Proyectos de simulación

Salazar C. Edgar M.

Primavera 2018

Índice general

Capítulo 1

Exploración de la ecuación de Van der Waals

Introducción

La ecuación de estado de Van der Waals

$$\left(P + \frac{N^2}{V^2}a\right)(V - Nb) = Nk_BT \tag{1.1}$$

$$a = -2\pi \int_\sigma^\infty u^{(1)}(r)\, r^2 dr \tag{1.2}$$

$$b = \frac{2}{3}\pi\sigma^3 \tag{1.3}$$

Donde σ es el diámetro de la esfera dura que corresponde al potencial de interacción *repulsivo* del sistema de referencia; y $u^{(1)}(r)$ el potencial de interacción *atractivo* que se considera como una perturbación al potencial de interacción entre las partículas.

$$u(r) = u^{(\mathrm{ND})}(r) + u^{(1)}(r) \tag{1.4}$$

Contexto

Para la obtención de la ecuación (1.2) se asume que la función de distribución radial para núcleos duros (ND).[1]

$$g_{\mathrm{ND}}(r) \approx \begin{cases} 0 & r \leqslant \sigma \\ 1 & r > \sigma \end{cases} \tag{1.5}$$

Claramente esta aproximación es para muy bajas concentraciones, donde la función de distribución radial de contacto $g_{\mathrm{ND}}(\sigma^+) \approx 1$, que corresponde al modelo (1.1) de Van der Waals.

En el contexto del ensemble canónico en la Física Estadística, la ecuación de estado se obtiene a partir de:

$$P = \frac{1}{\beta}\left(\frac{\partial \ln Z_N}{\partial V}\right)_T \tag{1.6}$$

[1] Aquí usaremos ND para hacer referencia al modelo de núcleo duro (HS o HD en inglés).

Como $g_{\text{ND}}(r)$ depende de la concentración, la ecuación de estado es de la forma

$$P = P_{\text{ND}} - \rho^2 \left[a(\rho) + \rho \left(\frac{\partial a(\rho)}{\partial \rho} \right)_T \right] \tag{1.7}$$

$$P_{\text{ND}} = \rho k_B T \left[1 + \frac{2}{3} \pi \sigma^3 \rho \, g_{\text{ND}}\left(\sigma^+\right) \right] \tag{1.8}$$

Metodología

La teoría de perturbaciones de Zwanzig no impone ninguna restricción sobre el potencial perturbativo $u^{(1)}(r)$, con tal que sea un potencial atractivo. Consideremos como potencial la parte atractiva del modelo de pozo cuadrado (PC) que depende de la distancia r entre partículas[2].

$$u^{(1)}(r) = \begin{cases} -\varepsilon & \sigma < r < \lambda\sigma \\ 0 & r \geqslant \lambda\sigma \end{cases} \tag{1.9}$$

Donde λ es un parámetro adimensional que escala el alcance del pozo. Procederemos a la adimensionalización o reducción de variables de modo que estemos en condiciones de implementar las simulaciones para los cálculos numéricos necesarios. Tomaremos como longitud característica al **diámetro** σ de las esferas duras[3] del sistema de referencia[4], y como energía característica a la **energía térmica** $k_B T = \beta^{-1}$.

1.1. Potencial

Multiplicamos la ecuación (1.9) por β (unidades de energía) y dividimos por σ (unidades de longitud) a la variable radial r.

$$\beta u^{(1)}(r) = \begin{cases} -\beta\varepsilon & \sigma < \dfrac{r}{\sigma} < \dfrac{\lambda\sigma}{\sigma} \\ 0 & \dfrac{r}{\sigma} \geqslant \frac{\lambda\sigma}{\sigma} \end{cases}$$

Definimos:

$$u^{*(1)}(r^*) = \beta u^{(1)}(r)$$

$$T^* = \frac{1}{\beta\varepsilon}$$

$$r^* = \frac{r}{\sigma}$$

Así, el potencial reducido está listo para la programación.

$$u^{*(1)}(r^*) = \begin{cases} -\dfrac{1}{T^*} & 1 < r^* < \lambda \\ 0 & r^* \geqslant \lambda \end{cases} \tag{1.10}$$

[2]En lo sucesivo se asume que la simulación es tridimensional.

[3]Hasta aquí solo hemos dicho que el núcleo es impenetrable y esto se refleja en la forma de escribir el potencial. Sin embargo la dependencia radial permite trabajar la forma (1.4) para discos como para esferas.

[4]Usamos esferas pues trabajaremos con un modelo en tres dimensiones, además de ser la forma más simple de sistema de referencia.

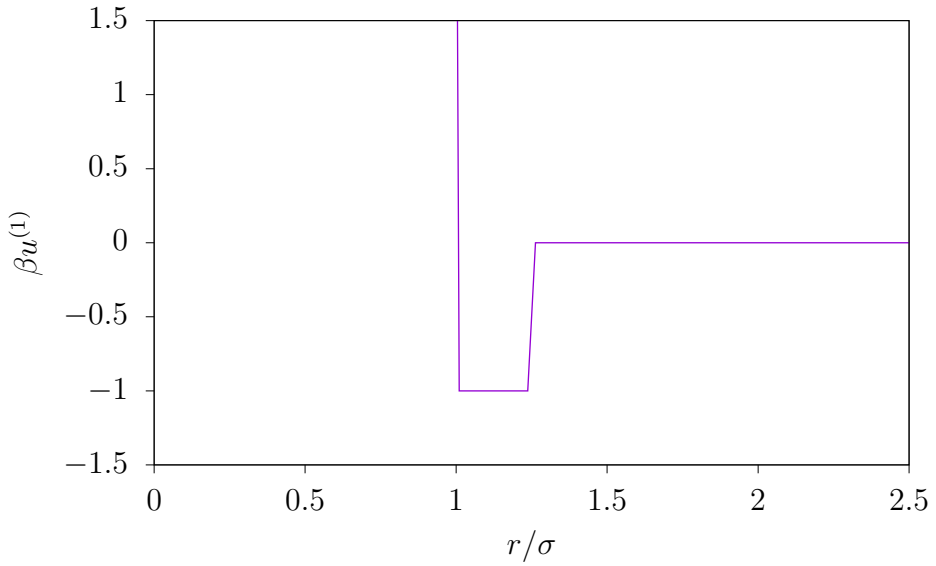


Figura 1.1: Potencial de ND y PC con alcance $\lambda = 1.25$ y profundidad $T^* = 1$.

1.2. Ecuación de estado

Para reducir la presión y la densidad volumétrica definimos a las siguientes cantidades:

$$P^* = \beta\sigma^3 P \qquad \rightarrow \qquad P = \frac{P^*}{\beta\sigma^3} \tag{1.11}$$

$$n^* = \sigma^3\rho \qquad \rightarrow \qquad \rho = \frac{n^*}{\sigma^3} \tag{1.12}$$

Ahora reducimos la ecuación (1.8) usando estas definiciones.

$$\begin{aligned}
P_{\text{ND}} &= \rho k_B T \left[1 + \frac{2}{3}\pi\sigma^3\rho\, g_{\text{ND}}\left(\sigma^+\right)\right] \\
\beta\, P_{\text{ND}} &= \rho \left[1 + \frac{2}{3}\pi\sigma^3\rho\, g_{\text{ND}}\left(\sigma^+\right)\right] \\
\beta\left(\frac{P^*_{\text{ND}}}{\beta\sigma^3}\right) &= \frac{n^*}{\sigma^3}\left[1 + \frac{2}{3}\pi\sigma^3\frac{n^*}{\sigma^3} g_{\text{ND}}\left(1^+\right)\right] \\
P^*_{\text{ND}} &= n^*\left[1 + \frac{2}{3}\pi n^* g_{\text{ND}}\left(1^+\right)\right] \\
P^*_{\text{ND}} &= n^*\left[1 + n^* b^*\right]
\end{aligned} \tag{1.13}$$

Donde hemos definido (a la Zwanzig)

$$b^* = \frac{2}{3}\pi g_{\text{ND}}\left(1^+\right) \tag{1.14}$$

Para la ecuación completa (1.7) hacemos

$$
\begin{aligned}
P &= P_{\text{ND}} - \rho^2 \left[a\left(\rho\right) + \rho \left(\frac{\partial a(\rho)}{\partial \rho} \right)_T \right] \\
\frac{P^*}{\beta\sigma^3} &= \frac{P^*_{\text{ND}}}{\beta\sigma^3} - \frac{(n^*)^2}{\sigma^6} \left[a\left(\rho\right) + \frac{n^*}{\sigma^3} \left(\frac{\partial a(\rho)}{\partial \frac{n^*}{\sigma^3}} \right)_T \right] \\
P^* &= P^*_{\text{ND}} - \frac{\beta(n^*)^2}{\sigma^3} \left[a\left(n^*\right) + n^* \left(\frac{\partial a(n^*)}{\partial n^*} \right)_{T^*} \right] \\
P^* &= P^*_{\text{ND}} - (n^*)^2 \left[\frac{\beta}{\sigma^3} a\left(n^*\right) + n^* \left(\frac{\partial \frac{\beta}{\sigma^3} a(n^*)}{\partial n^*} \right)_{T^*} \right] \\
P^* &= P^*_{\text{ND}} - (n^*)^2 \left[a^*\left(n^*\right) + n^* \left(\frac{\partial a^*(n^*)}{\partial n^*} \right)_{T^*} \right]
\end{aligned} \tag{1.15}
$$

Donde hemos definido

$$
a^* = \frac{\beta}{\sigma^3} a\left(n^*\right) \tag{1.16}
$$

Utilizando esta expresión en (1.2) (pero a la Zwanzig) obtenemos una reducción para a:

$$
\begin{aligned}
a &= -2\pi \int_\sigma^\infty u^{(1)}\left(r\right)\, g_{\text{ND}}\left(r\right) r^2 dr \\
\frac{\sigma^3}{\beta} a^* &= -2\pi \int_\sigma^\infty u^{(1)}\left(r\right)\, g_{\text{ND}}\left(r\right) r^2 dr \\
a^* &= -2\pi \int_\sigma^\infty \left[\beta u^{(1)}\left(r\right)\right]\, g_{\text{ND}}\left(r^*\right) \left(\frac{r}{\sigma}\right)^2 d\left(\frac{r}{\sigma}\right) \\
a^* &= -2\pi \int_1^\infty \left[u^{*(1)}\left(r^*\right)\right]\, g_{\text{ND}}\left(r^*\right) \left(r^*\right)^2 dr^*
\end{aligned} \tag{1.17}
$$

Tomando el caso de Van der Waals ($g_{\text{ND}}\left(r^*\right) = 1$) se reduce a:

$$
a^* = -2\pi \int_1^\infty \left[u^{*(1)}\left(r^*\right)\right] \left(r^*\right)^2 dr^* \tag{1.18}
$$

1.3. Parámetros Críticos

1.3.1. Relación con los coeficientes de Van der Waals (termo clásica)

La ecuación de Van der Waals (1.1) muestra una temperatura crítica T_c:

$T > T_c$ La ecuación de estado $P - V$ es monovaluada y no hay transiciones al estado líquido.

$T < T_c$ Como la ecuación es cúbica en V, tiene dos extremales que se juntan en $T = T_c$.

Pensemos en la ecuación de Van der Waals como una función del volumen a una temperatura dada $P = P(V;T)$.

- Para temperaturas $T < T_c$ tendremos dos extremales para los cuales se cumple $(\partial_V P)_T = 0$.
- A medida que $T \to T_c$ tenemos un punto de inflexión con coordenadas (P_c, V_c, T_c) para el cual la segunda derivada se desvanece de modo que también se cumple $(\partial_V^2 P)_T = 0$.

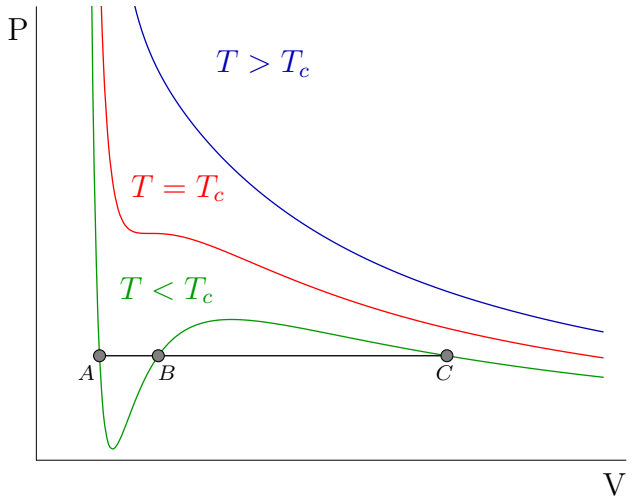


Figura 1.2: Diagrama de fases cualitativo de Van der Waals donde se muestran tres isotermas. Realmente el sistema sigue el camino ABC pues los puntos extremos son inestables.

Partiendo de (1.1):

$$\begin{aligned} P &= \frac{Nk_BT}{V - Nb} - \frac{aN^2}{V^2} \\ \left(\frac{\partial P}{\partial V}\right)_T &= -\frac{Nk_BT}{(V - Nb)^2} + \frac{2aN^2}{V^3} \\ \left(\frac{\partial^2 P}{\partial V^2}\right)_T &= \frac{2Nk_BT}{(V - Nb)^3} - \frac{6aN^2}{V^4} \end{aligned} \tag{1.19}$$

Aplicamos la condición en los puntos críticos y obtenemos el siguiente sistema de ecuaciones para los coeficientes a y b.

$$\begin{bmatrix} -\dfrac{Nk_BT_c}{(V_c - Nb)^2} + \dfrac{2aN^2}{V_c^3} & = 0 \\ \dfrac{2Nk_BT_c}{(V_c - Nb)^3} - \dfrac{6aN^2}{V_c^4} & = 0 \end{bmatrix} \tag{1.20}$$

Multiplicamos a la primera ecuación de (1.20) por $\frac{2}{V_c - Nb}$ y sumamos ambas

$$\begin{aligned} \frac{2aN^2}{V_c^3}\left(\frac{2}{V_c - Nb} - \frac{3}{V_c}\right) &= 0 \\ \frac{2}{V_c - Nb} &= \frac{3}{V_c} \end{aligned}$$

$$V_c = 3Nb \tag{1.21}$$

$$b = \frac{V_c}{3N} \tag{1.22}$$

Usando este resultado (1.21) en la primera ecuación (1.20):

$$-\frac{Nk_BT_c}{(3Nb - Nb)^2} + \frac{2aN^2}{27N^3b^3} = 0$$

$$\frac{k_BT_c}{4} = \frac{2a}{27b}$$

$$T_c = \frac{8a}{27k_Bb} \tag{1.23}$$

Usando (1.22) obtenemos

$$a = \frac{9}{8}k_BT_c\frac{V_c}{N} \tag{1.24}$$

Finalmente, haciendo uso de (1.21) y (1.23) la presión dada por los coeficientes a y b queda:

$$P_c = \frac{a}{27b^2} \tag{1.25}$$

O bien

$$P_c = \frac{3}{8}\frac{Nk_BT_c}{V_c} \tag{1.26}$$

1.3.2. Potencial perturbativo (con Zwanzig)

Ahora obtendremos las expresiones para las variables termodinámicas críticas de Van der Waals para el caso de estudio con un potencial perturbativo de pozo cuadrado (1.9). Para ello utilizaremos la expresión para el coeficiente a^* (1.18) y el potencial reducido (1.10).

$$\begin{aligned}
a^* &= -2\pi\int_1^\infty \left[u^{*(1)}\left(r^*\right)\right]\, g_{\text{ND}}\left(r^*\right)\left(r^*\right)^2 dr^* \\
&= -2\pi\left[\int_1^\lambda \left[u^{*(1)}\left(r^*\right)\right]\, g_{\text{ND}}\left(r^*\right)\left(r^*\right)^2 dr^* + \int_\lambda^\infty \left[u^{*(1)}\, g_{\text{ND}}\left(r^*\right)\left(r^*\right)\right]\left(r^*\right)^2 dr^*\right] \\
&= -2\pi\int_1^\lambda -\frac{1}{T^*}\, g_{\text{ND}}\left(r^*\right)\left(r^*\right)^2 dr^* \\
&= \frac{2\pi}{T^*}\int_1^\lambda g_{\text{ND}}\left(r^*\right)\left(r^*\right)^2 dr^*
\end{aligned} \tag{1.27}$$

Para el caso de Van der Waals integramos y queda

$$a^* = \frac{2\pi}{3}\,\frac{\lambda^3 - 1}{T^*} \tag{1.28}$$

Para el coeficiente b^* tenemos ya la ecuación (1.14) que depende del potencial. Si tomamos la suposición sobre el modelo de Van der Waals (gas ideal), entonces obtenemos (1.3), que en su forma reducida queda:

$$b^* = \frac{2}{3}\pi \tag{1.3}$$

Una vez reducida. De (1.21), (1.23) y (1.25) se obtiene

$$V_c^* = 2\pi N \tag{1.29}$$

$$T_c^* = \frac{8}{27}\left(\lambda^3 - 1\right) \tag{1.30}$$

$$P_c^* = \frac{1}{18\pi}\left(\lambda^3 - 1\right)\frac{1}{T^*} \tag{1.31}$$

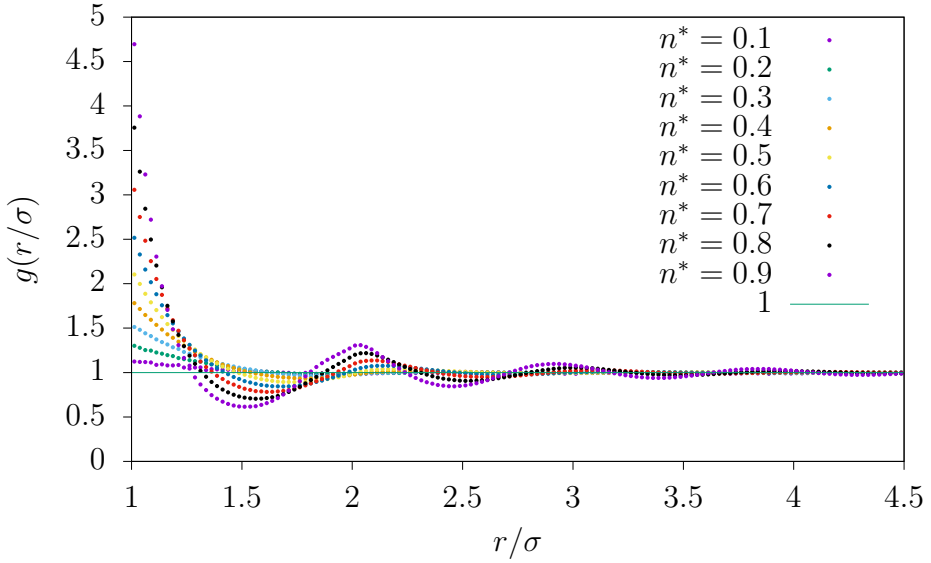
■ **Ejemplo** Veamos de qué orden son los valores para las variables críticas para un sistema dado. Sea $\lambda = 1.25$, resulta que $(\lambda^3 - 1) = 0.953125$, así que:

$$V_c^* \sim 6.2832\, N$$
$$T_c^* \sim 0.2824$$
$$P_c^* \sim \frac{0.0168}{T^*}$$

Para este proyecto se eligió $T^* = 1$, de modo que simplifique el cálculo de propiedades. ■

1.4. Función de distribución radial ND

Como sabemos la función de distribución radial $g(r)$ brinda información de estructura. Es de esperar que para $n^* \to 0$, $g(r) \to 1$, que implica descorrelación entre partículas. Esto es que las partículas están tan alejadas unas de otras que no hay «vecinas», por lo que no forman estructura definida. Ahora bien, a medida que aumenta n^* esperamos que el empaquetamiento permita la formación de estructuras pues la interacción entre partículas es mayor. A continuación se muestra un gráfico de $g(r) - n^*$ para el modelo de potencial ND solamente.



Se puede observar que en el punto de contacto (una distancia σ) hay mayor número de partículas a medida que aumenta la concentración. Las sinuosidades de $g(r)$ indican la presencia de «vecinos» segundos, terceros, etc., hasta alcanzar el bulto a mayores concentraciones.

1.5. Ecuación de presión ND

Como sabemos, una ecuación de estado muy utilizada en la termodinámica es la de presión contra volumen, en este caso $P^* - n^*$. Para el estudio de la presión con el modelo ND tenemos ya la ecuación (1.13) junto con (1.14).

$$P^*_{\text{ND}}\left(n^*\right) = n^*\left[1 + \frac{2\pi}{3} n^* g_{\text{ND}}\left(1^+\right)\right] \tag{1.13}$$

Dado que n^* es elegida por el simulador, basta con obtener el valor de contacto $g_{\text{ND}}\left(1^+\right)$. La ecuación de estado de Carnahan-Starling para un sistema de esferas duras está dada por:

$$P^*\left(n^*\right) = n^* \frac{1 + \phi + \phi^2 - \phi^3}{(1-\phi)^3} \qquad , \qquad \phi = \frac{\pi}{6} n^* \tag{1.32}$$

A continuación se muestra una gráfica con ambas ecuaciones de estado

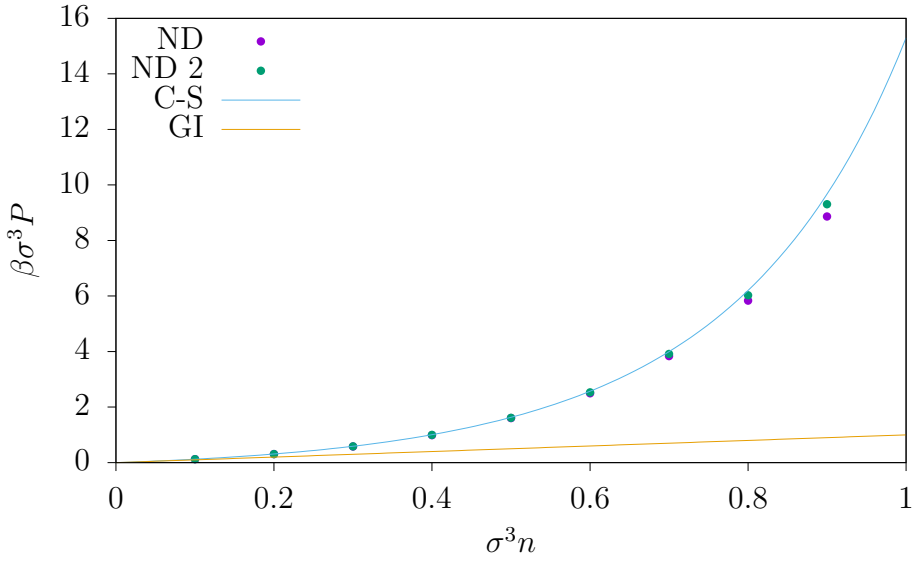


Figura 1.3

La diferencia entre ND y ND 2 es el tiempo de simulación. Para el caso ND se utilizaron 5×10^3 configuraciones; mientras que para ND 2 se simularon 1×10^6. Dado que para concentraciones de 0.7 en adelante se utilizó una configuración inicial regular, simular pocas configuraciones implica que el sistema puede continuar en su arreglo cristalino. De esta manera la estadística mejora permitiendo que se olvide de su situación inicial. Se puede apreciar que las simulaciones tienden a la expresión exacta C-S (1.32).

1.6. Coeficientes a y b

A partir de las ecuaciones (1.27) y (1.14) calculamos los valores para a^* y b^* haciendo uso de la información estructural obtenida de las simulaciones (de las funciones $g(r)$).

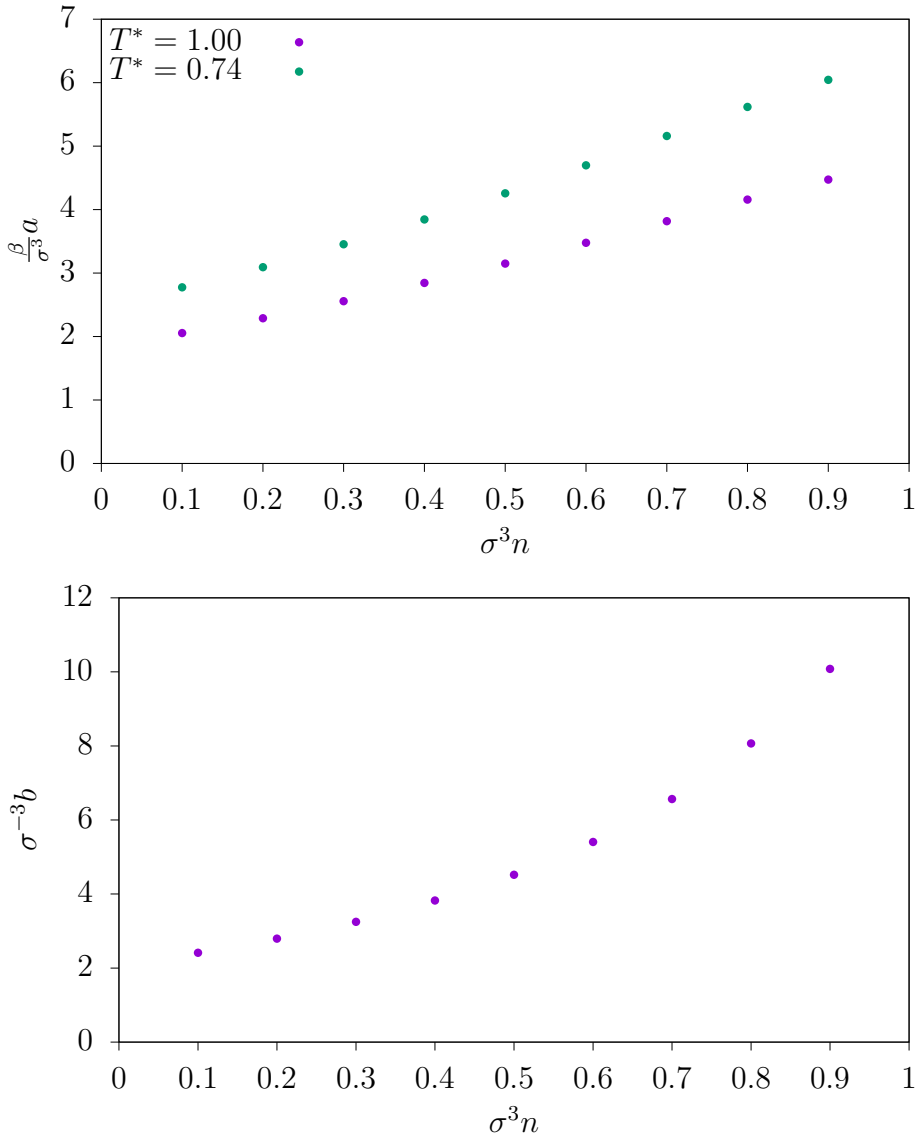
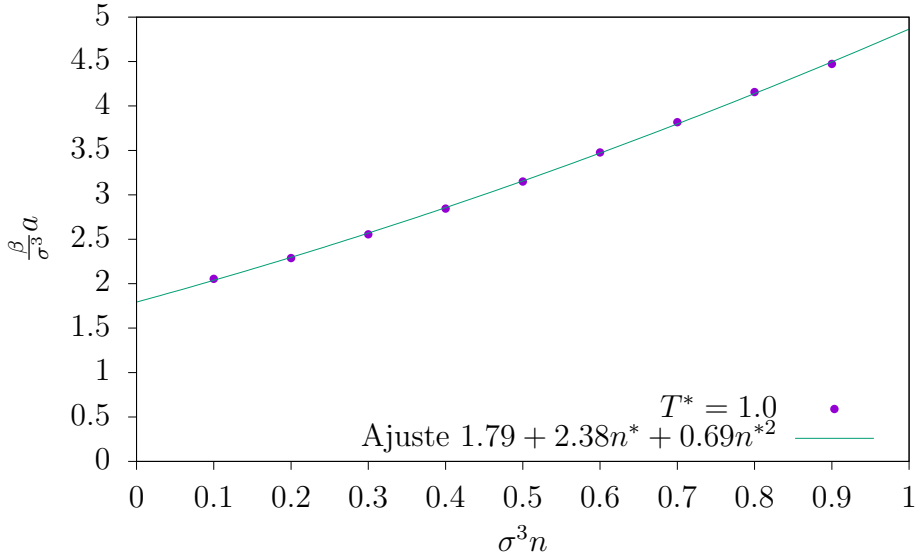


Figura 1.4

1.7. Ajuste para a

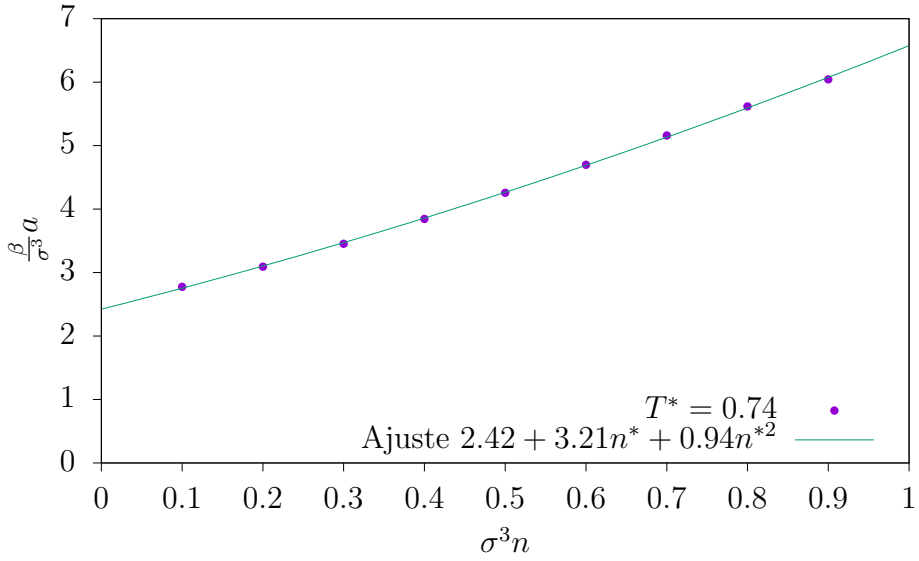
De los resultados para a se puede observar una ligera curvatura para concentraciones bajas, de modo que podemos pensar en ajustar una función polinomial de segundo orden.



Expresando a como función de la concentración reducida tenemos:

$$a^*(n^*) = 0.69n^{*2} + 2.38n^* + 1.79$$
$$\partial_{n^*} a^* = 0.69n^* + 2.38$$

Para el otro caso de temperatura reducida:



Expresando a como función de la concentración reducida tenemos:

$$a^*(n^*) = 0.94n^{*2} + 3.21n^* + 2.42$$
$$\partial_{n^*} a^* = 0.94n^* + 3.21$$

1.8. Ecuación de presión con la teoría de perturbaciones de Zwanzig

A partir de la presión para ND y utilizando (1.15):

$$P^* = P^*_{\text{ND}} - (n^*)^2 \left[a^*(n^*) + n^* \left(\frac{\partial a^*(n^*)}{\partial n^*} \right)_{T^*} \right] \tag{1.15}$$

Pasamos a calcular la presión utilizando los dos casos de T^* que llevamos hasta el momento.

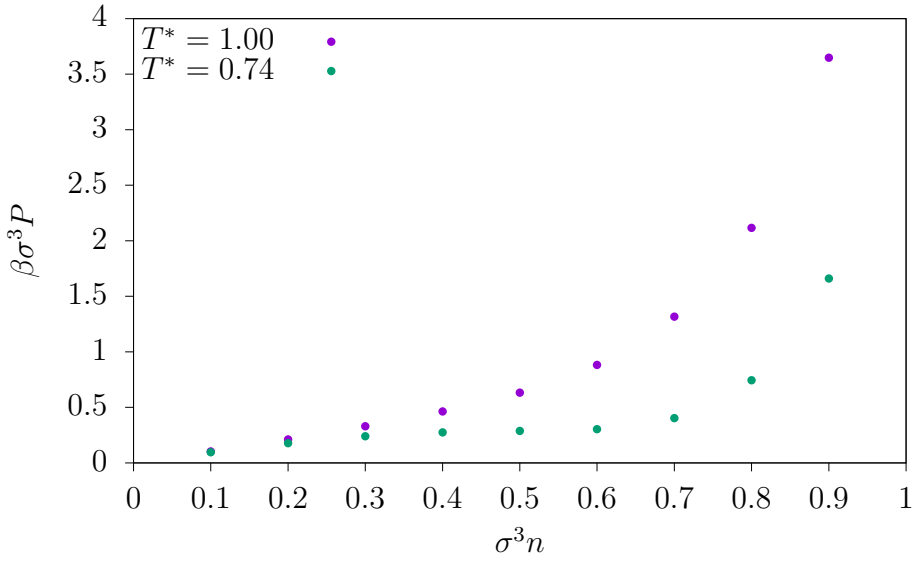


Figura 1.5

Como caso comparativo, incluimos las ecuaciones de presión de Van der Waals, de gas ideal y de esferas duras, junto con la que se obtuvo de la teoría de perturbaciones de Zwanzig. En este caso se compara tomando $T^* = 0.74$.

Modelo	Ecuación de presión
Gas ideal	n^*
ND	$n^* \left[1 + \frac{2\pi}{3} n^* g_{\text{ND}}(1^+) \right]$
Van der Waals	$\frac{n^*}{1 - n^* b^*} - n^{*2} a^*$
Zwanzig	$P^*_{\text{ND}} - (n^*)^2 \left[a^*(n^*) + n^* \left(\frac{\partial a^*(n^*)}{\partial n^*} \right)_{T^*} \right]$

De este cuadro comparativo notamos varias cosas:

- Todos los modelos coinciden para concentraciones muy bajas, i.e. el caso límite es el modelo de gas ideal.
- El modelo de Van der Waals es asintótico en $1/b^*$ por lo que al crecer la concentración a partir de cero, éste diverge del modelo de gas ideal rápidamente.

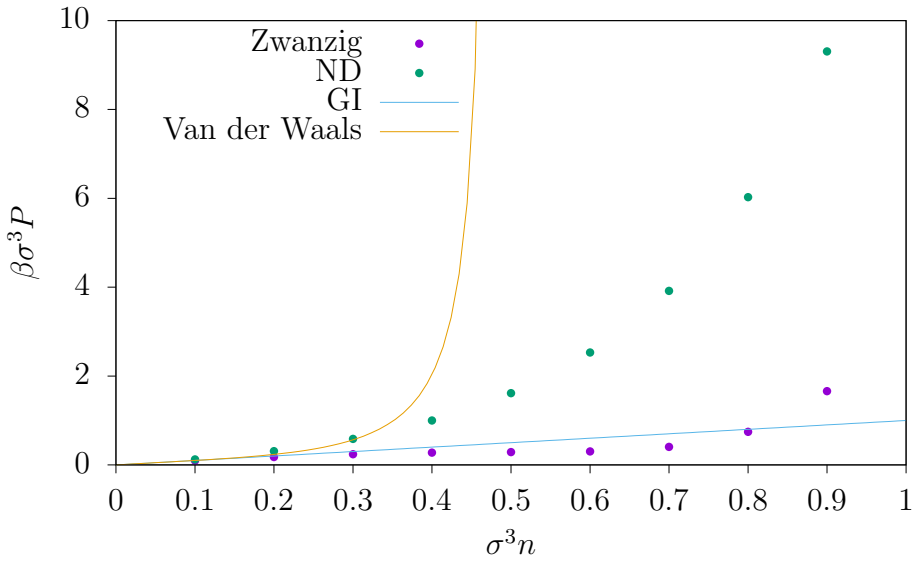


Figura 1.6

- El modelo de ND tiene un comportamiento creciente pero aparentemente regular.
- La teoría de perturbaciones de Zwanzig muestra que la presión se mantiene relativamente cercana a la de gas ideal, aunque es curioso que presenta cambio en la concavidad.

1.9. Energía promedio

De la teoría física estadística con la propuesta de Zwanzig para la función de partición es posible obtener una expresión para la energía media.

$$Z_N \approx (V - Nb)^N \, e^{-\beta N a^* n^*} \tag{1.33}$$

Siendo la energía media por partícula:

$$\bar{u} = \frac{1}{N} \frac{\partial \ln Z_N}{\partial \beta} = -a^* n^* \tag{1.34}$$

En la figura 1.7 se grafica la energía media por partícula para las dos isotermas que venimos analizando.

1.10. Validación de modelos teóricos

Ahora implementamos el código de simulación Monte Carlo para el sistema de esferas atractivas con pozo cuadrado para obtener la presión y la energía media por partícula en función de la concentración reducida para las isotermas, tomando $\lambda = 1.25$. Los resultados se incluyen a continuación.

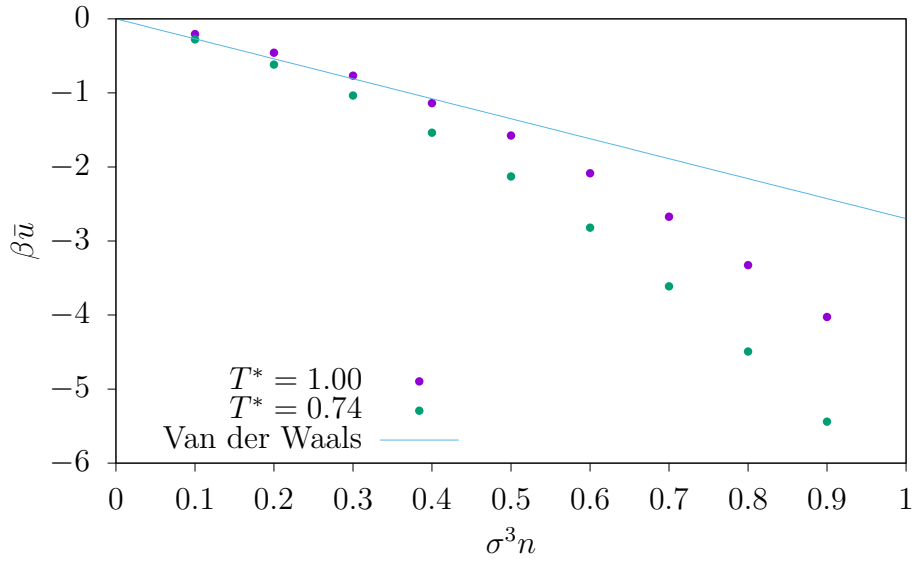


Figura 1.7

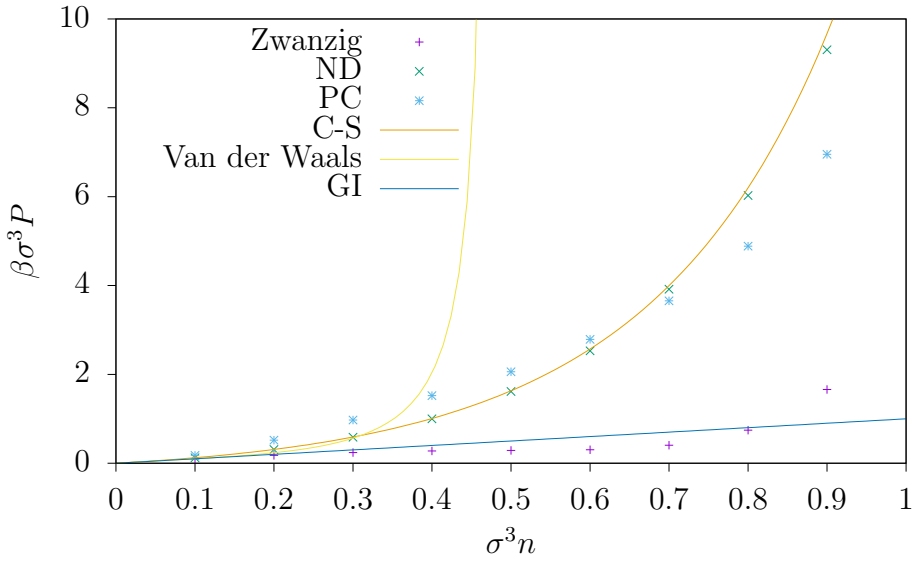


Figura 1.8

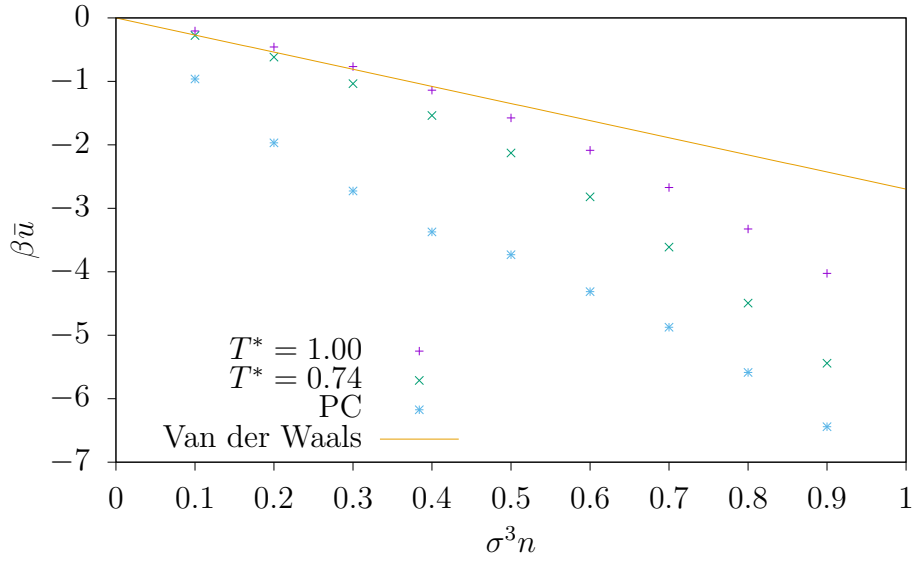

0
−1
−2
−3
−4
−5
−6
−7
$\beta\bar{u}$
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
$\sigma^3 n$
$T^* = 1.00$
$T^* = 0.74$
PC
Van der Waals

Figura 1.9

Capítulo 2

Exploración de propiedades estructurales y autodifusión de sistemas coloidales con un modelo de interacción FRAC

Introducción

A los modelos de potencial que tienen una parte atractiva y otra repulsiva se les conoce como modelos FRAC (Finite interparticle Repulsion with a longer range Attractive Component). En este proyecto se explorarán las propiedades estructurales y dinámicas de un sistema modelo de partículas que interaccionan entre sí con un potencial de interacción par FRAC de doble gaussiana (DG) como el siguiente:

$$u(r) = \epsilon e^{-\left(\frac{r}{\sigma}\right)^2} - \eta e^{-\left(\frac{r-\xi}{\sigma}\right)^2} \tag{2.1}$$

O bien

$$u(r) = \epsilon \left[e^{-\left(\frac{r}{\sigma}\right)^2} - \mu e^{-\left(\frac{r-\xi}{\sigma}\right)^2} \right] \qquad , \qquad \mu = \frac{\eta}{\epsilon} \tag{2.2}$$

Donde los parámetros son definidos positivos; ϵ y η son de carácter energético, y σ y ξ de longitud. Como puede verse en (2.1), corresponde a una superposición de una gaussiana repulsiva (positiva) y otra atractiva (negativa). Como puede verse, $\eta \to 0$ se recupera el caso del modelo de núcleo gaussiano (GCM[1]) característico de repulsiones ultrasuaves.

2.1. Potencial

Antes de implementar las simulaciones reducimos el potencial de interacción (2.1) tomando como parámetros característicos $\{\sigma, \beta^{-1}\}$. Primero definimos las siguientes cantidades adimensio-

[1]Gaussian Core Model por sus siglas en inglés

nales.

$$r^* = \frac{r}{\sigma} \qquad \rightarrow \qquad r = \sigma r^* \tag{2.3}$$

$$\xi^* = \frac{\xi}{\sigma} \qquad \rightarrow \qquad \xi = \sigma \xi^* \tag{2.4}$$

$$\epsilon^* = \beta\epsilon \qquad \rightarrow \qquad \epsilon = \frac{\epsilon^*}{\beta} \tag{2.5}$$

$$\tag{2.6}$$

Partimos de (2.2) aprovechando que μ es un parámetro adimensional.

$$\begin{aligned} u(r) &= \epsilon \left[e^{-\left(\frac{r}{\sigma}\right)^2} - \mu e^{-\left(\frac{r-\xi}{\sigma}\right)^2} \right] \\ \beta u\left(r^*\right) &= \beta\epsilon \left[e^{-\left(\frac{\sigma r^*}{\sigma}\right)^2} - \mu e^{-\left(\frac{\sigma r^* - \sigma\xi^*}{\sigma}\right)^2} \right] \\ u^*\left(r^*\right) &= \epsilon^* \left[e^{-r^{*2}} - \mu e^{-(r^* - \xi^*)^2} \right] \end{aligned} \tag{2.7}$$

O bien

$$u^*\left(r^*\right) = \frac{1}{T^*} \left[e^{-r^{*2}} - \mu e^{-(r^* - \xi^*)^2} \right] \tag{2.8}$$

Para este proyecto se trabajará con los siguientes valores de parámetros:

Parámetro	Valor
μ (GCM)	0
μ (DG)	0.0263
T^*	0.01
ξ^*	3.0

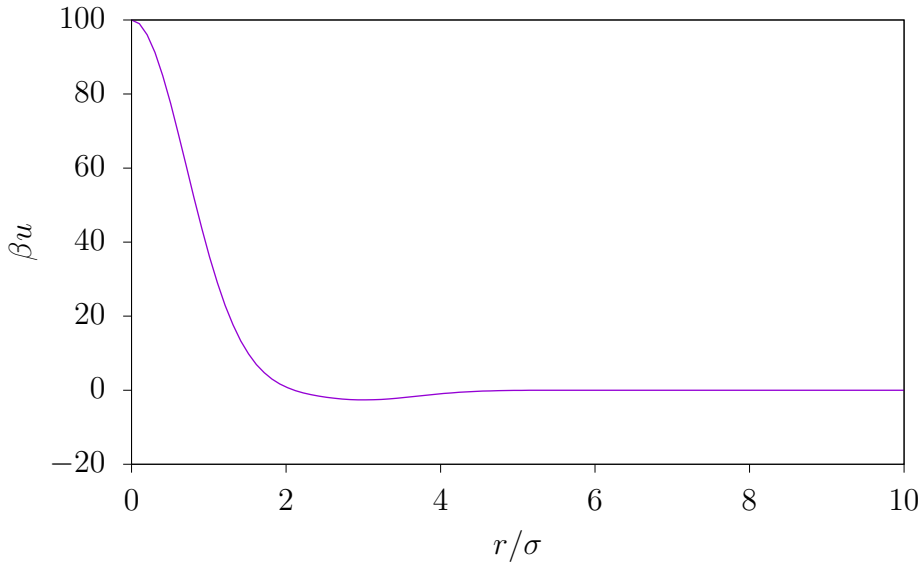


Figura 2.1: Potencial DG con $T^* = 0.01$, $\xi^* = 3.0$ y $\mu = 0.0263$.

Capítulo 3

Simulación con dinámica molecular para un sistema monodisperso de partículas que interaccionan con el potencial de Lennard-Jones

Introducción

En la simulación de dinámica molecular (DM) lo esencial es resolver las ecuaciones de movimiento de el sistema de N partículas, ya sea en el enfoque de Newton con N ecuaciones diferenciales de segundo orden o en el de Hamilton con $2N$ ecuaciones diferenciales de primer orden.

El algoritmo más utilizado es el de Verlet (1967) que es un método de solución numérica para la ecuación de Newton, y se resume en el siguiente algoritmo:

$$\vec{r}\,(t+\delta t) = 2\vec{r}\,(t+\delta t) - \vec{r}\,(t+\delta t) + \vec{a}\,(t)\,(\delta t)^2 \tag{3.1}$$

En este algoritmo no se requieren las velocidades para calcular trayectorias, sin embargo son necesarias para cálculos de energía cinética.

$$\vec{v}\,(t) = \frac{1}{2}\left[\vec{r}\,(t'+\delta t) - \vec{r}\,(t+\delta t)\right] \tag{3.2}$$

La expresión (3.1) posee error de cuarto orden, pero (3.2) lo tiene de tercer orden. Tildesley (1982) planea una versión del algoritmo de Verlet en dos etapas que se resume en el siguiente algoritmo:

$$\text{Primera etapa} \begin{cases} \vec{r}\,(t+\delta t) = \vec{r}\,(t) + \vec{v}\,(t)\,\delta t + \dfrac{1}{2}\vec{a}\,(t)\,(\delta t)^2 \\ \vec{v}\left(t+\dfrac{\delta t}{2}\right) = \vec{v}\,(t) + \dfrac{1}{2}\vec{a}\,(t)\,\delta t \end{cases} \tag{3.3}$$

$$\text{Segunda etapa} \begin{cases} \vec{v}\,(t+\delta t) = \vec{v}\,(t) + \dfrac{1}{2}\left[\vec{a}\,(t) + \vec{a}\,(t+\delta t)\right]\delta t \end{cases} \tag{3.4}$$

Metodología

Para este proyecto se busca identificar las partes esenciales de un código de simulación de DM NVT básico con el algoritmo equivalente de Verlet (versión de velocidad, Swope, Andersen, Berens y Wilson, 1982) para un sistema de átomos iguales que interaccionan entre sí mediante un modelo de potnecial central Lennard-Jones (LJ).

$$u^* = 4\epsilon^* \left[\left(\frac{1}{r^*} \right)^{12} - \left(\frac{1}{r^*} \right)^{6} \right] \tag{3.5}$$

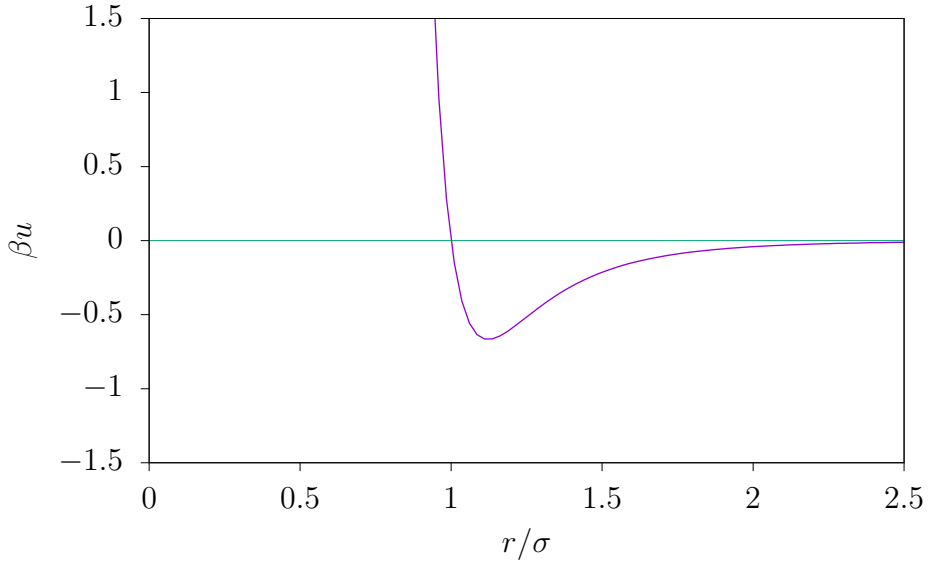


Figura 3.1: Potencial LJ con $T^* = 1.5$.

3.1. Código

A continuación se incluyen los fragmentos de código más relevantes del programa, se presentan comentarios y preguntas dentro de las mismas líneas de código.

Iniciamos definiendo la declaración implícita de variables reales con doble presición. Luego algunos parámetros:

```
1  implicit real(8)(a-h,o-z)
2  parameter(n=250)          ! Numero de particulas
3  parameter(nn2=5000)       ! Configuraciones de equilibrio a guardar
4  parameter(nener=50000)    ! Configuraciones antes del equilibrio
5  parameter(free=3.0)       ! Grados de libertad
6  parameter(pi=3.1415927)   !
7  real,external::zran       ! Funcion generadora de numeros aleatorios
```

Luego colocamos tres conjuntos de variables en contenedores (commons)

```
common/block1/rx,ry,rz,vx,vy,vz,fx,fy,fz
common/valores/dens,rcut,box,nstep
common/block2/rxc,ryc,rzc
```

Inicializamos los arreglos que contienen las posiciones, velocidades, fuerzas y configuraciones.

```
dimension cx(n,nn2),cy(n,nn2),cz(n,nn2)          ! Matrices de configuracion
dimension cxr(n,nn2),cyr(n,nn2),czr(n,nn2)       ! Matrices de conf. (dinamicas)
dimension fx(n),fy(n),fz(n),vx(n),vy(n),vz(n)    ! Fuerzas y velocidades
dimension rx(n),ry(n),rz(n),rxc(n),ryc(n),rzc(n)  ! Posiciones (componentes)
```

Le damos valor a los parámetros de la simulación. En este caso:

```
nstep  = 150000      ! Numero de configuraciones a calcular
iprint = 10000       ! Frecuencia de impresion en pantalla
nfrec  = 20          ! Frecuencia de guardado de configuraciones
dt     = 0.0001      ! Desplazamiento temporal
dens   = 0.6         ! Concentracion reducida
temp   = 1.5         ! Temperatura reducida de la configuracion
xm     = 1.0         ! Masa de la particula
sigma  = 1.0         ! Diametro de las particulas
```

Algunos cálculos previos

```
a     = 1.0/3.0      !
box   = (n/dens)**a  ! Longitud del lado de la caja cubica
rcut  = box/2.0      ! Radio de corte de la interaccion
tempi = temp         ! Temperatura inicial
ki2   = 0            ! Numero de configuraciones de equilibrio almacenadas
```

Se procede a imprimir en pantalla algunos datos de la simulación para informar al usuario. Luego se abren cuatro archivos para guardar la simulación: velocidades, configuración final, temperatura. Se llama la subrutina `configini` para generar la configuración inicial aleatoria tridimensional.

La subrutina `comvel` genera velocidades aleatorias a partir de una distribución gaussiana.

```
subroutine comvel(temp)
  implicit real(8)(a-h,o-z)
  parameter(n=250)
  common/block1/rx,ry,rz,vx,vy,vz,fx,fy,fz
  common/semillas/iseed3,iseed2,iseed1
  dimension rx(n),ry(n),rz(n)
  dimension fx(n),fy(n),fz(n),vx(n),vy(n),vz(n)

  iseed  = 43560        ! Semillas
```

```
10      iseed1 = 39467
11      iseed2 = 148420
12      iseed3 = 7845901
13      call azarg(iseed,ax)  ! Numero aleatorio gaussiano
14      call azarg(iseed,ay)
15      call azarg(iseed,az)
16      rtemp = sqrt(temp)    ! Raiz cuadrada del intervalo temporal
17      vx = rtemp*ax         ! Fija velocidades aleatorias
18      vy = rtemp*ay
19      vz = rtemp*az
20      sumx = 0.0; sumy = 0.0; sumz = 0.0
21      do i=1,n              ! Suma todas las velocidades
22        sumx = sumx + vx(i) ! ?: seria lo mismo tomar sumx = n*rtemp*ax ya
23        sumy = sumy + vy(i) ! que todas las entradas de vx tienen el mismo
24        sumz = sumz + vz(i) ! valor
25      end do
26      sumx = sumx/real(n) ! Promedio de vx
27      sumy = sumy/real(n)
28      sumz = sumz/real(n)
29      vx = vx - sumx      ! Desviacion de la media
30      vy = vy - sumy
31      vz = vz - sumz
32      return
33    end subroutine comvel
```

Los datos resultantes en las listas `vx`, `vy` y `vz` se escriben en un archivo `vidm0.dat`. Luego se calculan las correcciones de largo alcance para la energía y el desplazamiento cuadrático medio.

```
1   sr3 = (sigma/rcut)**3    !
2   sr9 = sr3**3             !
3   boxcub = 1.0/box**3      ! Inverso del volumen de la celda de simulacion
4     ! Correccion largo alcance energia
5   vlrc = (8./9.)*pi*dens*real(n)*(sr9-3.0*sr3)
6     ! Correccion largo alcance termino del virial para la presion
7   wlrc = (16./9.)*pi*dens*real(n)*(2.0*sr9-3.0*sr3)
```

Ahora se llama a la subrutina `moveA` la cual implementa la primera etapa del algoritmo (3.3), aplicando las condiciones de imagen mínima. Aquí una muestra de las líneas de código que implementan esta parte, lo mismo para `y` y para `z`.

```
1   dt2 = dt/2.
2   dtsq2 = dt*dt2
3   rx = rx + dt*vx + dtsq2*fx/xm    ! Para prop. estructurales
4   rx = rx - box*anint(rx/box)      ! Cond. imagen minima
5   rxc = rxc + dt*vx + dtsq2*fx/xm  ! Para prop. dinamicas
6   vx = vx + dt2*fx/xm              ! Primera etapa
```

Luego se llama a la subrutina `force` que calcula las fuerzas entre partículas utilizando el modelo de interacción, en este caso LJ.

```
v = v + vlrc              ! Energia potencial mas la correcion
w = (w + wlrc)*boxcub     ! Presion (termino del virial)
e = xk + v                ! Energia mecanica total (cinetica mas potencial)
vn = v/real(n)            ! Energia potencial por particula
xkn = xk/real(n)          ! Energia cinetica por particula
en = e/real(n)            ! Energia total por particula
temp = 2.0*xnk/free       ! Temperatura
pres = dens*temp + w      ! Presion
```

Después se llama a la subrutina `moveB` que implementa la segunda etapa (3.4) del algoritmo.

```
vx = vx + dt2*fx/xm                ! Segunda etapa
xk = xk + vx**2 + vy**2 + vz**2    ! Cuadrado de la velocidad
xk = 0.5*xm*xk                     ! Energia cinetica
```

Para mantener la temperatura del sistema consistente con la temperatura del baño térmico para una descripción NVT (comentario de LYR).

```
alfa = sqrt(tempi/temp)
vx = alfa*vx
vy = alfa*vy
vz = alfa*vz
```

Se procede a calcular todas las configuraciones (`nstep`).

Posteriormente se revisa si es momento de imprimir la información en pantalla. Se guarda la información de la energía por partícula, la presión y la temperatura en un archivo llamado `tedm0.dat`. Después, en caso de ser la última iteración del código, éste guarda la configuración final en un archivo.

Para guardar las configuraciones de equilibrio se utiliza una condición para evaluar si ya termalizó y si es momento de realizar el muestreo. Además de que se calculan algunas cantidades para luego evaluar ciertos valores medios. Sólo se incluye para un caso, aunque se calculan medias para la energía total, energía cinética, energía potencial y la presión.

```
if ((mod(ISTEP,nfrec).eq.0).and.(istep.GT.nener)) then
  if (istep.LE.nsteo) then
    ki2 = ki2 + 1          ! Se cuenta una configuracion de equilibrio mas
    ace = ace + en
    acesq = acesq + en**2
    cx(:,ki2) = rx(i)
    cxc(:,ki2) = rxc(i)
  end if
end if
```

3.2. Resultados

Tras correr el programa con los parámetros indicados obtuvimos los siguientes resultados:

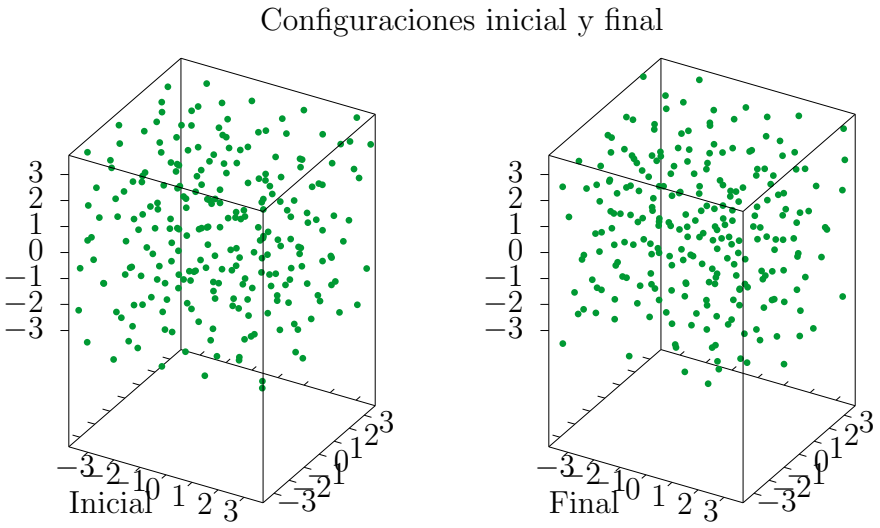


Figura 3.2

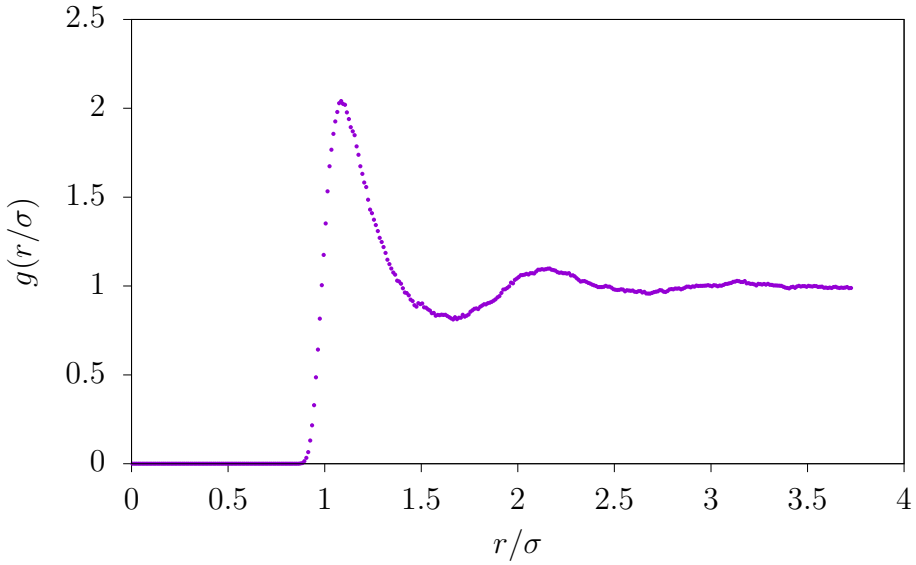


Figura 3.3: Función de distribución radial.

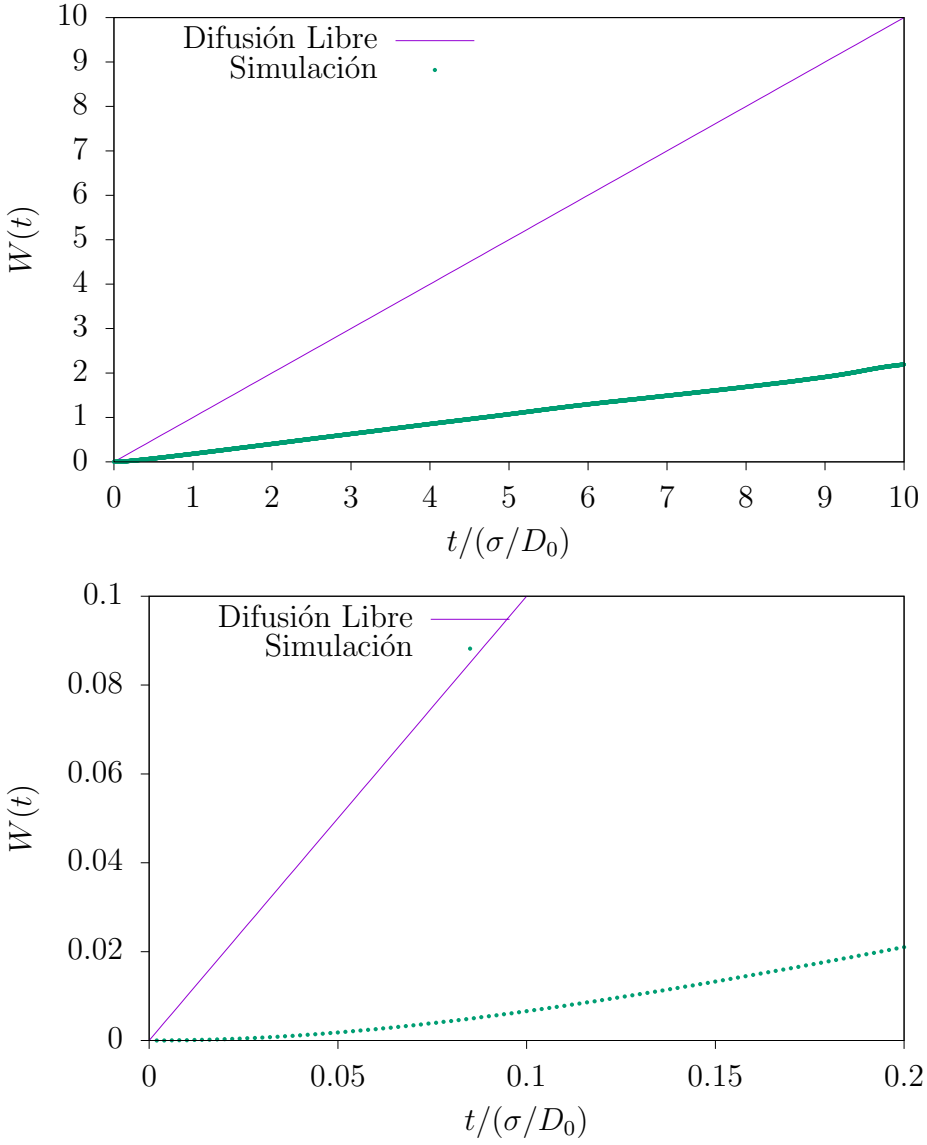


Figura 3.4: Desplazamiento cuadrático medio.

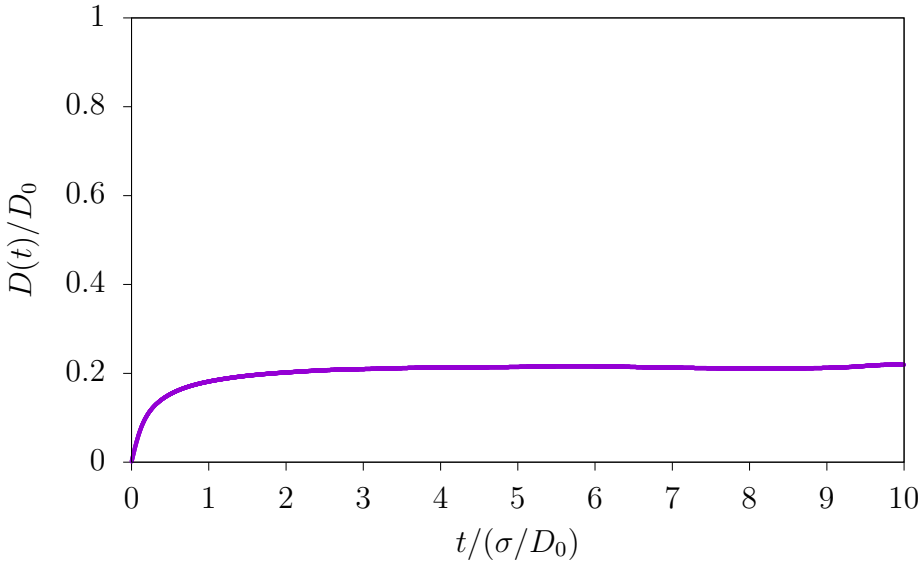


Figura 3.5: Difusión dependiente del tiempo.

Apéndice: Código Monte Carlo

A continuación se anexa el código completo utilizado para la simulación MC. Se han removido los bloques de comentarios iniciales y algunas funciones que no eran necesarias o fueron utilizadas para la implementación de la simulación. El programa contiene los siguientes módulos, subrutinas y funciones:

M `types` Definición de clases.

M `messages` Mensajes a pantalla.

M `iounits` Control de unidades de entrada y salida predeterminadas usando ISO Fortran.

M `runtime_vars` Variables re-utilizables durante la ejecución del programa.

M `user_vars` Parámetros y variables definidas por el simulador para la ejecución del programa.

S `exec_time` Calcula y despliega el tiempo de ejecución de un proceso.

S `random_seed_init` Inicializa una semilla aleatoria para el generador de números aleatorios.

F `random_normal` Genera un número aleatorio con distribución gaussiana normal.

M `modules` Carga todos los procedimientos enlistados anteriormente.

M `models` Los modelos de interacción (ND y PC).

S `configini` Configuración inicial regular o aleatoria tridimensional.

S `enemble_energy` Energía de la configuración.

S `particle_energy` Energía de potencial entre partículas dada por el modelo de interacción.

S `mc` Algoritmo MC.

S `gdr` Cálculo de la función de distribución radial.

P `main` Programa principal.

```
 module types
  implicit none
  save
  integer, parameter, public :: i1b = selected_int_kind(2)         !max 127
  integer, parameter, public :: i2b = selected_int_kind(4)         !max 32767
  integer, parameter, public :: i4b = selected_int_kind(9)         !max ~2.1e9
  integer, parameter, public :: i8b = selected_int_kind(16)        !max ~9.2e21
  integer, parameter, public :: sp  = selected_real_kind(5,30)     !max ~3.4e38
  integer, parameter, public :: dp  = selected_real_kind(12,200)   !max ~1.7e308
end module types

module messages
  public
  character(len=*), parameter :: suc_mess = '("Pass.")'
  character(len=*), parameter :: err_mess = '("Fail. Aborting...")'
  character(len=*), parameter :: baseskip = '("================================")'
  character(len=*), parameter :: fileopen = '("Opening file.")'
  character(len=*), parameter :: fileclse = '("Closing file.")'
  character(len=*), parameter :: alloc    = '("Allocating array.")'
  character(len=*), parameter :: dealloc  = '("Deallocating array.")'
end module messages

module iounits
  use types, only : i1b
  use iso_fortran_env, only : output_unit, input_unit
  implicit none
  public
  integer(i1b), parameter :: iounit1 = output_unit
  integer(i1b), parameter :: iounit2 = output_unit + 1
  integer(i1b), parameter :: iounit3 = output_unit + 2
end module iounits

module runtime_vars
  use types, only: i1b, sp
  implicit none
  public
  integer(i1b) :: ierror, istat     ! Flags
  real(sp) :: start, finish         ! Run-time variables
end module runtime_vars

module modules
  use messages
  use iounits
  use runtime_vars
  use user_vars

  contains

  subroutine exec_time(start,finish)
    use types, only: i1b, sp; use iounits
    implicit none
    real(sp), intent(in) :: start, finish
    real(sp) :: time, sec
    integer(i1b) :: mint, hr

```

```
56      time = finish - start
57      if (time.LT.60.) then
58        sec = time
59        write(iounit1,'(f6.3,A)') sec, " sec"
60      else if ((time.GE.60.) .AND. (time.LT.3600.)) then
61        mint = int(time/60.,i1b)
62        sec = time - mint*60
63        write(iounit1,'(i2,A,f6.3,A)') mint,":",sec," min"
64      else
65        hr = int(time/3600.,i1b)
66        mint = int((time - hr*3600.)/60.,i1b)
67        sec = time - hr*3600. - mint*60.
68        write(iounit1,'(i2,A,i2,A,f6.3,A)') hr,":",mint,":",sec," hr"
69      end if
70    end subroutine exec_time
71
72    subroutine random_seed_init()
73      integer :: i,n,clock
74      integer, allocatable :: seed(:)
75      call random_seed(size=n)
76      allocate(seed(n))
77      call system_clock(count=clock)
78      seed = clock + 37*(/ (i-1,i=1,n) /)
79      call random_seed(put=seed)
80      deallocate(seed)
81    end subroutine random_seed_init
82
83    real(dp) function random_normal()
84    ! Adapted from the following Fortran 77 code
85    !       ALGORITHM 712, COLLECTED ALGORITHMS FROM ACM.
86    !       THIS WORK PUBLISHED IN TRANSACTIONS ON MATHEMATICAL SOFTWARE,
87    !       VOL. 18, NO. 4, DECEMBER, 1992, PP. 434-435.
88    !
89    !  The function random_normal() returns a normally distributed pseudo-random
90    !  number with zero mean and unit variance.
91    !
92    !  The algorithm uses the ratio of uniforms method of A.J. Kinderman
93    !  and J.F. Monahan augmented with quadratic bounding curves.
94    implicit none
95    ! Local variables
96    real(dp) :: s = 0.449871, t = -0.386595, a = 0.19600, b = 0.25472, &
97                r1 = 0.27597, r2 = 0.27846, u, v, x, y, q
98    ! Generate P = (u,v) uniform in rectangle enclosing acceptance region
99    do
100     call random_number(u)
101     call random_number(v)
102     v = 1.7156 * (v - 0.5_dp)
103     ! Evaluate the quadratic form
104     x = u - s
105     y = abs(v) - t
106     q = x**2 + y*(a*y - b*x)
107     ! Accept P if inside inner ellipse
108     if (q < r1) exit
109     ! Reject P if outside outer ellipse
110     if (q > r2) cycle
111     ! Reject P if outside acceptance region
```

```
112     if (v**2 < -4.0*LOG(u)*u**2) exit
113   end do
114
115   ! Return ratio of P's coordinates as the normal deviate
116   random_normal = v/u
117   return
118   end function random_normal
119
120 end module modules
121
122 module user_vars
123   use types, only : i2b, i4b, dp
124   implicit none
125   public
126   save
127
128   integer(i4b), parameter :: npart = 512
129   integer(i4b), parameter :: nconfig = 55000
130   integer(i4b), parameter :: ntherm = 5000
131   integer(i4b), parameter :: isave = 10
132   integer(i4b), parameter :: dmax_rescale = 1000
133   real(dp), parameter :: sigma  = 1.0
134   real(dp), parameter :: deltaR = 0.025
135   real(dp), parameter :: pi = 4.0_dp*datan(1._dp)
136
137   ! NON-FIXED Common variables. May change during execution.
138   real(dp) :: dens = 0.
139   real(dp) :: phi = 0.
140   real(dp) :: ratio_max = 0.5
141   real(dp) :: dmax = 0.05
142   real(dp) :: ener_lrc = 0.
143   real(dp) :: ener = 0., sum_ener = 0.
144   real(dp) :: edgeL = 1.0
145   real(dp) :: rcut = 1.0
146   real(dp) :: x(npart), y(npart), z(npart)
147   real(dp), allocatable, dimension(:,:) :: cx, cy, cz
148
149   ! Square well model parameters
150   real(dp), parameter :: eps_sw = 1/0.74              ! 1/T*
151   real(dp), parameter :: lambda_sw = 1.25
152
153 end module user_vars
154
155 module models
156   use ieee_arithmetic, only : ieee_value, ieee_positive_inf
157   use user_vars
158   use types, only : dp
159   implicit none
160
161   contains
162
163   real(dp) function hard_nucleus(rij) result(u)
164     real(dp), intent(in) :: rij
165     if (rij.LT.sigma) then
166       u = ieee_value(1.0, ieee_positive_inf)
167     else
```

```
168       u = 0.
169     end if
170   end function hard_nucleus
171
172   real(dp) function square_well(rij,eps,lambda) result(u)
173     real(dp), intent(in) :: rij, eps, lambda
174     if (rij.LE.sigma) then
175         u = ieee_value(1.0,ieee_positive_inf)
176       elseif ((rij.GT.sigma).AND.(rij.LE.(sigma*lambda))) then
177         u = -eps
178       else
179         u = 0.
180     end if
181   end function square_well
182
183 end module models
184
185 subroutine configini_cube_3d()
186   use types, only : dp, i2b
187   use modules
188   implicit none
189   character(len=60) :: doc
190   real(dp) :: space, pini       ! Particle spacing and position of the first
191   integer(i2b) :: i, j, l, k=0, n2
192
193   call cpu_time(start)
194
195   n2 = int(npart**(1./3.),i2b)        ! Particles per side of the cube
196   space = edgeL/real(n2,dp)
197   pini = (space-edgeL)/2._dp
198
199   write(doc , '(A,F6.3,A)') "configs\configini", dens,".txt"
200   open(iounit2,file=doc, status='replace', action='write')
201
202   do l=1,n2
203     do j=1,n2
204       do i=1,n2
205         k = k + 1_i2b
206         x(k) = pini + real(i-1,dp)*space
207         y(k) = pini + real(j-1,dp)*space
208         z(k) = pini + real(l-1,dp)*space
209         write(iounit2,*) k, x(k), y(k), z(k)
210       end do
211     end do
212   end do
213
214   !====================== Finish procedure ====================================
215   close(iounit2)
216   call cpu_time(finish)
217   write(iounit1,baseskip)
218   write(iounit1,'("Initial configuration done!")')
219   call exec_time(start,finish)
220 end subroutine configini_cube_3d
221
222 subroutine configini_rand_3d()
223   use types, only : dp, i2b
```

```
224   use modules
225   implicit none
226   character(len=60) :: doc
227   integer(i2b) :: i, j
228   real(dp) :: r(1:3)
229   real(dp) :: xij, yij, zij, d
230   logical :: right, left, up, down, front, back
231
232   call cpu_time(start)
233
234   write(doc, '(A,F6.3,A,I4,A)') "configs\configini", dens,".txt"
235   open(iounit2,file=doc, status='replace', action='write')
236   call random_seed_init()
237
238   do i=1, npart
239 100 call random_number(r)
240     r = r - 0.5_dp
241     x(i) = r(1)*edgeL
242     y(i) = r(2)*edgeL
243     z(i) = r(3)*edgeL
244     right = (x(i)+sigma/2.).LT.(edgeL/2.)
245     left  = (x(i)-sigma/2.).GT.(-edgeL/2.)
246     up    = (y(i)+sigma/2.).LT.(edgeL/2.)
247     down  = (y(i)-sigma/2.).GT.(-edgeL/2.)
248     front = (z(i)+sigma/2.).LT.(edgeL/2.)
249     back  = (z(i)-sigma/2.).GT.(-edgeL/2.)
250     ! Check if particle is inside the cell
251     if (right.AND.left.AND.down.AND.up.AND.front.AND.back) then
252       do j=1, i-1_i2b
253         xij = x(i) - x(j)
254         yij = y(i) - y(j)
255         zij = z(i) - z(j)
256         d = sqrt(xij*xij+yij*yij+zij*zij)
257         ! Overlap between the i-th and j-th particle
258         if (d.LT.1.0) goto 100
259       end do
260     else
261       goto 100 ! Outside the cell: generate another position
262     end if
263     write(iounit2,*) i, x(i), y(i), z(i)
264   end do
265
266   !====================== Finish procedure ======================================
267   close(iounit2)
268   call cpu_time(finish)
269   write(iounit1,baseskip)
270   write(iounit1,'("Initial configuration done!")')
271   call exec_time(start,finish)
272 end subroutine configini_rand_3d
273
274 subroutine ensemble_energy_3d()
275   use types, only : dp, i2b
276   use modules
277   use models, only : square_well, hard_nucleus !!!! MODEL !!!!
278   implicit none
279   real(dp) :: xij, yij, zij, rij, enerij
```

```
280   integer(i2b) :: i, j
281
282   call cpu_time(start)
283
284   ener = 0.
285   do i = 1, npart-1, 1
286     do j = i+1_i2b, npart
287       xij = x(j)-x(i)
288       yij = y(j)-y(i)
289       zij = z(j)-z(i)
290       ! Minimal image condition
291       xij = xij - edgeL*anint(xij/edgeL)
292       yij = yij - edgeL*anint(yij/edgeL)
293       zij = zij - edgeL*anint(zij/edgeL)
294       rij = sqrt(xij*xij + yij*yij + zij*zij)
295       ! Interaction model implementation
296       if (rij.LT.rcut) then
297         !enerij = hard_nucleus(rij) ! Change for different models
298         enerij = square_well(rij,eps_sw,lambda_sw)
299         ener = ener + enerij
300       end if
301     end do
302   end do
303   !====================== Finish procedure ========================================
304   call cpu_time(finish)
305   write(iounit1,baseskip)
306   write(iounit1,'("Initial configuration energy done!")')
307   call exec_time(start,finish)
308 end subroutine ensemble_energy_3d
309
310 subroutine particle_energy_3d(xi,yi,zi,i,penergy)
311   use types, only : dp, i2b
312   use modules
313   use models, only : square_well, hard_nucleus !!!! MODEL !!!!
314   implicit none
315
316   real(dp), intent(in) :: xi, yi, zi
317   real(dp), intent(inout) :: penergy
318   integer(i2b), intent(in) :: i
319   real(dp) :: xij, yij, zij, rij, enerij
320   integer(i2b) :: j
321
322   penergy = 0._dp
323   do j = 1, npart
324     if (i.NE.j) then
325       xij = xi - x(j)
326       yij = yi - y(j)
327       zij = zi - z(j)
328       ! Minimal image condition
329       xij = xij - edgeL*dnint(xij/edgeL)
330       yij = yij - edgeL*dnint(yij/edgeL)
331       zij = zij - edgeL*dnint(zij/edgeL)
332       rij = sqrt(xij*xij + yij*yij + zij*zij)
333       ! Interaction model implementation
334       if (rij.LT.rcut) then
335         !enerij = hard_nucleus(rij) ! Change for different models
```

```
336         enerij = square_well(rij,eps_sw,lambda_sw)
337         penergy = penergy + enerij
338       end if
339     end if
340   end do
341
342 end subroutine particle_energy_3d
343
344 subroutine mc_3d()
345   use types, only : dp, i2b, i4b
346   use modules
347   implicit none
348   character(len=60) :: doc, doc2
349   integer(i4b) :: i, j, col
350   integer(i4b) :: accept, totalmove
351   real(dp) :: ener_old, ener_new, xnew, ynew, znew, ener_pp
352   real(dp) :: delta_ener, r(1:3), delta_cut, alpha!, ratio
353
354   call cpu_time(start)
355
356   ! Initialize variables
357   delta_cut = 75._dp
358   ener_old = 0._dp
359   ener_new = 0._dp
360   totalmove = 0_i4b
361   accept = 0_i4b
362   col = 0_i2b
363
364   call random_seed_init()
365   write(doc2 , '(A,F6.3,A)') "ener\ener_pp_mc", dens,".txt"
366   open(iounit2,file=doc2)
367
368 Configs:  do j=1, nconfig
369     Move: do i=1, npart
370       ! Current values
371       call particle_energy_3d(x(i),y(i),z(i),i,ener_old)
372       ! Tentative move
373       call random_number(r)
374       xnew = x(i) + (2.0*r(1)-1.0)*dmax
375       ynew = y(i) + (2.0*r(2)-1.0)*dmax
376       znew = z(i) + (2.0*r(3)-1.0)*dmax
377       ! Periodic conditions
378       xnew = xnew - edgeL*anint(xnew/edgeL)
379       ynew = ynew - edgeL*anint(ynew/edgeL)
380       znew = znew - edgeL*anint(znew/edgeL)
381       call particle_energy_3d(xnew,ynew,znew,i,ener_new)  ! "new" values
382       ! Tentative energy variation
383       delta_ener = ener_new - ener_old
384       ! Monte Carlo Algorithm ==========================================
385       if (delta_ener.LT.delta_cut) then
386         call random_number(alpha)
387         !print*, "True", j, i, delta_ener, ener_pp ! Debugging
388         if (delta_ener.LE.0.) then
389           x(i) = xnew
390           y(i) = ynew
391           z(i) = znew
```

```
392             ener = ener + delta_ener
393             accept = accept + 1_i4b
394           ! Evaluate Boltzmann factor
395           elseif (exp(-delta_ener).GT.alpha) then
396             x(i) = xnew
397             y(i) = ynew
398             z(i) = znew
399             ener = ener + delta_ener
400             accept = accept + 1_i4b
401           end if
402         end if
403         ener_pp = ener/real(npart,dp)
404       end do Move
405       totalmove = totalmove + 1_i4b
406
407       ! Save energy per particle for the j-th configuration
408       write(iounit2,*) j, ener_pp
409
410       ! Maximum displacement optimization (optional) ==============================
411       ! Removed!!
412
413       !Upper boundary
414       if (dmax.GT.2.) dmax = 2.0
415       ! Process monitoring on screen
416       if (mod(j,dmax_rescale).EQ.0) write(iounit1,'("Working",i10,f12.6)') totalmove, ener_pp
417
418       ! For estimating mean energy per particle
419       if (j.GT.ntherm) sum_ener = sum_ener + ener_pp
420
421       ! Verify if configuration is to be saved (cx,cy,cz)
422       if ((mod(j,isave).EQ.0).AND.(j.GT.ntherm)) then
423         col = col + 1_i4b
424         cx(:,col) = x
425         cy(:,col) = y
426         cz(:,col) = z
427       end if
428
429     end do Configs
430     close(iounit2)
431
432     ! Get mean energy per particle in over all ensembles.
433     sum_ener = sum_ener/real(nconfig-ntherm,dp)
434
435     ! Count particles inside the cell
436     j = 0_i2b
437     do i=1, npart
438       if ((abs(x(i)).LT.(edgeL/2.)).AND.(abs(y(i)).LT.(edgeL/2.)).AND.(abs(z(i)).LT.(edgeL/2.))
439     end do
440
441     ! ========== Save to file ==========
442     write(doc , '(A,F6.3,A)') "configs\configfin", dens,".txt"
443     open(iounit2,file=doc, status='replace', action='write', iostat=istat)
444       write(iounit1,baseskip)
445       write(iounit1,fileopen)
446       if (istat.EQ.0) then
447         write(iounit1,suc_mess)
```

```
448     else
449       write(iounit1,err_mess)
450       go to 100
451     end if
452
453     do i=1, npart
454       write(iounit2,*) i, x(i), y(i), z(i)
455     end do
456
457   !====================== Finish procedure ========================================
458   close(iounit2, iostat=ierror)
459     write(iounit1,fileclse)
460     if (ierror.EQ.0) then
461       write(iounit1,suc_mess)
462     else
463       write(iounit1,err_mess)
464       go to 100
465     end if
466
467   write(iounit1,'("Particles inside the cell",i10)') j
468   write(iounit1,'("Mean energy per particle =",f10.5)') sum_ener
469   write(iounit1,'("Monte Carlo done!")')
470 100   call cpu_time(finish)
471   call exec_time(start,finish)
472 end subroutine mc_3d
473
474 subroutine gdr_3d(gplus)
475   use types, only: dp, i4b
476   use modules
477   implicit none
478   character(len=60) :: doc
479   real(dp), intent(out) :: gplus
480   integer(i4b), allocatable, dimension(:) :: hist
481   integer(i4b) :: i, j, k, configs, totbins, bin
482   real(dp) :: xij, yij, zij, rij, g, r(1:3), parts, summ=0.5
483   real(dp), allocatable, dimension(:) :: integ
484
485   call cpu_time(finish)
486
487   totbins = int((edgeL/2._dp)/deltaR,i4b)
488
489   allocate(hist(1:totbins),integ(1:totbins))
490   hist = 0
491   integ = 0.
492   configs = size(cx,2,i4b)
493
494   do i=1, npart
495     do j=1, npart
496       if(j.EQ.i) cycle
497       do k=1, configs
498         xij = cx(j,k) - cx(i,k)
499         yij = cy(j,k) - cy(i,k)
500         zij = cz(j,k) - cz(i,k)
501         ! Periodic conditions
502         xij = xij - edgeL*anint(xij/edgeL)
503         yij = yij - edgeL*anint(yij/edgeL)
```

```
504           zij = zij - edgeL*anint(zij/edgeL)
505           rij = sqrt( xij*xij + yij*yij + zij*zij )
506           bin = ceiling( rij/deltaR ,i4b)
507           if ( bin.LE.totbins ) hist(bin) = hist(bin) + 1
508         end do
509       end do
510     end do
511
512     ! ========== Get g(r) & save to file ==========
513     write(doc, '(A,F6.3,A,I4,A)') "gdr\gdr3d", dens, ".txt"
514     open(iounit2,file=doc, status='replace', action='write', iostat=istat)
515       write(iounit1,baseskip)
516       write(iounit1,fileopen)
517       if (istat.EQ.0) then
518         write(iounit1,suc_mess)
519       else
520         write(iounit1,err_mess)
521         go to 100
522       end if
523
524     do bin=1, totbins
525       r(1) = real(bin-1,dp)*deltaR    ! Lower radius of the bin
526       r(2) = r(1) + deltaR            ! Upper radius of the bin
527       r(3) = r(1) + deltaR/2._dp      ! Representative radius of the bin
528       g = real(hist(bin),dp)/real(configs*npart,dp) / ((4./3.)*dens*pi*(r(2)**3 - r(1)**3))
529       write(iounit2,1000) r(3), g
530       1000 format(2(1X,f10.6))
531       if (bin.EQ.(ceiling(sigma/deltaR) + 1)) gplus = g
532       integ(bin) = g*r(3)
533     end do
534
535     do i=2, totbins-1
536       summ = summ + integ(i)
537     end do
538
539     parts = 24._dp*dens*deltaR*summ
540
541     !========== Finish procedure ==========
542     deallocate(hist)
543     close(iounit2, iostat=ierror)
544       write(iounit1,fileclse)
545       if (ierror.EQ.0) then
546         write(iounit1,suc_mess)
547       else
548         write(iounit1,err_mess)
549         go to 100
550       end if
551
552     write(iounit1,'("Bins width           =",f10.5)') deltaR
553     write(iounit1,'("Total number of bins =",i10)') totbins
554     write(iounit1,'("Total particles (int)=",i10)') nint(parts)
555     write(iounit1,'("G(r) done!")')
556 100   call cpu_time(finish)
557     call exec_time(start,finish)
558 end subroutine gdr_3d
559
```

```
560 program main
561   use types, only : sp, dp, i2b
562   use modules
563   implicit none
564
565   integer(i2b) :: iconfig
566   character(1) :: dim_sel
567   real(sp) :: startmain, finishmain
568   real(dp) :: gplus=0., press=0.
569
570   call cpu_time(startmain)
571   write(iounit1,'("Molecular Simulation")')
572   write(iounit1,baseskip)
573   write(iounit1,'("Input the reduced concentration <dens>")')
574   read(input_unit,*) dens
575
576   100 write(iounit1,'("Select (a) 2D, (b) 3D, (q) abort")')
577   !read(input_unit,*) dim_sel
578   dim_sel = 'b'
579   write(iounit1,baseskip)
580
581   ! Get the size of the box, interaction radius and number of configurations ===
582   ! Removed!!
583
584   rcut = 0.5_dp*edgeL                    ! Set biggest interaction radius possible
585   iconfig = int((nconfig-ntherm)/isave)! Get number of configurations to save
586   phi = dens*pi/6._dp
587
588   ! ========== Print settings for simulation ==========================================
589   write(iounit1,'("Number of particles   =",i10)') npart
590   write(iounit1,'("Reduced concentration =",f10.5)') dens
591   write(iounit1,'("Volume fraction       =",f10.5)') phi
592   write(iounit1,'("Side of the cell      =",f10.5)') edgeL
593   write(iounit1,'("Interaction radius    =",f10.5)') rcut
594   write(iounit1,'("Number of configs.    =",i10)') nconfig
595   write(iounit1,'("Save configs. after   =",i10)') ntherm
596   write(iounit1,'("Save every            =",i10)') isave
597   write(iounit1,'("Total configs to save =",i10)') iconfig
598
599   ! Initial configuration. Set model ==================================================
600   !
601   ! !!! Known error with rand_3d using dens>0.6, better use cube_3d
602   !
603   call configini_cube_3d()
604   !call configini_rand_3d()
605   call ensemble_energy_3d(rcut)
606
607   ener = ener + ener_lrc
608   write(iounit1,'("Initial configuration energy",f12.5)') ener/real(npart,dp)
609
610   ! ===================================================================================
611   ! ========================== MONTE CARLO ============================================
612   ! ===================================================================================
613   ! Set 'ratio_max' given 'dens'. Higher for lower concentrations ==========
614   ! Removed!!
615
```

```
616   ! Allocate C's matrices used for allocation of ensembles and run algorithm
617   write(iounit1,baseskip)
618   write(iounit1,alloc)
619   allocate(cx(1:npart,1:iconfig),cy(1:npart,1:iconfig),cz(1:npart,1:iconfig),stat=istat)
620     if (istat.EQ.0) then
621       write(iounit1,suc_mess)
622     else
623       write(iounit1,err_mess); go to 110
624     end if
625   ! Run algorithm
626   call mc_3d()
627
628   ! Get gdr ==========================================================
629   call gdr_3d(gplus)
630
631   ! Get pressure =====================================================
632   press = dens*(1._dp + (2._dp/3._dp)*pi*dens*gplus)
633   write(iounit1, '("Estimated pressure!",f10.5)') press
634
635   ! End program ======================================================
636   110 call cpu_time(finishmain)
637   !close(5)
638   write(iounit1, baseskip)
639   write(iounit1, '("Main done!")')
640   call exec_time(startmain,finishmain)
641   write(iounit1, baseskip)
642   !close(iounit1)
643
644 end program main
```