Voluntario 1: Simulación con dinámica molecular de un gas con un potencial de Lennard-Jones.

Física Computacional. Grado en Física. Universidad de Granada.

Alejandro José Zarzuela Moreno

10 de mayo, 2024

1. Objetivos

Usar el algoritmo de Verlet explicado en case para simular y estudiar las propiedades de un gas en dos dimensiones con un potencial de Lennard-Jones.

2. Planteamiento del problema y funciones básicas

Antes de hablar de las funciones usadas cabe indicar que el tratamiento de las variables $(r, v, a, ...)$ se hará en forma vectores numpy que contienen N vectores de 2 componentes, es decir, de la forma $([x_0, y_0], [x_1, y_1], ..., [x_{N-1}, y_{N-1}])$. Por otro lado, las variables usadas en la configuración del sistema (`h, nIteraciones, nParticulas, ...`) se encuentran al principio del programa (líneas 15 a 19). Justo después (líneas 25 a 32) se encuentran los parámetros que se cambiarán en función del apartado que estemos resolviendo.

- Distribución de las partículas en el cuadrado $L \times L$:

Para empezar, el objetivo es colocar N partículas ordenadas como una cuadrícula en el interior de un cuadrado $L \times L$. Comenzamos dando valor a `particulasPorFila` y `separacionInicialH` para saber cuántas partículas habrá por fila y cuál será la separación horizontal entre ellas. Por otro lado, a partir de la sentencia `if` se calcula la separación vertical, para lo que se tiene en cuenta si N es un cuadrado perfecto. Si lo es, como es el ejemplo de $N = 16$, entonces se necesitan 4 filas de 4 elementos y `separacionInicialV = separacionInicialH`. En caso de que no lo sea, como lo es el ejemplo de $N = 20$, se necesitan 4 filas de 4 elementos y 1 más de 4, por la que la separación horizontal no será la misma que la vertical y habrá que calcular `separacionInicialV = L/(particulasPorFila+1)`.

```
particulasPorFila = int(np.floor(np.sqrt(nParticulas)))
separacionInicialH = L/particulasPorFila
if nParticulas > particulasPorFila**2:
    separacionInicialV = L/(particulasPorFila+1)
else: separacionInicialV = separacionInicialH
```

Una vez hecho esto, solo queda llenar las filas de partículas, añadiendo de manera iterativa el espacio horizontal y vertical entre ellas.

```
for f in range(particulasPorFila):
    for i in range(particulasPorFila):
        r[f*particulasPorFila+i,1] = i*separacionInicialH  # Equiespaciado horizontal
particulasUltimaFila = nParticulas-particulasPorFila**2
if particulasUltimaFila != 0:
    for i in range(particulasUltimaFila):
        r[particulasPorFila**2+i,1] = i*separacionInicialH

for c in range(particulasPorFila):
    for i in range(particulasPorFila):
        r[c*particulasPorFila+i,0] = c*separacionInicialV  # Equiespaciado vertical
if particulasUltimaFila != 0:
    for i in range(particulasUltimaFila):
        r[particulasPorFila**2+i,0] = particulasPorFila*separacionInicialV
```

Nótese que en este algoritmo la última fila debe tener su propio bucle para tener en cuenta sus propiedades, ya que serán diferentes a las demás filas a no ser que N sea un cuadrado perfecto.

Más adelante se introducirá un algoritmo que dará lugar a otra configuración inicial en la que las partículas estarán en forma de red hexagonal.

- Desplazamientos y velocidades aleatorias:

Una vez las partículas se han colocado en forma de cuadrícula, cada una de ellas se desplaza ligeramente de manera aleatoria. Para ello, se itera sobre cada una de las partículas y a cada una de sus componentes se le suma $\pm$`margen`.

```
for i in range(nParticulas):
    r[i,0] += (2*np.random.rand()-1)*margen
    r[i,1] += (2*np.random.rand()-1)*margen
```

La variable `margen` se puede configurar al principio del programa junto a los demás parámetros.

Por otro lado, a cada partícula se le da una velocidad aleatoria de módulo `moduloVelocidad`. Se itera sobre cada una de las partículas y se le da una velocidad aleatoria entre `-moduloVelocidad` y `+moduloVelocidad` a la componente x. Luego se le da a la componente y el valor necesario para que el módulo de v sea `moduloVelocidad`, es decir, $\pm\sqrt{\texttt{moduloVelocidad}^2 - v[p,0]^2}$.

```
for p in range(nParticulas):
    v[p,0] = moduloVelocidad*(2*np.random.rand()-1)
    v[p,1] = np.random.choice(np.array([-1,1]))*np.sqrt(moduloVelocidad**2-v[p,0]**2)
```

Se usará el algoritmo de Verlet tal y como se ha aplicado en el problema 1 (dinámica del Sistema Solar), solo que con dos modificaciones:

1. Se usará el potencial de Lennard-Jones. Su expresión es la siguiente:

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \quad \Rightarrow \quad \left\{\begin{array}{l}\varepsilon = 1\\ \sigma = 1\end{array}\right\} \quad \Rightarrow \quad V(r) = 4\left[\left(\frac{1}{r}\right)^{12} - \left(\frac{1}{r}\right)^{6}\right] \tag{1}$$

donde se ha hecho el cambio $\varepsilon = \sigma = 1$ para disminuir el error debido a la diferencia entre los órdenes de magnitud con las que se opera. Este es un potencial con un punto de equilibrio en torno a $r = 1{,}1$. Si $r < 1{,}1$ la fuerza que genera es muy repulsiva, y si $r > 1{,}1$ es atractiva, hasta $r \simeq 3$ donde prácticamente no hay interacción.

2. Condiciones de contorno periódicas. Serán las condiciones de una superficie toroidal, de forma que en el cuadrado $L \times L$ las partículas que salen por el borde derecho aparecen por el izquierdo, y las que salen por el borde superior aparecen por el inferior (y viceversa).

Estas condiciones de contorno introducen dos dificultades nuevas al sistema: el cálculo de distancias y la recolocación de las partículas que atraviesan los bordes.

• Cálculo de distancias:

Se implementa la función `distancia()`, que dado un vector bidimensional de posiciones ordenado de la forma $([x_0, y_0], [x_1, y_1], ..., [x_{N-1}, y_{N-1}])$ calcula la distancia entre dos partículas sin tener en cuenta los bordes, restando eje por eje y obteniendo así `distX` y `distY`. Luego, se considera si esas distancias son mayores que $L/2$, y si es así se elige su complementario `(L-dist)` de modo que la distancia en cada eje sea la menor. Esto se hace para ambos ejes y se devuelve como resultado el vector `([distX, distY])`.

```
def distancia(vector,p,j,L,lMedios):
    distX = vector[p,0]-vector[j,0]
    distY = vector[p,1]-vector[j,1]
    if np.abs(distX) > lMedios:
        distX = -(L-np.abs(distX))*(distX/np.abs(distX))
    if np.abs(distY) > lMedios:
        distY = -(L-np.abs(distY))*(distY/np.abs(distY))
    return np.array([distX,distY])
```

• Recolocación de partículas:

En cada ciclo del algoritmo de Verlet se llama a la función `bordes()`, que comprueba si alguna partícula se ha salido de los bordes y si es así la desplaza en un múltiplo de L en la dirección contraria a la que se ha salido, haciendo uso de la operación resto '%'.

Las variables `fueraX` y `fueraY` están para dejar constancia de si la partícula se ha salido, y en caso afirmativo, de por qué pared se ha salido la partícula en cuestión. Se usará más adelante para el cálculo de presiones.

```
def bordes(v,p,L):
    evX = v[p,0]
    evY = v[p,1]
    fueraX = 0
    fueraY = 0
    if evX > L:
        evX = evX%L
        fueraX = 1
    if evX < 0:
        evX = evX%L
        fueraX = -1
    if evY > L:
        evY = evY%L
        fueraY = 1
    if evY < 0:
        evY = evY%L
        fueraY = -1
    return np.array([evX,evY]),np.array([fueraX,fueraY])
```

• Guardado de los resultados del algoritmo de Verlet:

En el interior del bucle de Verlet, se guardan los resultados (r y v) cada `skip` iteraciones. Esto se hace porque para valores muy bajos de h hay una gran cantidad de datos que corresponden a prácticamente el mismo instante temporal, de modo que al hacer una animación de la dinámica del sistema la evolución es muy lenta. Es por esto que el guardado de resultados del algoritmo de Verlet se hace cada `skip` iteraciones usando la condición `if t% skip == 0: ...`

Los datos r y v del bucle de Verlet se guardan en los vectores `posiciones` y `velocidades`. Estos vectores contienen `f+1 = nIteraciones/skip` vectores con la forma de r o v, cada uno correspondiente a

un instante temporal. Esto es, son similares a

$$\begin{aligned}([&[x_{00}, y_{00}], [x_{01}, y_{01}], ..., [x_{0,N-1}, y_{0,N-1}]\,,\\ &[[x_{10}, y_{10}], [x_{11}, y_{11}], ..., [x_{1,N-1}, y_{1,N-1}]]\,,\\ &\qquad\qquad ...\\ &[[x_{f,0}, y_{f,0}], [x_{f,1}, y_{f,1}], ..., [x_{f,N-1}, y_{f,N-1}]])\end{aligned}$$

Tras abrir el fichero deseado para escribir en él los datos, se llama a la función `verlet()` (línea 248) para obtener los vectores con las posiciones y velocidades de cada partícula del sistema en cada instante temporal. Hecho esto, se escriben las posiciones en el fichero *'posParticulas.dat'* mediante el bucle

```
for p in range(nParticulas):
    ficheroPlot.write(str(r[t,p,0]) + ", " + str(r[t,p,1]) + "\n")
ficheroPlot.write("\n")
```

de modo que se guardan en grupos de 20 líneas de la forma '$x_{t,n}$, $y_{t,n}$' separados por una línea vacía, correspondiendo cada línea con una partícula y cada grupo con un instante temporal.

3. Resultados

Todos los ficheros de interés para cada apartado están dentro de `voluntario1\Resultados`. Si para cualquiera de las ejecuciones del programa se quiere conocer los valores de `h, nIteraciones, N, skip`, el tiempo que ha tardado la ejecución, la temperatura promedio y la presión promedio, estos se pueden encontrar en las últimas líneas de los ficheros *'posParticulas.dat'* que hay en cada apartado.

3.1. Apartado 1: Energías, velocidades y temperatura media

Se simula un gas de 20 átomos en el interior de una caja 10×10. Sus velocidades iniciales tienen módulo `moduloVelocidad = 1`. Se realizan 40000 iteraciones con $h = 0{,}002$.

Por un lado, se busca estudiar las energías cinética T, potencial V y total $E = T + V$. Para ello se crean los vectores `eCinetica`, `ePotencial` y `eTotal`. Mediante un bucle que itera `t` hasta `nIteraciones/skip`. En cada iteración de `t`, se inicializan a cero las variables `sumaT` y `sumaV`. Luego, se itera sobre cada partícula `p` en ese instante temporal `t`.

```
temperaturaProm = 0
for t in range(int(nIteraciones/skip-1)):
    # CÁLCULO ENERGÍAS
    sumaT = 0
    sumaV = 0
    for p in range(nParticulas):
        sumaT += 0.5*np.linalg.norm(v[t,p])**2

        ePotencialAux = 0
        for j in range(nParticulas):
            if p != j:
                R = np.linalg.norm(distanciaGlobal(r,t,t,p,j,L,lMedios))
                ePotencialAux += (R**(-12)-R**(-6))
        sumaV += 2*ePotencialAux
        normaV[t,p] = np.linalg.norm(v[t,p])

        temperaturaProm += v[t,p,0]**2 + v[t,p,1]**2

    eCinetica.append(sumaT)
```

```
            ePotencial.append(sumaV)
            eTotal.append(sumaT+sumaV)
    temperaturaProm = 0.5*temperaturaProm/(nIteraciones/skip)
```

La energía cinética del instante `t` se calcula sumando la energía cinética de cada partícula en ese instante, `sumaT` $=$ `sumaT` $+ 0{,}5 \cdot |v|^2$ ($m = 1$). Lo mismo aplica para la energía potencial, solo que esta la calculamos sumando la contribución de las otras partículas $p' \neq p$. De nuevo, se usa la función `distancia()` para determinar la distancia entre las dos partículas que se estudian.

Una vez se ha calculado T y V para el instante temporal `t`, se añaden a la cola de los respectivos vectores `eCinetica` y `ePotencial`, y también se añade a `eTotal` la suma `sumaT+sumaV`.

Otro objetivo de este apartado es calcular la temperatura promedio del sistema. Para ello, se usa el teorema de equipartición $T = 0{,}5 \cdot \langle v_x^2 + v_y^2 \rangle$. En el bucle de arribe se aprecia cómo se inicializa la variable `temperaturaProm = 0`, y luego en cada instante temporal t se va sumando la contribución de cada partícula a la temperatura. Finalmente, al terminar el bucle se multiplica `temperaturaProm` por 0.5 (esto se hace aquí para ahorrar cálculos) y se divide entre el tiempo '`nIteraciones/skip`' para tener así el promedio.

Cabe notar que cuando arriba se dice 'tiempo', este no está expresado en segundos. Para que fuese así, sería `t = nIteraciones*h`.

Luego se guardan las velocidades en vectores para representarlas en un histograma.

```
# VECTORES PARA REPRESENTACIÓN DE VELOCIDADES
histogrVelocidades = np.zeros(int(nIteraciones/(2*skip)))
histogrVelocidadesX = np.zeros(int(nIteraciones/(2*skip)))
histogrVelocidadesY = np.zeros(int(nIteraciones/(2*skip)))
for t in range(0,int(nIteraciones/(2*skip))-nParticulas,nParticulas):
    for p in range(nParticulas):
        histogrVelocidades[t+p] = np.linalg.norm(v[t+int(nIteraciones/(2*skip)),p])
        histogrVelocidadesX[t+p] = v[t+int(nIteraciones/(2*skip)),p,0]
        histogrVelocidadesY[t+p] = v[t+int(nIteraciones/(2*skip)),p,1]
```

Finalmente se realiza un plot con estos datos usando matplotlib:

```
ax1 = plt.subplot(3,2,1)
plt.hist(normaV[0])
plt.title("Velocidades iniciales",fontsize=11)
plt.xlabel("|v|",fontsize=9)
plt.ylabel("Frecuencia",fontsize=9)

ax2 = plt.subplot(3,2,2)
plt.hist(histogrVelocidades,bins=100)
plt.title("Velocidades: t=t_f/2 a t=t_f",fontsize=11)
plt.xlabel("|v|",fontsize=9)
plt.ylabel("Frecuencia",fontsize=9)

ax3 = plt.subplot(3,2,3)
plt.hist(histogrVelocidadesX,bins=100)
plt.title("Velocidades (eje X): t=t_f/2 a t=t_f",fontsize=11)
plt.xlabel("|v_x|",fontsize=9)
plt.ylabel("Frecuencia",fontsize=9)

ax4 = plt.subplot(3,2,4)
plt.hist(histogrVelocidadesY,bins=100)
plt.title("Velocidades (eje Y): t=t_f/2 a t=t_f",fontsize=11)
```

```
plt.xlabel("|v_y|",fontsize=9)
plt.ylabel("Frecuencia",fontsize=9)

ax5 = plt.subplot(3,1,3)
plt.plot(eCinetica, label= "T")
plt.plot(ePotencial, label="V")
plt.plot(eTotal, label="E = T+V")
plt.title("Energías en función del tiempo")
plt.xlabel("tiempo")
plt.ylabel("Energía")
plt.legend()

plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=0.2, hspace=0.5)
    plt.show()
```

Los histogramas de velocidades se hacen usando los datos a partir de la segunda mitad del tiempo de ejecución, para que haya dado tiempo a alcanzar un estado de mayor equilibrio. La temperatura promedio es de 19.0178 K, y además se obtienen los siguientes resultados (ver la animación en la carpeta correspondiente):

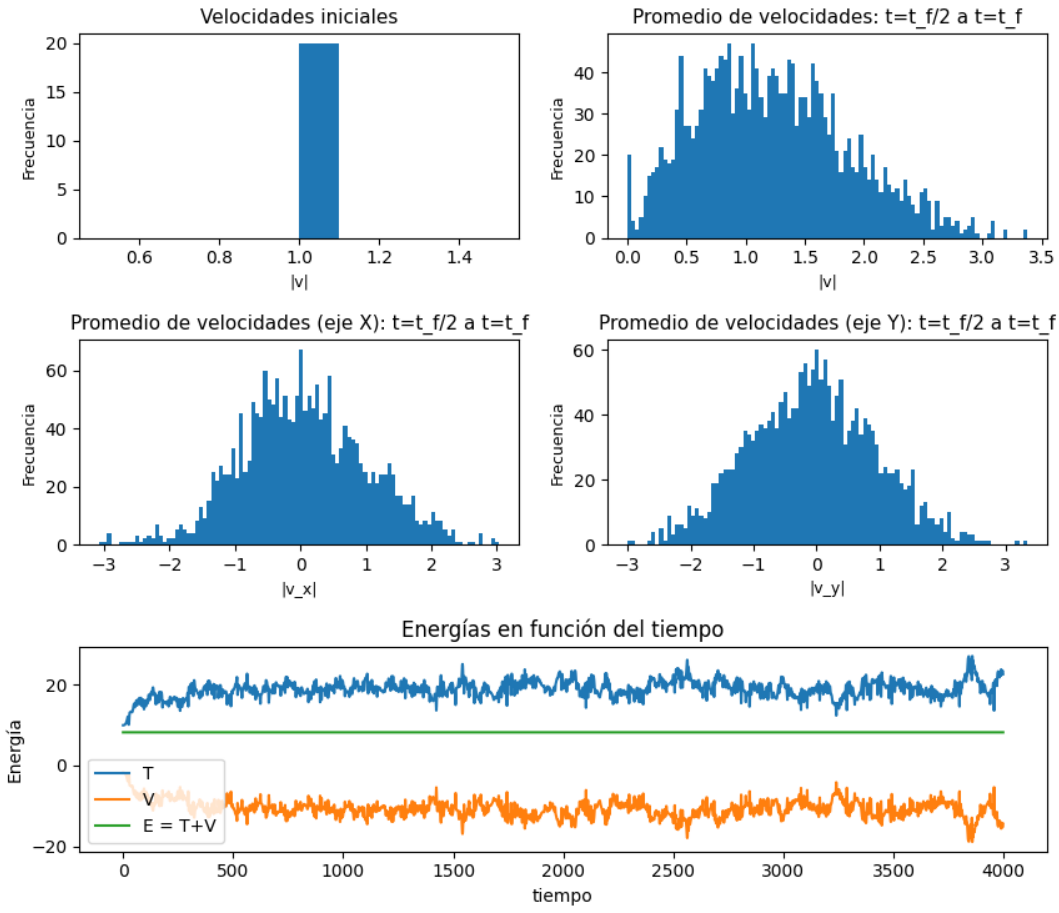


Figura 1: Resultados para una red 10×10 con 20 partículas.

3.2. Apartado 2: Variación de $|\mathbf{v_0}|$ (solo horizontal)

En este apartado se repiten los cálculos del apartado anterior cambiando las condiciones iniciales: la velocidad inicial de todas las partículas sólo tendrá componente X, con magnitud entre 0 y `moduloVelocidad`.

Para ello, se añade un bucle que sustituye al bucle de velocidades iniciales original cuando la variable `soloDesplHoriz = True`:

```
if soloDesplHoriz == True:
    for p in range(nParticulas):
        v[p,0] = moduloVelocidad*np.random.rand()
        v[p,1] = 0
```

Se realizan 4 simulaciones diferentes correspondientes a `moduloVelocidad = 2, 3, 4`. Las temperaturas obtenidas son, respectivamente, $T_2 = 24{,}3943$ K, $T_3 = 37{,}4676$ K y $T_4 = 54{,}3797$ K. Los gráficos con los resultados de velocidades y energías se recogen a continuación, aunque para verlos mejor y con sus animaciones se pueden encontrar en la carpeta `voluntario1\Resultados\Apartado 2`.

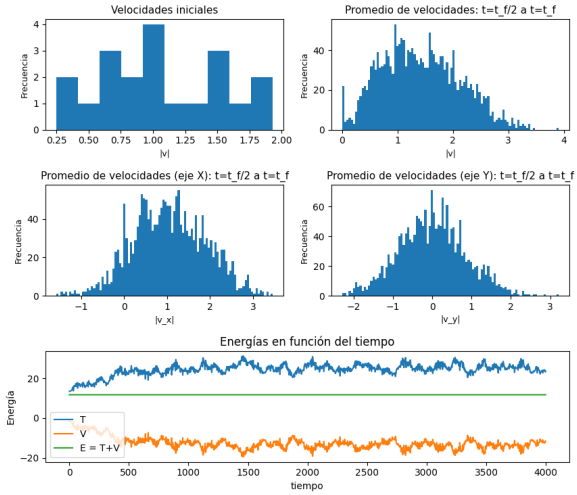


Figura 2: Resultados con $|v_{0,x}| \in [0, 2]$.

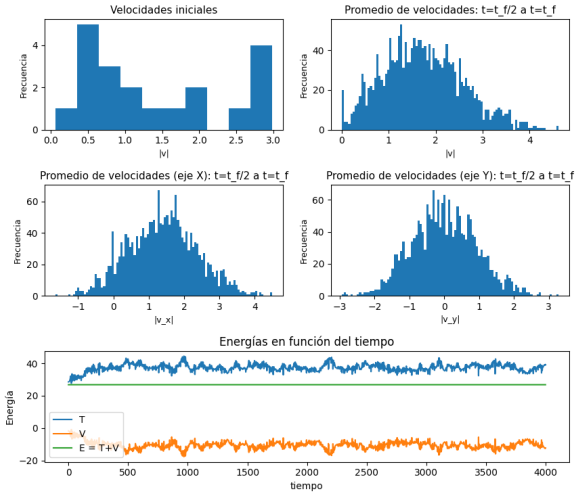


Figura 3: Resultados para $|v_{0,x}| \in [0, 3]$.

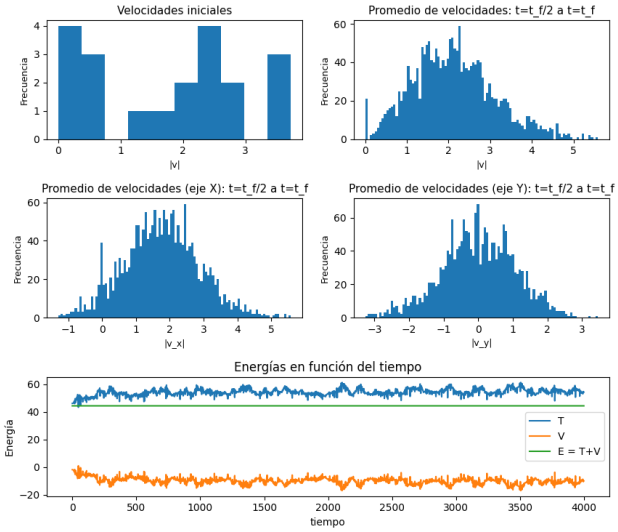


Figura 4: Resultados para $|v_{0,x}| \in [0, 4]$.

3.3. Apartado 3: Presión, temperatura y ecuación de estado

En este apartado se calculará la presión para, usando también la temperatura, determinar la ecuación de estado del gas.

Las condiciones de contorno varían: ahora la red es 10×10 y contiene 16 partículas, a las que se les otorga una velocidad inicial con `moduloVelocidad = 1, 2, 3, 4` y dirección aleatoria.

Para determinar la presión es necesario conocer las veces que las partículas atraviesan las paredes por unidad de tiempo y su velocidad en ese instante. Para esto se añade en la función `bordes()` el vector `np.array([fueraX,fueraY])`. Por ejemplo, en el eje X su valor en en es -1 para las partículas que cruzan el borde izquierdo ó +1 para las que cruzan el borde derecho, de modo que al multiplicar `fueraX` por `v[p,0]` el valor siempre será positivo. Haciendo esto también para el eje Y y luego dividiendo entre L para todas las partículas en un instante temporal, se obtiene el correspondiente elemento de presión, que se almacena en el vector `fuerzaParedes`:

```
for p in range(nParticulas):
    fuerzaParedes[t] += (fuera[p,0]*v[p,0]+fuera[p,1]*v[p,1])/L
```

Todo ese cómputo se realiza en el interior de la función `verlet()`, a la que se le ha añadido como output el vector `fuerzaParedes`.

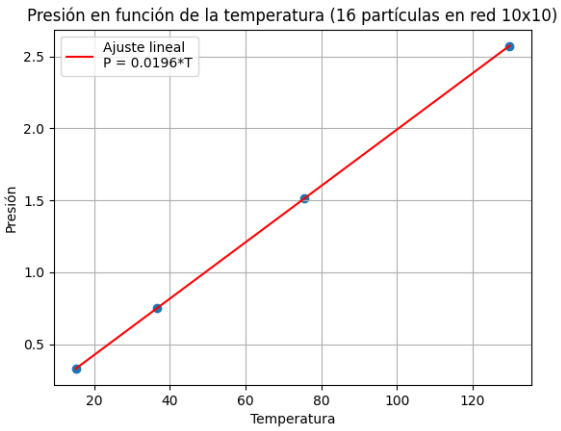
Para calcular la presión se suma `fuerzaParedes` a lo largo de cada unidad de tiempo, y se introduce en el vector `presion` que tiene dimensión `nIteraciones*h`, es decir, un valor por cada segundo de simulación. También se calcula la presión promedio.

```
presionPromedio = 0
presion = np.zeros(int(nIteraciones*h))
for i in range(int(nIteraciones*h)):
    for t in range(i*int(1/h),int(1/h)*(i+1)):
        presion[i] += fuerzaParedes[t]
    presionPromedio += presion[i]
presionPromedio = presionPromedio/(nIteraciones*h)
```

Se grafica de manera independiente usando matplotlib:

```
# REPRESENTACIÓN GRÁFICA - PRESIÓN
plt.rcParams["figure.figsize"] = (5,4)
plt.plot(presion, label="Presión")
plt.title("Presión en función del tiempo")
plt.xlabel("t")
plt.ylabel("Presión")
plt.axhline(y=presionPromedio, color='r', linestyle='-',
            label="Presión promedio: "+f"{(presionPromedio):.3f}")
plt.legend()
plt.show()
```

Tras ejecutar la simulación para todas los valores de `moduloVelocidad`, se recogen la presión promedio y la temperatura promedio de cada simulación, y se hace un ajuste lineal con el objetivo de obtener la ecuación de estado, es decir, expresar $\langle \mathrm{P} \rangle$ en función de $\langle T \rangle$.



Se obtiene la ecuación de estado $P = 0{,}0196 \cdot T$. El ajuste lineal $P = aT$ tiene una chi cuadrado de $\chi^2 = 0{,}9999983$.

Para realizar este ajuste se usa el programa `ecEstado.py`, que está en la carpeta del tercer apartado. Se ha usado PhindAI para escribir las líneas de código correspondientes a la regresión.

3.4. Apartado 4: Transición de fase sólido-líquido

De nuevo, las condiciones iniciales varían: se simularán 16 partículas en una caja 4×4, estando todas las partículas inicialmente en reposo. Para ello, se introducen todas las líneas de código correspondientes a la asignación de velocidades iniciales en el interior de la sentencia `if reposo == False`, de modo que cuando la variable `reposo` sea `True`, dichas líneas de código no se ejecutarán y no habrá velocidad inicial.

Se observa que, si bien inicialmente la distribución es en forma de red cuadrada, esta estructura colapsa inmediatamente y poco después se forma una red en la que las partículas se ordenan en forma de triángulos. Esta red triangular tiene dos estados cuasi-estables, uno con las bases de los triángulos dispuestas de manera vertical y otra con las bases horizontales.

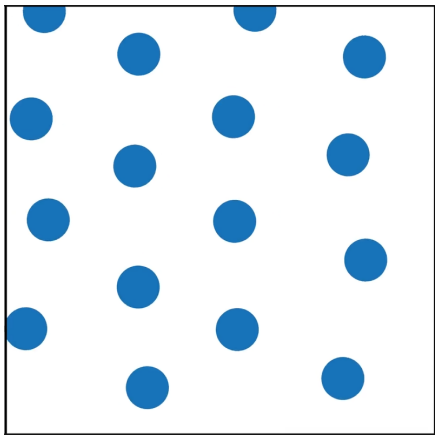


Figura 5: Red triangular con base vertical.

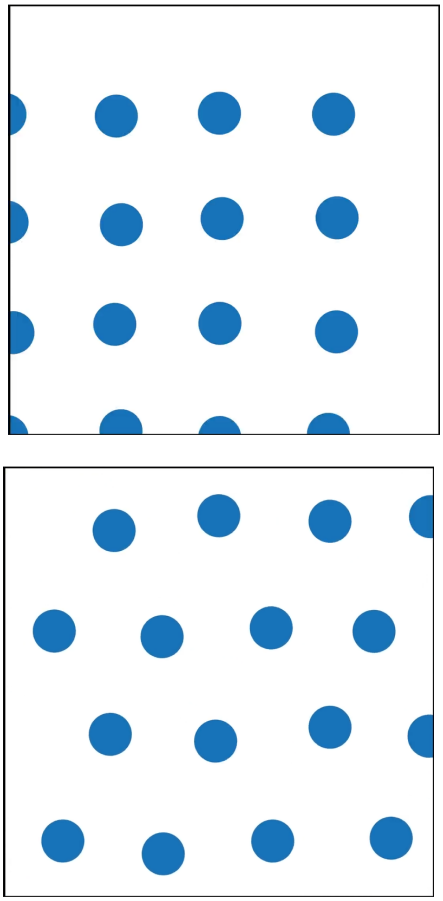


Figura 6: Red triangular con base horizontal.

3.5. Apartado 5: Evolución temporal de distribución en forma hexagonal

Para este apartado se quiere ver la evolución temporal del sistema si inicialmente disponemos las partículas en forma hexagonal. Para ello, se ha introducido una nueva variable llamada `redHexagonal24` que, cuando es `True`, establece las condiciones iniciales `nParticulas = 24`, `L = 4`, y aumenta la precisión para estudiar mejor qué pasa en los primeros instantes de evolución del sistema.

Cuando `redHexagonal24 = True`, en lugar del algoritmo que se ha usado hasta ahora de posiciones iniciales en forma de red cuadrada se ejecuta el siguiente.

Se calcula el espaciado entre partículas y/o huecos como `espaciado = L/huecosPorFila`, y se inicializa la coordenada $y = 0$. A continuación, se asignan las posiciones de las partículas de 4 en 4 (en cada fila hay 4 partículas y 2 huecos), de manera que las filas impares se desplazan en la coordenada x respecto a las impares para que quede finalmente la estructura de hexágonos. Tras colocar cada fila se incrementa el valor de y en `espaciado`.

```
huecosPorFila = 6
espaciado = L/huecosPorFila
y = 0
for f in range(huecosPorFila):
    if f%2==0:
        r[4*f] = np.array([0.0,y])
        r[4*f+1] = np.array([2*espaciado,y])
        r[4*f+2] = np.array([3*espaciado,y])
        r[4*f+3] = np.array([5*espaciado,y])
    else:
        r[4*f] = np.array([0.5*espaciado,y])
        r[4*f+1] = np.array([1.5*espaciado,y])
        r[4*f+2] = np.array([3.5*espaciado,y])
        r[4*f+3] = np.array([4.5*espaciado,y])
    y += espaciado
```

Este algoritmo sólo funciona para `nParticulas = 24`.

Al ejecutar la simulación, se observa cómo inicialmente parten de la estructura de red hexagonal pero esta colapsa inmediatamente. Eventualmente aparecen localmente redes triangulares como la del apartado anterior, solo que son menos estables debido a la mayor energía del sistema y se disipan y cambian de lugar rápidamente.

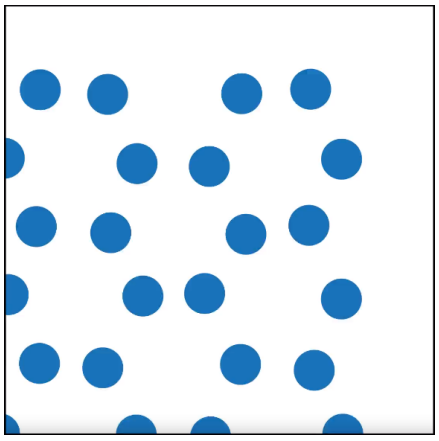


Figura 7: Red hexagonal en $t = 0$.

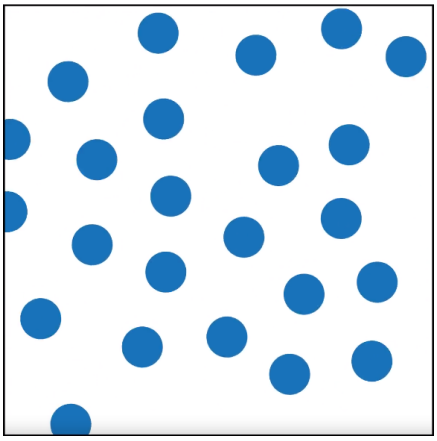


Figura 8: Red triang. en parte del sistema.

3.6. Apartado 6: Fluctuación en la posición de una partícula

Se realizarán varias simulaciones con las condiciones del apartado 3.4, solo que aumentando las velocidades iniciales cada cierto tiempo en la misma simulación para que aumente así la temperatura.

Se añade una nueva variable, `acelerarRapido`, que en caso de ser `True` hace que las velocidades de todas las partículas se multipliquen por 1.5 en $t = 20, 30, 35$ y 45 s. Esto se hace añadiendo en el interior del bucle de Verlet la siguiente sentencia `if:`

```
# ACELERAR LAS PARTÍCULAS CADA CIERTO TIEMPO
if acelerarRapido:
    if t*h==20 or t*h==30 or t*h==35 or t*h==45:
        for p in range(nParticulas):
            v[p] *= 1.5
```

Además, en cada simulación se elegirá una partícula aleatoria y se representará gráficamente la fluctuación de su posición, $\langle (r(t) - r(t_0))^2 \rangle$

```
# CÁLCULO DE LA FLUCTUACIÓN RESPECTO A LA POSICIÓN INICIAL DE UNA PARTÍCULA p
desvPos = np.zeros(int(nIteraciones*h))
p = np.random.randint(0,nParticulas)
for i in range(int(nIteraciones*h)):
    for t in range(i*int(1/(skip*h)),int(1/(skip*h))*(i+1)):
        desvPos[i] += np.linalg.norm(distanciaGlobal(r,0,t,p,p,L,lMedios))**2
    desvPos[i] = desvPos[i]*skip*h
```

donde, de nuevo, se usa la función `distancia()`, solo que se calcula la distancia entre la posición de p en t y en t_0. También se calcula ahora la temperatura en cada instante temporal para representarla junto a la fluctuación de la posición de p. Igual que antes, se usa el teorema de equipartición:

```
# CÁLCULO DE LA TEMPERATURA
temperatura = np.zeros(int(nIteraciones*h))
for i in range(int(nIteraciones*h)):
    for t in range(i*int(1/(skip*h)),int(1/(skip*h))*(i+1)):
        for p in range(nParticulas):
            temperatura[i] += v[t,p,0]**2 + v[t,p,1]**2
    temperatura[i] = 0.5*temperatura[i]*skip*h
```

Mediante matplotlib representamos las velocidades y energías y también estas magnitudes para ver qué relación guardan:

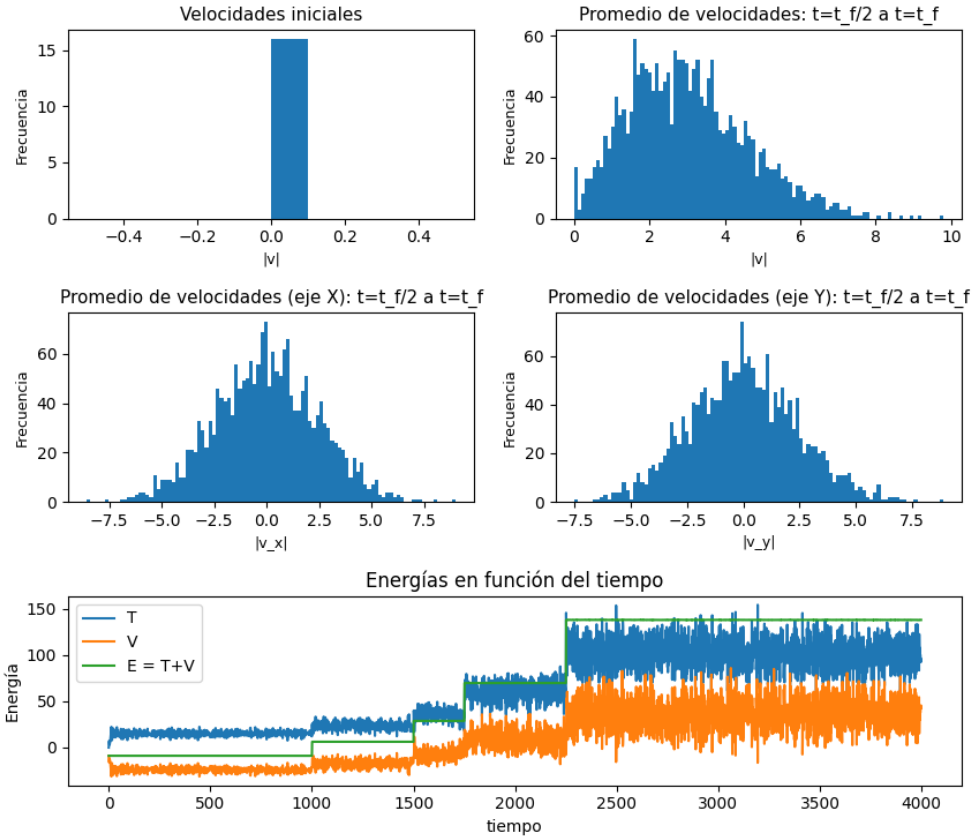


Figura 9: Velocidades y energías - |v| creciente.

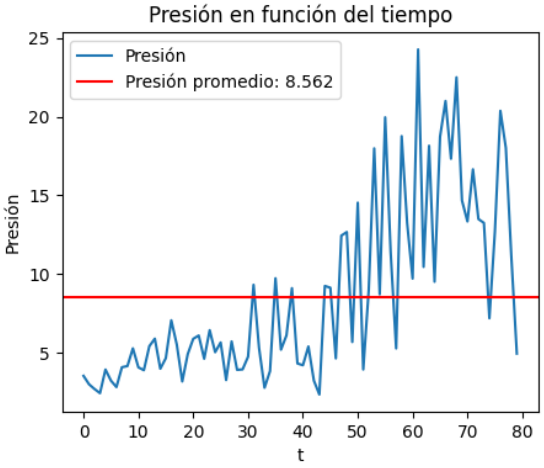


Figura 10: Presión en función de t.

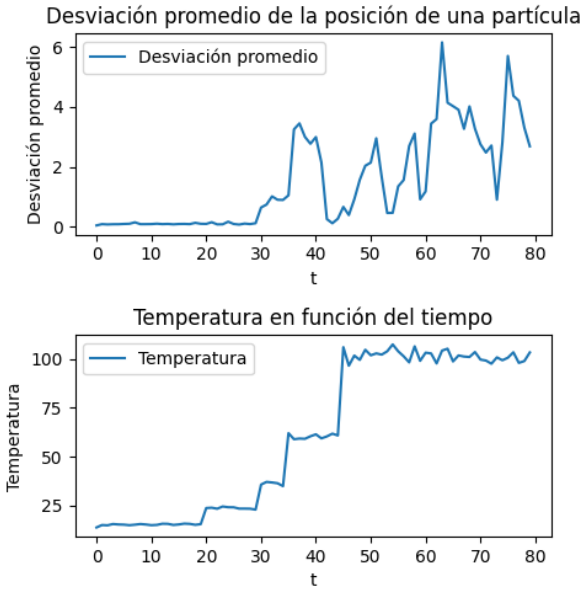


Figura 11: Desviación de una partícula y temperatura en función de t.

En la gráfica de energía se aprecia claramente los momentos en los que se incrementan las velocidades, en forma de cuatro escalones de distinta altura (nota: en la gráfica $E - t$ el tiempo no está expresado en segundos).

En la gráfica de desviación promedio de la posición de una partícula y temperatura en función del tiempo (fig. 11) se aprecia también que en el tercer aumento de velocidad ($t \simeq 35$) hay una gran subida en la desviación promedio de la posición de la partícula, y en general aumentan con la temperatura.

3.7. Apartado 7: Separación cuadrática media de un par de átomos

Trabajando de igual manera que en el apartado anterior, se añade otra nueva variable llamada `acelerarLento`, que multiplica las velocidades de todas las partículas por 1.1 cuando t es un múltiplo de 60 s.

```
elif acelerarLento:
    if (t*h)%60==0:
        for p in range(nParticulas):
            v[p] *= 1.1
```

También en cada simulación se elegirán dos partículas aleatorias p_1 y p_2 y se calculará la separación cuadrática media $\langle(\Delta r_{ij}(t))^2\rangle$ entre ellas,

```
# CÁLCULO DE LA SEPARACIÓN CUADRÁTICA MEDIA ENTRE UN PAR DE ÁTOMOS
desvCuad = np.zeros(int(nIteraciones*h))
p1 = np.random.randint(0,nParticulas)
p2 = np.random.randint(0,nParticulas)
while p2==p1:
    p2 = np.random.randint(0,nParticulas)
for i in range(int(nIteraciones*h)):
    for t in range(i*int(1/(skip*h)),int(1/(skip*h))*(i+1)):
        desvCuad[i] += np.linalg.norm(distanciaGlobal(r,t,t,p1,p2,L,lMedios))**2
    desvCuad[i] = desvCuad[i]*skip*h
```

El bucle `while` es para asegurarse de que no se ha elegido la misma partícula dos veces.

Tras realizar varias simulaciones con las mismas condiciones iniciales, se obtienen estos gráficos (cada columna corresponde con una ejecución):

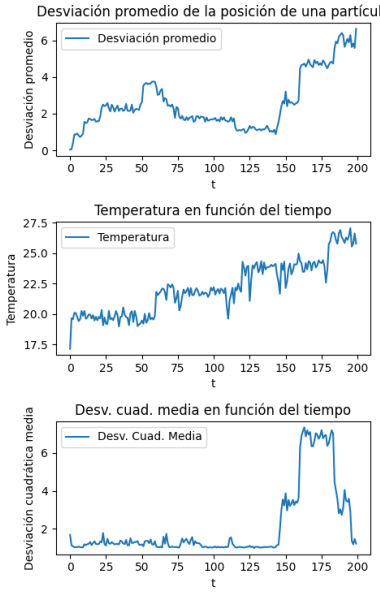


Figura 12: Primera ejecución.

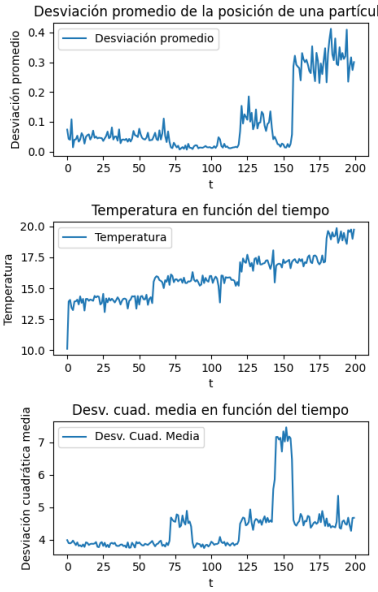


Figura 13: Segunda ejecución.

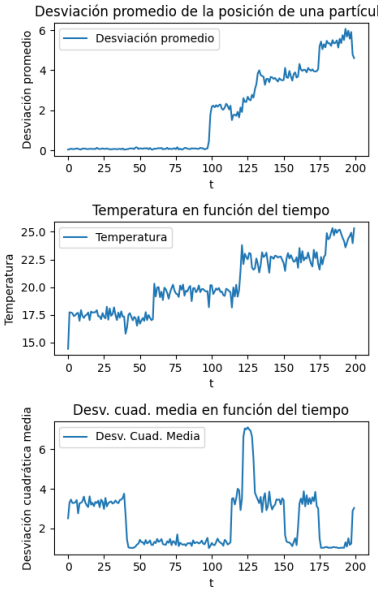


Figura 14: Tercera ejecución.

Para observar las gráficas mejor se recomienda verlas por separado en la carpeta correspondiente al apartado 7.

En la primera ejecución se produce el salto de la desviación cuadrática media para $T = 22{,}37$, en la segunda ejecución para $T = 17{,}12$ K y en la tercera para $T = 22{,}49$ K. A partir de estos valores se puede estimar una temperatura crítica $T_c = 20{,}67 \pm 2{,}53$ K con una confianza del 95 %.

4. Apéndice: Rendimiento

A continuación se harán dos análisis del rendimiento del programa: por un lado se comparará el rendimiento con y sin Numba, y por otro laldo se comparará el rendimiento del programa en un PC sobremesa que cuenta con un RYZEN 5 2600 con el rendimiento en JOEL. Se ha elegido comparar con y sin Numba en lugar del multithreading ya que la parte de este programa que se puede paralelizar más eficientemente son las funciones que calculan promedios estadísticos, pero en cuanto a tiempo de cómputo esto conforma una parte muy pequeña del tiempo de ejecución del programa, ya que la mayoría del tiempo se invierte en el bucle de la función `verlet()`. Es por esto que, al no ver cambios importantes en el rendimiento, se ha elegido esta otra comparación.

•Comparación con y sin Numba:

Se puede hacer este análisis incrementando progresivamente el número de partículas, pero en este caso se ha elegido incrementar el número de iteraciones en su lugar. Los resultados se recogen en la siguiente gráfica:

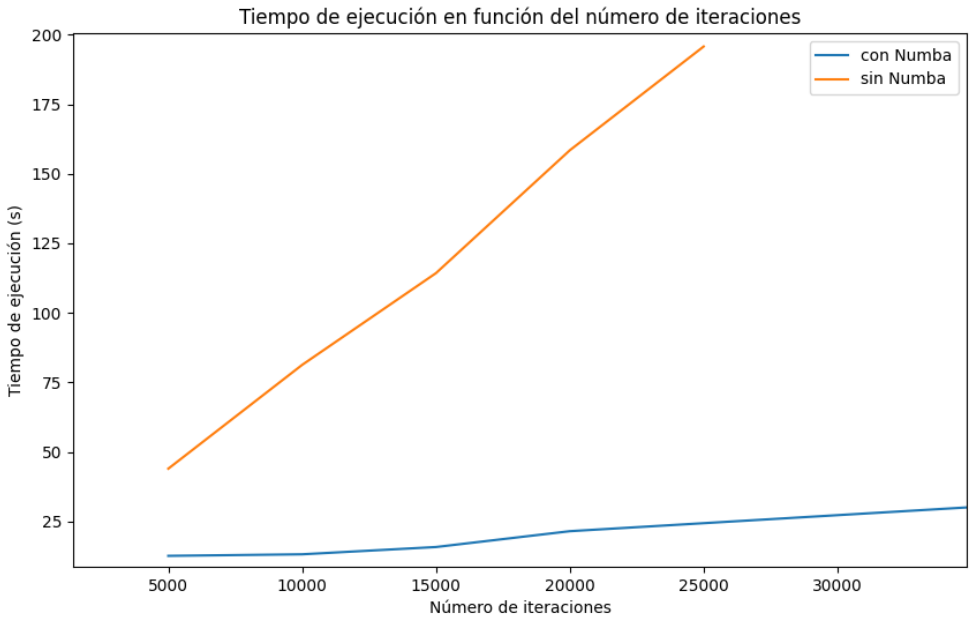


Figura 15: Comparación entre el tiempo que tarda la ejecución del programa en función de `nIteraciones`, según se use Numba o no.

Donde se aprecia la gran diferencia en el tiempo que tarda la simulación. Para tener una idea de cuántas iteraciones son necesarias usando Numba para que el tiempo de ejecución sea similar al de 25000 iteraciones del programa sin Numba, se ha realizado esta otra gráfica:

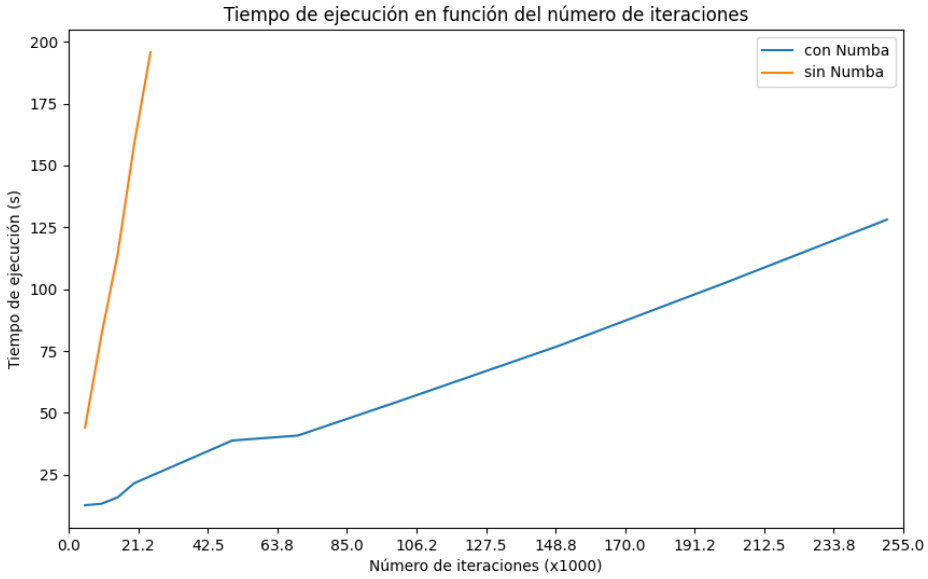


Figura 16: Comparación entre el tiempo que tarda la ejecución del programa en función de `nIteraciones`, según se use Numba o no.

Se aprecia que el programa con Numba sigue siendo más rápido incluso con 10 veces más iteraciones aproximadamente.

- **Comparación PC - JOEL:**

A continuación se recogen los datos de las ejecuciones del programa en JOEL y PC para comparar sus rendimientos. Se aprecia que en JOEL tarda un poco más, pero esto se debe a que el procesador del PC es un poco más potente que los de JOEL cuando se trata de trabajar con un solo núcleo. En un programa creado para usar muchos núcleos se apreciaría mucho más claramente la potencia de JOEL.

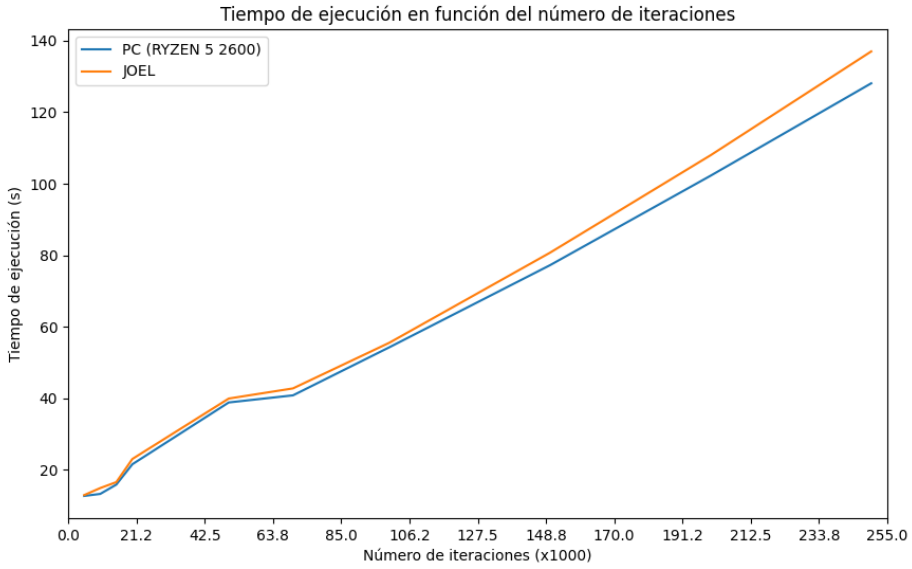


Figura 17: Comparación entre el tiempo que tarda la ejecución del programa en función de `nIteraciones`, en PC y en JOEL

Referencias

[1] N. J. Giordiano, H. Nakanishi. *"Computational Physics"*.