della mancanza di relazioni analitiche accurate. Le relazioni numeriche possono essere catalogate in due gruppi principali: metodi di equilibrio, in cui viene calcolata la viscosità a partire da quelle che sono le condizioni di equilibrio del sistema, ed i metodi di non-equilibrio, in cui si valuta la risposta del sistema a cui si impone un gradiente di velocità, flusso di quantità di moto, o una forza. In seguito la viscosità è calcolata per mezzo della formula di Newton generalizzata:

$$j_z(p_x) = -\mu \frac{\partial u_x}{\partial z} \tag{3.4}$$

dove $j_z(p_x)$ è il flusso della componente su x di quantità di moto p_x, in direzione z.

3.2.2 - Metodo di Green-Kubo

Il metodo di Green-Kubo è un metodo di equilibrio che calcola la viscosità dinamica a partire dalla funzione di autocorrelazione degli stress SACF. La spiegazione fisica che spiega il metodo si basa sull'ipotesi di Onsager [24], secondo la quale il raggiungimento di uno stato di equilibrio di un sistema perturbato da una forza esterna è legato alle dissipazioni delle fluttuazioni del sistema stesso. I fenomeni di rilassamento e di dissipazione delle fluttuazioni sono legati dal teorema di fluttuazione-dissipazione FDT descritto in Eq. (2.8). Il metodo di Green-Kubo per il calcolo della viscosità dinamica è descritto dalla seguente espressione:

$$\eta = \frac{V}{k_B T} \int_0^{+\infty} \left\langle \sigma_{\alpha\beta}(t_0)\, \sigma_{\alpha\beta}(t_0 + t) \right\rangle, \qquad (\alpha \neq \beta), \tag{3.5}$$

dove V è il volume del sistema e $\sigma_{\alpha\beta}$ corrispondono alle componenti fuori diagonale del tensore degli sforzi. La descrizione del tensore degli sforzi nelle simulazione DPD avviene per mezzo della formula di Irving-Kirkwood []:

$$\sigma_{\alpha\beta} = \frac{1}{V}\sum_{i}\sum_{j<i} r_{ij,\alpha}(t)\, F_{ij,\beta}(t) + \frac{1}{V}\sum_{i} m v_{i,\alpha}(t)\, v_{i,\beta}(t), \tag{3.6}$$

dove $F_{ij,\beta} = F^{C}_{ij,\beta} + F^{D}_{ij,\beta} + F^{R}_{ij,\beta}$, cioè la somma delle tre forze che caratterizzano il modello DPD (conservative, dissipative, stocastiche), gli indici α e β indicano gli indici del tensore degli sforzi mentre *i* e *j* sono indici relativi alle *beads*. I termini $v_{i,\alpha}$ e $r_{ij,\alpha}$ rappresentano rispettivamente la componente tensoriale α della velocità della *bead i*-esima e la componente tensoriale α della distanza tra *beads i*-esima e *j*-esima. Questo è il metodo scelto per il lavoro di tesi. Essendo un metodo di equilibrio risulta anche essere molto vantaggioso in quanto, non si va a modificare quella che è la configurazione di un sistema introducendo un gradiente di velocità come nei casi dei metodi di non-equilibrio.

3.2.3 - Metodo di Einstein-Helfand

Un altro metodo di equilibrio per calcolare la viscosità di un sistema è il metodo di Einstein-Halfand [20], le equazioni che descrivono il modello sono:

$$\eta = \lim_{t\to\infty} \frac{V}{2k_B T}\left\langle \left[G_{\alpha\beta}(t) - G_{\alpha\beta}(0) \right]^2 \right\rangle, \qquad (\alpha \neq \beta), \tag{3.7}$$

dove $G_{\alpha\beta}(t)$ è il momento di Helfand, descritto dalla relazione:

$$G_{\alpha\beta}(t) = \sum_{i} m_i r_{i,\alpha}(t)\, v_{i,\beta}(t). \tag{3.8}$$

Inoltre, dato che il metodo di Einstein-Helfand è equivalente al metodo di Green-Kubo, si può esprimere il momento di Helfand con la relazione tensoriale di Irving-Kirkwood:

$$G_{\alpha\beta}(t) = G_{\alpha\beta}(0) + \int_{0}^{t} \sigma_{\alpha\beta}(\tau)\, d\tau. \tag{3.9}$$

3.2.4 - Metodo di Müller-Plathe

Il metodo di Müller-Plathe [23], a differenza dei metodi appena descritti, è un metodo di non-equilibrio. Come già descritto precedentemente, i metodi di non equilibrio non si basano sul calcolo delle proprietà - in questo caso la viscosità dinamica - a partire dalle condizioni di equilibrio del fluido, ma si basa sull'imposizione nel sistema di un gradiente o di un flusso di proprietà e, successivamente, andare a ricavare la proprietà desiderata per mezzo della formula generalizzata di Newton. Il seguente metodo consiste nell'imporre un flusso di quantità di moto, esso genererà un gradiente di velocità che ci servirà per il calcolo della viscosità come prima descritto. Il sistema, rappresentato come una *box* periodica, viene diviso il *layers* sovrapposti e perpendicolari all'asse *z,* a questo punto si procede andando ad identificare due *beads* in particolare: la bead con la maggiore quantità di moto dal *top layer,* e la bead con la minor quantità di moto dal *middle layer*. A questo punto la quantità di moto delle due beads sono scambiati, in questo modo si va a creare un flusso più intenso di quantità di moto nel *middle layer* e questo crea un gradiente di velocità nel sistema. Questo meccanismo è descritto graficamente in Fig. 3.1.

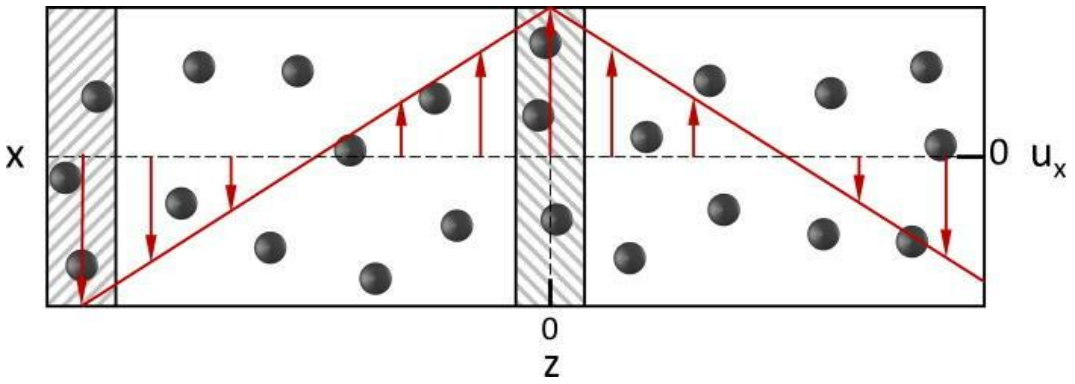


Fig. 3.1 - Metodo di non-equilibrio Müller-Plathe, []

Il flusso di quantità di moto può essere descritto dalla seguente relazione:

$$j_z(p_x) = \frac{P_x}{2tL_xL_y}, \tag{3.10}$$

dove i termini $L_{x,y}$ indicano le dimensioni della *box* periodica del sistema, P_x rappresenta la quantità di moto totale trasferita da un *layer* all'altro. Il valore di questo termine viene controllato andando ad imporre tasso di scambio delle quantità di moto delle *beads:* più è alto lo scambio infatti e più netto sarà il gradiente imposto.

3.2.5 - Diffusività

La diffusione è il processo fisico per il quale un gradiente di concentrazione in un sistema viene annullato senza un flusso macroscopico di materia, ma semplicemente grazie al moto particellare. Nel metodo DPD il coefficiente di diffusione può essere calcolato per mezzo della relazione di Einstein o l'approccio di Green-Kubo [28]. Nel metodo di Einstein il coefficiente di diffusione è calcolato per mezzo del *mean square displacement* (MSD) delle *beads*:

$$D = \lim_{t \to \infty} \frac{\left\langle \left[\boldsymbol{r}\left(t_0 + t \right) - \boldsymbol{r}\left(t_0 \right) \right]^2 \right\rangle}{6t}, \tag{3.11}$$

dove $\langle \cdots \rangle$ rappresenta la media d'insieme. Differentemente da questo metodo, l'approccio di Green-Kubo non comporta il calcolo della MSD: infatti esso relaziona il coefficiente di diffusione alla funzione di *autocorrelazione delle velocità* VACF. Questo secondo approccio si può esprimere matematicamente nel seguente modo:

$$D = \frac{1}{3} \int_0^{+\infty} \left\langle \boldsymbol{v}\left(t_0 \right) \cdot \boldsymbol{v}\left(t_0 + t \right) \right\rangle dt \tag{3.12}$$

Dal punto di vista fisico non si denotano particolari differenze tra i metodi. Si preferisce tuttavia utilizzare il metodo di Einstein nelle simulazioni DPD siccome la MSD è più facile valutare da un punto di vista numerico piuttosto che la VACF. A causa di questo motivo si è deciso quindi di calcolare la diffusività per mezzo della prima relazione.

Parte II

Capitolo 4

Simulazioni DPD: *LAMMPS* e Setup delle Simulazioni

Le simulazioni DPD eseguite nel lavoro di tesi, sono effettuate grazie al codice open source *Large-scale Atomic/Molecular Massively Parallel Simulator,* da qui indicato come LAMMPS.
LAMMPS permette di effettuare simulazioni di *Molecular Dynamics* di sistemi particellari gassosi, liquidi o solidi. Essendo un codice open source, esso è adattabile alle esigenze dello user. Inoltre permette la parallelizzazione delle operazioni di calcolo, cioè la redistribuzione del costo computazionale della simulazione su più *cores*.
Il metodo sviluppato per il calcolo della viscosità è stato testato su parametri di simulazione che vanno a definire un fluido semplice, in questo caso l'acqua. I metodi matematici che sono stati utilizzati in fase di calcolo delle proprietà di trasporto sono numerosi: la viscosità è stata calcolata con il metodo di Green-Kubo [28], quindi viene calcolata la SACF per mezzo di LAMMPS che viene successivamente viene integrata secondo L'Eq. (3.5). La viscosità viene calcolata come singolo valore in uscita da LAMMPS, ma nel lavoro di tesi verrà illustrato come in realtà siamo interessati all'andamento della viscosità più che al valore finale che viene calcolato. Infatti la SACF è una funzione che partirà da un valore diverso da zero per poi tendere, con l'avanzare delle iterazioni, ad un valore nullo affetto da fluttuazioni nell'intorno dello zero. L'andamento della viscosità è fortemente influenzato dai parametri in ingresso della simulazione, in particolare è influenzata dai parametri di *Nev*, *Nfreq* ed *Nrep*, che agiscono sull'andamento della SACF, e dal parametro Δt che va ad incidere sul raggiungimento del valore convergente finale.
La prima parte del lavoro di tesi è consistito nell'andare a ricercare un valore *minimo* di *Nrep*, *Nfreq*, ed *N* (=*Nrep***Nfreq*) per avere una convergenza della funzione di viscosità. Questo si è reso necessario in funzione della tecnica che si è deciso di sviluppare per la stima della viscosità stessa.

4.1 - Analisi dei set di simulazione DPD scelti

Le simulazioni DPD vengono impostate per mezzo di un file di testo in cui vengono raccolti tutti i parametri di ingresso e le istruzioni dei calcoli da eseguire con il codice LAMMPS. L'algoritmo sviluppato basa l'intero procedimento di gestione e sottomissione delle simulazione per mezzo della generazione automatica di questi file. I parametri più importanti da definire quando si considera una simulazione DPD sono contenuti nei file di input, che vanno a definire quelle che sono le caratteristiche del fluido simulato, in questo caso l'acqua.
Partiamo dai parametri che sono stati mantenuti costanti lungo tutto l'arco delle simulazioni: il primo di questi parametri è il grado di coarse-graining *Nm* delle particelle di fluido semplice. Questo parametro non è esplicito nel file di input ma viene definito per mezzo dell'Eq. (2.18): quindi per mezzo del parametro repulsivo *a* noi siamo in grado di andare a manipolare il valore di *Nm.*
Un altro aspetto molto importante, e che viene definito immediatamente nel file di input, è la dimensione ed il tipo di box di simulazione tridimensionale in cui andremo ad effettuare i calcoli. Le dimensioni sono state mantenute costanti e pari a 15 unità DPD (in *x, y, z*), con box *periodica.* Il termine *periodico* indica un'impostazione del sistema in cui le particelle simulate sono in grado di interagire anche oltre i limiti imposti dalla dimensionalità della box di simulazione. Sostanzialmente la particella è in grado di uscire dalla box senza essere costretta, come nel caso delle box non periodiche, a deviare la direzione del moto per rimanere all'interno dell'ambiente limitato di simulazione. Questo tipo di condizioni al contorno della box deve essere anche associata alla condizione che *simmetricamente* per ogni particella uscente deve necessariamente essercene una entrante in modo da mantenere costante la quantità di particelle simulate. Questo procedimento è garantito dal fatto che si considera che adiacenti ad ogni box che noi impostiamo sono presenti altre box omologhe in cui avviene lo stesso processo, quindi per ogni particella DPD uscente caratterizzata da una certa quantità di moto ed una traiettoria, si avrà una particella DPD entrante che presenta le stesse proprietà. L'intero processo è illustrato in Fig. 4.1

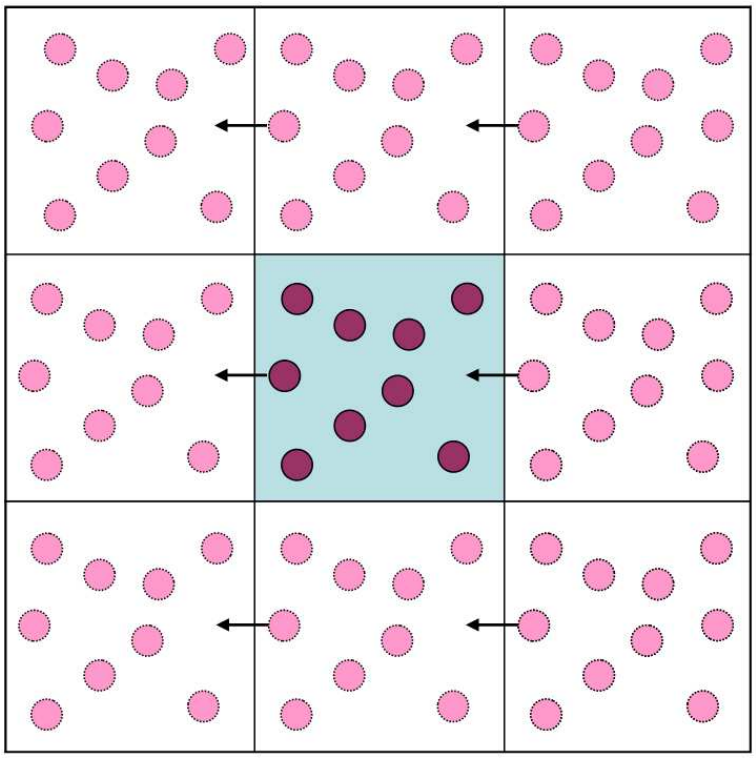


Fig. 4.1 - Rappresentazione periodica della box di simulazione

Altri parametri che vengono definiti nel file di input di LAMMPS sono legati alle equazioni caratteristiche del modello DPD. Troviamo infatti il termine ρ che indica la densità per unità di volume delle *beads,* i termini k_B e T, i termine r_c ed r_{cD} che indicano rispettivamente i raggi d'azione delle interazioni che intercorrono tra beads, il primo termine nel caso delle forze conservative ed il secondo nel caso delle forze dissipative/stocastiche, il termine *s,* cioè l'esponente delle funzioni peso delle forze, il termine *a* che abbiamo definito precedentemente, il timestep *dt*, il coefficiente dissipativo σ, ed il numero di beads.

L'algoritmo è stato provato su set di diversi parametri in modo tale da poter ricavare dei risultati in funzione di diverse condizione di simulazioni DPD. Non tutti i parametri dei set sono stati variati; infatti quelli che definiscono il numero di beads, la dimensionalità del sistema, il numero di *coarse-graining Nm* ed i coefficienti di dissipazione e rumore termico del sistema sono stati mantenuti costanti sia attraverso il processo di iterazione e convergenza del valore di viscosità per un singolo set, ma anche tra i set stessi. In Tab. 4.1 si possono osservare per i vari set quali sono stati i parametri studiati.

ρ	3
k_B	1
T	$1/k_B$
r_c	1
r_{cD}	1, 1.5, 1.75, 1.9, 2
s	1, 0.5, 0.25, 0.125, 0.0625
a	25
dt	0.04
γ	3
$nBeads$	10125

Tab. 4.1 - Set di parametri esplorati per mezzo dell'algoritmo sviluppato nel lavoro di tesi

Come si può vedere, alcuni parametri sono multipli, essi sono i parametri variati tra i diversi set analizzati. Ne consegue che quelli che invece compaiono con un singolo valore sono quelli che non vengono modificati.
Per ogni valore di s sono stati valutati tutti i valori di r_{cD}, di conseguenza l'algoritmo è stato valutato per un totale di 25 simulazioni.
Di seguito è aggiunto il segmento di codice che va ad identificare i parametri appena descritti:

```
# Initialization

units   lj
variable ndim   equal 3



# Box size

variable xsize  equal 15
variable ysize  equal 15
variable zsize  equal 15



# DPD parameters

variable rho    equal 3
variable kb     equal 1
variable T      equal 1/${kb}
variable rc     equal 1
variable rcD    equal 1, 1.5, 1.75, 1.9, 2
```

```
variable s      equal 1, 0.5, 0.25, 0.125, 0.0625
variable a      equal 25
variable dt     equal 0.04
variable sigma  equal 3
variable nBeads equal (${xsize}*${ysize}*${zsize})*${rho}
```

4.2 - Parametri di post-processing per la SACF

In seguito alla definizione dei parametri di input alle simulazione, si devono andare ad identificare quelli che sono i parametri di *post-processing*. Questi parametri sono estremamente importanti per quelli che sono i fini del seguente lavoro, infatti essi influenzano direttamente l'andamento della SACF e, di conseguenza, siccome si è scelto di utilizzare il metodo di Green-Kubo, anche l'andamento della funzione della viscosità.
I parametri che vengono definiti in questa sezione sono tre: *Nev*, *Nfreq* ed *Nrep*.
Nev indica ogni quanti *timesteps* vengono campionati i valori di input a partire da un tempo pari all'ultimo *output time* investigato e fino ad un tempo pari ad *Nfreq* passi di integrazione temporale. Andando ad evidenziare l'espressione matematica della serie correlazioni $C_{\alpha\beta}$, avremo che:

$$C_{\alpha\beta}(t) = \left\langle \sum_{\alpha \neq \beta} P_{\alpha\beta}(t)\, P_{\alpha\beta}(t+\delta) \right\rangle \tag{4.1}$$

Quindi si evidenzia come la correlazione al tempo t sia la media di coppie di valori di input separati dalla distanza temporale pari a δ. La relazione che ne definisce il valore massimo di δ è la seguente:

$$((Nrep - 1)\, Nev) \tag{4.2}$$

vediamo quindi come il numero di correlazioni totale sarà pari a *Nrep:*

$$C_{\alpha\beta}(0),\ C_{\alpha\beta}(Nev),\ C_{\alpha\beta}(2Nev),\ \ldots,\ C_{\alpha\beta}((Nrep-1)\, Nev). \tag{4.3}$$

Come detto prima, le medie di questi valori vengono effettuate ogni *Nfreq timesteps.*
In generale nel lavoro descritto in questa tesi si è sempre deciso di mantenere costante Nev unitario, questo perché a causa dell'elevato coefficiente di diffusione nelle simulazioni DPD si rende necessario campionare spesso i valori di input.
Gli altri due parametri di post processing vengono mantenuti sempre uguali tra loro, ma variano nell'algoritmo automatizzato per il calcolo della viscosità. Il motivo è che questi parametri vengono impostati per un parametro *minimo* pari a 500: questo valore poi verrà progressivamente incrementato in modo tale da valutare la convergenza della SACF e di conseguenza del valore di viscosità.

La sezione del file che prevede la definizione dei parametri di post-processing viene presentato di seguito:

```
# Post-processing correlation function parameters

variable Nev  equal 1       # correlation length
variable Nrep equal 1000    # sample interval
variable Nfreq  equal 1000  # dump interval
```

Un ulteriore parametro estremamente importante per il metodo, e direttamente correlato con *Nfreq,* è *nrun*. Questo parametro di solito viene definito autonomamente rispetto agli altri parametri, ma in questo caso viene definito dalla relazione $nrun = Nfreq \times N$, dove N è il numero di ripetizioni di *Nfreq* nella simulazione.

4.3 - Effetto di *Nfreq* ed *N* sull'andamento della Viscosità

Una volta definiti i parametri chiave per la simulazione DPD, è necessario andare a definire più approfonditamente quali sia il loro effettivo impatto sull'andamento della SACF. In generale sono stati studiati diversi set di simulazione disponibili per andare ad identificare l'effetto di questi parametri. In questa sezione faremo riferimento ad un set i cui parametri sono riassunti in Tab. 4.2. Va comunque detto che lo stesso comportamento è stato verificato anche su numerosi
altri set di parametri e di conseguenza le conclusioni tratte per questo set sono di carattere generale.

ρ	3
k_B	1
T	$1/k_B$
r_c	1
r_{cD}	2
s	0.7
a	265
dt	0.01
γ	3.46
nBeads	10125
Nfreq	8000
Nrep	8000
Nev	1
N	70, 100, 200, 320, 500, 625

Tab. 4.2 - Parametri del set di riferimento per lo studio dell'effetto di N

La viscosità, calcolata con il metodo di Green-Kubo, viene calcolata a partire dalla SACF che viene stampata all'interno del file *time_corr.txt*, questo file conterrà le SACF calcolate per ogni valore di *N*, di conseguenza è facile recuperare e graficare le SACF al variare di *N*.
In Fig. 4.1 sono rappresentati gli andamenti delle viscosità ed i rispettivi valori di *N*. Quello che possiamo notare è che all'aumentare di *N* le curve tendono a sovrapporsi ad un valore stabile della viscosità. Questo risultato è atteso in quanto per come è definito *Nfreq* è intuibile che se imponiamo alla simulazione più iterazioni da compiere, si avrà anche un minore effetto dovuto alle fluttuazioni nella coda della SACF, e di conseguenza anche un valore convergente della viscosità.

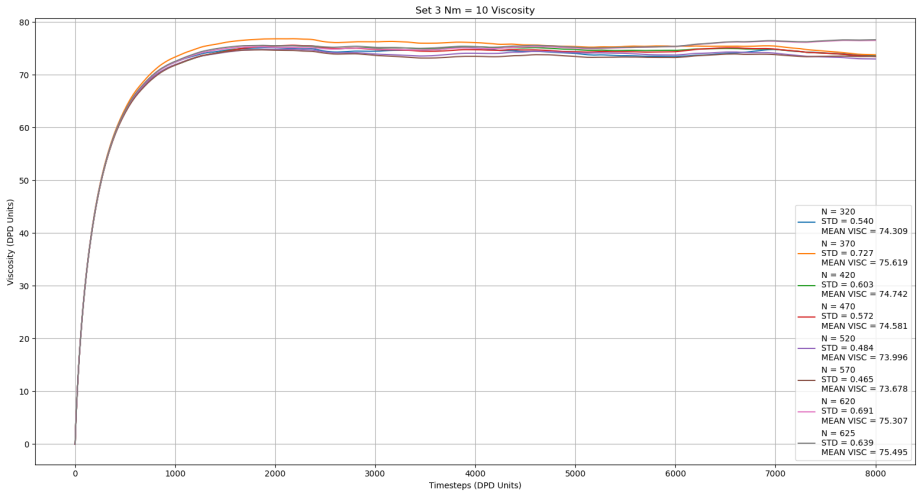


Fig. 4.1 - Variazione del valore finale di viscosità al variare di *N*

Questo risultato è particolarmente importante, in quanto ci permette di affermare che deve esserci un valore minimo di N per cui il valore della viscosità è sufficientemente accurato da poter essere considerato definitivo per il sistema preso in esame. Sulla base di questa osservazione si andrà poi a sviluppare uno dei controlli iterativi del metodo che è stato sviluppato durante la tesi.

4.4 - Effetto di *Nrep* sull'andamento della Viscosità

Ora che abbiamo analizzato l'effetto di *Nfreq* ed N sul calcolo della SACF, si deve necessariamente andare ad analizzare l'effetto dell'altro parametro di post-processing che influisce sulla velocità di calcolo della funzione di autocorrelazione, cioè *Nrep.*

L'influenza di *Nrep* risulta più intuitiva di quella di *Nfreq*. Infatti, per come è stato definito questo parametro in Sez. 4.2, risulta immediato constatare che il suo valore influenzerà la scala temporale della simulazione DPD. Intuitivamente si può affermare che più è alto il valore di *Nrep* e più la scala temporale dell'analisi DPD si allungherà, raggiungendo il valore di $Nrep \cdot dt$, tempo in unità DPD. Questa proprietà assume un senso nel contesto di questa tesi se si va a ripensare all'andamento della funzione di autocorrelazione dello stress SACF. Sappiamo infatti che questa funzione non è immediatamente convergente, ma tenderà a convergere a zero all'avanzare del tempo, formando un *plateau* in cui il valore medio sarà nullo ma affetto da fluttuazioni che ne fanno oscillare il valore istantaneo nell'intorno del valore medio. L'effetto di *Nfreq* in rapporto alle oscillazioni è stato discusso precedentemente in Sez. 4.3.

L'analisi dei risultati dimostra che non esiste un valore minimo di *Nrep* univoco per ogni sistema DPD che si vuole simulare, in compenso sappiamo che, a patto che ci sia un valore sufficientemente alto di *N*, si avrà sicuramente convergenza ad un *plateau* per un certo valore di *Nrep* minimo. Questo risultato è confortante in quanto significa che, anche in questo caso, si potrà sviluppare un algoritmo iterativo che preveda un qualche controllo sul valore di viscosità medio per individuare il valore minimo che garantisce accuratezza nel risultato finale. Questo sistema verrà spiegato nel capitolo seguente quando si andrà ad analizzare l'algoritmo finale.

Capitolo 5

Descrizione dell'algoritmo per il calcolo della viscosità

Il capitolo precedente si è concentrato sulla spiegazione di come si impostano e lanciano le simulazioni DPD per mezzo del codice LAMMPS, mettendo in luce alcuni degli aspetti caratteristici di come vengono gestiti gli output e di quali dei parametri possono essere di interesse concreto per la valutazione finale della viscosità.
Questo capitolo invece viene scritto con l'intento di spiegare approfonditamente come viene gestita la sequenza delle operazioni che compongono il lancio automatico delle simulazioni e, successivamente, i controlli sui file di output ed il calcolo delle proprietà di trasporto del fluido.
Il codice, nel suo complesso, è scritto in due linguaggi di programmazione diversi: *Python* e *Bash.*
Python è un linguaggio di programmazione di "alto livello", cioè un linguaggio che permette un facile ed intuitivo utilizzo da parte di un essere umano che lo utilizza, ed è orientato ad oggetti, cioè basato su un paradigma di programmazione che permette di definire oggetti software in grado di comunicare tra loro per mezzo di uno scambio di messaggi. Queste caratteristiche rendono Python un linguaggio di programmazione estremamente semplice da utilizzare ed interpretare, sia dal punto di vista dello user finale che del programmatore che lo modifica a seconda delle proprie esigenze. In ultimo, è importante notare che sono disponibili numerose librerie open source specifiche per applicazioni

matematico/ingegneristiche, in particolare nell'algoritmo vengono spesso utilizzate le librerie *pandas*, *numpy,* e *subprocess.*
Il secondo linguaggio di programmazione, *Bash*, è una *shell* del progetto GNU implementata nei sistemi operativi Unix-like. Esso è un linguaggio che permette di interpretare una serie di comandi e di comunicare con il sistema operativo, l'impiego di questo particolare linguaggio si è reso necessario dal momento che l'algoritmo è stato sviluppato in ambiente Linux, in particolare nella sua versione *Ubuntu 20.04 LTS,* ed è stato adattato per funzionare su diverse macchine il cui sistema operativo si basa su Linux. In particolare l'algoritmo è stato testato ed utilizzato sul cluster di ateneo Hactar fornito dal servizio HPC@POLITO.

5.1 - Flow chart del metodo sviluppato

Possiamo ora partire a definire quello che è il flow chart delle operazioni eseguite dal metodo sviluppato con la finalità di calcolare la viscosità del sistema DPD simulato. Come già detto in Sez. 3.2, la viscosità del sistema è calcolata con il metodo di Green-Kubo a partire dalla SACF calcolata per mezzo di LAMMPS e salvata nel file *time_corr.txt.* Abbiamo visto che la viscosità di un sistema non solo dipende dai parametri DPD, ma anche dai parametri di post-processing che vengono definiti nella simulazione. Inoltre abbiamo verificato quale sia l'influenza di questi parametri sull'andamento della SACF e, di conseguenza, sulla viscosità. A partire da queste osservazioni si è sviluppato quindi un algoritmo che prevede una serie di operazioni e controlli iterativi con la finalità di verificare la presenza del *plateau*, e quindi garantire che il valore calcolato della viscosità sia stato raggiunto.
In Fig. 5.1 viene raffigurato il flow chart del metodo successivamente implementato nel codice.
In input allo script daremo i parametri della simulazione e verrà generato un file LAMMPS di input con le caratteristiche volute. Per i parametri di post-processing il discorso è più ampio, in quanto non è banale scegliere dei parametri iniziali. Inizialmente si era proposto di procedere con i valori, di input e di incremento per iterazione, riassunti in Tab. 5.1.

Parametro	***Input***	***Incr./Decr.***
Nrep	100	100
Nfreq	100	100
N	70	70
Δt	0.05	33%

Tab. 5.1 - Primi Parametri ipotizzati come input delle simulazioni

Questi valori erano stati pensati con l'obiettivo di dare un possibile primo valore minimo di *Nrep* ed *N* in cui si verifica la presenza di *plateau* nella SACF/viscosità. Pur non essendo teoricamente errato, i valori iniziali sono stati valutati considerando solo i set per cui il *plateau* veniva raggiunto più velocemente, di conseguenza nella maggior parte delle simulazioni si è visto che servivano molte più simulazioni di quante fossero quelle effettivamente attese. Infatti, avendo questo *gap* iniziale si dovevano eseguire molte iterazioni in più per raggiungere una convergenza del valore. A seguito di questa verifica, si è deciso quindi di non utilizzare i valori precedentemente illustrati, ma di aggiornarli, mantenendo un margine cautelativo per cui siamo sicuri che probabilmente sarà presente un plateau per la maggior parte dei sistemi simulati. I nuovi valori sono riassunti in Tab.5.2 e, come si può vedere, sono nettamente più alti rispetto ai precedenti nel caso di *Nrep, Nfreq* ed *N,* mentre il valore di Δt è stato abbassato. Tutte le modifiche sono state fatte in modo da favorire la convergenza che si verifica con meno iterazioni rispetto ai valori iniziali, con conseguente risparmio computazionale.

Parametro	***Input***	***Incr./Decr.***
Nrep	500	500
Nfreq	500	500
N	500	100
Δt	0.04	33%

Tab. 5.2 - Nuovi valori proposti per inizio iterazioni

Una volta impostati i parametri iniziali ed i valori degli incrementi per iterazione, passiamo ad analizzare gli step fondamentali dello script.
Come si può vedere in Fig. 5.1, l'intero processo si basa su tre controlli iterativi principali, check di *Nrep*, check di *N*, e check di *dt*.

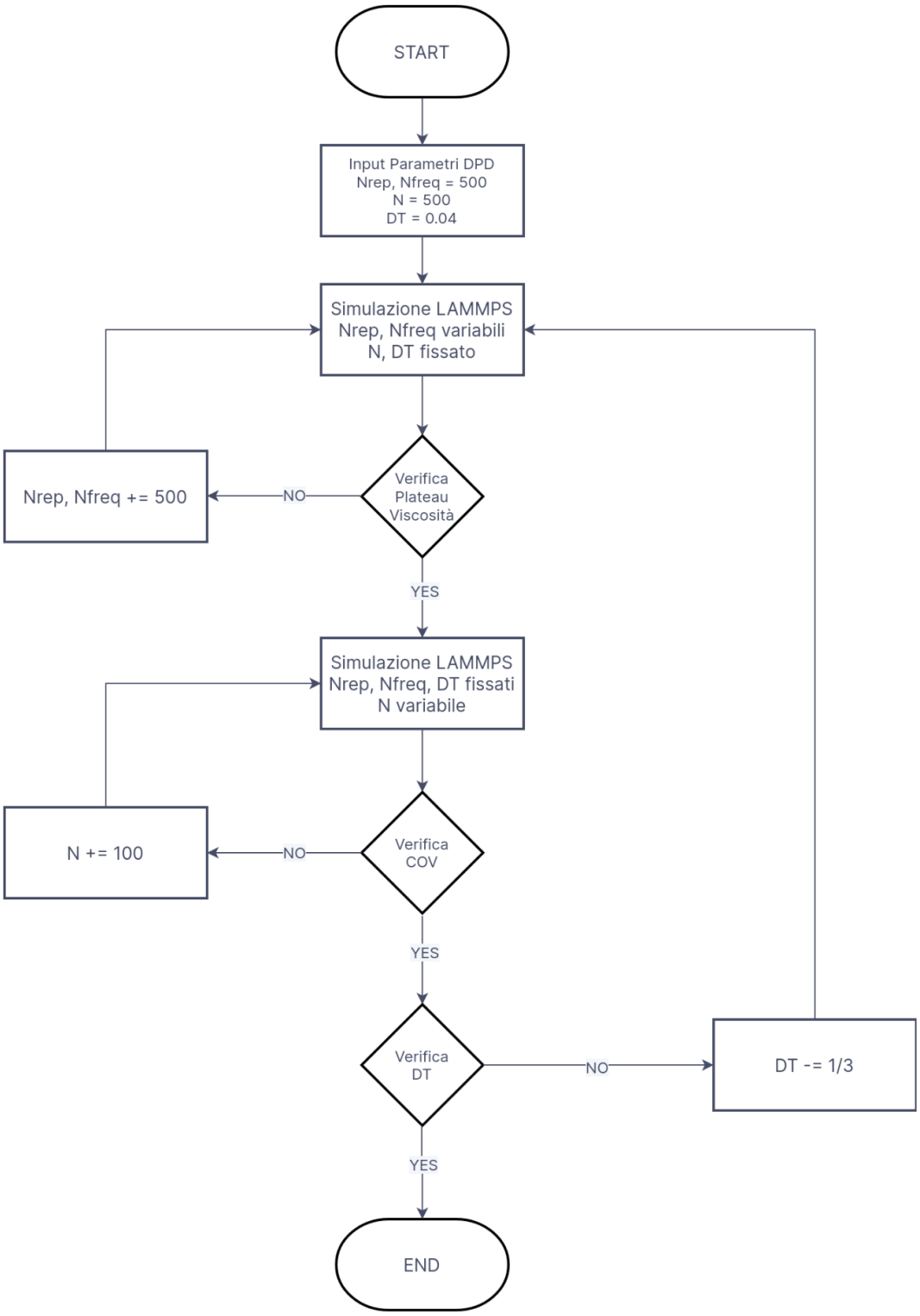


Fig. 5.1 - Flowchart completo del metodo

5.1.1 - Check di *Nrep*

Questo è il primo check ed è eseguito per la singola simulazione eseguita, cioè non serve *confrontare* tra loro due diverse simulazioni ma è sufficiente eseguire un controllo sull'andamento della funzione della viscosità per poter verificare la convergenza del valore di viscosità.

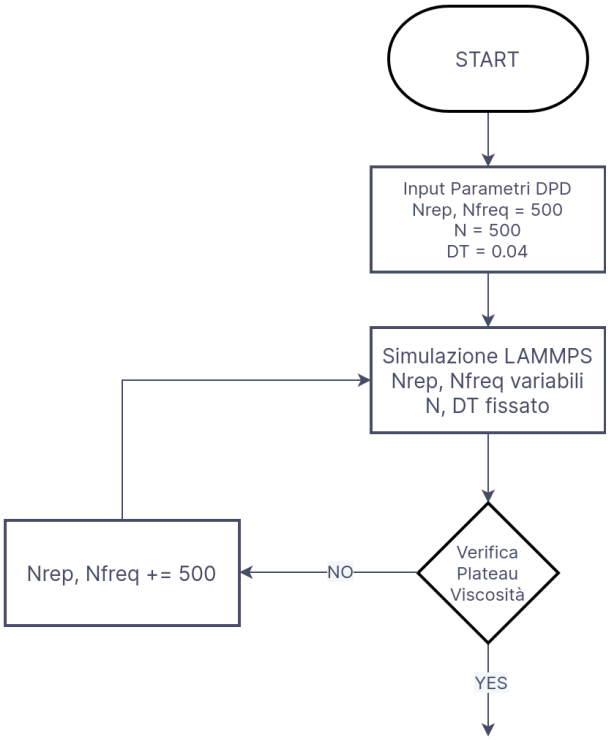


Fig. 5.2 - Flowchart Check di *Nrep*

5.1.2 - Check di *N*

Al contrario del check precedente, in questo caso è necessario confrontare due simulazioni consecutive per valutare che la riduzione della fluttuazione in coda alla viscosità sia sufficientemente bassa, il parametro di riferimento è il *Coefficient of Variation*, esso è definito dalla formula $COV = \sigma/\mu$, dove σ e μ sono rispettivamente la deviazione standard e la media della viscosità calcolata.
La prima simulazione che viene eseguita all'interno di questo check avrà come riferimento l'ultima uscita dal check prima. Nel caso la verifica non sia soddisfatta allora si inizierà una serie di simulazioni che verranno confrontate in serie. Vale sempre la regola che il $COV(n)$ della simulazione *n*-esima verrà confrontato con il $COV(n-1)$ della simulazione *n*-1.

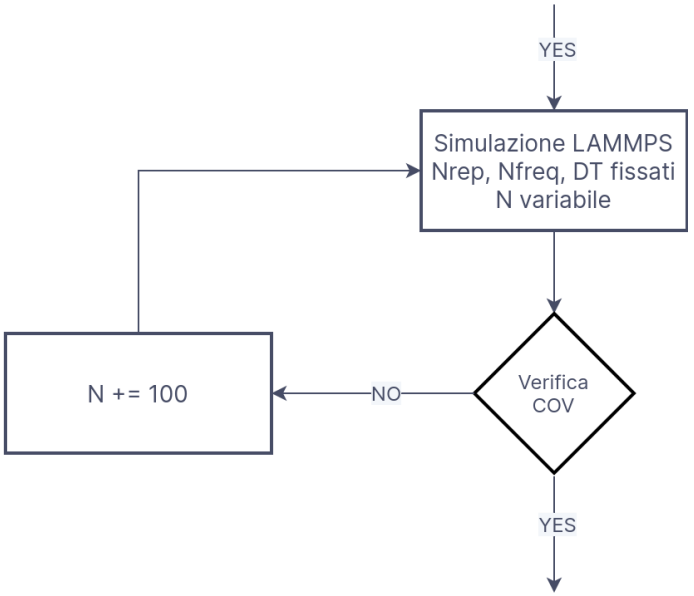


Fig. 5.3 - Flowchart Check di *N*

5.1.3 - Check di Δt

Esattamente come per *N*, in questo caso si deve valutare la convergenza facendo un confronto tra due simulazioni. La differenza sostanziale però è che in questo caso non viene fatto il confronto con la simulazione precedente, ma viene fatto con il valore finale della viscosità calcolata per il Δt precedente. Fino a questo ultimo check noi abbiamo calcolato il valore di convergenza della viscosità *per un dato* Δt, ma non in generale per il sistema. Questo significa che il controllo va fatto tra i valori di viscosità finali di due Δt consecutivi. Ovviamente alla prima iterazione non noi avremo un valore di viscosità finale *precedente* con cui confrontare la nostra prima iterazione, quindi verrà imposto un valore fittizio estremamente alto per fare in modo che il primo controllo fallisca sempre, indipendentemente dal valore di viscosità calcolato per $\Delta t = 0.04$, a questo punto il valore di timestep verrà decrementato del valore indicato in Tab. 5.2 e verrà reinizializzato tutto il processo di simulazione e controllo ripartendo dai valori di input di *Nrep, Nfreq* ed *N*, ma con il valore di Δt decrementato. In questo modo sarà possibile raggiungere nuovamente una convergenza del valore di viscosità per il nuovo Δt e sarà possibile confrontare a questo punto effettivamente i valori finali, per valutare se si è verificata una convergenza effettiva. Nel caso non sia stato possibile rispettare una tolleranza, impostata dallo user, tra i due valori finali calcolati, allora verrà eseguita una nuova iterazione come appena descritto. Se invece si è verificata la condizione di uscita dal check, allora si può considerare il risultato finale come accurato e definitivo.

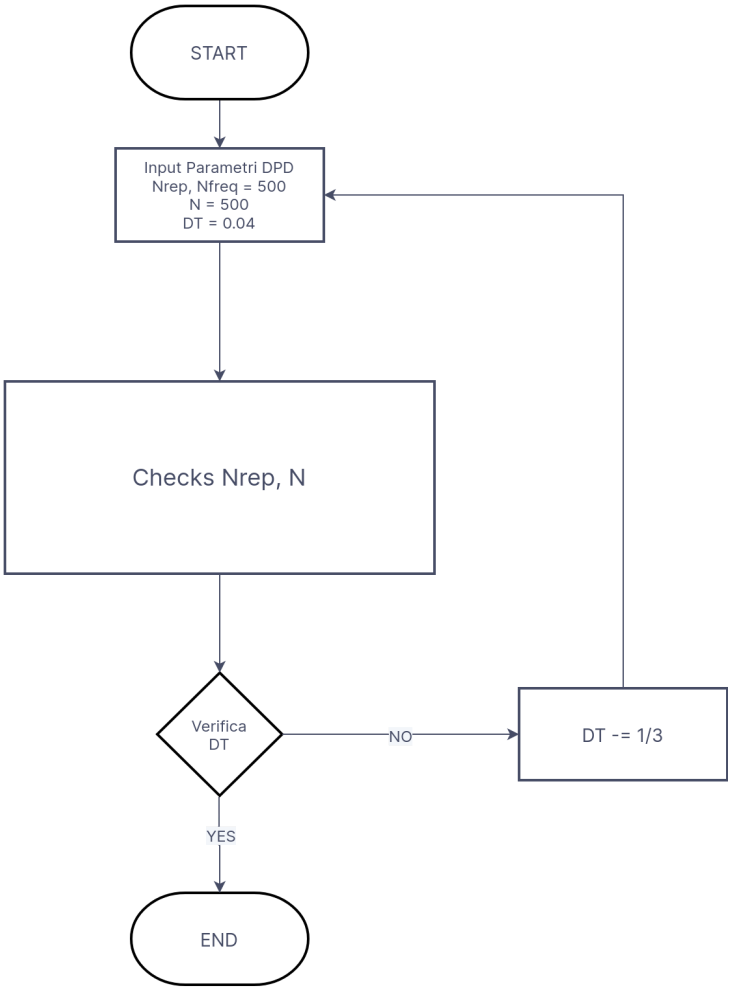


Fig.5.4 - Flowchart Check di dt

5.2 - Script del metodo sviluppato

Nel capitolo precedente si è spiegato concettualmente la struttura ed il funzionamento del metodo, è stato anche dimostrato e spiegato quali considerazioni ci hanno portato ad operare certe scelte e, per quale motivo si sono rese necessarie o convenienti determinate operazioni.
Il seguente capitolo andrà a spiegare passo passo come viene tradotto il metodo precedentemente sviluppato in *codice,* saranno presenti molti snippets del codice, ed ogni operazione eseguita, sia in *Python* che in LAMMPS, verrà dettagliatamente descritta.
Innanzitutto, conviene descrivere il tipo ed il numero dei file che sono stati sviluppati per il metodo. Il codice non si presenta sotto forma di un unico file in cui vengono eseguite le operazioni, ma, per una questione di *coding hygiene* e facilità di utilizzo, è stato scelto di suddividerlo in diversi file separati che verranno, quando necessario, richiamati ed utilizzati:

- *main.py* - il main file del nostro metodo. Questo è il file che verrà eseguito per iniziare un nuovo ciclo di simulazioni. Non contiene esplicitamente tutte le operazioni eseguite dal codice, ma viene utilizzato come supporto per coordinare la successione dei *task* eseguiti. Questa serie di *task* comprende quando inizializzare e lanciare le simulazioni, lettura degli output di simulazioni, e quali controlli eseguire in un determinato momento. Esso presenta quindi la stessa struttura del flow chart in Fig. 5.1 e descritto in Sez. 5.1 ma senza esplicitare le singole operazioni al suo interno.
- *ImportData.py* - Questo file contiene una funzione che permette di estrarre dal file di output da LAMMPS *time_corr.txt,* utilizza un sistema che è in grado di estrarre e convertire in un unico passaggio i valori desiderati dal file e convertirli direttamente in una *list* di valori facilmente manipolabili in *Python.*
- *MovingSpansAverage.py* - Il file che si occupa di verificare la presenza del plateau nell'andamento dei valori della viscosità. E' estremamente importante che sia efficiente ed adattabile al sistema che viene simulato.
- *launcher_cluster & launcher_local* - File che contiene tutte le informazioni per generare i file di input di LAMMPS, lanciare le simulazioni, e inizializzare i controlli sugli output. Ci sono due versioni di questo file in quanto il comando di lancio in locale e la stringa di lancio sul Cluster non coincidono, è stato quindi necessario prevedere due file in grado di eseguire il codice in entrambi gli ambienti di calcolo.
- *CumulativeIntegral.py* - File in cui sono presenti tutte le operazioni necessarie a calcolare il valore dell'integrale cumulativo del metodo di Green-Kubo per il valore della viscosità.

- *gamma_eff.py* - File contenente le funzioni necessarie per importare i valori della RDF calcolata dai file di output di LAMMPS e calcolare il valore di γ_{eff} .
- *v_fitslope_extr.py* - Il file contiene una singola funzione che estrae del file di output *log.lammps* i valori necessari al calcolo della diffusività con il metodo di Einstein.

Data questa prima descrizione esemplificativa, possiamo passare all'analisi più approfondita dei singoli file.

5.3 - *main.py*

Come già accennato, il file *main.py* è il file principale che coordina la successione delle operazioni che devono essere eseguite.
Il file a sua volta è divisa in più sottosezioni, la prima di cui consiste nella definizione dei parametri di ingresso:

```
from launcher_cluster import *
import subprocess
```

```
# DATAS

NFREQ = 500
NREP = 500
N = 500
DT = 0.04

# Input file

RHO = 3
rc = 1
rcD = 1
s = 1
a = 25
sigma = 3
neql = 2.0e5
nrun = N*NFREQ
nrunmsd = 5.0e4
```

Come si può osservare, vengono definiti i parametri di simulazione DPD e di post-processing necessari al lancio della simulazione. I parametri in sono ρ, la densità delle *beads* nel sistema DPD simulato, r_c ed r_{cD}, che sono i raggi d'azione delle interazioni conservative e dissipative/stocastiche agenti tra le *beads*, *s*, l'esponente della funzione peso associata alle interazioni e definita da Eq.(), *a,* il parametro repulsivo della forza conservativa correlato al parametro di *coarse-graining Nm*, il coefficiente di ampiezza delle forze randomiche σ, i

parametri di post-processing iniziali *Nfreq, Nrep* ed *N*, il timestep Δt iniziale, in ultimo vengono definiti i parametri di equilibratura del sistema *neql*, il parametro *nrun*, ed il parametro *nrunmsd.*

Una volta completata questa prima parte di definizione dei parametri di simulazione, viene impostata la prima fase di lancio delle simulazioni e controllo dei risultati, in particolare viene eseguito il primo dei *checks* descritti nella sezione precedente, cioè quello di *Nrep* e ricerca del *plateau:*

```
visc_list_2 = [1.0e3]

while True:

        visc_list_1 = []
        i = 0

        while True:
           simulation = Launcher_local(...)
           simulation.generateFiles("in.Sh")
           current_file, ran_code = simulation.currentFile()
           simulation.launchSims()
           WINDOW = 30
           ERR_REL = 1.0e-2
           JUMP_INDEX = 8

           try:
              visc_plateau, Nrep, flag_plateau = simulation.checkNrep(...)
              visc_list_1.append(visc_plateau)
           except TypeError:
              pass

           with open("results_{ran_code}.txt".format(ran_code = ran_code), "w") as res:

              res.write(
                "# Results\n\n\n"
                "filename: {file}\n\n"
                "NFREQ = {NFREQ}\n"
                "NREP = {NREP}\n"
                "N = {N}\n"
                "DT = {DT}\n\n"
                "Viscosity: {visc}\n".format(...)
                      )

           subprocess.run("...".format(...),shell=True)

           if i == 5:
              N += 250
           else:
              pass

           if flag_plateau == 0:
              NREP += 500
              NFREQ += 500
              print("WARNING: No Convergence @NREP,NFREQ,N\nRestarting the Process\n")
           else:
              break

           i += 1

        visc_list_2.append(visc_list_1[-1])
```

Il codice presentato riproduce il check di *Nrep* descritto in Sez.(). Il check comincia con la definizione di una lista chiamata *visc_list_2*, questa lista conterrà solo i valori definitivi di viscosità calcolata in questo step di controlli, mentre la seconda lista chiamata *visc_list_1* conterrà solo i valori che vengono calcolati all'interno del ciclo di iterazioni. Questa scelta è stata fatta per poter separare i valori definitivi senza il rischio di andare a valutare la convergenza su dei valori che in realtà sono stati valutati come non corretti all'interno del ciclo.
Siccome non sappiamo quanti cicli sono necessari alla convergenza del valore di viscosità, l'intero algoritmo è stato basato su due cicli *while* che non termineranno il processo fino al raggiungimento di una tolleranza accettabile impostata dallo user. Il primo ciclo *while* che incontriamo introduce al ciclo di controllo su Δt e racchiude in sé tutti i processi del metodo, come illustrato in Fig. 5.1, questo è detto il *master loop* del metodo.

5.3.1 - Ciclo *while* per *Nrep*

Il *while* interno comprende il check di *Nrep*. Inizialmente discuteremo le fasi di generazione del file di simulazione, denominato *in.Sh*, il lancio della simulazione corrispondente per mezzo della funzione `simulation.launchSims()`, la definizione dei parametri di check *WINDOW, ERR_REL, JUMP_INDEX* ed il controllo sugli output per mezzo della stringa di codice `simulation.checkNrep(...)`.
Di questa parte del codice ci sono diversi aspetti da approfondire, il metodo è stato sviluppato avendo in mente di poter accedere ai risultati *intermedi* della simulazione, in modo tale da permettere una supervisione completa di ogni passaggio e calcolo eseguito dal metodo senza la necessità di bloccare il processo completamente. In quest'ottica è stato sviluppato un sistema che permette di identificare univocamente ogni simulazione e file di input/output. Questo sistema si basa sul fatto che il file non viene univocamente generato in un solo step, ma viene costruito e modificato più volte fino al raggiungimento della sua versione finale pronta per essere effettivamente utilizzata. Le fasi di costruzione e generazione del file comprendono:

- *Importing* e salvataggio dei parametri con `Launcher_local(...)` - in questa fase vengono salvati i parametri specificati nella fase iniziale all'interno dell'oggetto di tipo *class* `Launcher_local(...)`. Non viene generato nessun file.
- Applicazione dei parametri ad un file template e generazione di un file randomico alfanumerico per mezzo del metodo di classe `simulation.generateFiles("in.Sh")` - il metodo di classe appena descritto compie due operazioni che verranno descritte in dettaglio successivamente in Sez. 5.4.2 ma che sinteticamente si possono riassumere nel seguente modo, grazie alla libreria *random* di *Python* viene generato un codice randomico alfanumerico che verrà applicato al nome del file specificato come input al metodo, in questo caso `"in.Sh"`.

di interesse facilmente anche se questa simulazione non ha raggiunto la convergenza e quindi non comparirà tra le ultime eseguite dal metodo. A questo punto verranno applicati i salvati in `Launcher_local(...)` ad un template di un file di input di LAMMPS definito in `simulation.generateFiles("in.Sh")`. Alla fine di questo secondo step non è ancora stato generato nessun file

- Generazione del file di input finale e salvataggio del codice randomico in variabile esterna per mezzo di `simulation.currentFile()` - questo ultimo step consiste nella generazione del file di input finale. Questo file avrà tutte le caratteristiche necessarie per comunicare al codice LAMMPS le istruzioni necessarie per eseguire la simulazione e sarà identificato dal codice randomico, che verrà salvato separatamente nella variabile `ran_code` per poter essere invocato nelle fasi successive dell'algoritmo. E' importante notare che non solo il file di input verrà associato al codice randomico, ma anche i file di output dalle simulazioni verranno etichettate in modo tale da ridurre al minimo la possibilità di confusione.

Si arriva a questo punto al lancio della simulazione. Questa azione verrà eseguita dalla stringa `simulation.launchSims()`, che risulta sempre essere un metodo della classe `Launcher_local(...)`. In seguito al lancio ed al completamento della simulazione, saranno stati generati i file di output di LAMMPS. A questo punto verrà iniziata la prima fase di controllo dei risultati per mezzo della sezione della seguente sezione del codice:

```
try:
   visc_plateau, Nrep, flag_plateau = simulation.checkNrep(...)
   visc_list_1.append(visc_plateau)
except TypeError:
   pass
```

L'insieme delle operazioni è inserito all'interno dei comandi *try/except.* Questo passaggio è stato implementato con l'unico scopo di evitare di interrompere il flusso dell'algoritmo nel caso di errori e permettergli di arrivare alla fase successiva ugualmente. Questo potrebbe sembrare un passaggio ridondante ma in realtà permette di identificare meglio l'errore in quanto sarà più facile identificare eventuali errori. Il check di Nrep è una serie di operazioni iniziate dalla stringa dal metodo di classe `simulation.checkNrep(...)`, in input verranno dati i valori specificati poco prima, cioè *WINDOW, ERR_REL, JUMP_INDEX* di cui anticipiamo solo la funzione di parametri che gestiscono l'accuratezza del controllo sulla funzione della viscosità. In output dal metodo invece troviamo le tre variabili `visc_plateau, Nrep, flag_plateau` che indicano rispettivamente il valore della viscosità a *plateau* raggiunto, nel caso sia stato effettivamente raggiunto, o l'ultimo valore di viscosità calcolato, nel caso contrario, il valore di *Nrep* necessario per il raggiungimento del *plateau*, quindi non quello impostato nella simulazione, ma quello effettivo che può essere diverso, e una variabile di *flag*.

La variabile di flag è particolarmente importante perché deve "avvisare" il resto del codice che effettivamente si sia raggiunto il plateau e che il valore salvato in `visc_plateau` non sia, come detto precedentemente, l'ultimo valore della viscosità senza l'effettiva convergenza. I suoi valori possono essere solo 0 o 1, valori che indicano rispettivamente il mancato raggiungimento del valore del plateau nel primo caso, ed il suo raggiungimento nel secondo caso: il proseguimento delle iterazioni dipendono quindi dal valore della variabile *flag*.
Una volta concluso il processo, e calcolato il valore del della viscosità al *plateau*, il codice prosegue con due operazioni molto importanti gestite dalle seguenti stringhe di codice:

```
with open("results_{ran_code}.txt".format(ran_code = ran_code), "w") as res:

   res.write(
     "# Results\n\n\n"
     "filename: {file}\n\n"
     "NFREQ = {NFREQ}\n"
     "NREP = {NREP}\n"
     "N = {N}\n"
     "DT = {DT}\n\n"
     "Viscosity: {visc}\n".format(...)
           )

subprocess.run("...".format(...),shell=True)
```

Qui non avvengono processi di calcolo, ma è una parte fondamentale per l'organizzazione chiara dei dati in ottica di studio ed analisi del metodo, viene infatti generato un file, nominato `"results_{ran_code}.txt"`, che riassume i valori finali dell'ultima simulazione eseguita nel ciclo iterativo precedente, indipendentemente dall'esito della stessa. Il file, come si legge dal nome, sarà anch'esso identificato dallo stesso codice randomico in modo da essere facilmente riconoscibile. Una volta chiuso il file dei risultati è stata prevista anche una stringa supplementare, `subprocess.run("...".format(...),shell=True)`, che per mezzo dei comandi *Bash* di *Linux* genera una cartella nominata come il codice randomico della simulazione ed in cui va a spostare tutti i file creati per la simulazione DPD appena eseguita, ovvero i file di input/output ma anche il file dei risultati creata immediatamente prima. Questo semplice passaggio permette una pulizia completa della cartella in cui viene eseguito lo script senza la perdita dei dati che, se fossero lasciati nello spazio condiviso anche dai file *Python* sarebbero molto difficili da reperire ed analizzare.
In ultimo, prima della fine del ciclo, vengono eseguite ancora tre operazioni per mezzo delle seguenti stringhe di codice:

```
            if i == 5:
               N += 250
            else:
               pass

            if flag_plateau == 0:
               NREP += 500
               NFREQ += 500
            else:
               break

            i += 1
         visc_list_2.append(visc_list_1[-1])
```

La prima è un'ulteriore misura di sicurezza del codice implementata a seguito di osservazioni fatte su simulazioni DPD eseguite in fase di sviluppo del metodo. Si è notato infatti che, indipendentemente dal valore di *Nrep*, se il valore di N è troppo basso allora non si raggiungerà mai un plateau in quanto il rumore è troppo intenso. Queste osservazioni sono riassunte in Sez., di conseguenza viene introdotta una variabile di conteggio *i* che tiene il conteggio di quante simulazioni DPD vengono fatte, con relativo incremento di *Nrep*, all'interno del ciclo di controllo, se il valore di *i* raggiunge 5 allora è ragionevole pensare che sia necessario aumentare sensibilmente il valore di N per raggiungere una convergenza, questa misura viene implementata per evitare che il check continui a lanciare in serie infinite simulazioni che non raggiungeranno mai un *plateau.*
Successivamente invece viene eseguito il controllo della variabile *flag* definita precedentemente: se la variabile presenta valore pari a 1 allora viene interrotto il ciclo *while* con il comando *break*, se invece il valore di flag è 0 allora si incrementano i valori di *Nfreq* ed *Nrep* di 500 e viene ricominciato tutto il ciclo analizzato fino ad ora.

5.3.2 - Ciclo *while* per *N*

All'interno del *master loop*, ed in seguito al primo loop che identifica la serie di operazioni per il check di *Nrep*, è presente anche un secondo loop che identifica il secondo check del nostro metodo, cioè quello di *N*.
La parte del codice a cui ci si riferisce è:

```
         cov_list = []
         N_list = []
         previous_file = current_file

         while True:
            simulation = Launcher_local(...)
            simulation.generateFiles("N_in.Sh")
            current_file,ran_code = simulation.currentFile()
            simulation.launchSims()

            cov_tmp = simulation.checkN()
            cov_list.append(cov_tmp)

         try:
            visc_plateau, Nrep, flag_plateau = simulation.checkNrep(...)
```

```
            visc_list_1.append(visc_plateau)
        except TypeError:
            pass

        with open("results_{ran_code}.txt".format(ran_code = ran_code), "w") as res:

            res.write(
                 "# Results\n\n\n"
                 "filename: {file}\n\n"
                 "NFREQ = {NFREQ}\n"
                 "NREP = {NREP}\n"
                 "N = {N}\n"
                 "DT = {DT}\n\n"
                 "Viscosity: {visc}\n".format(...)
                 )

        subprocess.run("...".format(...),shell=True)

        try:
            if abs(cov_list[-1]-cov_list[-2]) < 1.0e-2:
                break
        except:
            pass

        N += 100
```

Alcune operazioni sono omologhe alle precedenti analizzate in Sez.(). Ci si riferisce in particolare alle operazioni di generazione dei file e lancio delle simulazioni DPD, ed all'operazione di calcolo della viscosità, quest'ultima non necessaria per il superamento del controllo, ma inserita solo a scopo di valutare l'andamento dei calcoli e monitorare costantemente i valori di interesse. Successivamente a queste prime istruzioni arriviamo quindi al calcolo del valore di COV delle simulazioni ed al confronto tra essi, così come descritto in Sez.().
Il calcolo di COV è effettuato dal metodo di classe `simulation.checkN()`, il valore di output è quindi salvato nella variabile `cov_tmp` e successivamente aggiunto alla lista `cov_list = []` per mezzo della stringa di codice `cov_list.append(cov_tmp)`.
Il controllo in questo caso viene effettuato in modo diverso dal caso precedente. Infatti se prima era necessaria una singola simulazione per valutare la presenza del plateau, in questo caso servono due valori di COV di due simulazioni successive per poter verificare che sia raggiunto il valore minimo di fluttuazioni richiesto sul valore di viscosità. L'intero processo di confronto è descritto dalle seguenti righe di codice:

```
        try:
            if abs(cov_list[-1]-cov_list[-2]) < 1.0e-2:
                break
        except:
            pass

        N += 100
```

Si può notare come in questo caso il valore di tolleranza settato sia del 10%, cioè in questo caso il valore di due COV calcolati e salvati all'interno di `cov_list` devono essere affetti da una differenza massima del 10%. Nel caso questa condizione non sia soddisfatta allora viene incrementato di 100 il valore di N e viene reinizializzato il ciclo del check di *N*.
In ultimo, è utile notare che anche in questo caso viene eseguita sia la registrazione dei dati nel file `"results_{ran_code}.txt"`, sia l'operazione di pulizia e creazione dei file all'interno delle cartelle etichettate dal codice randomico.

5.3.3 - Calcolo del numero di Schmidt e di γ_{eff}

Arrivati a questo punto del metodo, l'ultima simulazione effettuata garantisce, per un dato *timestep*, il valore di viscosità accurato e calcolato con i valori minimi di *Nrep* ed *N*. Le operazioni restanti quindi consistono nel calcolo delle proprietà di trasporto del fluido utilizzando gli output delle simulazioni precedenti, le proprietà che vengono calcolate in questa sezione del codice sono il numero di *Schmidt*, con conseguente calcolo del valore di diffusività, ed il valore di γ_{eff}.

```
## Schmidt Number

      # v_fitslope
      from v_fitslope_extr import v_fits
      diff = v_fits("{ran_code}/log.lammps".format(...))

      schmidt_number = visc_list_2[-1]/diff/RHO

      # Effective Friction Factor

      from gamma_eff import *
      g_eff = gamma_eff("{ran_code}/tmp.rdf".format(...), 50, sigma, rcD, s)
```

Il calcolo del numero di Schmidt prevede il calcolo della diffusività del sistema simulato, la quale viene calcolata per mezzo della relazione di Einstein. La funzione che viene invocata e che esegue il calcolo è `v_fits(...)` che fa parte del file importato `v_fitslope_extr`. Questa funzione ha come input il file *log.lammps* in cui è definito il *mean square displacement* MSD necessario ai fini del calcolo. Una volta ottenuto il valore di diffusività e salvato nella variabile `diff`, si può procedere al calcolo del numero di Schmidt per mezzo della stringa di codice `schmidt_number = visc_list_2[-1]/diff/RHO`, dove `visc_list_2[-1]` è l'ultimo valore di viscosità in output dalle simulazioni.
Successivamente si procede con il calcolo di γ_{eff}, il calcolo viene eseguito dalla stringa di codice `gamma_eff(...)`. a cui viene data come input, a parte i parametri DPD necessari al calcolo, il file *tmp.rdf* di output dalle simulazioni.
Prima dell'ultimo check sul timestep, viene generato un ultimo file di testo che riassume i risultati trovati dall'ultima simulazione. Va ricordato che per come è stato descritto e sviluppato il metodo, il file generato in questa sezione finale di

main.py non contiene i valori intermedi delle simulazioni, ma contiene il valore di viscosità finale calcolato per quel determinato *timestep,* più tutti i valori delle proprietà di trasporto. In codice:

```
with open("results_{ran_code}.txt".format(ran_code = ran_code), "w") as res:

   res.write(
   "# Results\n\n\n"
   "filename: {file}\n\n"
   "NFREQ = {NFREQ}\n"
   "NREP = {NREP}\n"
   "N = {N}\n"
   "DT = {DT}\n\n"
   "Viscosity: {visc}\n"
   "Diffusivity: {diff}\n"
   "Schmidt Number: {sch}\n\n"
   "Effective Friction Factor: {gamma_eff}\n".format(...)
   )
```

5.3.4 - Check di Δt

L'ultimo check a cui viene sottoposto il valore di viscosità è legato al *timestep*. Come è stato infatti descritto in Sez. 5.1.3, il valore di timestep influenza il valore di convergenza della viscosità in quanto compare come termine in Eq.(). L'obiettivo quindi del controllo è di confrontare due valori di viscosità al *timestep* finale e valutare se la loro differenza rientra entro una certa *tolleranza* impostata dallo user. La tolleranza decisa nel lavoro di tesi è del 5% sull'errore relativo tra i valori consecutivi di viscosità.

```
if abs(visc_list_2[-1] - visc_list_2[-2])/visc_list_2[-1] < 5.0e-2:
   break
else:
   if DT == 0:
      break
   DT -= 1/3*DT
```

In queste righe di codice viene eseguito il procedimento sopra descritto, se la condizione di tolleranza non dovesse essere rispettata, allora il valore di timestep verrebbe decrementata del 33% e verrebbe reinizializzato l'intero *master loop* con il valore di *timestep* aggiornato.

5.4 - *launcher_cluster.py & launcher_local.py*

Il *main.py*, che abbiamo descritto nella sezione precedente, gestisce il flusso di operazioni e controlli che si susseguono nel metodo sviluppato nel lavoro di tesi. Ora vedremo come i singoli processi vengono implementati nel codice. I primi files che andiamo a vedere sono *launcher_local.py* e *launcher_cluster.py*: questi due file non si differenziano per il tipo di operazioni che eseguono, ma semplicemente nel modo in cui vengono lanciate le simulazioni. Come viene spiegato infatti, il codice è stato sviluppato per funzionare sia in locale, cioè sulla macchina privata dello user, che sul cluster di ateneo Hactar, e quindi deve essere possibile anche lanciare le simulazioni con dei comandi che siano interpretabile dal sistema Slurm installato, appunto, sul Cluster.
Detto ciò, partiamo quindi con la spiegazione dettagliata delle varie sezioni del codice e delle operazioni che esegue.

5.4.1 - La classe *Launcher_local*

Come già anticipato precedentemente, la generazione del file di input non avviene in una sola fase, ma in tre. Potremmo riassumere l'intero processo quindi nelle seguenti fasi:

- Importazione dei parametri DPD
- Applicazione dei parametri DPD importati al *template* e fase di etichettatura
- Generazione del file di input finale

In questa prima fase di definizione andremo quindi ad occuparci della prima fase di questo processo. Il processo è in realtà estremamente semplice, `Launcher_local` è un oggetto del tipo `class` che attraverso il metodo costruttore `__init__` riesce ad immagazzinare al suo interno i parametri di input. Questo processo comporta una serie di vantaggi, infatti per ogni metodo della classe saranno sempre disponibili tutti i parametri di input alla classe senza il bisogno di definirli nuovamente per ogni operazione. Questo permette non solo l'alleggerimento del codice in quanto si avranno meno variabili hard-coded da andare a modificare, ma abbassa sensibilmente la probabilità di errore dovuto all'inserimento ripetuto di parametri DPD in più punti del codice.
Come si può vedere dallo snippet del codice illustrato di seguito, compaiono tutti i parametri DPD che sono utili a definire il tipo di fluido che si vuole simulare ed anche i parametri di *post-processing* della simulazione.

```
class Launcher_local:

        def __init__(self, rc, rcD, s, a, dt, sigma, neql, nrun, nrunmsd, Nfreq, Nrep):
           self.ndim = 3
           self.xsize, self.ysize, self.zsize = 15,15,15
           self.rho = 3
           self.kb = 1
           self.T = 1/self.kb
           self.rc = rc
           self.rcD = rcD
           self.s = s
           self.a = a
           self.dt = dt
           self.sigma = sigma
           self.nBeads = (self.xsize*self.ysize*self.zsize)*self.rho
           self.neql = neql
           self.nrun = nrun
           self.nrunmsd = nrunmsd
           self.Nev = 1
           self.Nrep = Nrep
           self.Nfreq = Nfreq
```

5.4.2 - i metodi *generateFiles* & *currentFile*

Il metodo generateFiles si occupa del secondo step di creazione del file di input di LAMMPS. In questa fase i parametri definiti nello step precedente vengono applicati ad un template che è definito all'interno dello stesso metodo. Questa fase è estremamente critica per il procedere del metodo, infatti un errore nel codice del file di LAMMPS porterà alla creazione di file di input che conterranno errori e, di conseguenza, potrebbero essere generati files che non vengono letti dalle operazioni seguenti o addirittura potrebbe essere generato un file non funzionante del tutto.
Il template è completamente adattabile alle esigenze dello user, in questo caso è stato utilizzato come template il file di una simulazione di un fluido semplice, cioè l'acqua.

```
        def generateFiles(self,filename):

           import string
           import random
           S = 4
           ran = ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in range(S))
           random_code = str(ran)
           self.ran_code = random_code

           self.filename = str(filename) + "_" + random_code

           with open(self.filename, "w") as input:

              input.write(
                   "template_input_file".format(...)
                   )
```

Il file completo che è stato usato come template è stato inserito in Appendice e non nello snippet del codice per brevità. In questa fase del codice avviene anche il processo già citato di generazione del codice alfanumerico randomico e conseguente etichettatura dei file delle simulazioni. Il processo è eseguito dalle righe di codice sottostanti:

```
            import string
            import random
            S = 4
            ran = ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in range(S))
            random_code = str(ran)
            self.ran_code = random_code

            self.filename = str(filename) + "_" + random_code
```

La terza fase, cioè quella di effettiva generazione del file di input, viene eseguita da un altro metodo di classe, cioè `currentFile(self)`. Il codice eseguito da questo metodo è estremamente semplice e non aggiunge nessuna modifica ai parametri o ai file precedentemente specificati, semplicemente si limita a stampare in output i file ed il codice randomico per utilizzi successivi:

```
        def currentFile(self):
            return self.filename,self.ran_code
```

5.4.3 - il metodo *launchSims*

Il metodo `launchSims(self)` si occupa del lancio delle simulazioni, è in questo metodo che si differenziano le versioni degli algoritmi per Cluster e per macchina locale, la versione presentata è quella per Cluster:

```
        def launchSims(self):

        import subprocess

        subprocess.run('CASE_IN="{filename}"; prun singularity exec ../lammpsSH.sif
/home/lammps/src/lmp_mpi -in $CASE_IN'.format(...),shell=True)
```

La versione per Cluster presenta i comandi in *bash* comunicati al Cluster per mezzo della funzione `subprocess.run()`. E' presente inoltre un protocollo di parallelizzazione per le simulazioni su Cluster `prun` e l'esecuzione del container in cui è presente il codice LAMMPS `singularity`.

5.4.4 - il metodo *checkNrep*

In questo metodo vengono definite le operazioni che seguono le operazioni di check di *Nrep* descritte in Sez. 4. In questa fase viene controllato l'andamento della viscosità alla ricerca di un *plateau*. Questo fine è raggiunto per mezzo dell'invocazione di altre funzioni che verranno descritte successivamente, ma il cui operato può essere riassunto nel seguente ordine:

- Importazione dei dati necessari all'integrazione da `"time_cor.txt_{ran_code}"`
- Calcolo dell'integrale cumulativo
- Ricerca del plateau con il metodo Moving Average

Il codice è illustrato di seguito:

```
        def checkNrep(self,window,err_rel,jump_index):

            timestep, pxy, pxz, pyz = importdatas("time_cor.txt_{ran_code}".format(...),self.Nrep)

            cum_int = cum_integral(timestep, pxy, pxz, pyz, K=K(self.kb,self.Nev,15**3,self.dt,return_value="n"))

            visc_init = MovingSpans(cum_int)
            visc_plateau, Nrep, flag = visc_init.runAve(window,err_rel,jump_index,printNrep="n")

            return visc_plateau, Nrep, flag
```

La prima operazione è intuitiva, si tratta di una semplice operazione di importazione dal file `"time_cor.txt_{ran_code}"` dei valori `pxy, pxz, pyz` necessari poi per calcolare l'integrale cumulativo.
Il secondo step è il calcolo dell'integrale cumulativo, esso viene eseguito per mezzo della funzione `cum_integral()` che verrà descritta approfonditamente successivamente. I file di input sono `timestep, pxy, pxz, pyz, K`.
L'ultimo step viene eseguiti dall'oggetto di tipo *classe* `MovingSpans()` e dal suo metodo `runAve()`. I valori calcolati dell'integrale cumulativo vengono salvati all'interno di `MovingSpans()` e successivamente viene iniziato il processo di analisi valori calcolati per mezzo del metodo `runAve()`. Tutte le funzioni e le classi utilizzate in questi tre step, che consentono di eseguire il check di Nrep, vengono importate da tre files esterni all'inizio dei files *launcher_cluster.py* & *launcher_local.py*:

```
from CumulativeIntegral import K, cum_integral
from ImportDatas import importdatas
from MovingSpansAverage import MovingSpans
```

Il metodo restituisce infine i tre valori, il cui utilizzo è cruciale nello svolgimento e nel superamento del check di *Nrep*, così come descritto precedentemente in Sez. 4:

```
            return visc_plateau, Nrep, flag
```

5.4.5 - il metodo *checkN*

In questo metodo vengono eseguite le operazioni con la finalità di completare il check di *N*, come descritto in Sez. 4. In questo caso il processo di check non è possibile considerando una sola simulazione, l'ultima eseguita come nel caso del check di *Nrep*, ma è necessario confrontare due valori di COV che vengono calcolati in due simulazioni diverse e consecutive. Il confronto poi viene valutato sulla base di un errore relativo tra i due valori di COV: nel caso venga valutato un errore inferiore alla soglia imposta allora il check è superato.

```
def checkN(self):

    timestep, pxy, pxz, pyz = importdatas("time_cor.txt_{ran_code}".format(...),self.Nrep)

    cum_int = cum_integral(timestep, pxy, pxz, pyz, K=K(self.kb,self.Nev,15**3,self.dt,return_value="n"))

    import numpy as np
    cov = lambda x: np.std(x, ddof=1) / np.mean(x)
    cov_tmp = cov(cum_int)

    return cov_tmp
```

Come si può notare, le prime due operazioni eseguite sono omologhe al caso del check di *Nrep*, infatti sono le operazioni che sono già state descritte di importazione dei dati e calcolo dell'integrale cumulativo per ottenere l'andamento della viscosità. In seguito invece viene calcolato il valore di COV e salvato come output del metodo. In ultimo, si può notare che a differenza del metodo di check di Nrep in questo caso il controllo non viene attivamente eseguito all'interno del metodo e poi comunicato il risultato al file *main.py*, ma viene solo calcolato il valore di COV associato all'integrale cumulativo della viscosità. Il controllo effettivo viene effettuato all'esterno nel *main.py* come descritto in Sez. 5.3.

5.5 - *ImportDatas.py*

Buona parte dei processi descritti fino ad ora, come per esempio il calcolo dell'integrale cumulativo in *launcher_cluster.py* & *launcher_local.py*, utilizzano un sistema di importazione dei dati che permette l'estrazione dei dati dai file di output di LAMMPS, in formato di file di testo, e la loro rielaborazione in formati, o *type*, che siano leggibili e ed utilizzabili dallo script del metodo sviluppato.
Questo sistema di importazione dati è descritto all'interno del file *ImportDatas.py* illustrato di seguito:

```
import pandas as pd

def importdatas(path_to_file,Nrep):

        data = pd.read_csv(path_to_file, sep=" ", skiprows=4, header=None, engine='python')

        timestep = data[1].values.tolist()
        considered_timestep = timestep[-Nrep:]

        pxy = data[3].values.tolist()
        considered_pxy = pxy[-Nrep:]

        pxz = data[4].values.tolist()
        considered_pxz = pxz[-Nrep:]

        pyz = data[5].values.tolist()
        considered_pyz = pyz[-Nrep:]

        return considered_timestep, considered_pxy, considered_pxz, considered_pyz
```

Come si può vedere, il file comprende una sola funzione chiamata `importdatas(...)` che accetta in input due valori:

- **path_to_file** - una stringa di testo che indica alla funzione la posizione all'interno della macchina su cui viene eseguito il metodo, in generale questa stringa è automaticamente definita in *main.py* e comprende il file etichettato con il codice alfanumerico.
- **Nrep** - Qui viene definito il valore di *Nrep* impostato nella simulazione di cui si sta importando i dati. Questo parametro è molto importante per come viene eseguita l'azione di importazione, infatti la funzione utilizza questo parametro per riconoscere i valori di interesse da estrarre dal file, da esso infatti verranno estratti gli ultimi Nrep valori e se quest'ultimo valore dovesse essere errato, si andrebbe a calcolare un valore di viscosità nelle fasi successive dello script che erediterà l'errore generato in questa fase.

Questa funzione è molto semplice e non richiede ulteriori spiegazioni per definirne il funzionamento. L'unica cosa che è utile andare ancora a spiegare è l'effetto della stringa `import pandas as pd`. In questo punto del codice viene invocata

la libreria `pandas` che non esegue calcoli, ma serve solo come strumento per permettere la lettura dei file LAMMPS e la loro conversione in variabili di tipo *list.*

5.6 - *CumulativeIntegral.py*

Il file seguente contiene tutte le funzioni che permettono il calcolo dell'integrale cumulativo ed il valore di viscosità secondo il metodo di Green-Kubo illustrato in Sez. 3.2. La prima funzione definita è cum_integral(), il cui codice è illustrato di seguito:

```
def cum_integral(timestep, pxy, pxz, pyz,K):

        data_storage = [pxy,pxz,pyz]

        cum_int_storage0, cum_int_storage1, cum_int_storage2 = [], [], []

        for j in range(0,len(data_storage),1):

           current_ev_data = data_storage[j]

           for i in range(0,len(timestep),1):

              single_cum_int_value = np.trapz(current_ev_data[0:i],timestep[0:i])

              if j == 0:
                   cum_int_storage0.append(single_cum_int_value)
              if j == 1:
                   cum_int_storage1.append(single_cum_int_value)
              if j == 2:
                   cum_int_storage2.append(single_cum_int_value)

        funz_int_cum = __funzCum(K,cum_int_storage0,cum_int_storage1,cum_int_storage2,timestep)

        return funz_int_cum
```

In input alla funzione viene definito il *timestep,* in questo caso questa definizione è stata mantenuta nel codice ma può trarre in errore, infatti precedentemente è stato definito il *timestep* come Δt anche se, esclusivamente in questo caso, non si riferisce a quel parametro, ma al valore della scala temporale esplorata dalla simulazione DPD, cioè al valore $Nrep \cdot \Delta t$.
I parametri `pxy, pxz, pyz` sono le componenti spaziali del tensore dello stress, queste tre componenti calcolate da LAMMPS vengono prima integrate e successivamente il risultato viene mediato per ottenere il valore finale dell'integrale cumulativo necessario per il calcolo dell'equazione di Green-Kubo.

In ultimo, troviamo il parametro `K`, in esso sono racchiusi tutti i valori dei parametri DPD che compaiono all'esterno dell'integrale del metodo di Green-Kubo, esse sono state racchiuse in un'unica variabile per semplicità.

Il codice che comprende i parametri e le operazioni per il calcolo del parametro `K` sono illustrate di seguito:

```
def K(kb,Nev,V,dt,return_value='y'):

        T = 1/kb
        K_value = 1/(kb*T)*V*Nev*dt

        if return_value == 'y':
           print("Current K value: {K}\n".format(K = K_value))
        else:
           pass

        return K_value
```

Il file comprende anche una funzione privata, cioè una funzione che non può essere invocata se non nel file, e non ha alcuna utilità se non nel contesto dell'elaborazione dei calcoli all'interno del file stesso. Questa funzione è `__funzCum()` il cui codice è compreso di seguito:

```
def __funzCum(K,cum_int_storage0,cum_int_storage1,cum_int_storage2,timestep):

        viscosity_cum = []

        for i in range(0,len(timestep)):
           val_temp = (cum_int_storage0[i] + cum_int_storage1[i] + cum_int_storage2[i])/3*K
           viscosity_cum.append(val_temp)

        return viscosity_cum
```

Come si può notare, in input la funzione … raccoglie i risultati degli integrali cumulativi delle componenti spaziali del tensore di stress e successivamente ne media ogni termine. Questa operazione permette il calcolo della funzione della viscosità nella scala temporale considerata.

5.7 - *MovingSpansAverage.py*

Se il file *CumulativeIntegral.py* descritto precedentemente si occupa del calcolo dell'andamento della viscosità, il file *MovingSpansAverage.py* contiene tutte le operazioni necessarie per valutare la presenza del *plateau* di viscosità.
In questo caso conviene introdurre i concetti teorici che sono alla base del metodo: il calcolo che è stato implementato per il calcolo del *plateau* è un particolare adattamento dell'algoritmo di *Simple Moving Average* SMA. Il metodo è ampiamente utilizzato in ambito economico ed è un algoritmo che consente di calcolare l'andamento di una serie temporale con valori affetti da fluttuazioni. Non sorprende quindi che il suo utilizzo sia spesso necessario in ambito economico, pensiamo solo alle fluttuazioni di prezzo di un bene nel tempo.
La descrizione matematica della SMA è riportata in Eq.(5.1), qui di seguito:

$$SMA_k = \frac{1}{k} \sum_{i=n-k+1}^{n} p_i \tag{5.1}$$

dove p_i è il singolo valore osservato, nell'esempio precedente il prezzo del bene ad un dato tempo *t*, *n* è il numero totale di valori osservati, e *k* è il valore di window della SMA, cioè il numero di valori che vanno inclusi nel calcolo della media.
Immaginiamo l'applicazione tipica di questo algoritmo, cioè l'economia, non ci viene in mente come possa aiutarci questo metodo, ma noi abbiamo il vantaggio che sappiamo *a priori* quale dovrebbe essere l'andamento della nostra funzione di viscosità, in quanto sappiamo che prima o poi dovrà convergere ad un valore finale di viscosità perché, prima ancora, si avrà una convergenza del valore di SACF ad un valore nullo. Se noi partiamo da questo dato, allora ci viene molto facile immaginare un criterio di controllo che si basa sulle medie successive dei valori calcolati della funzione di viscosità. Se ad un certo punto si avrà che due valori medi rientreranno entro una certa tolleranza, **allora di conseguenza si può ragionevolmente affermare che la funzione, in quel punto, presenti il *plateau* di interesse**.
Il metodo tuttavia non è completo ed infallibile; infatti, data la fluttuazione ed il rumore legato ai fenomeni stocastici, si possono verificare dei casi in cui pur senza raggiungere la convergenza della viscosità si avrà che i valori di medie consecutive rientrino nella tolleranza impostata dallo user. Il problema è stato risolto con l'introduzione di un parametro, che verrà poi illustrato, che ha il compito di indicare *a quale distanza tra loro* vanno calcolate le medie, questo sistema permette quindi di valutare correttamente la presenza di convergenza, in

quanto permette di confrontare due valori di medie non consecutivi, ma ad una distanza tra loro definibile a priori.
Qui di seguito viene illustrato il codice che esegue le operazioni illustrate precedentemente. Verranno illustrati parametri chiave del controllo e le loro funzioni in ogni aspetto:

```
class MovingSpans:

        def __init__(self, cum_int_data):
           self.cum_int_data = cum_int_data
           self.count = 1

        def runAve(self,window,err_rel,jump_index,printNrep="y"):
           moving_averages, val_temp = [], []
           Nrep, val_media = 1, 0

           for i in range(0,len(self.cum_int_data),1):

              val_temp.append(self.cum_int_data[i])

              if len(val_temp) == window:
                   val_media = sum(val_temp)/window
                   moving_averages.append(val_media)

                   if len(moving_averages)>1+jump_index:
                   err_sol = abs((moving_averages[-1]-moving_averages[-2-jump_index]))/moving_averages[-1]

                      if err_sol <= err_rel:
                          while_flag = 1
                          return moving_averages[-1],Nrep, while_flag
                    else:
                         pass

                    val_media = []
                    val_temp = []

              Nrep += 1

           while_flag = 0
           return moving_averages[-1], Nrep, while_flag
```

Oltre al flusso di operazioni, che sono esattamente quelle descritte dalla SMA e che quindi non andremo ulteriormente a spiegare, conviene identificare i tre parametri più importanti specificati come input nel file, cioè `window,err_rel,jump_index`. Questi tre valori sono quelli che gestiscono i parametri di controllo precedentemente accennati:

- `window` - Questo parametro indica la quantità di valori della funzione di viscosità di cui viene fatta la media, più è alto questo numero e più è facile che le medie convergano.
- `err_rel` - Valore dell'errore relativo che deve essere rispettato dalle medie calcolate dei valori contenuti in `window`.
- `jump_index` - Parametro originale che indica a quanta distanza vanno calcolate due medie come spiegato precedentemente. Nel codice si può

vedere che i valori delle medie di `window` sono calcolati e salvati in una lista di valori. Il `jump_index` semplicemente va ad agire comunicando quanti valori di medie *saltare* tra i due che vengono invocati nel calcolo della tolleranza.

E' cruciale definire questi parametri in modo che non siano troppo stringenti, ma anche che non permettano il calcolo di risultati poco accurati. Questa fase di *tuning* dei valori è stata svolta nel corso del lavoro di tesi e ha portato all'identificazione dei seguenti valori specificati in *main.py*:

```
# Inputs of the checking algorithm
WINDOW = 30
ERR_REL = 1.0e-2
JUMP_INDEX = 8
```

5.8 - *GammaEff.py*

In questo file vengono definite le funzioni necessarie per il calcolo di γ_{eff}. Il procedimento ne calcola il valore per mezzo dell'implementazione in script. Andiamo ad illustrare il codice:

```
import pandas as pd
import numpy as np

def __import_rdf(path_to_file, n_values):
        tmp_rdf = pd.read_csv(...)
        r_ij = tmp_rdf.r_ij[-n_values:].values.tolist()
        g_ij = tmp_rdf.g_ij[-n_values:].values.tolist()
        return r_ij, g_ij

def gamma_eff(path_to_file, n_values, gamma, rcD, s):
        r_ij, g_ij = __import_rdf(path_to_file, n_values)
        w = [(1-r_ij[i]/rcD)**(2*s) for i in range(len(r_ij))]
        fun = [gamma*w[i]*g_ij[i]*4*np.pi*(r_ij[i]**2) for i in range(len(r_ij))]
        gamma_eff = np.trapz(fun, r_ij, rcD/50)
        return gamma_eff
```

Il codice appena illustrato mostra che vengono eseguite due operazioni da due funzioni diverse, abbiamo una fase di importazione dei dati definita dalla funzione privata `__import_rdf(...)` che, come precedentemente accennato, non viene esplicitamente invocata nel main.py, ma viene utilizzata all'interno della seconda funzione presente nel file, cioè `gamma_eff(...)`.
I valori necessari per il calcolo di γ_{eff} sono contenuti nel file di output LAMMPS *tmp.rdf*, quindi la prima fase del calcolo consiste quindi nell'importazione dei valori per poter eseguire il calcolo, successivamente i valori

vengono inseriti come input all'algoritmo che consente l'integrazione con il metodo dei trapezi del valore di γ_{eff} .

5.9 - *v_fitslope.py*

Il valore della diffusività viene calcolato con la relazione di Einstein, descritto in Eq.(). Il metodo prevede l'importazione e la media dei valori di *v_fitslope* calcolati da LAMMPS e contenuti nel file *log.lammps*, generato alla fine della simulazione DPD.
La struttura è molto simile ad funzioni definite precedentemente, cioè si basa su due processi consecutivi, l'importazione come già detto, ed il calcolo effettivo effettivo della diffusione. Il codice è presentato di seguito:

```
def v_fits(filename):

        string_start = "Step Temp Press c_msd[4] v_fitslope \n"

        line_flag = 0
        string_flag = 0

        with open(filename, "r") as file:
           with open("v_fitslope.txt", "w") as save:
               for line in file.readlines():

                    j = 1

                    if line == string_start or string_flag == 1:

                       for number in line.split():

                             if number == "Loop":
                             line_flag = 1
                             break
                             if j != 5:
                             number = number + " "
                             save.write(number)
                             if j == 5:
                             number = number + "\n"
                             save.write(number)

                             string_flag = 1
                             j += 1
                    else:
                       pass

                    if line_flag == 1:
                       break

        import pandas as pd
        save_df = pd.read_csv("v_fitslope.txt", sep=" ")
        mean_vfits = save_df["v_fitslope"][1:].mean()

        return mean_vfits
```

Capitolo 6

Analisi dei dati e risultati finali

Il metodo sviluppato si pone come obiettivo il calcolo del valore di viscosità per mezzo delle modalità che sono state illustrate nella Parte II della tesi.
Abbiamo quindi illustrato approfonditamente sia la base teorica per cui si vuole raggiungere questo risultato, che l'effettiva implementazione in forma di script delle operazioni necessarie per raggiungere l'obiettivo.
Questo ultimo capitolo invece presenterà i risultati che sono stati raggiunti, e gli andamenti dei valori calcolati delle proprietà di trasporto durante l'esecuzione del metodo. Come anticipato in Sez. (4.3), il metodo è stato testato per vari set di parametri, di cui vengono riassunti i valori di input in Tab. (4.1).
Le conclusione della fase di *testing* del metodo sono principalmente due:

- Verifica dell'effettiva efficacia del metodo e dello script - Il metodo quindi, sia teoricamente che come codice, deve garantire che i risultati siano sufficientemente accurati da poter essere considerato funzionante, questa prima fase dell'analisi dei risultati consiste nell'osservazione dei dei risultati alla ricerca di pattern attesi dalla teoria.
- La seconda parte invece consiste nella ricerca di relazioni tra i risultati calcolati per mezzo del metodo. In particolare vengono esaminati gli andamenti delle proprietà di trasporto in funzione della γ_{eff} .

6.1 - *Convergenza delle proprietà di trasporto in funzione di* r_{cD}

Le proprietà monitorate per ogni set dal metodo sono riassunte in Tab. (6.1):

D	Diffusività
Sc	Numero di Schmidt
ν	Viscosità
γ_{eff}	Gamma Effettivo

Tab. 6.1 - Proprietà investigate dal metodo

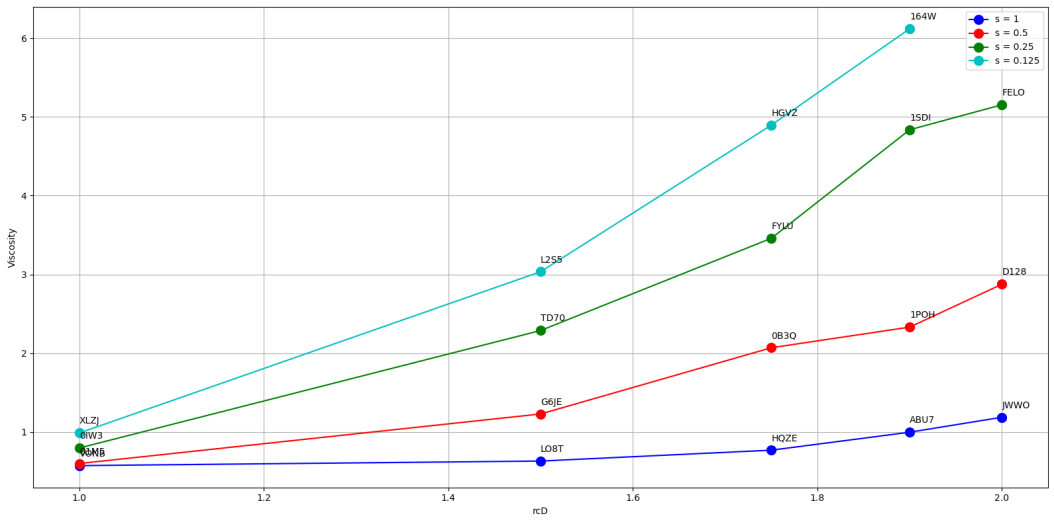
I risultati delle simulazioni verranno esposti in cinque sezioni, per ogni sezione verrà inizialmente illustrata una tabella riassuntiva dei parametri della simulazione DPD.
Come abbiamo anticipato in Sez. (4.3), le simulazioni in totale sono venticinque: sono stati esaminati infatti l'influenza di cinque valori diversi dell'esponente *s* e cinque valori di r_{cD}. Nelle sezioni vengono esaminate cinque simulazioni per volta in cui viene mantenuto costante il valore di *s* al variare di r_{cD} e successivamente vengono analizzate le proprietà di trasporto in funzione di r_{cD} per valutare se i risultati mostrano dei *patterns* attesi. Il fine di questa analisi consiste nel verificare che il metodo sia valido, di conseguenza siamo interessati a cercare degli andamenti delle proprietà di trasporto che possiamo spiegare teoricamente. Infine, se in tutte le simulazioni si verificano gli stessi *patterns* dei dati allora possiamo concludere che il metodo sviluppato è valido e che può essere considerato corretto.
Verrà anche verificata l'accuratezza dei risultati, infatti nel corso della descrizione dello script abbiamo visto come siano state implementate numerose tolleranze che devono essere rispettate, in particolare verrà controllata l'accuratezza del valore di viscosità finale.

6.1.1 - Risultati dei Set di simulazione

ρ	3
k_B	1
T	$1/k_B$
r_c	1
r_{cD}	1, 1.5, 1.75, 1.9, 2
s	1, 0.5, 0.25, 0.125
a	25
dt	0.04
γ	3
nBeads	10125

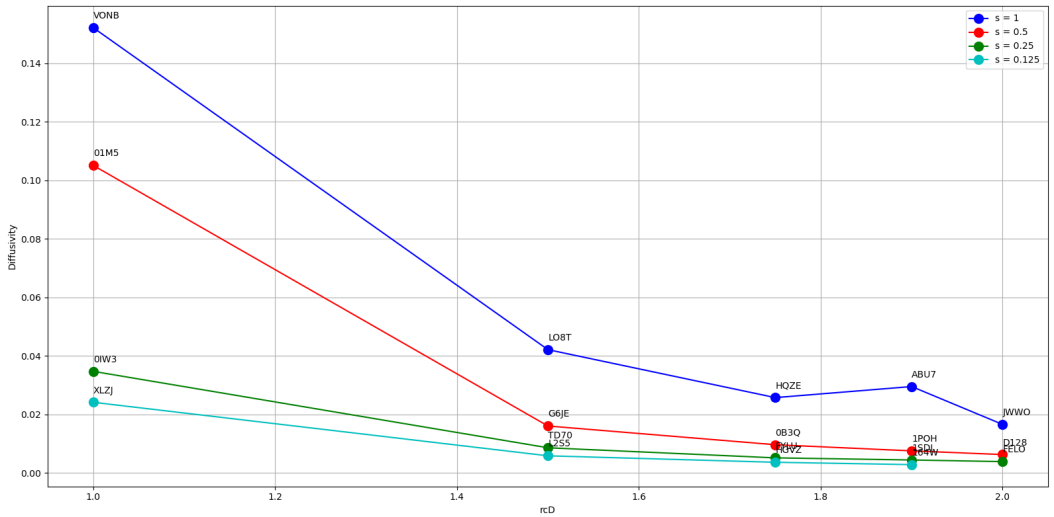
Tab. 6.2 - Parametri di input delle simulazioni

6.1.1.1 - Viscosità



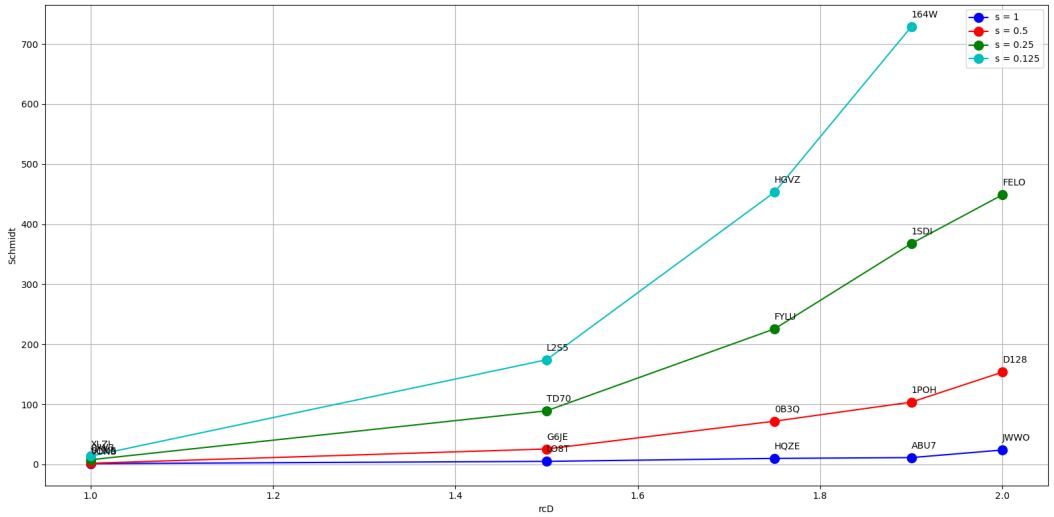
I plot mostrati della viscosità dimostrano in tutti i casi un andamento crescente del valore calcolato in funzione di r_{cD}. Il risultato in questo caso è atteso, infatti per come siamo andati a definire il parametro r_{cD} in Sez. 1.2, sappiamo che esso rappresenta il raggio d'azione associato alle interazioni di tipo dissipativo. Le interazioni dissipative nella DPD devono andare a simulare le forze di attrito che intercorrono tra le *beads* e, quindi, la viscosità di un fluido.
Possiamo anche notare come la viscosità sia influenzata dal valore del parametro *s*. Il parametro *s* è l'esponente della funzione peso del modello DPD, una sua variazione comporta quindi una variazione del valore finale calcolato. Si può notare come all'aumentare di *s* si abbassa il valore finale di viscosità, questo è atteso in quanto si ha che per un esponente tendente a 0 il valore della potenza tenderà ad 1, quindi si avrà che l'interazione dissipativa sarà più intensa entro il raggio d'azione definito da r_{cD}.

6.1.1.2 - Diffusività



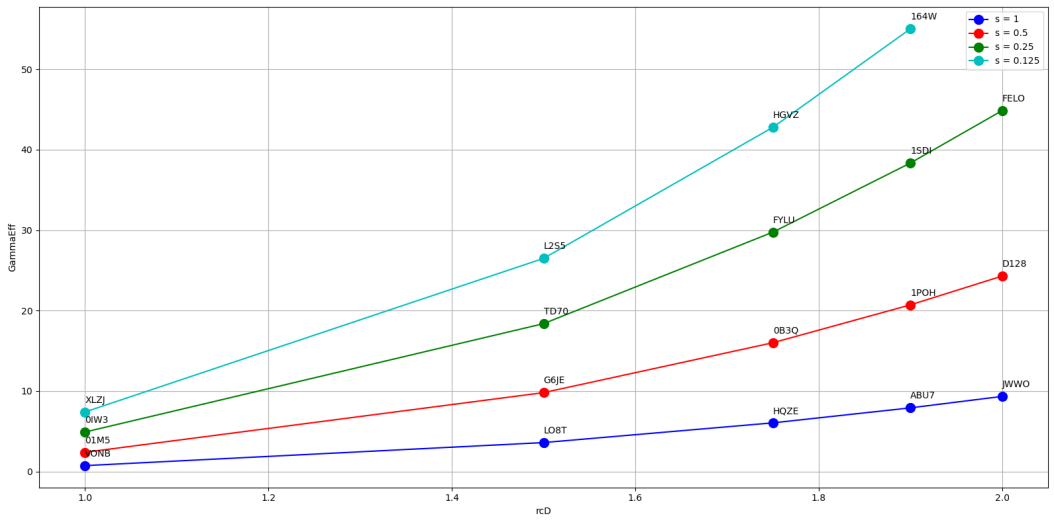
La diffusività è la proprietà di trasporto di un fluido che tiene conto della velocità con cui una particella diffonde. Nel modello DPD questa proprietà viene legata non alle singole particelle ma alle *beads* che, come abbiamo visto, rappresentano cluster di molecole. Il fenomeno di diffusione, essendo un fenomeno di trasporto di materia, è favorito nei fluidi a bassa viscosità, con conseguente incremento del valore di coefficiente di diffusività *D*.
Sapendo che la tendenza osservata è che all'aumentare della viscosità di un fluido, il coefficiente di diffusività tenderà ad abbassarsi, allora possiamo trarre la conclusione che i risultati dei plot dei valori di diffusività al variare di r_{cD} presentano un andamento atteso. Se infatti il valore di viscosità ci si aspetta che aumenti all'aumentare del raggio di azione delle forze dissipative, allo stesso modo allora si può dedurre che invece il valore di diffusività tenderà ad abbassarsi per lo stesso andamento di r_{cD}.
Anche in questo caso quindi si può affermare che il risultato è quello atteso.

6.1.1.3 - Numero di Schmidt



Il numero adimensionale di Schmidt, definito matematicamente in Eq. (3.3), quantifica il rapporto tra diffusione cinematica e diffusione di materia in un fluido. Data la sua definizione matematica, è facile intuire quindi che ad alto *Sc* si avrà una prevalenza degli effetti dovuti alla dissipazione dovuta alla viscosità di un fluido. Al numeratore infatti è presente la viscosità cinematica $\nu = \mu/\rho$ definita come il rapporto tra viscosità dinamica e densità delle *beads*, mentre un basso numero di *Sc* è associato ad una preponderanza dei fenomeni diffusivi legati al coefficiente di diffusione *D*, che troviamo infatti al denominatore.
Data l'interpretazione che viene data a *Sc*, è facile intuire che all'aumentare di r_{cD} il risultato atteso è che si avrà un aumento del numero di Schmidt, andamento confermato per i valori di output dal metodo sviluppato.

6.1.1.4 - Gamma Effettivo



Il termine γ_{eff} definisce l'attrito complessivo tra le *beads* e dipende dai parametri DPD s, r_{cD}, γ e, implicitamente, Nm. Anche in questo caso l'andamento atteso è l'aumento del valore di γ_{eff} in funzione di r_{cD} per come abbiamo definito il parametro, come negli altri casi abbiamo che il risultato effettivo atteso è riscontrato per tutti i set analizzati.

6.2 - Analisi delle relazioni tra PdT in funzione di γ_{eff}

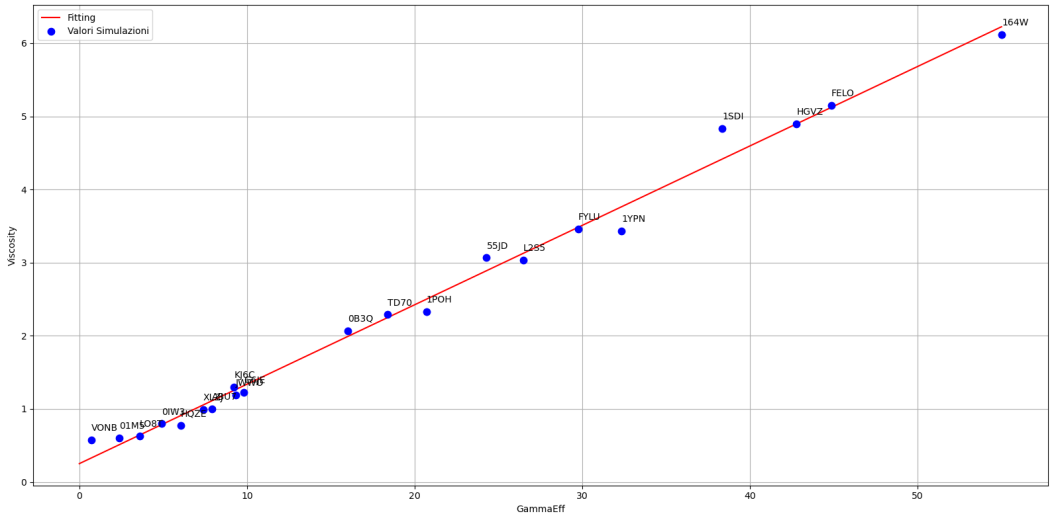
Nella seguente sezione verranno discussi dei risultati originali elaborati durante il lavoro di tesi. In particolare, andremo a evidenziare le relazioni che legano i valori delle proprietà di trasporto in funzione della variazione di γ_{eff}, i risultati non sono catalogati per Sets come nel caso precedente, ma vengono esaminati gli andamenti complessivi di tutti i set *finali*, cioè quei set che hanno raggiunto la convergenza completa. La scelta di esplorare queste particolari relazioni è dovuta al fatto che si può osservare che, nonostante i valori di ingresso differenti, si calcolano valori simili di PdT per valori simili di γ_{eff}, come confermato in Tab.6.1.

ν	D	Sc	γ_{eff}	$CODE$
3.035	0.0058	174.16	26.48	L2S5
3.066	0.0062	164.05	24.29	55JD
4.895	0.0036	453.22	42.78	HGVZ
5.154	0.0038	448.91	44.86	FELO

Tab. 6.1 - Due esempi di PdT calcolati

Quest'osservazione ci porta quindi a pensare che non solo i Sets con valori simili di γ_{eff} avranno anche valori simili di PdT, ma che ci sia una relazione che può descrivere l'andamento tra questi risultati.

6.2.1 - Viscosità

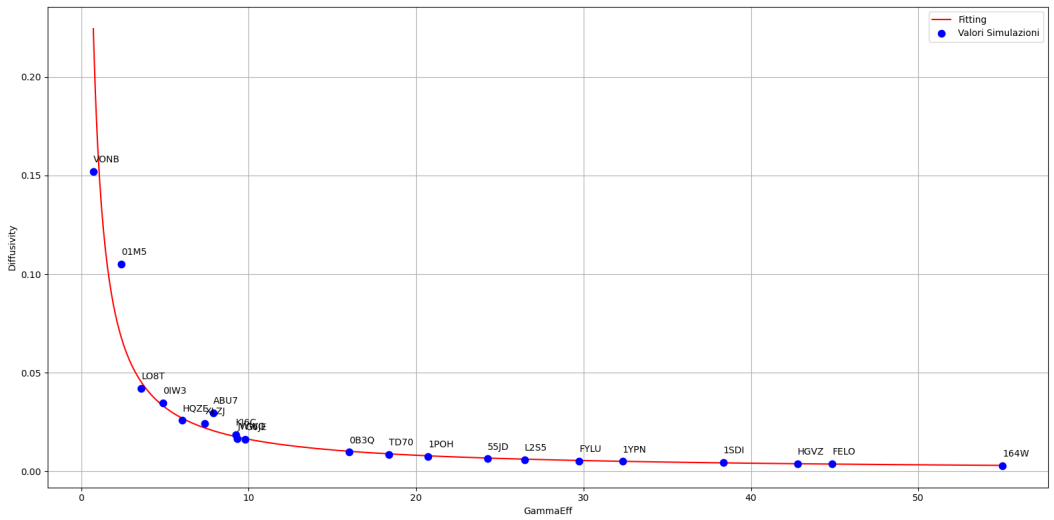


I valori trovati della viscosità per i set finali sono rappresentati nella figura qui riportata, si può immediatamente notare come al variare del valore di γ_{eff} si ha un andamento regolare del valore di viscosità. L'andamento è lineare e può essere descritto matematicamente dell'equazione e dai parametri riportati in nella tabella seguente:

Equazione	*a*	*b*	R^2
$y = a \cdot x + b$	0.1086	0.2522	0.9954

Come si può notare, il coefficiente di determinazione R^2 conferma una forte linearità dei dati, da qui è possibile dedurre l'accuratezza della relazione trovata.

6.2.1 - Diffusività

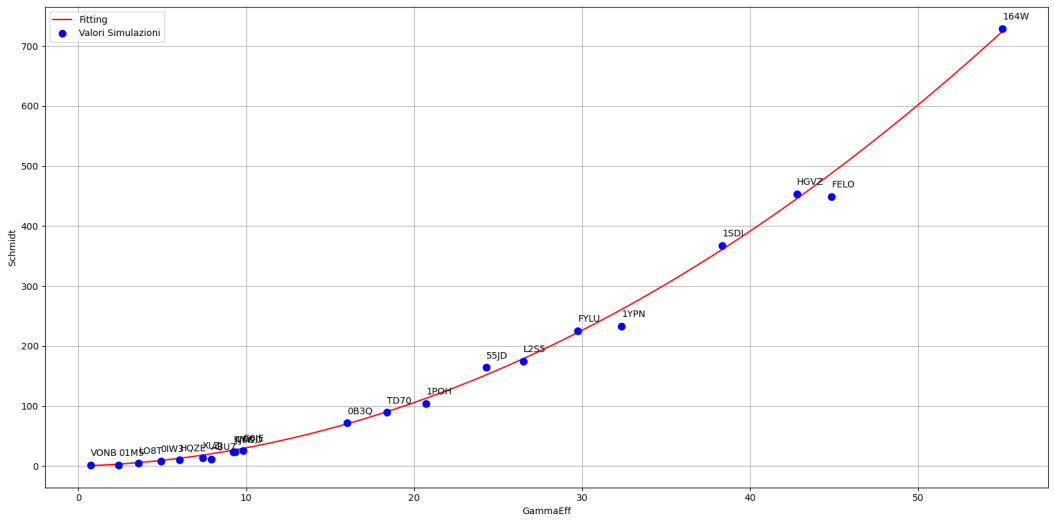


L'andamento dei valori della diffusività è molto diverso rispetto al caso precedente, si può notare infatti che non segue un trend lineare. Il fitting in questo caso è stato possibile per mezzo di una legge di potenza i cui parametri sono descritti nella tabella seguente:

Equazione	*a*	*b*	R^2
$y = a \cdot x^b$	0.1632	-1.004	0.9783

Anche in questo caso si può notare come il coefficiente R^2 sia molto prossimo ad 1, confermando l'accuratezza dell'interpolazione.

6.2.1 - Schmidt



In figura si può notare come i valori calcolati del numero di Schmidt segua un andamento descritto per mezzo di una polinomiale di grado 2. Vengono presentati i parametri del fitting nella tabella sottostante:

Equazione	*a*	*b*	R^2
$y = a \cdot x^2 + b \cdot x$	0.2248	0.7970	0.9974

Anche in questo ultimo caso si può notare che il valore del coefficiente R^2 conferma l'accuratezza del fitting.

Conclusioni

L'obiettivo della tesi è di sviluppare e implementare in codice un metodo automatizzato per il calcolo della viscosità di un fluido utilizzando la *Dissipative Particle Dynamics*. L'intero processo che ha portato alla realizzazione ed al testing del metodo è durato diversi mesi ed è stato possibile solo con una serie di studi precedenti allo sviluppo del metodo stesso. Queste fasi possono essere riassunte in:

1. **Studio della convergenza della viscosità nelle simulazione DPD**

In questa fase iniziale del lavoro di tesi si sono voluti studiare gli effetti dei parametri di post-processing sull'andamento della viscosità nelle simulazioni DPD, in particolare sono state considerati dei set di simulazioni e si è andato ad osservare l'andamento della viscosità al variare dei parametri di *N*, *Nrep/Nfreq*.

2. **Sviluppo teorico del metodo automatizzato**

In seguito alle conclusioni tratte nella fase precedente è stato possibile lo sviluppo di un metodo, come illustrato in Sez. 5, si tratta di un metodo automatizzato che si basa sul lanciare iterativamente delle simulazioni ed osservarne i risultati generati. Si sono anche ipotizzati dei controlli sui risultati che permettono al metodo di riconoscere dei risultati sufficientemente accurati e quindi stampare i risultati ottenuti.
Nello studio precedente, quello descritto al punto **1.**, si sono anche definiti i valori minimi per impostare la prima simulazione.

3. **Implementazione in codice del metodo automatizzato**

Questa fase è stata la più delicata e complessa, infatti l'implementazione in codice è stato un aspetto inedito del seguente lavoro di tesi. Il codice riprende il metodo sviluppato automatizzato ma presenta anche alcuni accorgimenti inediti, tra cui un sistema di sicurezza implementato per evitare ciclo infiniti in fase di calcolo di *Nrep* ed i diversi sistemi di inizializzazione del processo.
Il codice, nella sua interezza, è stato analizzato in Sez. 5.2, l'analisi comprende non solo la spiegazione delle operazioni eseguite, ma anche la spiegazione delle librerie usate che non sono state direttamente sviluppate durante il lavoro di tesi.

4. ***Testing* e analisi dei dati**

La fase finale del lavoro di tesi consiste nella validazione del metodo per mezzo del *testing* dell'analisi dati. In questa fase, i cui risultati sono presentati e discussi in Sez. 6, si sono generati dei plot delle proprietà di trasporto delle simulazioni calcolate e se ne sono analizzati gli andamenti alla ricerca di pattern attesi che confermassero l'effettiva efficacia del metodo. Oltre alla validazione del metodo,

in questa fase vengono anche analizzati le proprietà di trasporto in funzione di γ_{eff} , i valori poi sono stati interpolati e commentati.

Tutti i risultati analizzati alla fine del lavoro di tesi hanno confermato la presenza di andamenti noti al variare dei parametri di input. Anche le verifiche che sono state fatte sui valori di convergenza finale, quindi l'errore relativo associato al valore finale di viscosità, hanno dimostrato che effettivamente il metodo automatizzato si conclude solo quando effettivamente tutte le soglie di accuratezza sono state rispettate.
L'analisi delle proprietà di trasporto in funzione di γ_{eff} è stata eseguita considerando tutti i valori estrapolati dai Sets che hanno raggiunto la convergenza con il metodo, quindi al *timestep* per la quale non si osserva più un cambiamento del valore di viscosità. Partendo dall'osservazione descritta in Sez. 6.2, si è deciso di andare a valutare le relazioni che intercorrono tra le proprietà di trasporto e γ_{eff} . Si è eseguito un fitting dei dati che ha portato alla definizione di 3 relazioni, il cui significato fisico però deve ancora essere spiegato completamente e di cui sono stati riportati i risultati.
Per concludere si può affermare che il metodo automatizzato è in grado di calcolare un valore di viscosità, e valori di proprietà di trasporto, accurato per un determinato sistema che simula un fluido semplice per mezzo di *Dissipative Particle Dynamics*. Questo risultato è stato raggiunto in meno iterazioni e quindi con un notevole risparmio di risorse computazionali.

Bibliografia

[1] P. Español, P. Warren.
Perspective: Dissipative Particle Dynamics.
The Journal of Chemical Physics, 146:15090,1 (2017).

[2] C. Hijón, P. Español, E. Vanden-Eijnden, R. Delgado-Buscalioni.
Mori–zwanzig formalism as a practical computational tool.
Faraday Discuss., 144:301–322, (2010).

[3] P.J. Hoogerbrugge, J.M.V.A. Koelman.
Simulating microscopic hydrodinamic phenomena with dissipative particle dynamics.
Europhysics Letters, 19:155–160, (1992).

[4] P. Español, P. Warren.
Statistical Mechanics of Dissipative Particle Dynamics.
Europhysics Letters, 30:191-196, (1995).

[5] I. Pivkin, G. Karniadakis.
Coarse-Graining Limits in Open and Wall-Bounded Dissipative Particle Dynamics Systems.
The Journal of Chemical Physics, 124:184101-184101-7 (2006).

[6] M.B. Liu, G.R. Liu, L.W. Zhou.
Dissipative Particle Dynamics (DPD): An Overview and Recent Developments.
Arch Computat Methods Eng, 22:529-556, (2015).

[7] H.C. Andersen, J.D. Weeks, D. Chandler.
Relationship between the hard-sphere fluid and fluids with realistic repulsive forces.
Physical Review A, 4:1597–1607, (1971).

[8] R.D. Groot, P. Warren.
Dissipative Particle Dynamics: Bridging the Gap between Atomistic and Mesoscopic Simulation.
The Journal of Chemical Physics, 11:4423-4435, (1997).

[9] P.V. Coveney, P. Español.
Dissipative particle dynamics for interacting multicomponent systems.
Journal of Physics A: Mathematical and General, 30:779–784, (1997).

[10] G. Besold, J. M. Polson, I. Vattulainen, M. Karttunen.
Integration schemes for dissipative particle dynamics simulations: From softly interacting systems towards hybrid models.
Journal of Chemical Physics, 116:3967–3979, (2002).

[11] K.E. Novik, P.V. Convey.
Finite-difference methods for simulation models incorporating nonconservative forces.
Journal of Chemical Physics, 109:7667–7677, (1998).

[12] M.H.J. Hagen, I.Pagonabarraga, D.Frenkel.
Self-consistent dissipative particle dynamics algorithm.
Europhysics Letters, 42:377–382, (1998).

[13] T.Shardlow.
Splitting for dissipative particle dynamics.
SIAM Journal on Scientific Computing, 24:1267–1282, (2003).

[14] G. Besold, O. G. Mouritsen, A. F. Jakobsen.
Multiple time step update schemes for dissipative particle dynamics.
Journal of Chemical Physics, 124:094104, (2006).

[15] I. Pivkin, G. Karniadakis, W. Pan.
Single-particle hydrodynamics in dpd: a new formulation.
Nature Materials, 84:10012, (2008).

[7] X. Fan, N. Phan-Thien, S. Chen, X. Wu, T. Yong Ng.
Simulating Flow of DNA Suspension Using Dissipative Particle Dynamics.
Physics of Fluids, 18:063102-63102-10, (1994).

[17] G. Jung, F. Schimid.
Computing bulk and shear viscosities from simulations of fluids with dissipative and stochastic interactions.
Journal of Chemical Physics, 144:204104, (2016).

[18] M.S. Green.
Markoff random processes and the statistical mechanics of time dependent phenomena.
Journal of Chemical Physics, 20:1281–1295, (1952).

[19] R. Kubo.
Statistical-mechanical theory of irreversible processes. General theory and simple applications to magnetic and conduction problems.
J. Phys. Soc. Jpn., 12:570–586, (1957).

[20] E. Helfand.
Transport coefficients from dissipation in a canonical ensemble.
Physical Review, 119:1, (1960).

[21] A. W. Lees, A.F. Edwards.
The computer study of transport processes under extreme conditions.
Journal of Chemical Physics, 5, (1972).

[22] H.C. J. Hoefsloot J.A. Backer, C.P. Lowe, P.D. Iedema.
Poiseuille flow to measure the viscosity of particle model fluids.
Journal of Chemical Physics, 122:154503, (2005).

[23] F. Müller-Plathe.
Reversing the perturbation in nonequilibrium molecular dynamics: An easy way to calculate the shear viscosity of fluids.
Physical Review E., 59:5, (1999).

[24] L. Onsager.
Reciprocal relations in irreversible processes.
Physical Review, 37:405–426, (1931).

[25] P. Carbone, P.Asinari, H. Droghetti, I. Pagonabarra, D. Marchisio.
Dissipative particle dynamics simulations of tri-block co-polymer and water: Phase diagram validation and microstructure identification.
Journal of Chemical Physics, 149:184903, (2018).

[26] D.Frenkel and B.Smit.
Understanding Molecular Simulation, 2nd Edition.
Academic Press, 2001.

[27] Español, P.
Hydrodynamics from Dissipative Particle Dynamics.
Physical Review. E, 52:1734-1742, (1995).

[28] N. Lauriello, J. Kondracki, A. Buffo, G. Boccardo, M. Bouaifi, M. Lisal, D. Marchisio.
Simulation of High Schmidt Number Fluids with Dissipative Particle Dynamics: Parameter Identification and Robust Viscosity Evaluation.
Physics of Fluids, 33:73106, (2021).

[29] J. Larentzos, J. Brennan, J. Moore, M. Lísal, W. Mattson.
Parallel Implementation of Isothermal and Isoenergetic Dissipative Particle Dynamics Using Shardlow-like Splitting Algorithms.
Computer Physics Communications, 185:1987-1998, (2014).

[30] F. Lahmar, B. Rousseau.
Influence of the Adjustable Parameters of the DPD on the Global and Local Dynamics of a Polymer Melt.
Polymer (Guilford), 48:3584-3592, (2007).

[31] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, S. J. Plimpton.
LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales.
Comp Phys Comm, 271:10817, (2022).

[33] N. Lauriello.
Dissipative Particle Dynamics Come Strumento per Reologia Computazionale: Applicazione a Fluidi Semplici e Complessi = Dissipative Particle Dynamics as a Tool for Computational Rheology: Application to Simple and Complex Fluids.
(2021).

[34] R.D. Groot, K.L. Rabone.
Mesoscopic simulation of cell membrane damage, morphology change and rupture by nonionic surfactants.
Biophysical journal, 81:725–736, (2001)

[35] A. Chaudhri, J. R. Lukes.
Velocity and Stress Autocorrelation Decay in Isothermal Dissipative Particle Dynamics.
Physical Review E, 81:026707, (2010).

[36] P. Español, M. Serrano.
Dynamical Regimes in the Dissipative Particle Dynamics Model.
Physical Review E, 59:6340–6347, (1999).

[37] C.A. Marsh.
Static and Dynamic Properties of Dissipative Particle Dynamics.
Physical Review E, 56:1676–1691, (1997).