Informe de Laboratorio N°5

Alumno: Méndez Martín
Docentes: Dra. Marconi Verónica I.; Dr. Banchio Adolfo
Universidad Nacional de Córdoba (UNC)
Facultad de Matemática, Astronomía, Física y Computación (FaMAF)
Curso de Física Computacional: Problemas N°1,2,3

(*) Para mayor claridad se recomienda ver figuras en la versión digital.

I. Introducción

I-1. Dinámica Molecular

La simulación de tipo dinámica molecular (DM) es una técnica útil en cálculos de equilibrio y propiedades de transporte de sistemas clásicos de muchos cuerpos. Y se trata de sistemas clásicos debido a que los centros de masa de las partículas constituyentes del sistema se mueven de acuerdo las leyes de la mecánica clásica y donde se estudian comportamientos en rangos donde los efectos cuánticos son despreciables. Por otro lado, las simulaciones de dinámica molecular comparten muchas propiedades de los experimentos reales, es decir, primeramente preparamos la muestra, aplicamos un sistema modelo consistente de n_p partículas y resolvemos las ecuaciones clásicas de movimiento (mecánica Newtoniana, Lagrangiana o Hamiltoniana) para este sistema hasta que las propiedades del sistema no cambien más con el tiempo (sistema en equilibrio), luego del equilibrio realizamos las mediciones de las propiedades correspondiente y para ello debemos expresar los observables que mediremos en función de las coordenadas generalizadas del sistema (posiciones y momentos).

I-2. Ensamble Microcanónico

Teniendo en cuenta que nos centramos en los posibles estados del sistema mecánico manteniendo el numero de partículas fijo ($n_p = cte$), el volumen total del sistema ($L^3 = cte$) y la energía total del sistema ($E_{tot} = cte$), tratamos con un ensamble microcanónico.

I-3. Temperatura y corrección de velocidades

El significado físico clásico de la temperatura es que es una cantidad estadística y usando el teorema de equipartición de la energía, que involucra a todos los grados de libertad que aparecen como términos cuadráticos en el hamiltoniano del sistema, podremos expresar la energía cinética promedio por grado de libertad de la siguiente manera,

$$\left\langle \frac{1}{2} m (v_\alpha)^2 \right\rangle = \frac{1}{2} k_B T \tag{1}$$

ahora bien, como la energía cinética total del sistema fluctua podemos definir la temperatura instantánea como muestra la ecuación 2.

$$T(t) = \sum_{i=1}^{n_p} \frac{m_i [v_i(t)]^2}{k_B N_f}; N_f = 3n_p \tag{2}$$

donde N_f se refiere a los grados de libertad del sistema.

Por otro lado, debido al numero finito de partículas la temperatura puede modificarse demasiado debido al comportamiento de ciertas partículas en la simulación que al tener un sistema finito es relevante. Por ello, en la termalización es necesario asegurarse de que no se produzcan cambios muy grandes de la temperatura respecto a la de referencia, para lograr esto, se produce un re-escaleo de las velocidades utilizando el *termostáto* más usual que se muestra en la ecuación 3.

$$\vec{v}(t_{new}) = \lambda \vec{v}(t_{old}); \lambda = \sqrt{\frac{T_{ref}}{T(t_{new})}} \tag{3}$$

teniendo en cuenta que altera la dinámica de las partículas, pues produce discontinuidades en las velocidades y no se logran cumplir las ecuaciones de Newton, se utiliza únicamente en el régimen transitorio hasta lograr la equilibración del sistema.

I-4. Potencial de Lennard-Jones

En el presente trabajo consideramos como potencial de interacción el potencial interatómico de Lennard-Jones (ver figura 1) el cual sirve como modelo de fuerzas de enlace entre pares de moléculas y es utilizado para calcular la fuerza de van der Waals. Este potencial depende básicamente de tres parámetros (ver 4) r_{ij} que es la distancia relativa entre centros de masa de un par de partículas, ϵ que es la magnitud del pozo de potencial y σ es es la distancia interatómica para la cual el potencial se anula. Además, podemos notar que el primer término del potencial tiene en cuenta qué tan intensa es la repulsión entre el par de partículas a medida que aumenta (o disminuye) la distancia interatómica, y el segundo término tiene en cuenta qué tan intensa es la atracción entre el par de partículas a media que aumenta (o disminuye) la distancia interatómica.

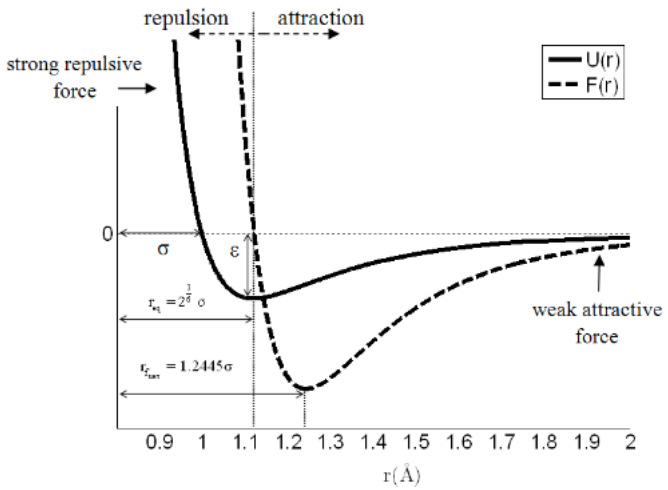


Fig. 1. Lennard-Jones: Potencial y fuerzas de interacción vs distancia interatómica

$$u_{ij} = 4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \quad (4)$$
$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

Ahora bien, para las simulaciones es necesario adimensionalizar todas las magnitudes que tengan dimensiones, entonces el potencial individual quedará de la forma según 5.

$$u^*_{ij} = 4\left(\frac{1}{r^*_{ij}}\right)^6\left[\left(\frac{1}{r^*_{ij}}\right)^6 - 1\right]; u^*_{ij} = \frac{u_{ij}}{\epsilon}; r^*_{ij} = \frac{r_{ij}}{\sigma} \quad (5)$$

Por otro lado, la fuerza interatómica se define como el opuesto al gradiente del potencial entre pares y en forma adimensional quedará según la ecuación 6. Y en componentes será según 7.

$$f^*_{r,ij} = -\frac{\partial u^*_{ij}}{\partial r^*_{ij}} = 24\frac{1}{r^*_{ij}}\left(\frac{1}{r^*_{ij}}\right)^6\left[2\left(\frac{1}{r^*_{ij}}\right)^6 - 1\right] \quad (6)$$

$$f^*_{x_k,ij} = \frac{r^*_{x_k,ij}}{r^*_{ij}} f^*_{r,ij}; x_k = \{x, y, z\} \quad (7)$$

Finalmente el potencial total será la suma de los potenciales individuales como muestra la ecuación 8. Notemos que se ha definido una distancia de corte entre pares r_{cutoff} este importante parámetro nos permite decidir hasta que distancia podemos considerar al potencial de interacción entre pares relevante, es decir, para distancias interatómicas mayores a este radio el potencial es despreciable y lo consideramos, a efectos prácticos, nulo, en otras palabras, este radio de corte nos permite determinar cuántas partículas vecinas interactúan cuando nos situamos en una dada partícula, la magnitud de este radio estará fuertemente influenciado a si estamos tratando con interacciones de largo o corto alcance, además, deberá ser menor a la mitad de la longitud L de la celda periódica como veremos luego.

$$U_{tot} = \sum_{j=2}^{n_p}\sum_{i=1}^{j-1} u^*_{ij}; \forall r^*_{ij} \leq r_{cutoff} \quad (8)$$

II. Resultados y Discusiones

Primeramente se consideró un sistema de $n_p = 256$ partículas, densidad $\rho = 0{,}8$ (partículas por unidad de volumen), temperatura de referencia adimensional $T_{ref} = 1{,}1$, paso temporal adimensional de $\Delta t = 0{,}005$ radio de corte adimensional de $r_{cutoff} = 2{,}5$.

II-A. Configuración de estructura cristalina inicial

La configuración de las posiciones iniciales de las partículas considerada fue la de una estructura cúbica centrada en las caras (FCC), la cual tiene un total de 4 átomos por celda unidad, y cuyos vectores primitivos son $\vec{a}_1 = \frac{a}{2}(\hat{e}_x + \hat{e}_y)$; $\vec{a}_2 = \frac{a}{2}(\hat{e}_y + \hat{e}_z)$; $\vec{a}_3 = \frac{a}{2}(\hat{e}_x + \hat{e}_z)$, entonces, como el total de partículas debe conservarse, la celda unidad deberá repetirse un cierto número de veces de tal forma de respetar este vínculo y además cumplir con la densidad impuesta externamente ρ. El código desarrollado corresponde a la subrutina **initial_lattice_configuration** dentro del módulo **module_md_lennard_jones.f90** y en la figura 2 se puede observar la disposición de partículas en el estado inicial, la misma fue centrada en el rango $[-L/2; L/2]$.

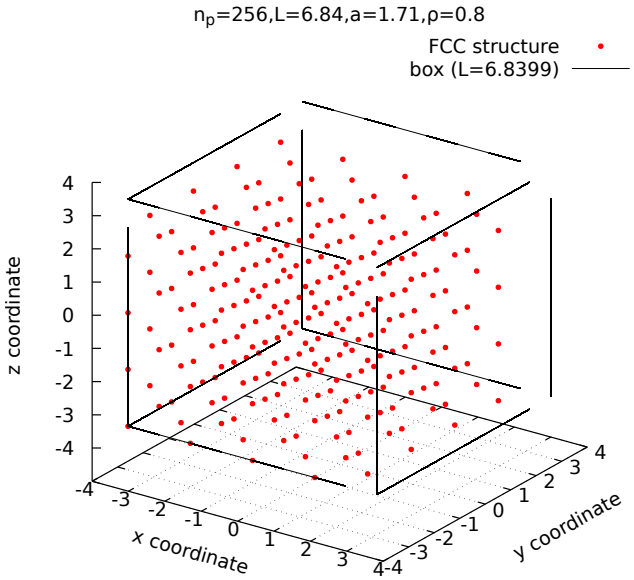


Fig. 2. Posiciones iniciales de las partículas: Estructura FCC

II-B. Condiciones de contorno periódicas y corrección por imagen mínima

Teniendo en cuenta que quieren estudian propiedades en el equilibrio termodinámico ($n_p \to \infty$) y debido a que no nos interesan estudiar efectos de superficie, es decir, excluyendo el comportamiento de las partículas cerca del borde físico del sistema macroscópico, podremos eliminar dicho borde considerando una celda de estudio que se repite infinitamente, es decir, llenamos el espacio de copias idénticas de estas regiones de simulación. De esta manera una partícula que sale de esta región de estudio por alguna delimitación particular de esta, deberá ser remplazada inmediatamente por otra que tiene el mismo momento lineal desde la delimitación opuesta a la original, de esta forma actuará una condición de contorno periódica (PBC) sobre cada partícula. En la figura 3 se muestra esquemáticamente en qué consisten las PBC en un sistema partículas en 2D, en este caso caso cada partícula podrá cruzar la región de simulación por cualquiera de los cuatro borde, sin embargo, en 3D (como es nuestro caso) las partículas podrán cruzar la región por cualquiera de los 6 bordes de la celda tridimensional de estudio. Por el hecho de considerar PBC el cálculo de las fuerzas también requerirán ciertas correcciones, llamadas corrección de desplazamientos por imagen mínima las cuales consisten en que, al momentos de centrarnos en una determinada partícula para computar la fuerza neta actuando sobre ella debido a la interacción con las otras partículas del entorno, deberemos centrar también la celda de estudio en la propia partícula y considerar las distancias interatómicas dentro de esta nueva celda y, notando que las distancias relativas serán siempre menores o iguales a la mitad de la celda cúbica se deberán corregir todas aquellas distancias que superen la mitad de la celda cúbica de simulación. En la figura 4 se observa esquemáticamente esta convención para corregir el desplazamiento según imagen mínima. El código desarrollado para estas correcciones de posiciones y desplazamientos corresponde a las funciones **pbc_correction**,**rel_pos_correction** y a la subrutina **position_correction** dentro del módulo **module_md_lennard_jones.f90**.

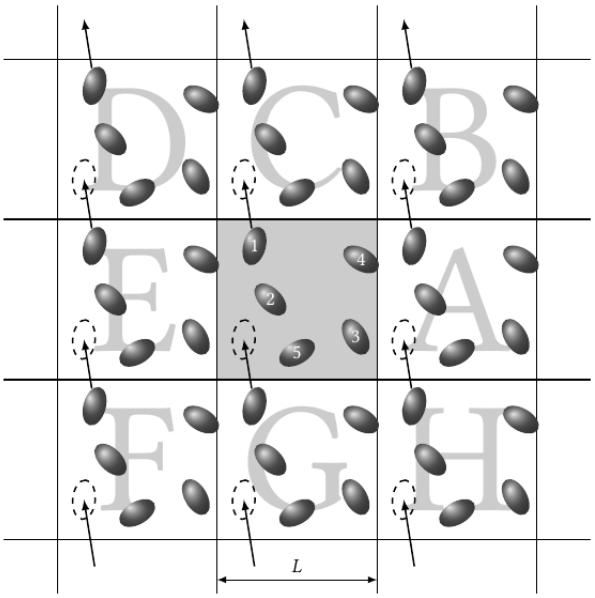


Fig. 3. Condiciones de contorno periódicas en un sistema 2D

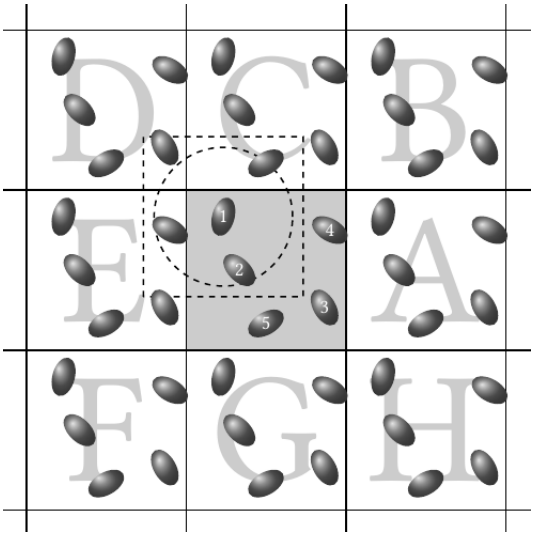


Fig. 4. Correcciones según imagen mínima en un sistema 2D

II-C. Configuración inicial de parámetros

Teniendo en cuenta que las simulaciones de DM se encargan de muestrear todo el espacio de configuraciones, el cual su éxito en dicha exploración dependerá únicamente del tiempo que le demos para recorrer el espacio y no de las configuraciones iniciales dadas, por ello, si bien sabemos que, en el equilibrio, las componentes de la velocidad seguirán una distribución Gaussiana y la velocidad total seguirá una distribución de Maxwell-Boltzmann, para reducir tiempo de simulación posiblemente se deba invertir un esfuerzo en proponer estas distribuciones de velocidad para el estado inicial, sin embargo, como veremos la ganancia de tiempo de CPU en las simulaciones de DM están fuertemente influenciadas por el cuello de botella del algoritmo, que es, el cálculo de las fuerzas de interacción entre pares de partículas. Entonces, debido a que no se gana demasiado en proponer distribuciones especificas en las velocidades iniciales, se propuso una distribución random uniforme en el rango $[-0{,}5; 0{,}5]$ para cada componente de la velocidad (utilizando el generador de números pseudoaleatorios Mersenne Twister), por otro lado, cuando nos referimos a las velocidades de las partículas, las medimos respecto del centro de masas del sistema, por tanto, la presión y la temperatura del sistema de partículas no se modifican si el recipiente que lo contiene está en movimiento, para ello a las velocidades aleatorias se le resta la velocidad del centro de masas. Luego, se escalan las velocidades según el termostato descrito en la sección I-3. Cabe mencionar que las posiciones iniciales según la estructura FCC podrían modificarse restándoles las distancias que recorrería cada partícula con las velocidades iniciales definidas anteriormente y luego corregir dichas posiciones según PBC, sin embargo, esta consideración es irrelevante respecto a los resultados obtenidos en el equilibrio e introduce cierto tiempo de CPU que desea reducirse, por ello no fueron consideradas en las simulaciones y se partió de una configuración posicional de las partículas según la estructura FCC. El código desarrollado para estas configuraciones iniciales corresponde a la subrutina **md_initial_parameters** dentro del módulo **module_md_lennard_jones.f90**.

II-D. Integración de las ecuaciones de movimiento

Teniendo en cuenta que las simulaciones de Dinámica Molecular, a diferencia de las simulaciones configuracionales de Monte Carlo, siguen una dinámica realista, por ello, las ecuaciones de movimiento deben cumplir con leyes de conservación de momento lineal, momento angular y la energía total del sistema (para sistemas con vínculos holónomos e independientes del tiempo, y cuyo potencial no depende de las velocidades entonces, el hamiltoniano es igual a la energía del sistema y si además el sistema es invariante ante traslaciones temporales el hamiltoniano, y en consecuencia la energía, se conserva). Las PBC inducen pérdida de simetría rotacional del sistema y en consecuencia el momento angular no puede conservarse en simulaciones que utilicen PBC. Los métodos de integración más utilizados son algoritmo de Verlet clásico, algoritmo de Leap-Frog, los cuales son inestables (estrictamente el algoritmo de Verlet clásico no conserva la energía del sistema sino que conserva la energía de un pseudo-hamiltoniano que en el límite de tiempos cortos es igual al hamiltoniano real), y el algoritmo de velocity-Verlet, que es estable. Este último es el que aplicamos en el presente trabajo. El código desarrollado para la integración de las ecuaciones de movimiento corresponde a la subrutina **velocity_verlet** dentro del módulo **module_md_lennard_jones.f90**. Cabe aclarar que se aplicó una variante del algoritmo que no requiere mantener guardado el vector fuerza en el tiempo anterior para computar las velocidades finales, debido a que se integra en dos pasos las ecuaciones de movimiento, utilizando un paso intermedio ficticio para aproximar las soluciones y disminuyendo es espacio en memoria de las simulaciones por cada llamada al integrador.

II-E. Estado transitorio y estacionario

Para llevar a cabo la equlibración del sistema (estado transitorio) se implementaron $t_{eq} = 1000$ pasos de integración, de los cuales cada $t_{scal} = 50$ pasos se escalaron las velocidades según el termostato descrito en la sección I-3 y todos los pasos se corrigieron las velocidades restándole la velocidad del centro de masa del sistema. En la figura 5 se puede observar la temperatura instantánea en el régimen transitorio y cómo se produce el escaleo de velocidades cada cierto tiempo para corregir la temperatura y llevarla a la temperatura de referencia. Una vez relajado el sistema, se implementaron $t_{run} = 1000$ pasos de integración, sin escale de velocidades y sin correcciones respecto a la velocidad del centro de masa

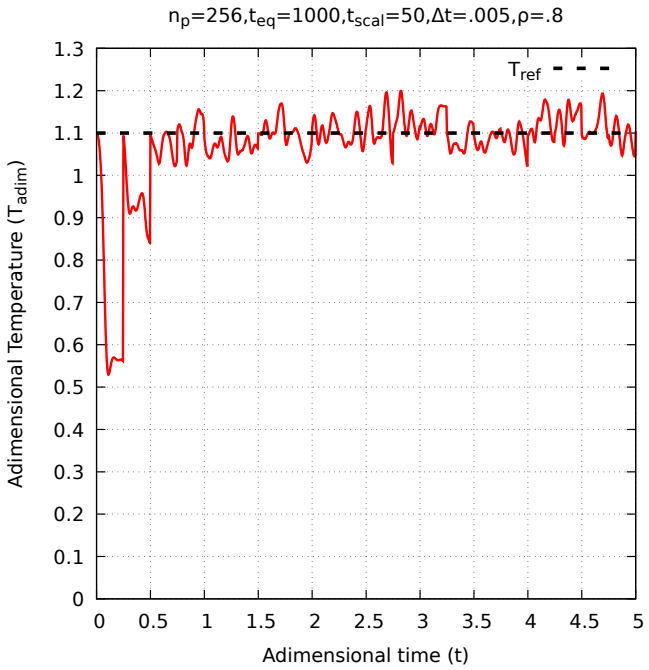


Fig. 5. Temperatura instantánea vs tiempo: régimen transitorio

del sistema, esto último es debido a que la velocidad total del centro de masa del sistema equilibrado produce correcciones a la temperatura que son despreciables respecto al orden de magnitud de las fluctuaciones de temperatura en el equilibrio, como se puede apreciar en la figura 7.

En la sección III se puede consultar un vídeo realizado con GNUPLOT para observar la dinámica de partículas, esta técnica nos permite observar qué tanto se mueven las partículas y corroborar si las PBC están aplicadas correctamente.

II-F. Distribución de velocidades

Luego de la termalización realizamos histogramas de las componentes de la velocidad y de la velocidad total, y se compararon los resultados con las soluciones exactas también calculadas según la temperatura de referencia y las magnitudes de la velocidad. Los histogramas de probabilidad se llevaron a cabo mediante GNUPLOT, a través de la función **histogram** utilizando directamente las componentes de la velocidad y de la velocidad total tomadas a cada tiempo de integración y, también, realizando un bineo explícito (en este caso particular se utilizaron 50 bins) desde FORTRAN. Los histogramas se compararon con las soluciones exáctas correspondientes a una distribución Gaussiana para las componentes de la velocidad y una distribución de Maxwell-Boltzmann para la velocidad total. Como podemos observar en la figura 6, las distribuciones están en acuerdo con las predicciones teóricas, lo cual nos permite corroborar que efectivamente a pesar de iniciar una distribución uniforme de velocidades el sistema relaja a la distribución adecuada según la teoría cinética de los gases. Las diferencias apreciables de la velocidad total respecto a la distribución de Maxwell-Boltzmann podrían deberse a la normalización de las distribuciones y/o a la ligera desviación del sistema de partículas respecto al gas ideal, para ver este último aspecto podría variarse la densidad de partículas y ver si la distribución se ajusta más a la exacta. El código principal para la obtención de estas distribuciones corresponde a **molecular_dynamic_lennard_jones_03.f90**.

II-G. Calculo de observables

II-G1. Energía potencial,cinética y total

Conseguida la equilibración del sistema se graficaron las energías cinética, potencial y total del sistema de partículas para cada tiempo de integración, corroborando la conservación de la energía total del sistema (ver 7). Además, se graficó la presión instantánea del sistema en función del tiempo y la temperatura instantánea del sistema.

La presión se calculó teniendo en cuenta el término correspondiente al gas ideal y el término correspondiente al teorema del virial de la siguiente manera:

$$P = \rho k_B T + \frac{1}{dV}\left\langle \sum_{\langle ij \rangle} \overrightarrow{f}(\overrightarrow{r}_{ij}) \cdot \overrightarrow{r}_{ij} \right\rangle \qquad (9)$$
$$d = 3; V \equiv L^3 = n_p/\rho$$

Entonces, las gráficas asociadas a cálculos de presión corresponden a la presión adimensional total del sistema, presión debida al gas ideal y presión osmótica. Ahora bien, adimensionalmente y trabajando un poco la ecuación tendremos una expresión para la presión adimensional, la cual fue implementada en las simulaciones

$$P^* = \rho\left[T^* + \frac{1}{3n_p}\left\langle \sum_{\langle ij \rangle}\left(f^*_{x,ij}r^*_{x,ij} + f^*_{y,ij}r^*_{y,ij} + f^*_{z,ij}r^*_{z,ij}\right)\right\rangle\right]$$
$$\langle ... \rangle = \left\langle \sum_{\langle ij \rangle}\left(\frac{(r^*_{x,ij})^2}{r^*_{ij}}f^*_{r,ij} + \frac{(r^*_{y,ij})^2}{r^*_{ij}}f^*_{r,ij} + \frac{(r^*_{k,ij})^2}{r^*_{ij}}f^*_{r,ij}\right)\right\rangle$$
$$\langle ... \rangle = \left\langle \sum_{\langle ij \rangle} r^*_{ij}f^*_{r,ij}\right\rangle$$
$$\Rightarrow P^* = \rho\left[T^* + \frac{1}{3n_p}\left\langle \sum_{\langle ij \rangle} r^*_{ij}f^*_{r,ij}\right\rangle\right] \qquad (10)$$

Notemos que $\langle ... \rangle$ se refiere al promedio temporal. Además, teniendo en cuenta que para la reducción del tiempo de CPU en las simulaciones se computan los cálculos relacionados a la fuerza de interacción hasta un radio de corte r_{cutoff} sin embargo, el potencial y la fuerza, más allá de este radio no es nula (lo cuál suponemos en los cálculos), entonces, se inducirá un error sistemático respecto a la solución real. En el caso particular de la presión la corrección a la misma se puede calcular de la siguiente manera:

$$\Delta P_{tail} = \frac{16}{3}\pi\rho^2\epsilon\sigma^3\left[\frac{2}{3}\left(\frac{\sigma}{r_{cutoff}}\right)^9 - \left(\frac{\sigma}{r_{cutoff}}\right)^3\right] \qquad (11)$$

y de forma adimensional tendremos la siguiente expresión;

$$\Delta P^*_{tail} = \frac{16}{3}\pi\rho^2\left(\frac{1}{r^*_{cutoff}}\right)^3\left[\frac{2}{3}\left(\frac{1}{r^*_{cutoff}}\right)^6 - 1\right] \qquad (12)$$

que, en el caso en que, $\rho = 0{,}8$ y $r^*_{cutoff} = 2{,}5$ tendremos un tail corrección de $\Delta P^*_{tail} = -0{,}6844$.

El código desarrollado para el cálculo de las energías, presión y temperatura corresponden a las funciones **u_lj_total** para la energía potencial total, **kinetic_ergy_total** para la energía cinética total, **temperature** para la temperatura y **pressure** para la presión dentro del módulo **module_md_lennard_jones.f90**. Y el código principal corresponde a **molecular_dynamic_lennard_jones_01.f90**.

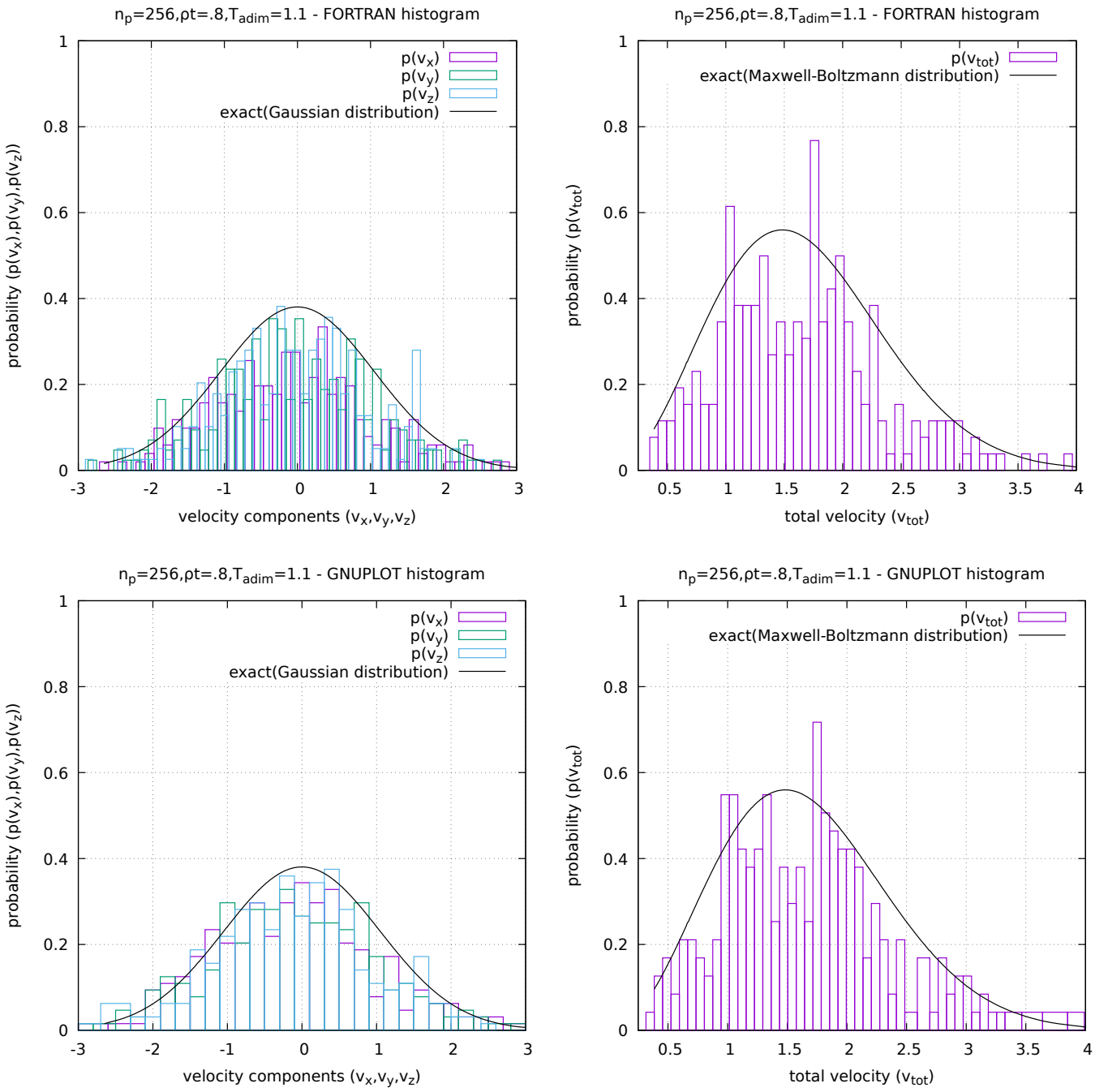


Fig. 6. Distribuciones de probabilidad para las velocidades

II-G2. Valores medios y fluctuaciones de observables

Luego de de termalización del sistema también se calcularon los valores medios (primer momento) y fluctuaciones (varianza) de las energías potencial, cinética y total; de la presión y de la temperatura. Los resultados obtenidos se pueden apreciar en la figura 15.

Para el cálculo de los valores promedios y fluctuaciones nos valemos de las propiedades de ergodicidad del sistema el cual nos permite calcular valores de expectación de una única corrida (experimento) suficientemente larga (promedio temporal) y no muchos experimentos con configuraciones completamente descorrelacionadas entre si (promedio en el ensamble). En la figura 15 se puede apreciar cómo al calcular promedios temporales e ir incluyendo mayor cantidad de términos las fluctuaciones disminuyen convergiendo a un valor medio el cual, por hipótesis ergódica, es igual al promedio del observable en el ensamble (microcanónico en este caso particular). En linea punteada, se graficaron los valores medios del último paso de integración y se aclaran los errores obtenidos. Como podemos observar las todos los observables fluctúan, sin embargo, contrario a nuestra primera intuición la energía total también fluctúa pero entorno a valores con orden de magnitud tres veces menores al del orden de magnitud de las fluctuaciones asociadas a las energías potencial y cinética, aunque la energía total se conserva, la existencia de fluctuaciones en esta podría deberse al paso temporal Δt elegido el cual, como veremos luego, influye significativamente en las fluctuaciones de la energía total. El código principal para la obtención de valores medios y fluctuaciones corresponde a **molecular_dynamic_lennard_jones_02.f90**.

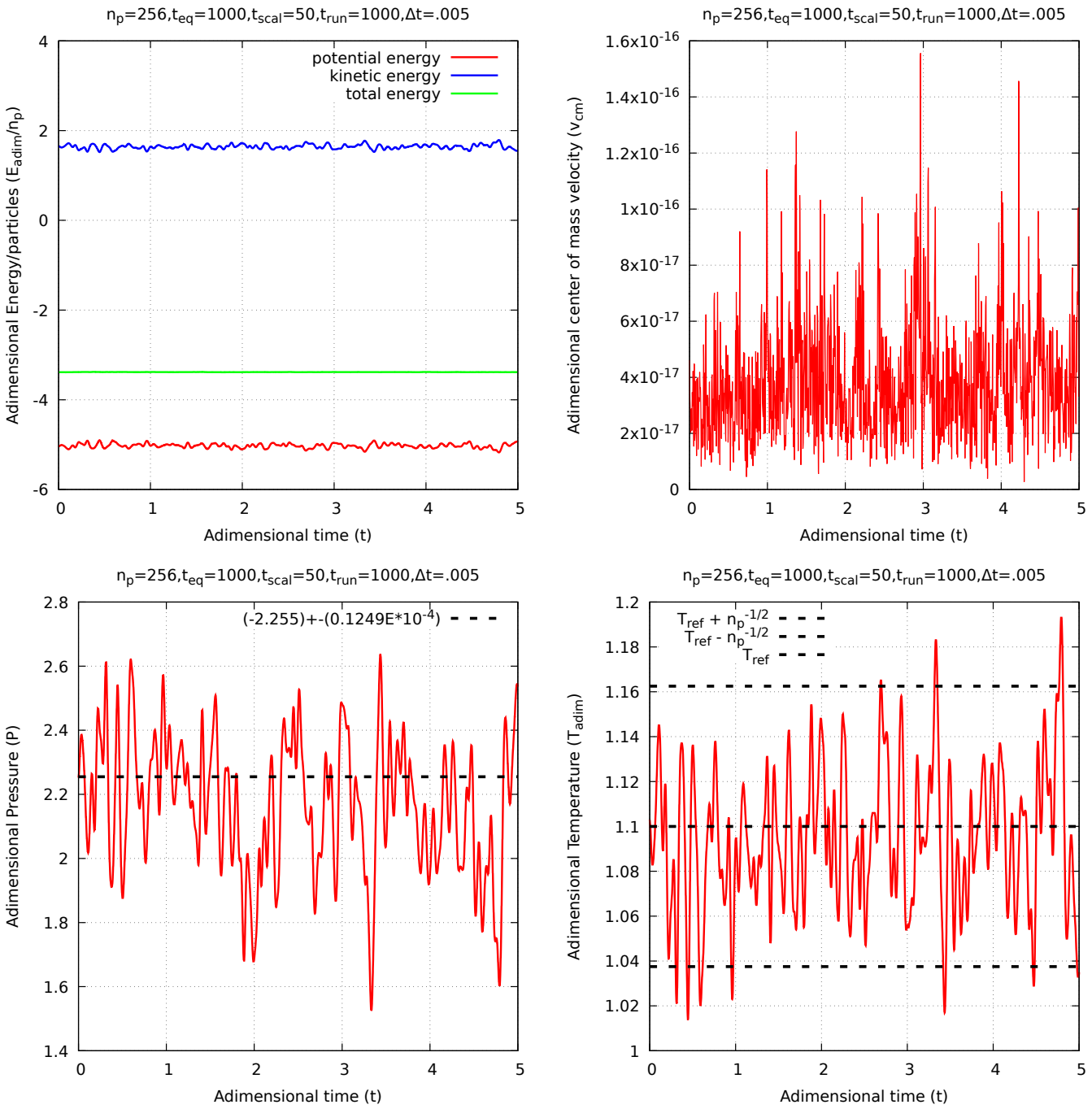


Fig. 7. Energías, velocidad del centro de masa, presión y temperaturas instantáneas vs tiempo

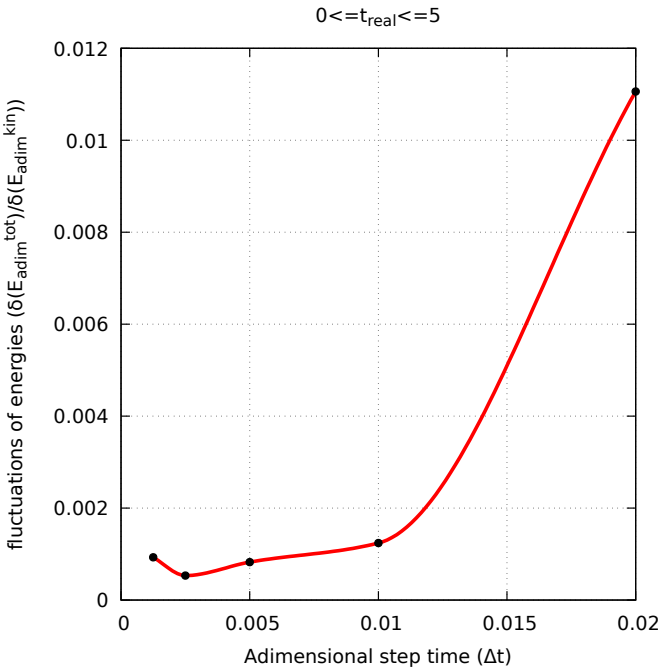
II-H. Dependencia de errores y tiempos de CPU

Se calcularon las razones entre fluctuaciones de energía total y fluctuaciones de energía cinética en función del paso de integración Δt, para poder comparar los resultados a cada paso de tiempo propuesto se definieron los pasos de equilibración, pasos de escaleo de velocidades y pasos de corrida de tal forma de mantener constante el tiempo real físico adimensional en cada etapa de la siguiente manera:

$$t_{eq} = \frac{t^*_{real-eq}}{\Delta t}; t_{scal} = \frac{t^*_{real-scal}}{\Delta t}; t_{run} = \frac{t^*_{real-run}}{\Delta t} \quad (13)$$
$$t^*_{real-eq} \equiv t^*_{real-run} = 5; t^*_{real-scal} = 0{,}25$$

El código principal para la obtención de las razones entre fluctuaciones de energía total y fluctuaciones de energía cinética en función del paso de integración corresponde a **molecular_dynamic_lennard_jones_04.f90**.

Por otro lado se calcularon las razones entre fluctuaciones de energía total y fluctuaciones de energía cinética en función del radio de corte r_{cutoff} de las interacciones entre pares de partículas, para ello se tomo un paso de integración pequeño $\Delta t = 0{,}005$ de tal manera que las fluctuaciones no dependan apreciablemente de este parámetro sino que su comportamiento esté determinado esencialmente por el radio de corte. Los resultados obtenidos se muestran en la figura 8, donde se aprecia aproximadamente una dependencia lineal de a tramos de la razón de fluctuaciones con el paso de integración una dependencia exponencial decreciente de la razón de fluctuaciones con el radio de corte para las interacciones. Estos resultados nos estarían mostrando que para mejorar la precisión de las simulaciones de DM, es decir, reducir las fluctuaciones de la energía total del sistema deberemos utilizar pasos de integración suficientemente pequeños



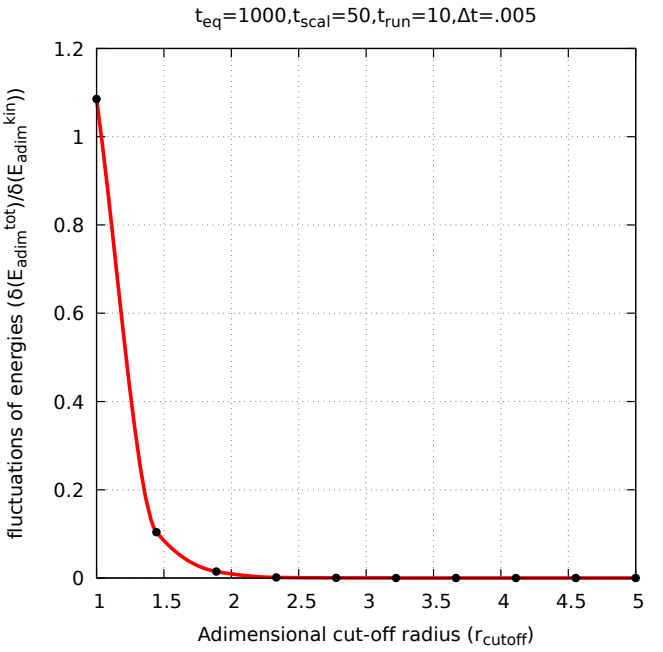


Fig. 8. Razón entre fluctuaciones de energía vs pasos temporales

como para que las fluctuaciones dependan fuertemente del radio de corte y además, considerar un radio de corte mayor a la mitad de la dimensión del cubo periódico de estudio ($\Delta t \ll 1 \wedge r_{cutoff} < L/2$). El código principal para la obtención de las razones entre fluctuaciones de energía total y fluctuaciones de energía cinética en función del radio de corte corresponde a **molecular_dynamic_lennard_jones_05.f90**.

Finalmente, se realizó un estudio de escalamiento del sistema, es decir, se simularon distintos tamaños de sistema aumentando el número de partículas en la forma $n_p = 4i^3; i \in N \wedge 3 \leq i \leq 8$. Para definir el radio de corte de interacciones se tomó como criterio que el mismo sea igual a $r_{cutoff} = 0{,}3L$ donde $L = \sqrt[3]{\frac{n_p}{\rho}}$ es el lado del cubo de estudio el cual se repite según las PBC. En la gráfica 9 se observan los resultados obtenidos, en donde ser llevó a cabo un fiteo ajustando una ecuación polinómica a través de dos

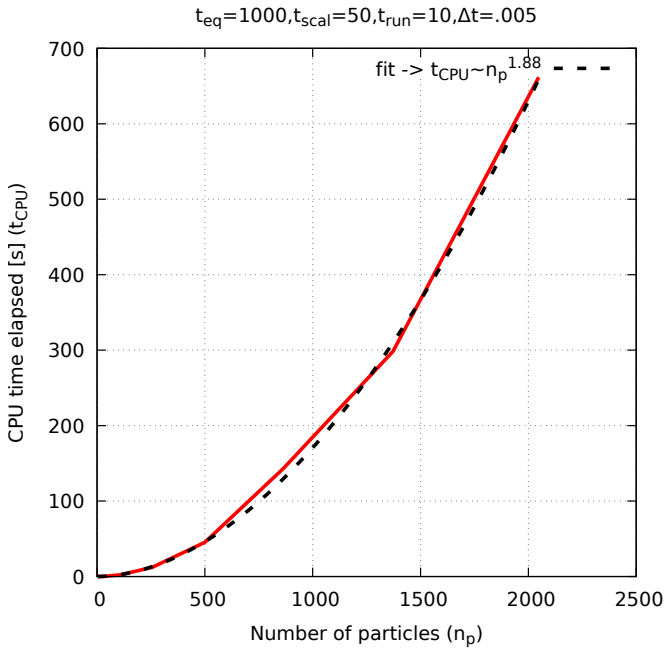


Fig. 9. Escaleo del sistema vs tiempo de CPU

parámetros de ajuste, y en donde podemos observar que el coste computacional crece cuadráticamente con el escaleo del sistema. El código principal para la medida de *performance* corresponde a **molecular_dynamic_lennard_jones_06.f90**. Para lograr un escalamiento lineal del coste computacional se emplean mejoras algorítmicas como las llamadas **linked list**, estas listas son las comunmente utilizadas en cualquier simulación de dinámica molecular.

II-I. Ensamble canónico (Problema 2)

Para realizar las simulaciones en el ensamble canónico se utilizó el termostato descrito en la sección I-3 en todos los pasos de integración de las ecuaciones de movimiento, de tal forma de mantener constante la temperatura. Además, los parámetro de simulación fueron modificados respecto a lo utilizado hasta el momento con $n_p = 500$ partículas, temperatura adimensional $T_{adim} = 1{,}1$, densidades de $\rho = 0{,}8$ y $\rho = 1{,}2$ y, para asegurarnos de que el sistema equilibre totalmente se aumentaron los pasos de termalización a $t_{eq} = 2000$.

II-I1. Función de distribución radial

La función de distribución radial se define de las siguiente manera

$$g(r) = \frac{(\langle n_p \rangle \in [r; r+dr])_{sistema}}{(\langle n_p \rangle \in [r; r+dr])_{gas\ ideal}} = \frac{n_{histogram}(r)}{n_{gas\ ideal}(r)} \qquad (14)$$
$$n_{gas\ ideal}(r) = \frac{4\pi\rho}{3}[(r+dr)^3 - r^3]$$

entonces la función de correlación de pares nos muestra cómo varía la densidad de partículas en función de la distancia interatómica medida respecto de alguna posición de referencia. La normalización respecto al gas ideal es debido a que, para este los pares de partículas están totalmente descorrelacionados y la correlación aumenta a mediad que el sistema se solidifica, siendo máximo para un sólido e intermedio para un fluido. Cabe mencionar que para fabricar el histograma de distribución de partículas se utilizaron $n_{bins} = 1000$ bins. En la figura 10 podemos observar el resultado obtenido para la función de correlación espacial. En la misma, vemos que para

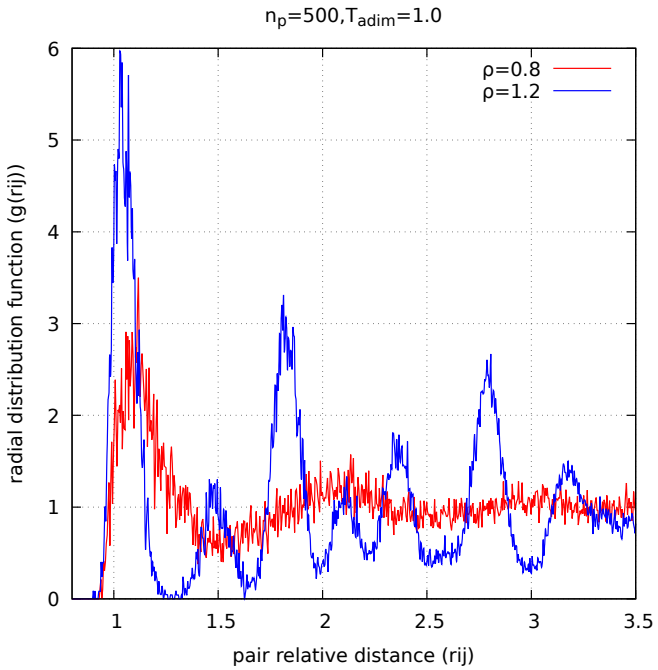


Fig. 10. función de correlación espacial vs distancia iteratómica

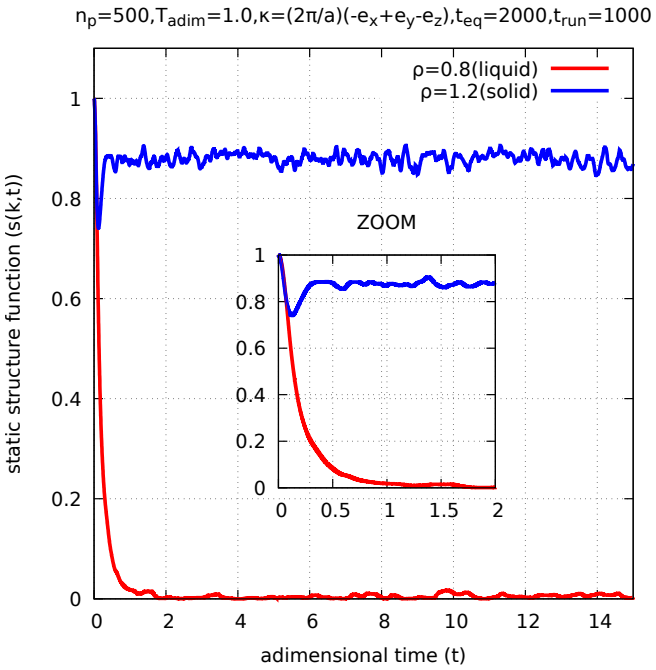


Fig. 11. función de estructura estática vs tiempo adimensional

el caso de mayor densidad se obtiene una curva con más picos localizados, lo que nos muestra que se trata de una estructura bastante sólida, es de esperar que en en una región donde el material es completamente sólido los picos se encuentren en las posiciones de red de Bravais correspondiente, simulando una función tipo peine de Dirac. Por otro lado, para el caso de menor densidad la función de correlación es más suave evidenciando una mayor descorrelación espacial, y un mayor grado de amorfismo del sistema, y podríamos asociarlo a una fase líquida.

El código principal para la obtención de la función de correlación espacial corresponde a **md_lj_canonical_ensamble_01.f90** el cual utiliza la subrutina **radial_ditribution_function** dentro del módulo **module_md_lennard_jones.f90**.

II-I2. Factor de estructura cristalino

La definición del factor de estructura estático es la siguiente:

$$S(\vec{k},t) = \frac{1}{N^2}\left|\sum_j \exp\left(i\vec{k}\cdot\vec{r}_j(t)\right)\right|^2 \qquad (15)$$
$$\vec{k} = \frac{2\pi}{a}(-\hat{e}_x+\hat{e}_y-\hat{e}_z); a = \sqrt[3]{\frac{n_{FCC}}{\rho}}$$

donde $\vec{k}$ es un vector de onda (estrictamente un momento cristalino) perteneciente a la red recíproca, a es el parámetro de red de la red cristalina y $n_{FCC} = 4$ es la cantidad de átomos por celda unidad.

En la figura 11 se puede observar los resultados obtenidos para el factor de estructura estático, el cual es un parámetro de orden cristalino del sistema. Como podemos notar, para el caso del sistema líquido el factor de estructura se anula evidenciando un completo desorden estático del sistema (el decaimiento está asociado directamente con la dimensión del sistema con una pendiente proporcional $1/n_p$) y para el caso del sistema sólido el factor de estructura se estabiliza en un valor menor a la unidad, lo cual es esperable.

El código principal para la obtención del parámetro de orden cristalino corresponde a **md_lj_canonical_ensamble_02.f90**, el cual utiliza la función **static_structure_factor** dentro del módulo **module_md_lennard_jones.f90**.

II-I3. Desplazamiento cuadrático medio y coeficiente de difusión

La definición de desplazamiento cuadrático medio (MSD) y su relación con el coeficiente de difusión es la siguiente:

$$\left\langle\left|\vec{r}(t)-\vec{r}(0)\right|^2\right\rangle = \frac{1}{n_p}\sum_{i=1}^{n_p}\left[\Delta\vec{r}_i(t)\right]^2 \cong 6Dt \qquad (16)$$

donde D es el coeficiente de difusión. En la figura 12 se pueden apreciar los resultados obtenidos, donde se puede notar que para el caso del fluido ($\rho = 0{,}8$) la dependencia del MSD con el tiempo de correlación está en acuerdo con la teoría (proceso difusivo, comportamiento lineal) y para el sistema sólido tenemos un comportamiento un poco extraño, evidenciando, posiblemente, a una no difusión de partículas en el medio (comportamiento balístico). Para las simulaciones se consideraron $\tau_{corr}^{max} = 5000$ pasos máximos de correlación, almacenando $nmax_{\tau_{corr}} = 500$ valores distintos para el computo del MSD.

El código principal para la obtención del desplazamiento cuadrático medio corresponde a **md_lj_canonical_ensamble_03.f90**

II-J. Transición sólido-líquido (Problema 3)

En este caso se implementó una simulación de DM en el ensamble canónico escaleando las velocidades (según el termostáto descrito en la sección I-3) en cada paso de integración de la ecuaciones de movimiento. Los parámetros elegidos para este caso fueron $n_p = 256$ partículas, un paso temporal de integración de $\Delta t = 0{,}005$, $t_{eq} = 1000$ pasos de termalización, $t_{run} = 2000$ pasos de corrida en régimen estacionario, un radio de corte de interacciones de $r_{cutoff} = 2{,}5$, temperatura adimensional de referencia de

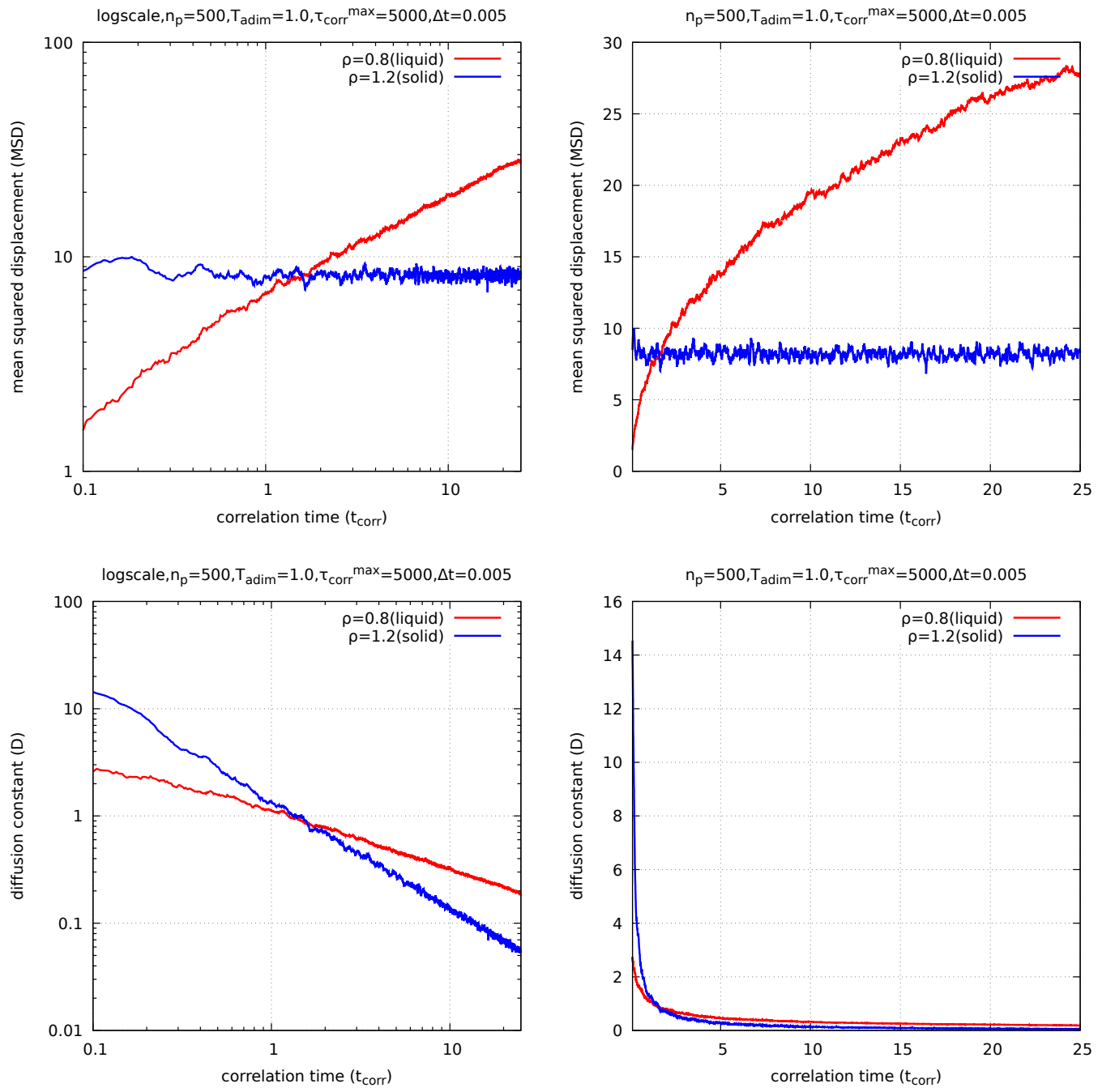


Fig. 12. MSD y coeficiente de difusión vs tiempo de correlación

$T_{adim} = 1$. Además, para estudiar la transición de fase se varió la densidad en el rango crítico entre $\rho_{min} = 0{,}8$ y $\rho_{max} = 1{,}2$. Los resultados obtenidos se muestran en la figura 13 donde se observan la presión total adimensional (P) (incluyendo presión osmótica y presión del gas ideal), factor de estructura estático $(S(k))$, coeficiente de difusión (D) y desplazamiento cuadrático medio (MSD), todas estas propiedades en función de la densidad adimensional del sistema. Se pueden observar cambios de curvatura en cada una de las propiedades en los puntos críticos de densidades $\rho_1 = 0{,}8889$ y $\rho_2 = 0{,}9333$, lo cual evidencia una transición de fase de sólido-líquido, para valores de densidad mayores a rho_2 tendremos fase sólida y para valores de densidad menores a rho_1 tendremos fase líquida. Para apreciar aún más estas curvaturas, se realizaron gráficas de estas funciones respuesta en la zona de interés (ver figura 14). Quizás, para mayor claridad y caracterización de la transición se podrían graficar isotérmas parametrizando las funciones respuesta en función de la temperatura, además, se podría estudiar el comportamiento de la dimensión del sistema (número de partículas) en el resultado.

El código principal para el estudio de la transición de fase corresponde a **molecular_dynamic_lennard_jones_06.f90**

III. Códigos

Repositorio de GitHub

- https://github.com/mendzmartin/fiscomp2022.git

Repositorio GitHub del problema

- **Problema 1**
- https://github.com/mendzmartin/fiscomp2022/tree/main/lab05/prob01
- **Problema 2**

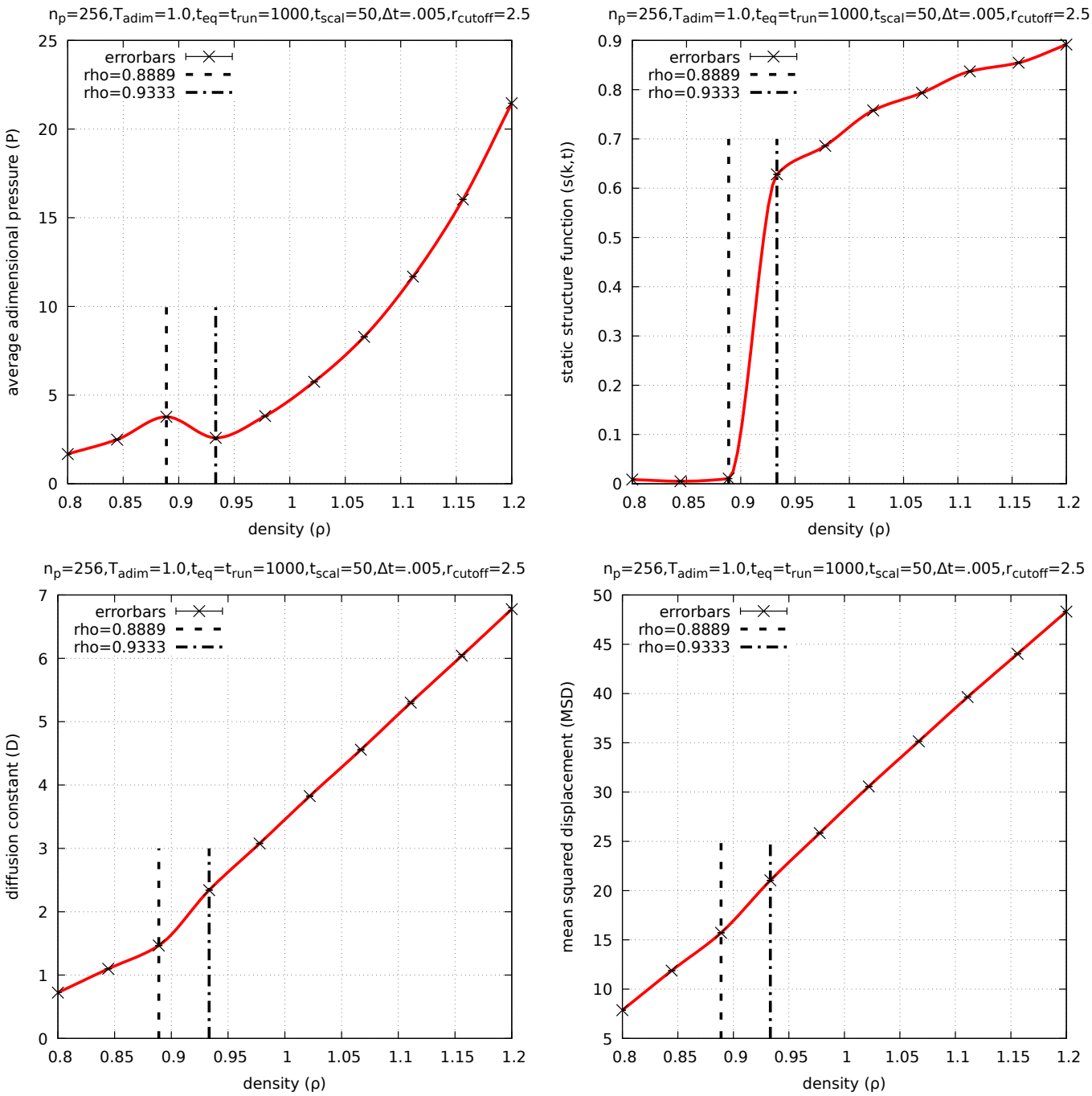


Fig. 13. Presión, factor de estructura estático, coeficiente de difusión y MSD vs densidad

- https://github.com/mendzmartin/fiscomp2022/tree/main/lab05/prob02
- **Problema 3**
- https://github.com/mendzmartin/fiscomp2022/tree/main/lab05/prob03

Códigos principales y Makefile

- **Problema 1**
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/code/molecular_dynamic_lennard_jones_01.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/code/molecular_dynamic_lennard_jones_02.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/code/molecular_dynamic_lennard_jones_03.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/code/molecular_dynamic_lennard_jones_04.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/code/molecular_dynamic_lennard_jones_05.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/code/molecular_dynamic_lennard_jones_06.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/code/Makefile
- **Problema 2**
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob02/code/md_lj_canonical_ensamble_01.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob02/code/md_lj_canonical_ensamble_02.f90

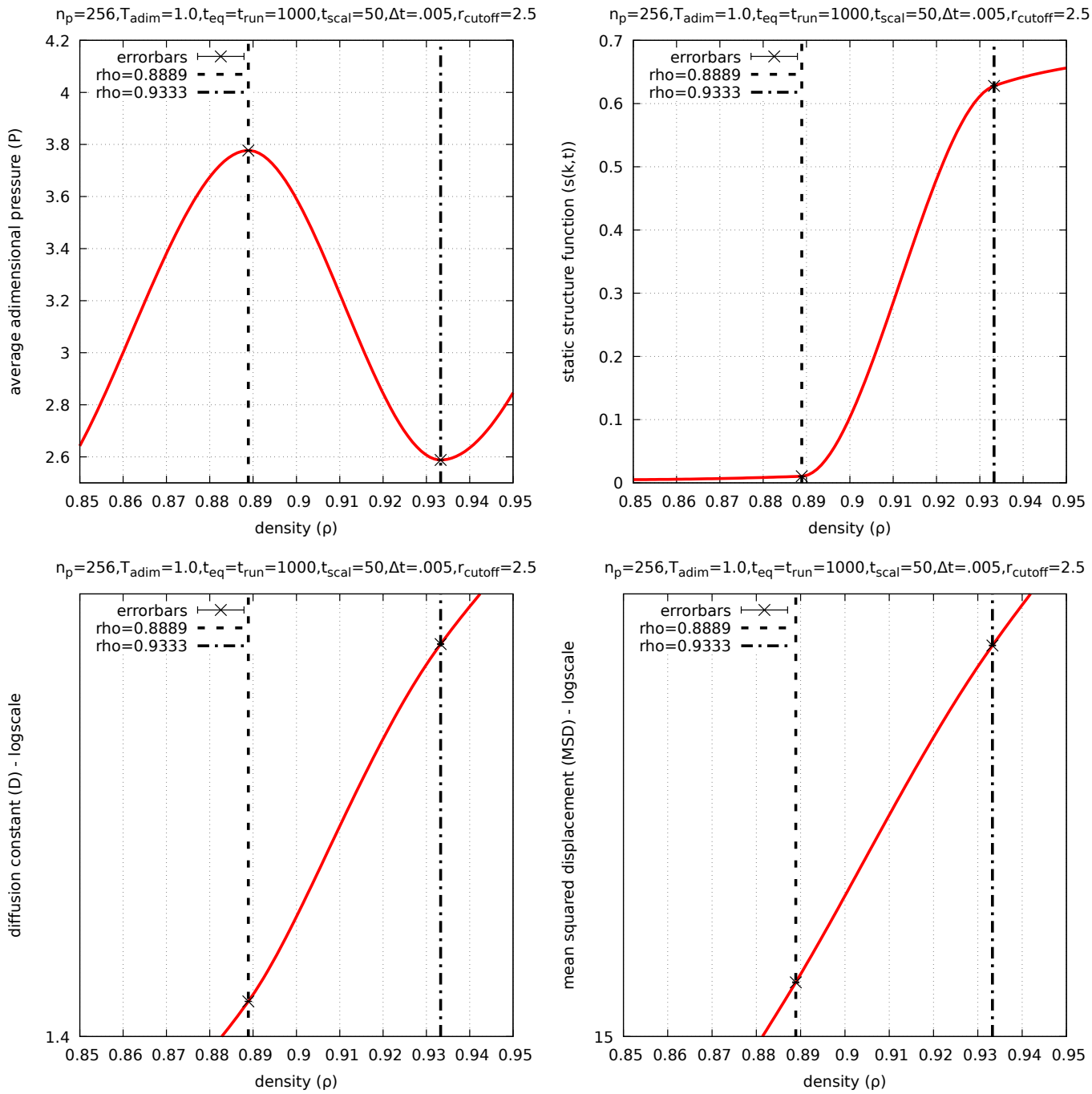
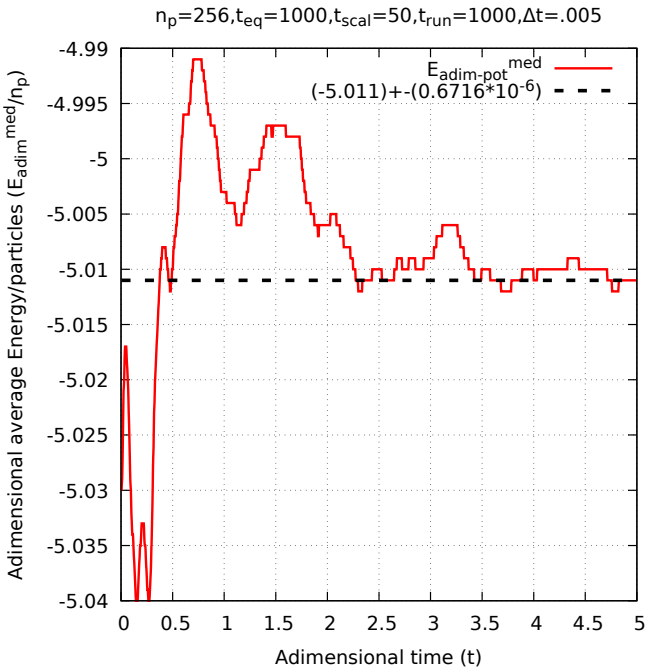
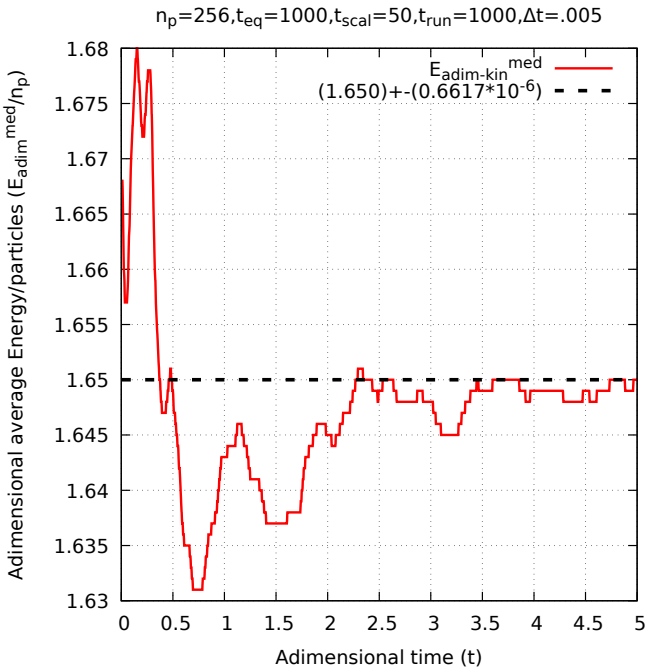
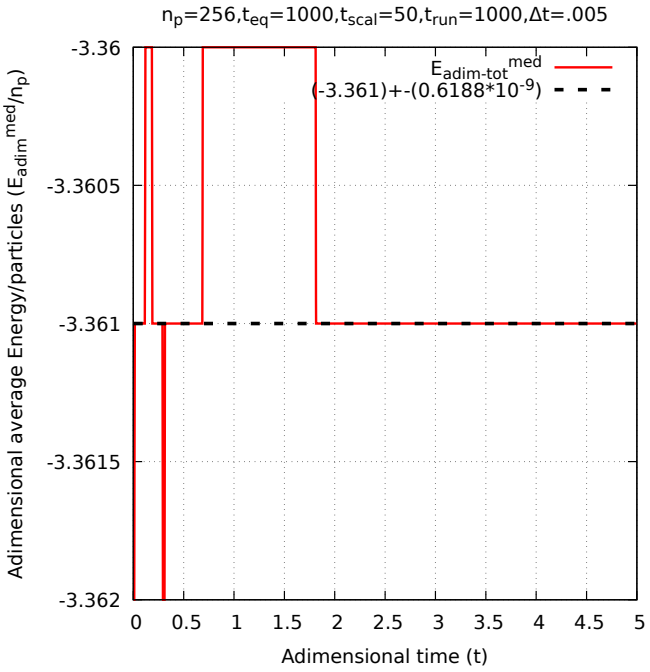


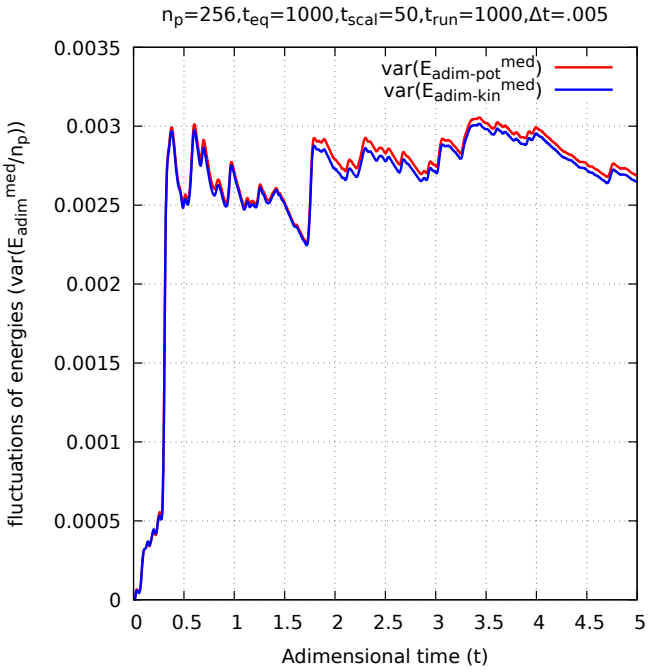
Fig. 14. Presión, factor de estructura estático, coeficiente de difusión y MSD vs densidad - ZOOM

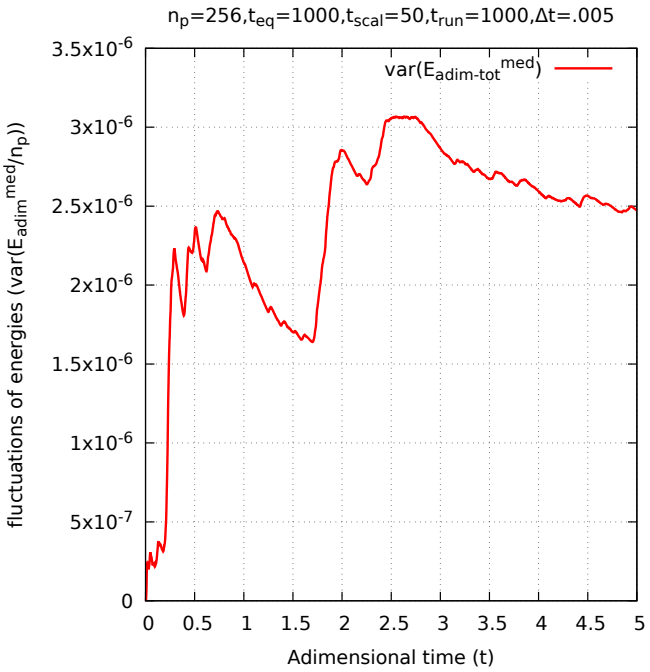
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob02/code/md_lj_canonical_ensamble_03.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob02/code/Makefile
- **Problema 3**
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob03/code/md_lj_order_transition_01.f90
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob03/code/Makefile
- **Video de dinámica molecular**
- https://github.com/mendzmartin/fiscomp2022/blob/main/lab05/prob01/plots/movie.mp4

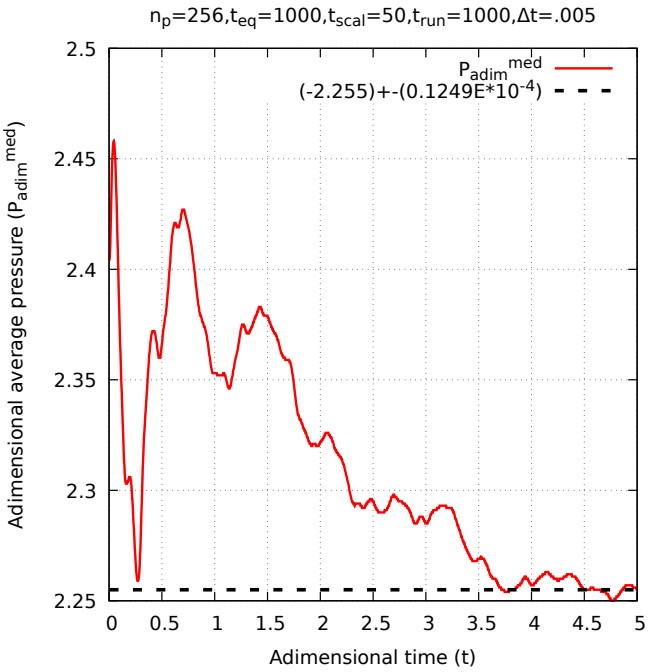












np=256,teq=1000,tscal=50,trun=1000,Δt=.005

np=256,teq=1000,tscal=50,trun=1000,Δt=.005

Laboratorio 05 - Problema 01 - Códigos

molecular_dynamic_lennard_jones_01.f90

```
! make clean && make molecular_dynamic_lennard_jones_01.o && ./molecular_dynamic_lennard_jones_01.o
program molecular_dynamic_lennard_jones_01
    use module_precision;use module_md_lennard_jones
    implicit none
    integer(sp), parameter   :: n_p=256_sp                              ! cantidad de partículasa
    real(dp),    parameter   :: delta_time=0.005_dp                     ! paso temporal
    integer(sp), parameter   :: time_eq=1000_sp,time_scal=50_sp,&       ! pasos de equilibración y de escaleo de veloc.
                                time_run=1000_sp                        ! pasos de evolucion en el estado estacionario
    real(dp),    parameter   :: T_adim_ref=1.1_dp                       ! temperatura de referencia adimensional
    real(dp),    parameter   :: density=0.8_dp                          ! densidad (particulas/volumen)
    real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp              ! radio de corte de interacciones y masa
    real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)     ! componentes de las posiciones/particula
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)        ! componentes de la fuerza/particula
    integer(sp)              :: i,istat,index                           ! loop index
    real(dp)                 :: U_adim,Ec_adim,time,press,v_mc,T_adim   ! observables
    real(dp)                 :: vx_mc,vy_mc,vz_mc                       ! componentes de la velocidad del centro de masas
    real(dp)                 :: time_end,time_start                     ! tiempos de CPU
    logical                  :: movie_switch,fcc_init_switch,&
                                Tadim_trans_switch,energies_switch

    movie_switch       =.false. ! escribir pelicula con partículas en la caja
    fcc_init_switch    =.false. ! escribir estructura fcc inicial
    Tadim_trans_switch =.false. ! escribir tempereratura en el estado transitorio
    energies_switch    =.true. ! escribir energías en el estado estacionario

    call cpu_time(time_start)
    22 format(5(E12.4,x),E12.4);23 format(5(A12,x),A12)

    allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
    x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp

    ! generamos configuración inicial (FCC structure)
    call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)

    ! ESCRIBIMOS DATOS
    if (fcc_init_switch.eqv..true.) then
        open(90,file='../results/fcc.dat',status='replace',action='write',iostat=istat)
        if (istat/=0) write(*,*) 'ERROR! istat(90file) = ',istat
        write(90,"(2(A12,x),A12)") 'rx_fcc','ry_fcc','rz_fcc'
        do i=1,n_p;write(90,"(2(E12.4,x),E12.4)") x_vector(i),y_vector(i),z_vector(i);end do;close(90)
    else if (movie_switch.eqv..true.) then
        index=10;call create_movie(index,x_vector,y_vector,z_vector,n_p)
    end if

    allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
    vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
    call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
    vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)

    ! computamos fuerzas en el tiempo inicial
    allocate(force_x(n_p),force_y(n_p),force_z(n_p))
    force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
    call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)

    ! ESCRIBIMOS DATOS
    if (fcc_init_switch.eqv..true.) then
        open(90,file='../results/init_force.dat',status='replace',action='write',iostat=istat)
        if (istat/=0) write(*,*) 'ERROR! istat(90file) = ',istat
        write(90,"(2(A12,x),A12)") 'fx','fy','fz'
        do i=1,n_p;write(90,"(2(E12.4,x),E12.4)") force_x(i),force_y(i),force_z(i);end do;close(90)
    else if (Tadim_trans_switch.eqv..true.) then
        open(90,file='../results/Tadim_transitorio.dat',status='replace',action='write',iostat=istat)
        if (istat/=0) write(*,*) 'ERROR! istat(90file) = ',istat
        write(90,"(A12,x,A12)") 'time','Tadim'
    end if

    ! TRANSITORIO
    do i=1,time_eq
        write(*,*) i
        if (mod(i,time_scal)==0_sp) call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
        call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        ! velocity center of mass to zero
        vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
        vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
        vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
        T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
        if (Tadim_trans_switch.eqv..true.) write(90,"(E12.4,x,E12.4)") real(i,dp)*delta_time,T_adim
    end do
    if (Tadim_trans_switch.eqv..true.) close(90)
```

```
82
83      ! ESTACIONARIO
84      time=0._dp
85
86      ! ESCRIBIMOS DATOS
87      if (energies_switch.eqv..true.) then
88          open(12,file='../results/result_03.dat',status='replace',action='write',iostat=istat)
89          write(*,*) 'istat(12file) = ',istat;write(12,23) 'time','pot_ergy','kin_ergy','v_mc','press','T_adim'
90      end if
91
92      U_adim=u_lj_total(n_p,x_vector,y_vector,z_vector,r_cutoff,density)
93      Ec_adim=kinetic_ergy_total(n_p,vx_vector,vy_vector,vz_vector,mass)
94      press=pressure(n_p,density,mass,r_cutoff,x_vector,y_vector,z_vector,&
95      vx_vector,vy_vector,vz_vector)
96      T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
97      v_mc=sqrt(vx_mc*vx_mc+vy_mc*vy_mc+vz_mc*vz_mc)
98
99      ! ESCRIBIMOS DATOS
100     if (energies_switch.eqv..true.) then
101         write(12,22) time,U_adim*(1._dp/real(n_p,dp)),Ec_adim*(1._dp/real(n_p,dp)),v_mc,press,T_adim
102     end if
103
104
105     do i=1,time_run
106         write(*,*) time_eq+i
107         call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
108         vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
109         ! velocity center of mass to zero
110         vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
111         vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
112         vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
113         U_adim=u_lj_total(n_p,x_vector,y_vector,z_vector,r_cutoff,density)
114         Ec_adim=kinetic_ergy_total(n_p,vx_vector,vy_vector,vz_vector,mass)
115         press=pressure(n_p,density,mass,r_cutoff,x_vector,y_vector,z_vector,&
116         vx_vector,vy_vector,vz_vector)
117         T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
118         v_mc=sqrt(vx_mc*vx_mc+vy_mc*vy_mc+vz_mc*vz_mc)
119         time=real(i,dp)*delta_time
120
121         ! ESCRIBIMOS DATOS
122         if (energies_switch.eqv..true.) then
123             write(12,22) time,U_adim*(1._dp/real(n_p,dp)),Ec_adim*(1._dp/real(n_p,dp)),v_mc,press,T_adim
124         else if ((movie_switch.eqv..true.).and.(mod(i,100)==0_sp)) then
125             index=index+1;call create_movie(index,x_vector,y_vector,z_vector,n_p)
126         end if
127     end do
128     if (energies_switch.eqv..true.) close(12)
129
130     deallocate(x_vector,y_vector,z_vector)
131     deallocate(vx_vector,vy_vector,vz_vector)
132     deallocate(force_x,force_y,force_z)
133
134     call cpu_time(time_end)
135     write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
136 end program molecular_dynamic_lennard_jones_01
137
138 ! subrutina para crear película de partículas en la caja
139 subroutine create_movie(index,x_vector,y_vector,z_vector,n_p)
140     use module_precision
141     implicit none
142     integer(sp), intent(in) :: index,n_p
143     real(dp),    intent(in) :: x_vector(n_p),y_vector(n_p),z_vector(n_p)
144     character(len=24)       :: file_name
145     character(len=2)        :: index_str
146     integer(sp)             :: i,istat
147     50 format(2(A12,x),A12);51 format(2(E12.4,x),E12.4)
148     write (index_str,'(I2)') index
149     file_name='../results/picture'//trim(index_str)//'.dat'
150     open(52,file=file_name,status='replace',action='write',iostat=istat)
151     if (istat/=0) write(*,*) 'ERROR! istat(52file) = ',istat
152     write(52,50) 'rx_fcc','ry_fcc','rz_fcc'
153     do i=1,n_p;write(52,51) x_vector(i),y_vector(i),z_vector(i);end do;close(52)
154 end subroutine create_movie
```

molecular_dynamic_lennard_jones_02.f90

```
1   ! make clean && make molecular_dynamic_lennard_jones_02.o && ./molecular_dynamic_lennard_jones_02.o
2   program molecular_dynamic_lennard_jones_02
3       use module_precision;use module_md_lennard_jones
4       implicit none
5       integer(sp), parameter   :: n_p=256_sp                          ! cantidad de partículasa
6       real(dp),    parameter   :: delta_time=0.005_dp                 ! paso temporal
7       integer(sp), parameter   :: time_eq=1000_sp,time_scal=50_sp,&   ! pasos de equilibración y de escaleo de veloc.
8                                   time_run=1000_sp                    ! pasos de evolucion en el estado estacionario
9       real(dp),    parameter   :: T_adim_ref=1.1_dp                   ! temperatura de referencia adimensional
10      real(dp),    parameter   :: density=0.8_dp                      ! densidad (particulas/volumen)
```

```
11      real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp             ! radio de corte de interacciones y masa
12      real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)    ! componentes de las posiciones/particula
13      real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
14      real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)       ! componentes de la fuerza/particula
15      integer(sp)              :: i,j,istat,index                        ! loop index
16      real(dp)                 :: U_adim,U_med,var_U,err_U
17      real(dp)                 :: Ec_adim,Ec_med,var_Ec,err_Ec
18      real(dp)                 :: Etot_adim,Etot_med,var_Etot,err_Etot
19      real(dp)                 :: press,press_med,var_press,err_press
20      real(dp)                 :: T_adim,T_med,var_T,err_T
21      real(dp)                 :: s1_U,s2_U
22      real(dp)                 :: s1_Ec,s2_Ec
23      real(dp)                 :: s1_Etot,s2_Etot
24      real(dp)                 :: s1_press,s2_press
25      real(dp)                 :: s1_T,s2_T
26      real(dp)                 :: vx_mc,vy_mc,vz_mc                      ! componentes de la velocidad del centro de masas
27      real(dp)                 :: time_end,time_start                    ! tiempos de CPU
28
29      call cpu_time(time_start)
30
31      allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
32      x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp
33
34      ! generamos configuración inicial (FCC structure)
35      call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)
36
37      allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
38      vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
39      call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
40      vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)
41
42      ! computamos fuerzas en el tiempo inicial
43      allocate(force_x(n_p),force_y(n_p),force_z(n_p))
44      force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
45      call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)
46
47
48      index=0
49      ! TRANSITORIO
50      do i=1,time_eq
51          index=index+1
52          write(*,*) 'paso temporal =',index,' de',time_eq+time_run
53          if (mod(i,time_scal)==0_sp) call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
54          call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
55          vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
56          ! velocity center of mass to zero
57          vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
58          vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
59          vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
60          T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
61      end do
62
63      write(*,*) 'termino el transitorio'
64
65      ! ESTACIONARIO
66      open(10,file='../results/fluctuations.dat',status='replace',action='write',iostat=istat)
67      if (istat/=0) write(*,*) 'ERROR! istat(10file) = ',istat
68      11 format(9(E12.4,x),E12.4);12 format(9(A12,x),A12)
69      write(10,12) 'U_med','err_U','Ec_med','err_Ec','Etot_med','err_Etot','press_med','err_press','T_med','err_T'
70
71      open(20,file='../results/fluctuations_vs_time.dat',status='replace',action='write',iostat=istat)
72      if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
73      21 format(10(E12.4,x),E12.4);22 format(10(A12,x),A12)
74      write(20,22) 'time','U_med','var_U','Ec_med','var_Ec','Etot_med','var_Etot','press_med','var_press','T_med','var_T'
75
76      U_med=0._dp
77      Ec_med=0._dp
78      Etot_med=0._dp
79      press_med=0._dp
80      T_med=0._dp
81
82      s1_U=0._dp;s2_U=0._dp
83      s1_Ec=0._dp;s2_Ec=0._dp
84      s1_Etot=0._dp;s2_Etot=0._dp
85      s1_press=0._dp;s2_press=0._dp
86      s1_T=0._dp;s2_T=0._dp
87
88      do i=1,time_run
89          index=index+1
90          write(*,*) 'paso temporal =',index,' de',time_eq+time_run
91          call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
92          vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
93
94          U_adim=u_lj_total(n_p,x_vector,y_vector,z_vector,r_cutoff,density)
95          Ec_adim=kinetic_ergy_total(n_p,vx_vector,vy_vector,vz_vector,mass)
96          Etot_adim=(U_adim+Ec_adim)*(1._dp/real(n_p,dp))
97          press=pressure(n_p,density,mass,r_cutoff,x_vector,y_vector,z_vector,&
```

```
98          vx_vector,vy_vector,vz_vector)
99          T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
100
101         ! computamos 1er y 2do momento
102         s1_U=s1_U+U_adim*(1._dp/real(n_p,dp));s2_U=s2_U+U_adim*U_adim*(1._dp/real(n_p*n_p,dp))
103         s1_Ec=s1_Ec+Ec_adim*(1._dp/real(n_p,dp));s2_Ec=s2_Ec+Ec_adim*Ec_adim*(1._dp/real(n_p*n_p,dp))
104         s1_Etot=s1_Etot+Etot_adim;s2_Etot=s2_Etot+Etot_adim*Etot_adim
105         s1_press=s1_press+press;s2_press=s2_press+press*press
106         s1_T=s1_T+T_adim;s2_T=s2_T+T_adim*T_adim
107
108         ! computamos valores medios (mejor a mayor paso evolucionado)
109         U_med=s1_U*(1._dp/real(i,dp))
110         Ec_med=s1_Ec*(1._dp/real(i,dp))
111         Etot_med=s1_Etot*(1._dp/real(i,dp))
112         press_med=s1_press*(1._dp/real(i,dp))
113         T_med=s1_T*(1._dp/real(i,dp))
114
115         ! computamos varianzas (mejor a mayor paso evolucionado)
116         var_U=(real(i,dp)*s2_U-s1_U*s1_U)*(1._dp/real(i*i,dp))
117         var_Ec=(real(i,dp)*s2_Ec-s1_Ec*s1_Ec)*(1._dp/real(i*i,dp))
118         var_Etot=(real(i,dp)*s2_Etot-s1_Etot*s1_Etot)*(1._dp/real(i*i,dp))
119         var_press=(real(i,dp)*s2_press-s1_press*s1_press)*(1._dp/real(i*i,dp))
120         var_T=(real(i,dp)*s2_T-s1_T*s1_T)*(1._dp/real(i*i,dp))
121
122         write(20,21) delta_time*real(i,dp),U_med,var_U,Ec_med,var_Ec,Etot_med,var_Etot,press_med,var_press,T_med,var_T
123     end do
124     close(20)
125
126     write(*,*) 'termino el estacionario'
127
128     ! computamos errores en el último paso
129     err_U=(var_U*0.25_dp)*(1._dp/real(time_eq-1,dp))
130     err_Ec=(var_Ec*0.25_dp)*(1._dp/real(time_eq-1,dp))
131     err_Etot=(var_Etot*0.25_dp)*(1._dp/real(time_eq-1,dp))
132     err_press=(var_press*0.25_dp)*(1._dp/real(time_eq-1,dp))
133     err_T=(var_T*0.25_dp)*(1._dp/real(time_eq-1,dp))
134
135     write(10,11) U_med,err_U,Ec_med,err_Ec,Etot_med,err_Etot,press_med,err_press,T_med,err_T
136     close(10)
137
138     deallocate(x_vector,y_vector,z_vector)
139     deallocate(vx_vector,vy_vector,vz_vector)
140     deallocate(force_x,force_y,force_z)
141
142     call cpu_time(time_end)
143     write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
144 end program molecular_dynamic_lennard_jones_02
```

molecular_dynamic_lennard_jones_03.f90

```
1   ! Inciso c)
2   ! make clean && make molecular_dynamic_lennard_jones_03.o && ./molecular_dynamic_lennard_jones_03.o
3   program molecular_dynamic_lennard_jones_03
4       use module_precision;use module_md_lennard_jones
5       implicit none
6       integer(sp), parameter   :: n_p=256_sp                              ! cantidad de partículasa
7       real(dp),    parameter   :: delta_time=0.005_dp                     ! paso temporal
8       integer(sp), parameter   :: time_eq=1000_sp,time_scal=50_sp         ! pasos de equilibración y de escaleo de veloc.
9       real(dp),    parameter   :: T_adim_ref=1.1_dp                       ! temperatura de referencia adimensional
10      real(dp),    parameter   :: density=0.8_dp                          ! densidad (particulas/volumen)
11      real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp              ! radio de corte de interacciones,masa
12      real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)     ! componentes de las posiciones/particula
13      real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
14      real(dp),    allocatable :: vtot_vector(:)
15      real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)        ! componentes de la fuerza/particula
16      integer(sp)              :: i,index,istat                           ! loop index
17      real(dp)                 :: T_adim
18      real(dp)                 :: vx_mc,vy_mc,vz_mc                       ! componentes de la velocidad del centro de masas
19      real(dp)                 :: time_end,time_start                     ! tiempos de CPU
20
21      real(dp),   allocatable  :: probability_vx(:),probability_vy(:),&
22                                  probability_vz(:),probability_vtot(:)
23      real(dp),   allocatable  :: exact_probability_vx(:),exact_probability_vy(:),&
24                                  exact_probability_vz(:),exact_probability_vtot(:)
25      real(dp),   allocatable  :: variable_vx(:),variable_vy(:),&
26                                  variable_vz(:),variable_vtot(:)
27      integer(sp)              :: n_bins   ! numbers of bins
28
29      call cpu_time(time_start)
30
31      allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
32      x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp
33
34      ! generamos configuración inicial (FCC structure)
35      call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)
36
```

```
37      allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
38      vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
39      call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
40      vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)
41
42      ! computamos fuerzas en el tiempo inicial
43      allocate(force_x(n_p),force_y(n_p),force_z(n_p))
44      force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
45      call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)
46
47      index=0
48      ! TRANSITORIO
49      do i=1,time_eq
50          index=index+1
51          write(*,*) 'paso temporal =',index,' de',time_eq
52          if (mod(i,time_scal)==0_sp) call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
53          call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
54          vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
55          ! velocity center of mass to zero
56          vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
57          vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
58          vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
59          T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
60      end do
61
62      deallocate(x_vector,y_vector,z_vector)
63      deallocate(force_x,force_y,force_z)
64
65      ! computamos distribuciones exactas de velocidad (Maxwell-Boltzmann y Gauss)
66      allocate(vtot_vector(n_p));vtot_vector(:)=0._dp
67      allocate(exact_probability_vx(n_p),exact_probability_vy(n_p),exact_probability_vz(n_p))
68      allocate(exact_probability_vtot(n_p))
69      open(10,file='../results/exact_velocities_distributions.dat',status='replace',action='write',iostat=istat)
70      if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
71      24 format(7(E12.4,x),E12.4);25 format(7(A12,x),A12)
72      write(10,25) 'vx','p(vx)','vy','p(vy)','vz','p(vz)','v_tot','p(vtot)'
73      do i=1,n_p
74          exact_probability_vx(i)=sqrt(mass*0.125_dp*(1._dp/atan(1._dp))*(1._dp/T_adim_ref))*&
75              exp(-mass*vx_vector(i)*vx_vector(i)*0.5_dp*(1._dp/T_adim_ref))
76
77          exact_probability_vy(i)=sqrt(mass*0.125_dp*(1._dp/atan(1._dp))*(1._dp/T_adim_ref))*&
78              exp(-mass*vy_vector(i)*vy_vector(i)*0.5_dp*(1._dp/T_adim_ref))
79
80          exact_probability_vz(i)=sqrt(mass*0.125_dp*(1._dp/atan(1._dp))*(1._dp/T_adim_ref))*&
81              exp(-mass*vz_vector(i)*vz_vector(i)*0.5_dp*(1._dp/T_adim_ref))
82
83          vtot_vector(i)=sqrt(vx_vector(i)*vx_vector(i)+vy_vector(i)*vy_vector(i)+vz_vector(i)*vz_vector(i))
84
85          exact_probability_vtot(i)=sqrt(0.5_dp*(1._dp/atan(1._dp))*(mass*(1._dp/T_adim_ref))**3)*&
86              vtot_vector(i)*vtot_vector(i)*exp(-mass*vtot_vector(i)*vtot_vector(i)*0.5_dp*(1._dp/T_adim_ref))
87
88          write(10,24) vx_vector(i),exact_probability_vx(i),vy_vector(i),exact_probability_vy(i),&
89                       vz_vector(i),exact_probability_vz(i),vtot_vector(i),exact_probability_vtot(i)
90      end do
91      deallocate(exact_probability_vx,exact_probability_vy,exact_probability_vz)
92      deallocate(exact_probability_vtot)
93      close(10)
94
95      ! hacemos histogramas de las componentes de la velocidad
96      n_bins=50
97      allocate(probability_vx(n_bins),variable_vx(n_bins+1))
98      call histogram(vx_vector,n_p,variable_vx,probability_vx,n_bins)
99
100     allocate(probability_vy(n_bins),variable_vy(n_bins+1))
101     call histogram(vy_vector,n_p,variable_vy,probability_vy,n_bins)
102
103     allocate(probability_vz(n_bins),variable_vz(n_bins+1))
104     call histogram(vz_vector,n_p,variable_vz,probability_vz,n_bins)
105
106     open(10,file='../results/components_velocities_histogram.dat',status='replace',action='write',iostat=istat)
107     if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
108     20 format(5(E12.4,x),E12.4);21 format(5(A12,x),A12)
109     write(10,21) 'vx','p(vx)','vy','p(vy)','vz','p(vz)'
110     do i = 1,n_bins
111         write(10,20) variable_vx(i),probability_vx(i)*(1._dp/(variable_vx(i+1)-variable_vx(i))),&
112                      variable_vy(i),probability_vy(i)*(1._dp/(variable_vy(i+1)-variable_vy(i))),&
113                      variable_vz(i),probability_vz(i)*(1._dp/(variable_vz(i+1)-variable_vz(i)))
114         ! write(10,20) variable_vx(i),probability_vx(i),&
115         !              variable_vy(i),probability_vy(i),&
116         !              variable_vz(i),probability_vz(i)
117     end do
118     deallocate(vx_vector,vy_vector,vz_vector)
119     deallocate(probability_vx,variable_vx)
120     deallocate(probability_vy,variable_vy)
121     deallocate(probability_vz,variable_vz)
122
123     ! hacemos histograma de la velocidad total
```

```
    n_bins=50
    allocate(probability_vtot(n_bins),variable_vtot(n_bins+1))
    call histogram(vtot_vector,n_p,variable_vtot,probability_vtot,n_bins)

    open(11,file='../results/total_velocities_histogram.dat',status='replace',action='write',iostat=istat)
    if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
    22 format(E12.4,x,E12.4);23 format(A12,x,A12)
    write(11,23) 'vtot','p(vtot)'
    do i = 1,n_bins
        write(11,22) variable_vtot(i),probability_vtot(i)*(1._dp/(variable_vtot(i+1)-variable_vtot(i)))
    end do
    close(11)
    deallocate(vtot_vector)
    deallocate(probability_vtot,variable_vtot)

    call cpu_time(time_end)
    write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
end program molecular_dynamic_lennard_jones_03

! subrutina para crear e imprimir histograma
subroutine histogram(x_vector,x_dim,variable,probability,n_bins)
    use module_precision
    implicit none
    integer(sp),      intent(in)    :: x_dim           ! dimension
    integer(sp),      intent(in)    :: n_bins
    real(dp),         intent(in)    :: x_vector(x_dim) ! data to do histogram
    real(dp),         intent(inout) :: variable(n_bins+1),probability(n_bins)

    integer(sp), allocatable :: counter(:)      ! counter vector of bins
    real(sp)                 :: max_value       ! maximun counter value
    real(dp)                 :: min_bin_point,max_bin_point
    real(dp)                 :: bins_step       ! step of points between bins
    integer(sp)              :: i,j       ! loop and control variables

    allocate(counter(n_bins))

    ! armamos el vector de bins (o variable(:)) en el rango [x_min,x_max]
    max_bin_point=maxval(x_vector(:));min_bin_point=minval(x_vector(:))
    bins_step=abs(max_bin_point-min_bin_point)*(1._dp/n_bins)
    do i=1,n_bins+1;variable(i)=min_bin_point+bins_step*real(i-1,dp);end do

    ! llenamos el vector contador de bins
    do i=1,n_bins
        counter(i)=0
        do j=1,x_dim
            if ((variable(i)<=x_vector(j)).and.(variable(i+1)>=x_vector(j))) then
                counter(i)=counter(i)+1
            endif
    enddo;enddo

    max_value=abs(real(maxval(counter(:)),dp))
    if (max_value==0._dp) write(*,*) 'math error'

    ! escribimos distribución de probabilidad
    probability(:)=real(counter(:),dp)*(1._dp/max_value)

    deallocate(counter)
end subroutine histogram
```

molecular_dynamic_lennard_jones_04.f90

```
! make clean && make molecular_dynamic_lennard_jones_04.o && ./molecular_dynamic_lennard_jones_04.o
program molecular_dynamic_lennard_jones_04
    use module_precision;use module_md_lennard_jones
    implicit none
    integer(sp), parameter   :: n_p=256_sp                          ! cantidad de partículasa
    real(dp),    parameter   :: T_adim_ref=1.1_dp                   ! temperatura de referencia adimensional
    real(dp),    parameter   :: density=0.8_dp                      ! densidad (particulas/volumen)
    real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp          ! radio de corte de interacciones y masa
    real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:) ! componentes de las posiciones/particula
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)    ! componentes de la fuerza/particula
    integer(sp)              :: time_eq,time_scal,&                 ! pasos de equilibración y de escaleo de veloc.
                                time_run                            ! pasos de evolucion en el estado estacionario
    real(dp)                 :: delta_time                          ! paso temporal
    integer(sp)              :: i,j,istat,index                     ! loop index
    real(dp)                 :: U_adim,U_med,var_U,err_U
    real(dp)                 :: Ec_adim,Ec_med,var_Ec,err_Ec
    real(dp)                 :: Etot_adim,Etot_med,var_Etot,err_Etot
    real(dp)                 :: T_adim
    real(dp)                 :: s1_U,s2_U
    real(dp)                 :: s1_Ec,s2_Ec
    real(dp)                 :: s1_Etot,s2_Etot
    real(dp)                 :: vx_mc,vy_mc,vz_mc                   ! componentes de la velocidad del centro de masas
    real(dp)                 :: time_end,time_start                 ! tiempos de CPU

```

```
26      call cpu_time(time_start)
27
28      allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
29      x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp
30
31      ! generamos configuración inicial (FCC structure)
32      call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)
33
34      allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
35      vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
36      call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
37      vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)
38
39      ! computamos fuerzas en el tiempo inicial
40      allocate(force_x(n_p),force_y(n_p),force_z(n_p))
41      force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
42      call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)
43
44
45      open(10,file='../results/enrgy_fluct_relations.dat',status='replace',action='write',iostat=istat)
46      if (istat/=0) write(*,*) 'ERROR! istat(10file) = ',istat
47      11 format(E12.4,x,E12.4);12 format(A12,x,A12)
48      write(10,12) 'delta_time','var_Etot/var_Ec'
49
50      delta_time=0.02_dp
51      do j=1,5
52          ! definimos los pasos temporales y pasos de MD
53          time_eq=5._dp*(1._dp/delta_time)
54          time_scal=0.25_dp*(1._dp/delta_time)
55          time_run=5._dp*(1._dp/delta_time)
56
57          index=0
58          ! TRANSITORIO
59          do i=1,time_eq
60              index=index+1
61              write(*,*) 'paso temporal =',index,' de',time_eq+time_run
62              if (mod(i,time_scal)==0_sp) call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
63              call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
64              vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
65              ! velocity center of mass to zero
66              vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
67              vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
68              vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
69              T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
70          end do
71
72          write(*,*) 'termino el transitorio'
73
74          ! ESTACIONARIO
75          U_med=0._dp
76          Ec_med=0._dp
77          Etot_med=0._dp
78
79          s1_U=0._dp;s2_U=0._dp
80          s1_Ec=0._dp;s2_Ec=0._dp
81          s1_Etot=0._dp;s2_Etot=0._dp
82
83          do i=1,time_run
84              index=index+1
85              write(*,*) 'paso temporal =',index,' de',time_eq+time_run
86              call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
87              vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
88
89              U_adim=u_lj_total(n_p,x_vector,y_vector,z_vector,r_cutoff,density)
90              Ec_adim=kinetic_ergy_total(n_p,vx_vector,vy_vector,vz_vector,mass)
91              Etot_adim=(U_adim+Ec_adim)*(1._dp/real(n_p,dp))
92
93              ! computamos 1er y 2do momento
94              s1_U=s1_U+U_adim*(1._dp/real(n_p,dp));s2_U=s2_U+U_adim*U_adim*(1._dp/real(n_p*n_p,dp))
95              s1_Ec=s1_Ec+Ec_adim*(1._dp/real(n_p,dp));s2_Ec=s2_Ec+Ec_adim*Ec_adim*(1._dp/real(n_p*n_p,dp))
96              s1_Etot=s1_Etot+Etot_adim;s2_Etot=s2_Etot+Etot_adim*Etot_adim
97
98              ! computamos valores medios (mejor a mayor paso evolucionado)
99              U_med=s1_U*(1._dp/real(i,dp))
100             Ec_med=s1_Ec*(1._dp/real(i,dp))
101             Etot_med=s1_Etot*(1._dp/real(i,dp))
102
103             ! computamos varianzas (mejor a mayor paso evolucionado)
104             var_U=(real(i,dp)*s2_U-s1_U*s1_U)*(1._dp/real(i*i,dp))
105             var_Ec=(real(i,dp)*s2_Ec-s1_Ec*s1_Ec)*(1._dp/real(i*i,dp))
106             var_Etot=(real(i,dp)*s2_Etot-s1_Etot*s1_Etot)*(1._dp/real(i*i,dp))
107
108         end do
109
110         write(*,*) 'termino el estacionario'
111
112         ! computamos errores en el último paso
```

```
        err_U=(var_U*0.25_dp)*(1._dp/real(time_eq-1,dp))
        err_Ec=(var_Ec*0.25_dp)*(1._dp/real(time_eq-1,dp))
        err_Etot=(var_Etot*0.25_dp)*(1._dp/real(time_eq-1,dp))

        write(10,11) delta_time,var_Etot*(1._dp/var_Ec)

        delta_time=delta_time*0.5_dp
    end do

    close(10)

    deallocate(x_vector,y_vector,z_vector)
    deallocate(vx_vector,vy_vector,vz_vector)
    deallocate(force_x,force_y,force_z)

    call cpu_time(time_end)
    write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
end program molecular_dynamic_lennard_jones_04
```

molecular_dynamic_lennard_jones_05.f90

```
! Inciso f)
! make clean && make molecular_dynamic_lennard_jones_05.o && ./molecular_dynamic_lennard_jones_05.o
program molecular_dynamic_lennard_jones_05
    use module_precision;use module_md_lennard_jones
    implicit none
    integer(sp), parameter   :: n_p=256_sp                          ! cantidad de partículasa
    real(dp),    parameter   :: T_adim_ref=1.1_dp                   ! temperatura de referencia adimensional
    real(dp),    parameter   :: density=0.8_dp                      ! densidad (particulas/volumen)
    real(dp),    parameter   :: mass=1._dp                          ! masa
    real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:) ! componentes de las posiciones/particula
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)    ! componentes de la fuerza/particula
    integer(sp)              :: time_eq,time_scal,&                 ! pasos de equilibración y de escaleo de veloc.
                                time_run                            ! pasos de evolucion en el estado estacionario
    real(dp)                 :: r_cutoff                            ! radio de corte de interacciones
    real(dp)                 :: delta_time                          ! paso temporal
    integer(sp)              :: i,j,istat,index                     ! loop index
    real(dp)                 :: U_adim,U_med,var_U,err_U
    real(dp)                 :: Ec_adim,Ec_med,var_Ec,err_Ec
    real(dp)                 :: Etot_adim,Etot_med,var_Etot,err_Etot
    real(dp)                 :: T_adim
    real(dp)                 :: s1_U,s2_U
    real(dp)                 :: s1_Ec,s2_Ec
    real(dp)                 :: s1_Etot,s2_Etot
    real(dp)                 :: vx_mc,vy_mc,vz_mc                   ! componentes de la velocidad del centro de masas
    real(dp)                 :: time_end,time_start                 ! tiempos de CPU

    call cpu_time(time_start)

    allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
    x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp

    ! generamos configuración inicial (FCC structure)
    call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)

    allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
    vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
    call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
    vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)

    ! computamos fuerzas en el tiempo inicial
    allocate(force_x(n_p),force_y(n_p),force_z(n_p))
    force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
    call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)


    open(10,file='../results/enrgy_fluct_relations_vs_rcutoff.dat',status='replace',action='write',iostat=istat)
    if (istat/=0) write(*,*) 'ERROR! istat(10file) = ',istat
    11 format(E14.6,x,E14.6);12 format(A14,x,A14)
    write(10,12) 'r_{cutoff}','varEtot/varEc'

    ! definimos pasos temporales pequeños
    delta_time=0.005_dp
    time_eq=5._dp*(1._dp/delta_time)
    time_scal=0.25_dp*(1._dp/delta_time)
    time_run=5._dp*(1._dp/delta_time)

    r_cutoff=0._dp
    do j=1,10

        ! definimos r_cutoff en el rango [1;5]
        r_cutoff=1.0_dp+(4._dp*(1._dp/9._dp)*real(j-1,dp))

        index=0
        ! TRANSITORIO
```

```
66          do i=1,time_eq
67              index=index+1
68              write(*,*) 'paso temporal =',index,' de',time_eq+time_run
69              if (mod(i,time_scal)==0_sp) call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
70              call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
71              vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
72              ! velocity center of mass to zero
73              vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
74              vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
75              vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
76              T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
77          end do
78
79          write(*,*) 'termino el transitorio'
80
81          ! ESTACIONARIO
82          U_med=0._dp
83          Ec_med=0._dp
84          Etot_med=0._dp
85
86          s1_U=0._dp;s2_U=0._dp
87          s1_Ec=0._dp;s2_Ec=0._dp
88          s1_Etot=0._dp;s2_Etot=0._dp
89
90          do i=1,time_run
91              index=index+1
92              write(*,*) 'paso temporal =',index,' de',time_eq+time_run
93              call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
94              vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
95
96              U_adim=u_lj_total(n_p,x_vector,y_vector,z_vector,r_cutoff,density)
97              Ec_adim=kinetic_ergy_total(n_p,vx_vector,vy_vector,vz_vector,mass)
98              Etot_adim=(U_adim+Ec_adim)*(1._dp/real(n_p,dp))
99
100             ! computamos 1er y 2do momento
101             s1_U=s1_U+U_adim*(1._dp/real(n_p,dp));s2_U=s2_U+U_adim*U_adim*(1._dp/real(n_p*n_p,dp))
102             s1_Ec=s1_Ec+Ec_adim*(1._dp/real(n_p,dp));s2_Ec=s2_Ec+Ec_adim*Ec_adim*(1._dp/real(n_p*n_p,dp))
103             s1_Etot=s1_Etot+Etot_adim;s2_Etot=s2_Etot+Etot_adim*Etot_adim
104
105             ! computamos valores medios (mejor a mayor paso evolucionado)
106             U_med=s1_U*(1._dp/real(i,dp))
107             Ec_med=s1_Ec*(1._dp/real(i,dp))
108             Etot_med=s1_Etot*(1._dp/real(i,dp))
109
110             ! computamos varianzas (mejor a mayor paso evolucionado)
111             var_U=(real(i,dp)*s2_U-s1_U*s1_U)*(1._dp/real(i*i,dp))
112             var_Ec=(real(i,dp)*s2_Ec-s1_Ec*s1_Ec)*(1._dp/real(i*i,dp))
113             var_Etot=(real(i,dp)*s2_Etot-s1_Etot*s1_Etot)*(1._dp/real(i*i,dp))
114
115         end do
116
117         write(*,*) 'termino el estacionario'
118
119         ! computamos errores en el último paso
120         err_U=(var_U*0.25_dp)*(1._dp/real(time_eq-1,dp))
121         err_Ec=(var_Ec*0.25_dp)*(1._dp/real(time_eq-1,dp))
122         err_Etot=(var_Etot*0.25_dp)*(1._dp/real(time_eq-1,dp))
123
124         write(10,11) r_cutoff,var_Etot*(1._dp/var_Ec)
125     end do
126
127     close(10)
128
129     deallocate(x_vector,y_vector,z_vector)
130     deallocate(vx_vector,vy_vector,vz_vector)
131     deallocate(force_x,force_y,force_z)
132
133     call cpu_time(time_end)
134     write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
135 end program molecular_dynamic_lennard_jones_05
```

molecular_dynamic_lennard_jones_06.f90

```
1   ! Inciso g)
2   ! make clean && make molecular_dynamic_lennard_jones_06.o && ./molecular_dynamic_lennard_jones_06.o
3   program molecular_dynamic_lennard_jones_06
4       use module_precision;use module_md_lennard_jones
5       implicit none
6       integer(sp), parameter   :: nc_max=8_sp                          ! factor máximo paa definir número de partículas
7       integer(sp), parameter   :: time_eq=1000_sp,time_scal=50_sp,&    ! pasos de equilibración y de escaleo de veloc.
8                                   time_run=10_sp                       ! pasos de evolucion en el estado estacionario
9       real(dp),    parameter   :: T_adim_ref=1.1_dp                    ! temperatura de referencia adimensional
10      real(dp),    parameter   :: density=0.8_dp                       ! densidad (particulas/volumen)
11      real(dp),    parameter   :: mass=1._dp                           ! masa
12      real(dp),    parameter   :: delta_time=0.005_dp                  ! paso temporal
13      real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)  ! componentes de las posiciones/particula
```

```
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)       ! componentes de la fuerza/particula
    integer(sp)              :: n_p                                    ! cantidad de partículas
    integer(sp)              :: i,j,istat,index                        ! loop index
    real(dp)                 :: r_cutoff,L                             ! radio de corte de interacciones
    real(dp)                 :: T_adim                                 ! temperatura adimensional
    real(dp)                 :: vx_mc,vy_mc,vz_mc                      ! componentes de la velocidad del centro de masas
    real(dp)                 :: time_end,time_start                    ! tiempos de CPU

    open(10,file='../results/num_particles_vs_cpu_time.dat',status='replace',action='write',iostat=istat)
    if (istat/=0) write(*,*) 'ERROR! istat(10file) = ',istat
    11 format(E12.4,x,I12);12 format(A12,x,A12)
    write(10,12) 'CPU_time','n_p'

    do j=1,nc_max
        call cpu_time(time_start)
        write(*,*) 'corrida=',j,' de ',nc_max
        n_p=4_sp*j*j*j
        L=(real(n_p,dp)*(1._dp/density))**(1._dp/3._dp)
        r_cutoff=L*0.3_dp ! definimos r_cutoff < L/2

        allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
        x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp

        ! generamos configuración inicial (FCC structure)
        call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)

        allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
        vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
        call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)

        ! computamos fuerzas en el tiempo inicial
        allocate(force_x(n_p),force_y(n_p),force_z(n_p))
        force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
        call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)

        index=0
        ! TRANSITORIO
        do i=1,time_eq
            index=index+1
            if (mod(i,time_scal)==0_sp) call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
            call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
            vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
            ! velocity center of mass to zero
            vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
            vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
            vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
            T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
        end do

        write(*,*) 'termino el transitorio'

        ! ESTACIONARIO
        do i=1,time_run
            index=index+1
            call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
            vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        end do
        write(*,*) 'termino el estacionario'

        deallocate(x_vector,y_vector,z_vector)
        deallocate(vx_vector,vy_vector,vz_vector)
        deallocate(force_x,force_y,force_z)
        call cpu_time(time_end)
        write(10,11) time_end-time_start,n_p
    end do
    close(10)
end program molecular_dynamic_lennard_jones_06
```

Laboratorio 05 - Problema 02 - Códigos

md_lj_canonical_ensamble_01.f90

```
! make clean && make md_lj_canonical_ensamble_01.o && ./md_lj_canonical_ensamble_01.o
program md_lj_canonical_ensamble_01
    use module_precision;use module_md_lennard_jones
    implicit none
    integer(sp), parameter   :: n_p=500_sp                             ! cantidad de partículasa
    real(dp),    parameter   :: delta_time=0.005_dp                    ! paso temporal
    integer(sp), parameter   :: time_eq=2000_sp,&                      ! pasos de equilibración
                                time_run=1000_sp                       ! pasos de evolucion en el estado estacionario
    real(dp),    parameter   :: T_adim_ref=1.0_dp                      ! temperatura de referencia adimensional
    real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp             ! radio de corte de interacciones y masa
    real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)    ! componentes de las posiciones/particula
```

```
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)       ! componentes de la fuerza/particula
    integer(sp)              :: i,index                                ! loop index
    real(dp)                 :: T_adim                                 ! Temperatura
    real(dp)                 :: vx_mc,vy_mc,vz_mc                      ! componentes de la velocidad del centro de masas
    real(dp)                 :: time,time_end,time_start               ! tiempos de CPU
    real(dp),   allocatable  :: g(:)                                   ! radial ditribution vector
    real(dp),    parameter   :: density=1.2_dp                         ! densidad (particulas/volumen)
    integer(sp)              :: n_bins                                 ! numero total de bins

    call cpu_time(time_start)

    allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
    x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp

    ! generamos configuración inicial (FCC structure)
    call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)

    allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
    vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
    call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
    vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)

    ! computamos fuerzas en el tiempo inicial
    allocate(force_x(n_p),force_y(n_p),force_z(n_p))
    force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
    call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)

    ! TRANSITORIO
    index=0
    do i=1,time_eq
        index=index+1
        write(*,*) 'paso temporal =',index,' de',time_eq+time_run
        call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
        call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        ! velocity center of mass to zero
        vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
        vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
        vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
        T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
    end do

    ! ESTACIONARIO
    time=0._dp

    do i=1,time_run
        index=index+1
        write(*,*) 'paso temporal =',index,' de',time_eq+time_run
        call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
        call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        time=real(i,dp)*delta_time
    end do

    n_bins=1000
    allocate(g(n_bins))
    ! descomentar para density=0.8
    ! call radial_ditribution_function('../results/radial_ditribution_function_rho1.dat',n_p,density,&
    !                                  x_vector,y_vector,z_vector,n_bins,g)
    ! descomentar para density=1.2
    call radial_ditribution_function('../results/radial_ditribution_function_rho2.dat',n_p,density,&
                                     x_vector,y_vector,z_vector,n_bins,g)
    deallocate(g)

    deallocate(x_vector,y_vector,z_vector)
    deallocate(vx_vector,vy_vector,vz_vector)
    deallocate(force_x,force_y,force_z)

    call cpu_time(time_end)
    write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
end program md_lj_canonical_ensamble_01
```

md_lj_canonical_ensamble_02.f90

```
! make clean && make md_lj_canonical_ensamble_02.o && ./md_lj_canonical_ensamble_02.o
program md_lj_canonical_ensamble_02
    use module_precision;use module_md_lennard_jones
    implicit none
    integer(sp), parameter   :: n_p=500_sp                             ! cantidad de partículasa
    real(dp),    parameter   :: delta_time=0.005_dp                    ! paso temporal
    integer(sp), parameter   :: time_eq=2000_sp,&                      ! pasos de equilibración
                                time_run=1000_sp                       ! pasos de evolucion en el estado estacionario
    real(dp),    parameter   :: T_adim_ref=1.0_dp                      ! temperatura de referencia adimensional
    real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp             ! radio de corte de interacciones y masa
    real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)    ! componentes de las posiciones/particula
```

```
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)       ! componentes de la fuerza/particula
    integer(sp)              :: i,index,istat                              ! loop index
    real(dp)                 :: T_adim                                 ! Temperatura
    real(dp)                 :: vx_mc,vy_mc,vz_mc                      ! componentes de la velocidad del centro de masas
    real(dp)                 :: time,time_end,time_start               ! tiempos de CPU
    real(dp),    parameter   :: density=0.8_dp                         ! densidad (particulas/volumen)
    !real(dp),    parameter   :: density=1.2_dp                         ! densidad (particulas/volumen)

    ! DESCOMENTAR PARA density=0.8
    open(10,file='../results/structure_function_rho1.dat',status='replace',action='write',iostat=istat)
    ! DESCOMENTAR PARA density=1.2
    !open(10,file='../results/structure_function_rho2.dat',status='replace',action='write',iostat=istat)
    if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
    24 format(E12.4,x,E12.4);25 format(A12,x,A12)
    write(10,25) 'time','S(k,t)'

    call cpu_time(time_start)

    allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
    x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp

    ! generamos configuración inicial (FCC structure)
    call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)
    write(10,24) 0._dp,static_structure_factor(n_p,density,x_vector,y_vector,z_vector)

    allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
    vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
    call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
    vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)

    ! computamos fuerzas en el tiempo inicial
    allocate(force_x(n_p),force_y(n_p),force_z(n_p))
    force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
    call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)

    ! TRANSITORIO
    index=0
    time=0._dp
    do i=1,time_eq
        index=index+1
        write(*,*) 'paso temporal =',index,' de',time_eq+time_run
        call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
        call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        ! velocity center of mass to zero
        vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
        vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
        vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
        T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
        time=real(i,dp)*delta_time
        write(10,24) time,static_structure_factor(n_p,density,x_vector,y_vector,z_vector)
    end do

    ! ESTACIONARIO
    time=0._dp
    do i=1,time_run
        index=index+1
        write(*,*) 'paso temporal =',index,' de',time_eq+time_run
        call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
        call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        time=(real(time_eq,dp)+real(i,dp))*delta_time
        write(10,24) time,static_structure_factor(n_p,density,x_vector,y_vector,z_vector)
    end do
    close(10)

    deallocate(x_vector,y_vector,z_vector)
    deallocate(vx_vector,vy_vector,vz_vector)
    deallocate(force_x,force_y,force_z)

    call cpu_time(time_end)
    write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
end program md_lj_canonical_ensamble_02
```

md_lj_canonical_ensamble_03.f90

```
 ! make clean && make md_lj_canonical_ensamble_03.o && ./md_lj_canonical_ensamble_03.o
 program md_lj_canonical_ensamble_03
     use module_precision;use module_md_lennard_jones
     implicit none
     integer(sp), parameter   :: n_p=500_sp                          ! cantidad de partículasa
     real(dp),    parameter   :: delta_time=0.005_dp                 ! paso temporal
     integer(sp), parameter   :: time_eq=2000_sp,&                   ! pasos de equilibración
                                 time_run=5000_sp                    ! pasos de evolucion en el estado estacionario
     integer(sp), parameter   :: tau_max_corr=5000_sp                ! pasos maximos de correlación
```

```
    real(dp),    parameter   :: T_adim_ref=1.0_dp                      ! temperatura de referencia adimensional
    real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp             ! radio de corte de interacciones y masa
    real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)    ! componentes de las posiciones/particula
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)       ! componentes de la fuerza/particula
    real(dp),    allocatable :: wxx_matrix(:,:),wyy_matrix(:,:),&      ! matrices auxiliares para cálculo de msd
                                wzz_matrix(:,:)
    real(dp),    allocatable :: sum_wxx_vector(:),sum_wyy_vector(:),&  ! vectores auxiliares para cálculo de msd
                                sum_wzz_vector(:),counter_data(:)
    integer(sp)              :: i,j,index,istat,counter                 ! loop index
    real(dp)                 :: T_adim                                  ! Temperatura
    real(dp)                 :: vx_mc,vy_mc,vz_mc                       ! componentes de la velocidad del centro de masas
    real(dp)                 :: time,time_end,time_start                ! tiempos de CPU
    real(dp)                 :: msd                                     ! desplazamiento cuadrático medio
    real(dp),    parameter   :: density=0.8_dp                          ! densidad (particulas/volumen)
    !real(dp),    parameter   :: density=1.2_dp                         ! densidad (particulas/volumen)

    ! DESCOMENTAR PARA density=0.8
    open(10,file='../results/msd_rho1.dat',status='replace',action='write',iostat=istat)
    ! DESCOMENTAR PARA density=1.2
    !open(10,file='../results/msd_rho2.dat',status='replace',action='write',iostat=istat)
    if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
    24 format(E12.4,x,E12.4);25 format(A12,x,A12)
    write(10,25) 'time','msd'

    call cpu_time(time_start)

    allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
    x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp

    ! generamos configuración inicial (FCC structure)
    call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)

    counter=0
    allocate(wxx_matrix(n_p,tau_max_corr),wyy_matrix(n_p,tau_max_corr),wzz_matrix(n_p,tau_max_corr))
    allocate(sum_wxx_vector(tau_max_corr),sum_wyy_vector(tau_max_corr),sum_wzz_vector(tau_max_corr))
    allocate(counter_data(tau_max_corr))

    sum_wxx_vector(:)=0._dp;sum_wxx_vector(:)=0._dp;sum_wxx_vector(:)=0._dp
    counter_data(:)=0_sp

    call mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
    wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
    counter_data,counter)

    allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))
    vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
    call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
    vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)

    ! computamos fuerzas en el tiempo inicial
    allocate(force_x(n_p),force_y(n_p),force_z(n_p))
    force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
    call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)

    ! TRANSITORIO
    index=0
    time=0._dp
    do i=1,time_eq
        index=index+1
        write(*,*) 'paso temporal =',index,' de',time_eq+time_run
        call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
        call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        ! velocity center of mass to zero
        vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
        vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
        vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
        T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
        time=real(i,dp)*delta_time
        call mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
        wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
        counter_data,counter)
    end do

    ! ESTACIONARIO
    time=0._dp
    do i=1,time_run
        index=index+1
        write(*,*) 'paso temporal =',index,' de',time_eq+time_run
        call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
        call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        time=(real(time_eq,dp)+real(i,dp))*delta_time
        call mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
        wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
        counter_data,counter)
```

```
    end do

    do j=1,tau_max_corr
        time=real(j,dp)*delta_time
        msd=(sum_wxx_vector(j)+sum_wyy_vector(j)+sum_wzz_vector(j))*(1._dp/real(counter_data(j),dp))*(1._dp/real(n_p,dp))
        write(10,24) time,msd
    end do
    close(10)

    deallocate(x_vector,y_vector,z_vector)
    deallocate(vx_vector,vy_vector,vz_vector)
    deallocate(force_x,force_y,force_z)

    deallocate(wxx_matrix,wyy_matrix,wzz_matrix)
    deallocate(sum_wxx_vector,sum_wyy_vector,sum_wzz_vector)
    deallocate(counter_data)

    call cpu_time(time_end)
    write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
end program md_lj_canonical_ensamble_03

! subrutina para calcular el desplazamiento cuadrático medio
subroutine mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
    wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
    counter_data,counter)
    use module_precision

    implicit none
    integer(sp), intent(in)    :: n_p                                    ! numero total de partículas
    integer(sp), intent(in)    :: tau_max_corr                           ! pasos maximos de autocorrelación
    real(dp),    intent(in)    :: x_vector(n_p),y_vector(n_p),z_vector(n_p) ! componentes del vector posición
    real(dp),    intent(inout) :: wxx_matrix(n_p,tau_max_corr),&         ! matrices de acumulación
                                  wyy_matrix(n_p,tau_max_corr),&
                                  wzz_matrix(n_p,tau_max_corr)
    real(dp),    intent(inout) :: sum_wxx_vector(tau_max_corr),&         ! vectores de sumas auxiliares
                                  sum_wyy_vector(tau_max_corr),&
                                  sum_wzz_vector(tau_max_corr)
    real(dp),    intent(inout) :: counter_data(tau_max_corr)             ! contador de datos
    integer(sp), intent(inout) :: counter                                ! contador de entradas

    integer(sp)            :: i,j
    integer(sp)            :: tau_corr_0,tau_corr_t ! tiempos de correlación
    integer(sp), parameter :: nmax_tau_corr_0=500_sp ! maximo número de tau_corr_0 que almacenamos

    counter=counter+1                       ! numero de veces que entro a la subrutina
    tau_corr_0=mod(counter-1,tau_max_corr)+1 ! tiempo de correlación actual tau_corr_0={1,2,...,tau_max_corr}

    ! guardamos cíclicamente los últimos tau_max_corr valores
    !  de las componentes x,y,z de cada partícula
    do i=1,n_p
        wxx_matrix(i,tau_corr_0)=x_vector(i)
        wyy_matrix(i,tau_corr_0)=y_vector(i)
        wzz_matrix(i,tau_corr_0)=z_vector(i)
    end do

    if ((mod(counter,nmax_tau_corr_0)==0).and.(counter>tau_max_corr)) then
        do j=1,tau_max_corr
            tau_corr_t=mod(counter-j,tau_max_corr)+1
            do i=1,n_p
                sum_wxx_vector(j)=sum_wxx_vector(j)+(wxx_matrix(i,tau_corr_0)-wxx_matrix(i,tau_corr_t))*&
                                                   (wxx_matrix(i,tau_corr_0)-wxx_matrix(i,tau_corr_t))
                sum_wyy_vector(j)=sum_wyy_vector(j)+(wyy_matrix(i,tau_corr_0)-wyy_matrix(i,tau_corr_t))*&
                                                   (wyy_matrix(i,tau_corr_0)-wyy_matrix(i,tau_corr_t))
                sum_wzz_vector(j)=sum_wzz_vector(j)+(wzz_matrix(i,tau_corr_0)-wzz_matrix(i,tau_corr_t))*&
                                                   (wzz_matrix(i,tau_corr_0)-wzz_matrix(i,tau_corr_t))
            end do
            ! actualizamos el contador de datos para cada tiempo de correlación
            counter_data(j)=counter_data(j)+1._dp
        end do
    end if
end subroutine mean_squared_displacement
```

Laboratorio 05 - Problema 03 - Códigos

md_lj_order_transition_01.f90

```
! make clean && make md_lj_order_transition_01.o && ./md_lj_order_transition_01.o
program md_lj_order_transition_01
    use module_precision;use module_md_lennard_jones
    implicit none
    integer(sp), parameter   :: n_p=256_sp                          ! cantidad de partículasa
    real(dp),    parameter   :: delta_time=0.005_dp                 ! paso temporal
    integer(sp), parameter   :: time_eq=2000_sp,&                   ! pasos de equilibración
                                time_run=1000_sp                    ! pasos de evolucion en el estado estacionario
    real(dp),    parameter   :: T_adim_ref=1.0_dp                   ! temperatura de referencia adimensional
```

```
    real(dp),    parameter   :: r_cutoff=2.5_dp,mass=1._dp             ! radio de corte de interacciones y masa
    real(dp),    allocatable :: x_vector(:),y_vector(:),z_vector(:)    ! componentes de las posiciones/particula
    real(dp),    allocatable :: vx_vector(:),vy_vector(:),vz_vector(:) ! componentes de la velocidad/particula
    real(dp),    allocatable :: force_x(:),force_y(:),force_z(:)       ! componentes de la fuerza/particula
    integer(sp)              :: i,j,k,index,istat                        ! loop index
    real(dp)                 :: T_adim                                 ! Temperatura
    real(dp)                 :: vx_mc,vy_mc,vz_mc                      ! componentes de la velocidad del centro de masas
    real(dp)                 :: time,time_end,time_start               ! tiempos de CPU
    logical                  :: pressure_switch,structure_factor_switch,&   ! variables para decidir escritura de datos
    diffusion_coeff_switch
    ! VARIABLES PARA REALIZAR BARRIDO DE DENSIDADES
    real(dp)                 :: density                                ! densidad (particulas/volumen)
    real(dp),    parameter   :: density_min=0.8_dp,density_max=1.2_dp  ! rango de densidades
    integer(sp), parameter   :: n_density=10_sp                        ! cantidad de densidades simuladas
    real(dp),    parameter   :: step_density=abs(density_max-density_min)*& ! paso de variació de densidades
                                  (1._dp/real(n_density-1,dp))
    ! VARIABLES PARA COMPUTAR PRESIÓN Y FACTOR DE ESTRUCTURA ESTÁTICO
    real(dp)                 :: press,press_med,var_press,err_press
    real(dp)                 :: s1_press,s2_press
    real(dp)                 :: Sk,Sk_med,var_Sk,err_Sk
    real(dp)                 :: s1_Sk,s2_Sk
    real(dp)                 :: D,D_med,var_D,err_D
    real(dp)                 :: s1_D,s2_D
    real(dp)                 :: msd_med,var_msd,err_msd
    real(dp)                 :: s1_msd,s2_msd
    ! VARIABLES PARA COMPUTAR COEFICIENTE DE DIFUSIÓN
    integer(sp), parameter   :: tau_max_corr=1000_sp                   ! pasos maximos de correlación
    real(dp),    allocatable :: wxx_matrix(:,:),wyy_matrix(:,:),&      ! matrices auxiliares para cálculo de msd
                                wzz_matrix(:,:)
    real(dp),    allocatable :: sum_wxx_vector(:),sum_wyy_vector(:),&  ! vectores auxiliares para cálculo de msd
                                sum_wzz_vector(:),counter_data(:)
    real(dp)                 :: msd                                    ! desplazamiento cuadrático medio
    integer(sp)              :: counter

    pressure_switch         =.false. ! escribir presión vs densidad
    structure_factor_switch =.false. ! escribir factor de estructura vs densidad
    diffusion_coeff_switch  =.true. ! escribir coeficiente de difusión vs densidad

    20 format(2(E12.4,x),x,E12.4);21 format(2(A12,x),x,A12)
    22 format(4(E12.4,x),x,E12.4);23 format(4(A12,x),x,A12)
    if (pressure_switch.eqv..true.) then
        open(10,file='../results/pressure_vs_density.dat',status='replace',action='write',iostat=istat)
        if (istat/=0) write(*,*) 'ERROR! istat(10file) = ',istat
        write(10,21) 'density','pressure','error'
    else if (structure_factor_switch.eqv..true.) then
        open(11,file='../results/struct_factor_vs_density.dat',status='replace',action='write',iostat=istat)
        if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
        write(11,21) 'density','S(k)_med','error'
    else if (diffusion_coeff_switch.eqv..true.) then
        open(12,file='../results/diffsuion_vs_density.dat',status='replace',action='write',iostat=istat)
        if (istat/=0) write(*,*) 'ERROR! istat(12file) = ',istat
        write(12,23) 'density','D','error','msd','error'
    end if

    call cpu_time(time_start)

    ! allocación de memoria
    allocate(x_vector(n_p),y_vector(n_p),z_vector(n_p))
    allocate(vx_vector(n_p),vy_vector(n_p),vz_vector(n_p))

    allocate(force_x(n_p),force_y(n_p),force_z(n_p))

    allocate(wxx_matrix(n_p,tau_max_corr),wyy_matrix(n_p,tau_max_corr),wzz_matrix(n_p,tau_max_corr))
    allocate(sum_wxx_vector(tau_max_corr),sum_wyy_vector(tau_max_corr),sum_wzz_vector(tau_max_corr))
    allocate(counter_data(tau_max_corr))

    do j=1,n_density
        ! seteo de variables y parámetros
        x_vector(:)=0._dp;y_vector(:)=0._dp;z_vector(:)=0._dp
        vx_vector(:)=0._dp;vy_vector(:)=0._dp;vz_vector(:)=0._dp
        force_x(:)=0._dp;force_y(:)=0._dp;force_z(:)=0._dp
        sum_wxx_vector(:)=0._dp;sum_wxx_vector(:)=0._dp;sum_wxx_vector(:)=0._dp
        counter=0;counter_data(:)=0_sp

        ! definimos r_cutoff en el rango [density_min;density_max]
        density=density_min+step_density*real(j-1,dp)

        ! generamos configuración inicial (FCC structure)
        call initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,2)
        call mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
        wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
        counter_data,counter)

        call md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,T_adim_ref,delta_time,density,mass)

        ! computamos fuerzas en el tiempo inicial
```

```
        call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)

        ! TRANSITORIO
        index=0
        time=0._dp
        do i=1,time_eq
            index=index+1
            write(*,*) 'paso temporal =',index,' de',time_eq+time_run,j
            call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
            call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
            vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
            ! velocity center of mass to zero
            vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp));vx_vector(:)=(vx_vector(:)-vx_mc)
            vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp));vy_vector(:)=(vy_vector(:)-vy_mc)
            vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp));vz_vector(:)=(vz_vector(:)-vz_mc)
            T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
            call mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
            wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
            counter_data,counter)
            time=real(i,dp)*delta_time
        end do

        ! ESTACIONARIO
        press_med=0._dp;s1_press=0._dp;s2_press=0._dp
        Sk_med=0._dp;s1_Sk=0._dp;s2_Sk=0._dp

        time=0._dp
        do i=1,time_run
            index=index+1
            write(*,*) 'paso temporal =',index,' de',time_eq+time_run,j
            call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
            call velocity_verlet(n_p,x_vector,y_vector,z_vector,&
            vx_vector,vy_vector,vz_vector,delta_time,mass,r_cutoff,density,force_x,force_y,force_z)

            ! computamos observables,1er y 2do momento,valores medios y varianzas
            if (pressure_switch.eqv..true.) then
                press=pressure(n_p,density,mass,r_cutoff,x_vector,y_vector,z_vector,&
                vx_vector,vy_vector,vz_vector)
                s1_press=s1_press+press;s2_press=s2_press+press*press
                press_med=s1_press*(1._dp/real(i,dp))
                var_press=(real(i,dp)*s2_press-s1_press*s1_press)*(1._dp/real(i*i,dp))
            else if (structure_factor_switch.eqv..true.) then
                Sk=static_structure_factor(n_p,density,x_vector,y_vector,z_vector)
                s1_Sk=s1_Sk+Sk;s2_Sk=s2_Sk+Sk*Sk
                Sk_med=s1_Sk*(1._dp/real(i,dp))
                var_Sk=(real(i,dp)*s2_Sk-s1_Sk*s1_Sk)*(1._dp/real(i*i,dp))
            else if (diffusion_coeff_switch.eqv..true.) then
                call mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
                wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
                counter_data,counter)
            end if

            time=real(i,dp)*delta_time
        end do

        if (pressure_switch.eqv..true.) then
            ! computamos errores en el último paso
            err_press=(var_press*0.25_dp)*(1._dp/real(time_eq-1,dp))
            write(10,20) density,press_med,err_press
        else if (structure_factor_switch.eqv..true.) then
            ! computamos errores en el último paso
            err_Sk=(var_Sk*0.25_dp)*(1._dp/real(time_eq-1,dp))
            write(11,20) density,Sk_med,err_Sk
        else if (diffusion_coeff_switch.eqv..true.) then
            D_med=0._dp;s1_D=0._dp;s2_D=0._dp
            msd_med=0._dp;s1_msd=0._dp;s2_msd=0._dp
            do k=1,tau_max_corr
                time=real(k,dp)*delta_time
                ! computamos msd
                msd=(sum_wxx_vector(k)+sum_wyy_vector(k)+sum_wzz_vector(k))*&
                    (1._dp/real(counter_data(k),dp))*(1._dp/real(n_p,dp))
                ! computamos observables,1er y 2do momento,valores medios y varianzas
                D=msd*(1._dp/6._dp)*(1._dp/time)
                s1_D=s1_D+D;s2_D=s2_D+D*D
                D_med=s1_D*(1._dp/real(k,dp))
                var_D=(real(k,dp)*s2_D-s1_D*s1_D)*(1._dp/real(k*k,dp))

                s1_msd=s1_msd+msd;s2_msd=s2_msd+msd*msd
                msd_med=s1_msd*(1._dp/real(k,dp))
                var_msd=(real(k,dp)*s2_msd-s1_msd*s1_msd)*(1._dp/real(k*k,dp))
            end do
            ! computamos errores en el último paso
            err_D=(var_D*0.25_dp)*(1._dp/real(tau_max_corr-1,dp))
            err_msd=(var_msd*0.25_dp)*(1._dp/real(tau_max_corr-1,dp))
            write(12,22) density,D_med,err_D,msd_med,err_msd
        end if
    end do
```

```
184
185     if (pressure_switch.eqv..true.) close(10)
186     if (structure_factor_switch.eqv..true.) close(11)
187     if (diffusion_coeff_switch.eqv..true.) close(12)
188
189     deallocate(x_vector,y_vector,z_vector)
190     deallocate(vx_vector,vy_vector,vz_vector)
191     deallocate(force_x,force_y,force_z)
192     deallocate(wxx_matrix,wyy_matrix,wzz_matrix)
193     deallocate(sum_wxx_vector,sum_wyy_vector,sum_wzz_vector)
194     deallocate(counter_data)
195
196     call cpu_time(time_end)
197     write(*,*) 'elapsed time = ',time_end-time_start,'[s]'
198 end program md_lj_order_transition_01
199
200 ! subrutina para calcular el desplazamiento cuadrático medio
201 subroutine mean_squared_displacement(n_p,x_vector,y_vector,z_vector,tau_max_corr,&
202     wxx_matrix,wyy_matrix,wzz_matrix,sum_wxx_vector,sum_wyy_vector,sum_wzz_vector,&
203     counter_data,counter)
204     use module_precision
205
206     implicit none
207     integer(sp), intent(in)    :: n_p                                    ! numero total de partículas
208     integer(sp), intent(in)    :: tau_max_corr                           ! pasos maximos de autocorrelación
209     real(dp),    intent(in)    :: x_vector(n_p),y_vector(n_p),z_vector(n_p) ! componentes del vector posición
210     real(dp),    intent(inout) :: wxx_matrix(n_p,tau_max_corr),&         ! matrices de acumulación
211                                   wyy_matrix(n_p,tau_max_corr),&
212                                   wzz_matrix(n_p,tau_max_corr)
213     real(dp),    intent(inout) :: sum_wxx_vector(tau_max_corr),&         ! vectores de sumas auxiliares
214                                   sum_wyy_vector(tau_max_corr),&
215                                   sum_wzz_vector(tau_max_corr)
216     real(dp),    intent(inout) :: counter_data(tau_max_corr)             ! contador de datos
217     integer(sp), intent(inout) :: counter                                ! contador de entradas
218
219     integer(sp)            :: i,j
220     integer(sp)            :: tau_corr_0,tau_corr_t ! tiempos de correlación
221     integer(sp), parameter :: nmax_tau_corr_0=10_sp ! maximo número de tau_corr_0 que almacenamos
222
223     counter=counter+1                      ! numero de veces que entro a la subrutina
224     tau_corr_0=mod(counter-1,tau_max_corr)+1 ! tiempo de correlación actual tau_corr_0={1,2,...,tau_max_corr}
225
226     ! guardamos cíclicamente los últimos tau_max_corr valores
227     !  de las componentes x,y,z de cada partícula
228     do i=1,n_p
229         wxx_matrix(i,tau_corr_0)=x_vector(i)
230         wyy_matrix(i,tau_corr_0)=y_vector(i)
231         wzz_matrix(i,tau_corr_0)=z_vector(i)
232     end do
233
234     if ((mod(counter,nmax_tau_corr_0)==0).and.(counter>tau_max_corr)) then
235         do j=1,tau_max_corr
236             tau_corr_t=mod(counter-j,tau_max_corr)+1
237             do i=1,n_p
238                 sum_wxx_vector(j)=sum_wxx_vector(j)+(wxx_matrix(i,tau_corr_0)-wxx_matrix(i,tau_corr_t))*&
239                                                    (wxx_matrix(i,tau_corr_0)-wxx_matrix(i,tau_corr_t))
240                 sum_wyy_vector(j)=sum_wyy_vector(j)+(wyy_matrix(i,tau_corr_0)-wyy_matrix(i,tau_corr_t))*&
241                                                    (wyy_matrix(i,tau_corr_0)-wyy_matrix(i,tau_corr_t))
242                 sum_wzz_vector(j)=sum_wzz_vector(j)+(wzz_matrix(i,tau_corr_0)-wzz_matrix(i,tau_corr_t))*&
243                                                    (wzz_matrix(i,tau_corr_0)-wzz_matrix(i,tau_corr_t))
244             end do
245             ! actualizamos el contador de datos para cada tiempo de correlación
246             counter_data(j)=counter_data(j)+1._dp
247         end do
248     end if
249 end subroutine mean_squared_displacement
```

Laboratorio 05 - Módulo principal

module_md_lennard_jones.f90

```
1   ! module of molecular dynamic to lennard jones potential
2   ! gfortran -c module_precision.f90 module_mt19937.f90 module_md_lennard_jones.f90
3   module module_md_lennard_jones
4       use module_precision;use module_mt19937, only: sgrnd,grnd
5       implicit none
6       contains
7
8       ! FUNCIONES
9       ! FUNCION PARA CALCULAR EL FACTOR DE ESTRUCTURA ESTÁTICO (PARAMETRO DE ORDEN CRISTALINO)
10      function static_structure_factor(n_p,density,x_vector,y_vector,z_vector)
11          integer(sp), intent(in) :: n_p                                    ! numero de partículas
12          real(dp),    intent(in) :: density                                ! densidad de partículas
13          real(dp),    intent(in) :: x_vector(n_p),y_vector(n_p),z_vector(n_p) ! coordenadas del vector posicion
14          real(dp)                :: static_structure_factor                ! funcion de estructura
```

```
15          real(dp),    parameter  :: pi=4._dp*atan(1._dp)
16          real(dp)                :: a                                    ! parámetro de red
17          real(dp)                :: points_unitcells                     ! número de partículas por celda unidad
18          real(dp)                :: factor1,factor2
19          real(dp)                :: kx,ky,kz                             ! componentes del vector de onda
20          integer(sp)             :: i
21
22          ! valido únicamente para una red FCC
23          points_unitcells=4._dp
24          a=(points_unitcells*(1._dp/density))**(1._dp/3._dp)
25          factor1=0._dp;factor2=0._dp
26          do i=1,n_p
27              ! cambiar en caso de tener otro vector de onda
28              kx=-2._dp*pi*(1._dp/a)*x_vector(i)
29              ky=2._dp*pi*(1._dp/a)*y_vector(i)
30              kz=-2._dp*pi*(1._dp/a)*z_vector(i)
31              factor1=factor1+cos(kx+ky+kz)
32              factor2=factor2+sin(kx+ky+kz)
33          end do
34          factor1=factor1*factor1;factor2=factor2*factor2
35          static_structure_factor=(1._dp/real(n_p*n_p,dp))*(factor1+factor2)
36      end function static_structure_factor
37
38      function pressure(n_p,density,mass,r_cutoff,x_vector,y_vector,z_vector,&
39          vx_vector,vy_vector,vz_vector)
40          integer(sp), intent(in) :: n_p
41          real(dp),    intent(in) :: r_cutoff,density,mass
42          real(dp),    intent(in) :: x_vector(n_p),y_vector(n_p),z_vector(n_p)
43          real(dp),    intent(in) :: vx_vector(n_p),vy_vector(n_p),vz_vector(n_p)
44          real(dp)                :: pressure,rij_pow02,result,T_adim,force_indiv
45          integer(sp)             :: i,j
46          result=0._dp
47          do j=2,n_p
48              do i=1,j-1
49                  rij_pow02=rel_pos_correction(x_vector(i),y_vector(i),z_vector(i),&
50                  x_vector(j),y_vector(j),z_vector(j),n_p,density)
51                  if (rij_pow02<=r_cutoff*r_cutoff) then
52                      force_indiv=f_lj_individual(rij_pow02)
53                  else; force_indiv=0._dp; end if
54                  result=result+force_indiv*rij_pow02
55              end do
56          end do
57          T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
58          pressure=density*(T_adim+(1._dp/(3._dp*real(n_p,dp)))*result)
59      end function pressure
60
61      ! compute individual lennard jones potential (simple truncation)
62      function u_lj_individual(r12_pow02)
63          real(dp), intent(in) :: r12_pow02 ! distancia adimensional entre pares de particulas
64          real(dp)             :: r12_pow06 ! potencias de la distancia relativa
65          real(dp)             :: u_lj_individual     ! adimensional lennard jones potential
66          integer(sp)          :: i
67          ! calculamos distancia relativa corregida según PBC
68          r12_pow06=1._dp
69          do i=1,3;r12_pow06=r12_pow06*r12_pow02;end do  ! (r12)^6
70          u_lj_individual=4._dp*(1._dp/r12_pow06)*((1._dp/r12_pow06)-1._dp)
71      end function u_lj_individual
72      ! compute total lennard jones potential
73      function u_lj_total(n_p,x_vector,y_vector,z_vector,r_cutoff,density)
74          integer(sp), intent(in) :: n_p
75          real(dp),    intent(in) :: x_vector(n_p),y_vector(n_p),z_vector(n_p)
76          real(dp),    intent(in) :: r_cutoff,density
77          real(dp)                :: u_lj_total,u_indiv,rij_pow02
78          integer(sp)             :: i,j
79          u_lj_total=0._dp
80          do j=2,n_p
81              do i=1,j-1
82                  ! calculamos distancia relativa corregida según PBC
83                  rij_pow02=rel_pos_correction(x_vector(i),y_vector(i),z_vector(i),&
84                  x_vector(j),y_vector(j),z_vector(j),n_p,density)
85                  if (rij_pow02<=r_cutoff*r_cutoff) then
86                      u_indiv=u_lj_individual(rij_pow02)
87                  else; u_indiv=0._dp; end if
88                  u_lj_total=u_lj_total+u_indiv
89              end do
90          end do
91      end function u_lj_total
92
93      function kinetic_ergy_total(n_p,vx_vector,vy_vector,vz_vector,mass)
94          integer(sp), intent(in) :: n_p
95          real(dp),    intent(in) :: vx_vector(n_p),vy_vector(n_p),vz_vector(n_p)
96          real(dp),    intent(in) :: mass
97          real(dp)                :: kinetic_ergy_total
98          !integer(sp) :: i
99          kinetic_ergy_total=0.5_dp*mass*(sum(vx_vector(:)*vx_vector(:))+&
100         sum(vy_vector(:)*vy_vector(:))+sum(vz_vector(:)*vz_vector(:)))
101     end function kinetic_ergy_total
```

```
102
103     ! caclulo de la fuerza individual (par de partículas)
104     function f_lj_individual(r12_pow02)
105         real(dp), intent(in) :: r12_pow02   ! distancia adimensional entre pares de particulas
106         real(dp)             :: r12_pow06   ! factores potencia
107         real(dp)             :: f_lj_individual   ! adimensional individual lennard jones force
108         integer(sp)          :: i
109         r12_pow06=1._dp
110         do i=1,3;r12_pow06=r12_pow06*r12_pow02;end do ! (r12)^6
111         f_lj_individual=24._dp*(1._dp/(r12_pow02*r12_pow06))*(2._dp*(1._dp/r12_pow06)-1._dp)
112     end function f_lj_individual
113
114     ! calculo de la componente xi de la fuerza total
115     subroutine f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,&
116         fx_lj_total_vector,fy_lj_total_vector,fz_lj_total_vector)
117         integer(sp), intent(in)    :: n_p
118         real(dp),    intent(in)    :: x_vector(n_p),y_vector(n_p),z_vector(n_p)
119         real(dp),    intent(in)    :: r_cutoff,density
120         real(dp),    intent(inout) :: fx_lj_total_vector(n_p),fy_lj_total_vector(n_p),&
121                                       fz_lj_total_vector(n_p) ! vector de fuerzas netas
122         real(dp)                   :: rij_pow02
123         real(dp)                   :: force_indiv ! fuerza neta acuando en una determinada partícula
124         integer(sp)                :: i,j
125         real(dp)                   :: dx,dy,dz,L
126         fx_lj_total_vector(:)=0._dp;fy_lj_total_vector(:)=0._dp;fz_lj_total_vector(:)=0._dp
127         L=(real(n_p,dp)*(1._dp/density))**(1._dp/3._dp)
128         do j=2,n_p
129             do i=1,j-1
130                 ! calculamos distancia relativa corregida según PBC
131                 rij_pow02=rel_pos_correction(x_vector(i),y_vector(i),z_vector(i),&
132                 x_vector(j),y_vector(j),z_vector(j),n_p,density)
133
134                 if (rij_pow02<=r_cutoff*r_cutoff) then
135                     force_indiv=f_lj_individual(rij_pow02)
136                 else; force_indiv=0._dp;end if
137
138                 dx=pbc_correction((x_vector(i)-x_vector(j)),n_p,density)
139                 dy=pbc_correction((y_vector(i)-y_vector(j)),n_p,density)
140                 dz=pbc_correction((z_vector(i)-z_vector(j)),n_p,density)
141                 ! COMPONENTES DE LA FUERZA
142                 fx_lj_total_vector(i)=fx_lj_total_vector(i)+force_indiv*dx
143                 fx_lj_total_vector(j)=fx_lj_total_vector(j)-force_indiv*dx
144                 fy_lj_total_vector(i)=fy_lj_total_vector(i)+force_indiv*dy
145                 fy_lj_total_vector(j)=fy_lj_total_vector(j)-force_indiv*dy
146                 fz_lj_total_vector(i)=fz_lj_total_vector(i)+force_indiv*dz
147                 fz_lj_total_vector(j)=fz_lj_total_vector(j)-force_indiv*dz
148             end do
149         end do
150     end subroutine f_lj_total
151
152     function temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
153         integer(sp), intent(in) :: n_p
154         real(dp),    intent(in) :: vx_vector(n_p),vy_vector(n_p),vz_vector(n_p)
155         real(dp),    intent(in) :: mass
156         real(dp)                :: temperature
157         temperature=mass*(sum(vx_vector(:)*vx_vector(:))+&
158         sum(vy_vector(:)*vy_vector(:))+sum(vz_vector(:)*vz_vector(:)))*&
159         (1._dp/(3._dp*real(n_p,dp)))
160     end function temperature
161
162     ! corrección de las posiciones relativas (PBC)
163     function rel_pos_correction(x1,y1,z1,x2,y2,z2,n_p,density)
164         integer(sp), intent(in) :: n_p               ! numero total de partículas
165         real(dp),    intent(in) :: density           ! densidad de partículas
166         real(dp),    intent(in) :: x1,y1,z1,x2,y2,z2  ! coordenadas del par de partículas
167         real(dp)                :: rel_pos_correction ! posición relativa corregida (PBC) al cuadrado
168         real(dp)                :: dx,dy,dz          ! diferencia de posiciones segun coordenadas
169         dx=pbc_correction((x1-x2),n_p,density)
170         dy=pbc_correction((y1-y2),n_p,density)
171         dz=pbc_correction((z1-z2),n_p,density)
172         rel_pos_correction=dx*dx+dy*dy+dz*dz
173     end function rel_pos_correction
174
175     function pbc_correction(x,n_p,density)
176         integer(sp), intent(in) :: n_p      ! numero total de partículas
177         real(dp),    intent(in) :: density  ! densidad de partículas
178         real(dp),    intent(in) :: x        ! variable a corregir
179         real(dp)                :: pbc_correction,L
180         L=(real(n_p,dp)*(1._dp/density))**(1._dp/3._dp)
181         pbc_correction=(x-L*anint(x*(1._dp/L),dp))
182     end function pbc_correction
183
184     ! SUBRUTINAS
185     subroutine rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)
186         integer(sp), intent(in)    :: n_p
187         real(dp),    intent(inout) :: vx_vector(n_p),vy_vector(n_p),vz_vector(n_p)
188         real(dp),    intent(in)    :: T_adim_ref,mass
```

```
        real(dp)                    :: T_adim
        T_adim=temperature(n_p,mass,vx_vector,vy_vector,vz_vector)
        vx_vector(:)=sqrt(T_adim_ref*(1._dp/T_adim))*vx_vector(:)
        vy_vector(:)=sqrt(T_adim_ref*(1._dp/T_adim))*vy_vector(:)
        vz_vector(:)=sqrt(T_adim_ref*(1._dp/T_adim))*vz_vector(:)
    end subroutine rescaling_velocities

    ! corrección de las posiciones (PBC)
    subroutine position_correction(n_p,density,x,y,z)
        integer(sp), intent(in)    :: n_p       ! numero total de partículas
        real(dp),    intent(in)    :: density   ! densidad de partículas
        real(dp),    intent(inout) :: x,y,z     ! posiciones
        x=pbc_correction(x,n_p,density)
        y=pbc_correction(y,n_p,density)
        z=pbc_correction(z,n_p,density)
    end subroutine position_correction

    ! Subroutine to set up sc and fcc lattice
    subroutine initial_lattice_configuration(n_p,density,x_vector,y_vector,z_vector,type_structure)
        integer(sp), intent(in)    :: n_p                ! numero total de partículas
        integer(sp), intent(in)    :: type_structure     ! 1 by sc lattice/ 2 by fcc lattice
        real(dp),    intent(in)    :: density            ! densidad de partículas
        real(dp),    intent(inout) :: x_vector(n_p),&    ! coordenadas de los vectores posición
                                      y_vector(n_p),&
                                      z_vector(n_p)
        integer(sp)                :: i,j,k,index2,index ! loop index
        real(dp)                   :: L                  ! logitud macroscópica por dimensión
        real(dp)                   :: a                  ! parámetro de red
        real(dp)                   :: n_unitcells        ! numero de celdas unidad por dimension
        real(dp)                   :: points_unitcells   ! número de partículas por celda unidad
        real(dp),   allocatable    :: aux_matrix(:,:)    ! matriz auxiliar con vectores primitivos (FCC)
        L=(real(n_p,dp)*(1._dp/density))**(1._dp/3._dp)
        select case (type_structure)
            case(1) ! sc laticce
                points_unitcells=1._dp
                n_unitcells=anint((real(n_p,dp)*(1._dp/points_unitcells))**(1._dp/3._dp),dp)
                a=L*(1._dp/(n_unitcells-1._dp))!a=(4*(1._dp/density))**(1._dp/3._dp)
                ! cargamos vectores de coordenadas
                index=0
                do i=1,int(n_unitcells,sp)
                    do j=1,int(n_unitcells,sp)
                        do k=1,int(n_unitcells,sp)
                            index=index+1
                            ! CENTRAMOS LA CELDA EN EL RANGO [-L/2:L/2]
                            x_vector(index)=real(i-1,dp)*a-0.5_dp*L
                            y_vector(index)=real(j-1,dp)*a-0.5_dp*L
                            z_vector(index)=real(k-1,dp)*a-0.5_dp*L
                        end do
                    end do
                end do
            case(2) ! fcc laticce
                points_unitcells=4._dp
                n_unitcells=anint((real(n_p,dp)*(1._dp/points_unitcells))**(1._dp/3._dp),dp)
                a=L*(1._dp/(n_unitcells))!a=(4*(1._dp/density))**(1._dp/3._dp)
                ! cargamos matriz con vectores primitivos (specific for FCC structure)
                allocate(aux_matrix(4,3));aux_matrix(:,:)=a*0.5_dp
                aux_matrix(1,:)=0.0_dp;aux_matrix(2,3)=0.0_dp
                aux_matrix(3,2)=0.0_dp;aux_matrix(4,1)=0.0_dp
                ! cargamos vectores de coordenadas
                index=0
                do i=1,int(n_unitcells,sp)
                    do j=1,int(n_unitcells,sp)
                        do k=1,int(n_unitcells,sp)
                            do index2=1,4
                                index=index+1
                                ! CENTRAMOS LA CELDA EN EL RANGO [-L/2:L/2]
                                x_vector(index)=(aux_matrix(index2,1)+real(i-1,dp)*a)-0.5_dp*L
                                y_vector(index)=(aux_matrix(index2,2)+real(j-1,dp)*a)-0.5_dp*L
                                z_vector(index)=(aux_matrix(index2,3)+real(k-1,dp)*a)-0.5_dp*L
                            end do
                        end do
                    end do
                end do
                deallocate(aux_matrix)
            case(3) ! fcc laticce (other method)
                points_unitcells=4._dp
                n_unitcells=anint((real(n_p,dp)*(1._dp/points_unitcells))**(1._dp/3._dp),dp)
                a=(points_unitcells*(1._dp/density))**(1._dp/3._dp)
                index=0
                do i=0,int(n_unitcells,sp)-1
                    do j=0,int(n_unitcells,sp)-1
                        do k=0,int(n_unitcells,sp)-1
                            index=index+1
                            x_vector(index)=a*real(i,dp)-L*0.5_dp
                            y_vector(index)=a*real(j,dp)-L*0.5_dp
                            z_vector(index)=a*real(k,dp)-L*0.5_dp
                            index=index+1
```

```
                        x_vector(index)=a*(real(i,dp)+0.5_dp)-L*0.5_dp
                        y_vector(index)=a*(real(j,dp)+0.5_dp)-L*0.5_dp
                        z_vector(index)=a*real(k,dp)-L*0.5_dp
                        index=index+1
                        x_vector(index)=a*real(i,dp)-L*0.5_dp
                        y_vector(index)=a*(real(j,dp)+0.5_dp)-L*0.5_dp
                        z_vector(index)=a*(real(k,dp)+0.5_dp)-L*0.5_dp
                        index=index+1
                        x_vector(index)=a*(real(i,dp)+0.5_dp)-L*0.5_dp
                        y_vector(index)=a*real(j,dp)-L*0.5_dp
                        z_vector(index)=a*(real(k,dp)+0.5_dp)-L*0.5_dp
                    enddo
                enddo
            enddo
        end select
    end subroutine initial_lattice_configuration

    ! SUBRUTINA DE INTEGRACIÓN DE ECUACIONES DE MOVIMIENTO
    subroutine velocity_verlet(n_p,x_vector,y_vector,z_vector,&
        vx_vector,vy_vector,vz_vector,&
        delta_time,mass,r_cutoff,density,force_x,force_y,force_z)
        integer(sp), intent(in)    :: n_p
        real(dp),    intent(inout) :: x_vector(n_p),y_vector(n_p),z_vector(n_p)
        real(dp),    intent(inout) :: vx_vector(n_p),vy_vector(n_p),vz_vector(n_p)
        real(dp),    intent(in)    :: delta_time,mass,r_cutoff,density
        real(dp),    intent(inout) :: force_x(n_p),force_y(n_p),force_z(n_p)
        integer(sp)                :: i
        real(dp)                   :: factor
        factor=delta_time*0.5_dp*(1._dp/mass)
        do i=1,n_p
            ! COMPONENTES DE LA VELOCIDAD A TIEMPO SEMI-EVOLUCIONADO
            vx_vector(i)=vx_vector(i)+force_x(i)*factor
            vy_vector(i)=vy_vector(i)+force_y(i)*factor
            vz_vector(i)=vz_vector(i)+force_z(i)*factor
            ! COMPONENTES DE LA POSICION A TIEMPO EVOLUCIONADO
            x_vector(i)=x_vector(i)+vx_vector(i)*delta_time
            y_vector(i)=y_vector(i)+vy_vector(i)*delta_time
            z_vector(i)=z_vector(i)+vz_vector(i)*delta_time
            call position_correction(n_p,density,x_vector(i),y_vector(i),z_vector(i))
        end do
        ! ACTUALIZAMOS COMPONENTES DE LA FUERZA A TIEMPO EVOLUCIONADO
        call f_lj_total(x_vector,y_vector,z_vector,r_cutoff,n_p,density,force_x,force_y,force_z)
        do i=1,n_p
            ! COMPONENTES DE LA VELOCIDAD A TIEMPO EVOLUCIONADO
            vx_vector(i)=vx_vector(i)+force_x(i)*factor
            vy_vector(i)=vy_vector(i)+force_y(i)*factor
            vz_vector(i)=vz_vector(i)+force_z(i)*factor
        end do
    end subroutine velocity_verlet

    ! SUBRUTINA PARA DEFINIR PARAMETRO INICIALES DE DINAMICA MOLECULAR
    subroutine md_initial_parameters(n_p,x_vector,y_vector,z_vector,&
                                     vx_vector,vy_vector,vz_vector,&
                                     T_adim_ref,delta_time,density,mass)

        integer(sp), intent(in)    :: n_p ! number of particles
        real(dp),    intent(inout) :: x_vector(n_p),y_vector(n_p),z_vector(n_p)    ! position vectors
        real(dp),    intent(inout) :: vx_vector(n_p),vy_vector(n_p),vz_vector(n_p) ! velocities vectors
        real(dp),    intent(in)    :: T_adim_ref,delta_time,density,mass ! temperature,time step,densidad,masa
        real(dp)    :: nrand
        integer(sp) :: seed,seed_val(8),i
        real(dp)    :: vx_mc,vy_mc,vz_mc ! velocity center of mass

        call date_and_time(values=seed_val)
        seed=seed_val(8)*seed_val(7)*seed_val(6)+seed_val(5);call sgrnd(seed)
        do i=1,n_p ! give random velocities
            nrand=real(grnd(),dp);vx_vector(i)=(nrand-0.5_dp)
            nrand=real(grnd(),dp);vy_vector(i)=(nrand-0.5_dp)
            nrand=real(grnd(),dp);vz_vector(i)=(nrand-0.5_dp)
        end do
        ! calculamos velocidad del centro de masa
        vx_mc=sum(vx_vector(:))*(1._dp/real(n_p,dp))
        vy_mc=sum(vy_vector(:))*(1._dp/real(n_p,dp))
        vz_mc=sum(vz_vector(:))*(1._dp/real(n_p,dp))
        ! velocity center of mass to zero and rescaling
        vx_vector(:)=vx_vector(:)-vx_mc
        vy_vector(:)=vy_vector(:)-vy_mc
        vz_vector(:)=vz_vector(:)-vz_mc
        call rescaling_velocities(n_p,vx_vector,vy_vector,vz_vector,T_adim_ref,mass)

        ! descomentar si se quere re-asignar posiciones según velocidades iniciales
        ! do i=1,n_p
        !     ! position previous time step
        !     x_vector(i)=x_vector(i)-vx_vector(i)*delta_time
        !     y_vector(i)=y_vector(i)-vy_vector(i)*delta_time
        !     z_vector(i)=z_vector(i)-vz_vector(i)*delta_time
        !     ! corregimos posiciones según PBC
```

```
363         !     call position_correction(n_p,density,x_vector(i),y_vector(i),z_vector(i))
364         ! end do
365     end subroutine md_initial_parameters
366 
367     ! SUBRUTINA PARA CALCULAR LA FUNCION DE DISTRIBUCIÓN RADIAL (FUNCION DE CORRELACIÓN)
368     subroutine radial_ditribution_function(file_name,n_p,density,x_vector,y_vector,z_vector,n_bins,g)
369         character(len=*), intent(in)    :: file_name                                ! nombre del archivo de datos
370         integer(sp),      intent(in)    :: n_p,n_bins                               ! numero de partículas y numero total de
    bins
371         real(dp),         intent(in)    :: x_vector(n_p),y_vector(n_p),z_vector(n_p) ! coordenadas del vector posicion
372         real(dp),         intent(in)    :: density                                  ! densidad de partículas
373         real(dp),         intent(inout) :: g(n_bins)                                ! funcion distribución
374 
375         integer(sp) :: ngr              ! contador de particulas
376         real(dp)    :: rij_pow02,rij    ! distancia relativa entre par de partículas
377         real(dp)    :: L                ! logitud macroscópica
378         real(dp)    :: step_bins        ! tamaño de bins
379         real(dp)    :: volume           ! volumen de particulas
380         real(dp)    :: nid              ! numero de partículas del gas ideal en volume
381         integer(sp) :: i,j,index,istat  ! loop variables
382 
383         ! initialization
384         L=(real(n_p,dp)*(1._dp/density))**(1._dp/3._dp)
385         step_bins=L/(2*n_bins)
386         g(:)=0._dp
387 
388         ! sample
389         ngr=0_sp;ngr=ngr+1_sp
390         do j=2,n_p;do i=1,j-1
391             ! relative distances (pow 2) with pbc correction
392             rij_pow02=rel_pos_correction(x_vector(i),y_vector(i),z_vector(i),&
393             x_vector(j),y_vector(j),z_vector(j),n_p,density)
394             ! relative distances
395             rij=sqrt(rij_pow02)
396             if (rij<=L/2) then ! only within half the box length
397                 index=int(rij/step_bins)
398                 g(index)=g(index)+2._dp ! contribution for particle i and j
399             end if
400         end do;end do
401 
402         ! result initialization (determine g(rij))
403         open(100,file=file_name,status='replace',action='write',iostat=istat)
404             if (istat/=0) write(*,*) 'ERROR! istat(11file) = ',istat
405             101 format(E12.4,x,E12.4);102 format(A12,x,A12)
406             write(100,102) 'rij','g(rij)'
407             do i=1,n_bins
408                 rij=step_bins*(i+0.5_dp)
409                 ! volumen between bin i+1 and bin i
410                 volume=(real(i+1,dp)**3-real(i,dp)**3)*(step_bins**3)
411                 ! number of ideal gas particles in volume
412                 nid=(1._dp/3._dp)*16._dp*atan(1._dp)*volume*density
413                 ! normalize g(rij)
414                 g(i)=g(i)/(real(ngr*n_p,dp)*nid)
415                 write(100,101) rij,g(i)
416             end do
417         close(100)
418     end subroutine radial_ditribution_function
419 
420 end module module_md_lennard_jones
```