Universidad de Extremadura

Facultad de Ciencias

Simulación de Dinámica Molecular de Discos Duros en 2D

Tercer Trabajo Evaluable

Autor: Juan Miguel Polo Barahona

Asignatura: Ampliación de Física del Estado Sólido

Curso: 2024/2025

1. Análisis del Código

Se comienza importando las librerías necesarias para la ejecución de la simulación. Cabe destacar el uso de librerías como *random* y *os*, que permiten respectivamente la generación de números aleatorios y la gestión de archivos y rutas del sistema operativo.

```
import math
import numpy as np
import random
import bisect # libreria de ordenacion de listas (para lista de cols.)
from operator import itemgetter, attrgetter
# import matplotlib # esta es para usar graficos python. a implementar en nuevas versiones
from os import system, remove
import os
```

Lista 1: Importación de librerías.

A continuación, se verifica que exista una carpeta denominada *Datos*, la cual se utilizará para almacenar los distintos archivos que contienen las velocidades y posiciones de las partículas. El sistema simulado consiste en 10 partículas confinadas en una caja bidimensional cuadrada de lado $L = 10R$, lo que corresponde a un factor de empaquetamiento aproximado de $\nu \sim 0{,}3$, siendo $R = 1$ el radio de cada partícula en unidades arbitrarias.

Un parámetro crucial en esta simulación es el coeficiente de restitución, α, que determina el grado de conservación de la energía cinética durante las colisiones. En este caso, $\alpha = 1$, lo que implica que la energía cinética se conserva completamente en cada choque. Este hecho tendrá implicaciones importantes en la interpretación de los resultados, como se discutirá más adelante.

```
os.makedirs('Datos', exist_ok=True)

# radio de las particulas
R=1.
# tamano del sistema
LX = 10*R
LY = 10*R
# tamano del sistema menos un radio (para situar las particulas)
LXR = LX*0.5-R
LYR = LY*0.5-R
# fraccion de empaquetamiento
#nu = 0.72
# numero de particulas
#npart = int(math.floor(nu*LX*LY/(math.pi*R*R)))

npart = 10
# numero de pasos temporales (cols.)
#nt = 100 * npart
nt = 2000

# coef. de restitucion
alfa = 1.0
```

```
    # parametro de control para evitar solapacion de parts. por error numerico
    # es posible que tol=0 funcione
    tol = 1.0e-20
    # colisiones/part. debe ser real (no borrar nunca el prefactor 1.0)
    # es mas, este parametro no debe modificarse
    ncp=1.0*nt/npart
    # numero de ncps entre snapshots. si icp=1.0*npart/nt -> se guardan todas las cols.
    # se recomienda icp=npart/nt para etapa de desarrollo de codigo
    # en todo caso icp < nt  si se quieren guardar datos; icp > nt  si no se quiere guardar

    # iteraciones entre snapshots que sale
    utermo = 1
    #utermo=int(math.ceil(icp))
```

Lista 2: Parámetros del sistema.

El siguiente paso consiste en inicializar el sistema. Para ello, se asignan valores nulos a las velocidades y posiciones de cada uno de los discos, así como a la temperatura y al parámetro a_2, el cual proporciona información sobre la forma de la distribución de velocidades. En equilibrio térmico, $a_2 \approx 0$, por lo que puede utilizarse como un indicador del estado del sistema. Asimismo, se inicializa el tiempo de simulación y se utiliza la librería *random* para generar configuraciones estadísticamente distintas en cada ejecución.

```
    #   inicializa listas de velocidades y posiciones
    vx = [0. for i in range(npart)]
    vy = [0. for i in range(npart)]

    x = [0. for i in range(npart)]
    y = [0. for i in range(npart)]

    #   inicializa listas temporales de T y a2
    temp = [0. for i in range(nt+1)]
    a2 = [0. for i in range(nt+1)]

    #inicializa listas relacionadas con las colisiones
    listacol = []
    listacol_orden = []
    ij = []

    # inicializa el tiempo
    t = 0.
    dt = 0.
    it = 0

    #   inicializa el generador aleatorio. cada vez que se lanza la simulacion usa una semilla aleatoria
    # es decir, ejecuciones consecutivas hacen simulaciones estadisticamente diferentes (replicas)
```

```
    # si no se quiere esta propiedad, escribir: random.seed(1)
    random.seed()
```

Lista 3: Inicialización del sistema.

La función *propaga* determina la evolución temporal de las partículas. Dado que se trata de discos duros, la única interacción que experimentan es a través de colisiones elásticas entre ellos o con las paredes de la caja. Por tanto, entre colisiones, las partículas se desplazan libremente con velocidad constante, cumpliendo las ecuaciones del movimiento rectilíneo uniforme.

Por su parte, la función *midedist* calcula la distancia al cuadrado entre dos partículas y le resta el cuadrado del diámetro ($4R^2$). El resultado, almacenado en la variable `dist2`, permite identificar situaciones no físicas: si `dist2` < 0, significa que las partículas se superponen, lo cual contradice el modelo de discos duros, en el que las partículas son impenetrables.

```
    #   avanza las particulas con v cte un intervalo de tiempo dt

    def propaga(dt):
        for i in range(npart):
            x[i]=x[i]+vx[i]*dt
            y[i]=y[i]+vy[i]*dt


    #   calcula los tiempos de colision p-p. para un par (i,j)


    def midedist(i,j):
        dx=x[i]-x[j]
        dy=y[i]-y[j]
        dist2=(dx*dx+dy*dy)-4*R*R
```

Lista 4: Funciones *propaga y midedist*.

La función *tcol* calcula el tiempo que tardarán en colisionar dos partículas i y j. Se parte de las posiciones relativas (dx, dy) y velocidades relativas (dv_x, dv_y) entre las dos partículas. A partir de estos datos, se calcula el producto escalar $\vec{r}_{ij} \cdot \vec{v}_{ij}$, almacenado en la variable `drdv`, el cual indica si las partículas se están acercando (`drdv` < 0) o alejando (`drdv` > 0). Solo si se están acercando se considera la posibilidad de colisión.

Posteriormente, se evalúa si existe una solución real. Si (`raiz` < 0), las trayectorias no llevan a una colisión, y el tiempo se fija como infinito.

En caso de existir una colisión, se calcula el instante de colisión `vdt` y se verifica que esta ocurra dentro de los límites del sistema. Si alguna de las posiciones futuras está fuera de la caja, se descarta la colisión. Finalmente, si es válida, se inserta en una lista ordenada (`listacol`), de modo que el evento más próximo esté siempre en la primera posición.

```
    def tcol(i,j):
    dx=x[i]-x[j]
    dy=y[i]-y[j]
    dvx=vx[i]-vx[j]
    dvy=vy[i]-vy[j]
    drdv=dx*dvx+dy*dvy
    # estructura condicional de colision p-p
```

```
8       # condicion de acercamiento
9       if drdv > 0:
10              vct=float('inf')
11      else:
12          dist2=(dx*dx+dy*dy)-4*R*R # distancia instantanea entre
               dos particulas
13          raiz=drdv*drdv-dist2*(dvx*dvx+dvy*dvy) # condicion de
               solucion real en la condicion de col.
14      if raiz < 0:
15          vct=float('inf')
16          # si hay sol. real, guarda en dt el tiempo de col.
17      else:
18          vdt=dist2/(math.sqrt(raiz)-drdv)
19          # posicion de la colision. si en realidad la colision
               ocurriria fuera del sistema, descartala
20          xicol=x[i]+vx[i]*vdt
21          yicol=y[i]+vy[i]*vdt
22          xjcol=x[j]+vx[j]*vdt
23          yjcol=y[j]+vy[j]*vdt
24          # estructura condicional de col. fuera del sistema
25      if math.fabs(xicol)>LXR:
26          vdt=float('inf')
27      elif math.fabs(xjcol)>LXR:
28          vdt=float('inf')
29      elif math.fabs(yicol)>LYR:
30          dt=float('inf')
31      elif math.fabs(yjcol)>LYR:
32          vdt=float('inf')
33      else:
34          # coloca en la lista de colisiones ordenada de menor a
               mayor
35          # usa un algoritmo rapido 'binary search' para la
               colocacion
36          bisect.insort(listacol,[vdt,[i,j]])
```

Lista 5: Función *tcol*.

A continuación, se definen las funciones responsables de describir la interacción de una partícula con las paredes del sistema. En particular, la función `tpcol` calcula el tiempo que tarda la partícula i en colisionar con alguna de las paredes de la caja.

Si la componente x (o y) de la velocidad es nula, la partícula nunca colisionará con las paredes verticales (u horizontales, respectivamente), por lo que el tiempo hasta dicha colisión se considera infinito. En caso contrario, se calcula el tiempo que falta para alcanzar la pared correspondiente, dependiendo de la dirección del movimiento. Por ejemplo, si la partícula se desplaza hacia la derecha, se evalúa el tiempo hasta colisionar con la pared derecha y se le asigna un identificador (en este caso, -3). Se procede de forma análoga para las restantes direcciones, usando los identificadores -1, -2 y -4 para las paredes izquierda, inferior y superior, respectivamente.

Finalmente, se comparan los tiempos de colisión con las paredes y se selecciona el menor, junto con su identificador. Después, esta información se añade a la lista de próximos eventos de colisión (`listacol`).

La función `pcolisiona` actualiza la velocidad de la partícula tras una colisión con una de las paredes del sistema. Tras la colisión, la partícula invierte la componente de su velocidad correspondiente al eje normal a la pared con la que ha chocado. Así, si la colisión se produce con una pared vertical (izquierda o derecha), se invierte la componente v_x, mientras que si se produce con una pared horizontal (superior o inferior), se invierte la componente v_y.

```
    def tpcol(i):
        if vx[i]==0:
            tx=float('inf')
        elif vx[i]<0:
            ltx=[-(LXR+x[i])/vx[i],-1]
        elif vx[i]>0:
            ltx=[(LXR-x[i])/vx[i],-3]

        if vy[i]==0:
            ty=float('inf')
        elif vy[i]<0:
            lty=[-(LYR+y[i])/vy[i],-2]
        elif vy[i]>0:
            lty=[(LYR-y[i])/vy[i],-4]

        ltm=sorted([ltx,lty],key=itemgetter(0))
        vdt=ltm[0][0]
        im=ltm[0][1]
        bisect.insort(listacol,[vdt,[i,im]])

    # actualiza velocidad de part. que colisiona con pared

    def pcolisiona(ii):
        if ii[1]==-1 or ii[1]==-3:
            vx[ii[0]]=-vx[ii[0]]
        elif ii[1]==-2 or ii[1]==-4:
            vy[ii[0]]=-vy[ii[0]]
```

Lista 6: Función *tpcol y pcolisiona*.

La función `colisiona` simula la colisión entre dos partículas. Primero, calcula la distancia entre las dos partículas, obteniendo las diferencias en las coordenadas x y y, es decir, dx y dy. A continuación, determina la norma de esta distancia, llamada σ_{norma}, y utiliza esta norma para normalizar el vector que apunta en la dirección de la colisión, obteniendo los vectores unitarios σ_x y σ_y.

Luego, calcula la velocidad relativa de las partículas en la dirección de la colisión, proyectando las velocidades de las partículas sobre el vector unitario normal.

Finalmente, las velocidades de las partículas se actualizan atendiendo al coeficiente de restitución α.

```
    def colisiona(par):

        # la 1a particula la llamamos i y la 2a, j
        i=par[0]
        j=par[1]

```

```
        dx=x[i]-x[j]
        dy=y[i]-y[j]

        # construye sigma_ij unitario
        sigma_norma=math.sqrt(dx*dx+dy*dy)
        sigmax=dx/sigma_norma
        sigmay=dy/sigma_norma

        # construye g \cdot sigma (g, vel relativa)
        gsigma=(vx[i]-vx[j])*sigmax+(vy[i]-vy[j])*sigmay

        # actualiza vel. de 1a. part.
        vx[i]=vx[i]-0.5*(1+alfa)*gsigma*sigmax
        vy[i]=vy[i]-0.5*(1+alfa)*gsigma*sigmay

        # actualiza vel. de 2a. part.
        vx[j]=vx[j]+0.5*(1+alfa)*gsigma*sigmax
        vy[j]=vy[j]+0.5*(1+alfa)*gsigma*sigmay
```

Lista 7: Función *colisiona*.

El siguiente paso consiste en colocar las partículas dentro de una caja rectangular que esté centrada en el origen. Las coordenadas x e y de la primera partícula se generan de manera aleatoria utilizando una distribución uniforme.

Luego, se van colocando las partículas restantes una por una. Para cada nueva partícula, se genera una posición aleatoria dentro de la caja, asegurándose de que no se superponga con ninguna de las partículas que ya están colocadas. Para garantizar que no haya solapamiento, se verifica que la distancia euclidiana entre el centro de la nueva partícula y cada una de las partículas anteriores sea mayor que $2R$, que es el doble del radio de las partículas. Si no se cumple esta condición, se descarta la posición y se genera una nueva hasta que se logre.

Una vez que todas las partículas están colocadas sin solapamientos, se les asignan velocidades iniciales aleatorias. Estas velocidades se generan siguiendo una distribución gaussiana normal estándar para ambas componentes x e y.

Finalmente, se calcula el tiempo hasta la próxima colisión entre cada par de partículas distintas (i, j) utilizando la función `tcol(i,j)`. También se determina, para cada partícula, el tiempo hasta la colisión con las paredes del recinto, llamando a la función `tpcol(i)`.

```

    # colocacion de las particulas
    x[0]=random.uniform(-LXR, LXR)
    y[0]=random.uniform(-LYR, LYR)

    # condicion de solapamiento
    for i in range(1,npart):
        dr=False
        while dr==False:
            x[i]=random.uniform(-LXR, LXR)
            y[i]=random.uniform(-LYR, LYR)
            # condicion de no solapamiento con las pos. ya
               generadas
            for j in range(0,i):
```

```
                dr=((x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j
                    ])>4*R*R)
                if dr==False:
                    break


    #   velocidades aleatorias para las velocidades, distribucion
        gaussiana
    for i in range(npart):
        vx[i]=np.random.randn()
        vy[i]=np.random.randn()

    #### bucle en particulas. Calcula tiempos iniciales de
       colision

    for i in range(npart-1):
        for j in range(i+1,npart):
            tcol(i,j)   # para todos los pares de particulas (i,j
                ) con j>i
    for i in range(npart):
        tpcol(i)    # con la pared
```

Lista 8: Estado inicial del sistema.

A continuación, se guarda en disco el estado inicial del sistema, es decir, las posiciones y velocidades de todas las partículas en el instante $t = 0$.

Para ello, se utiliza un contador de iteraciones (`it`) y se almacenan los archivos de tipo .dat en la carpeta `Datos`.

Cada línea representa una partícula. Los archivos de tipo `xy****.dat` contienen las coordenadas x e y de cada partícula, escritas en columna, mientras que los archivos `vxvy****.dat` guardan las componentes v_x y v_y de las velocidades de cada partícula.

```
    it=0
    # formatea el nombre de archivo de posiciones y escribelo en
       disco
    inum='{0:04d}'.format(it)
    nombre='Datos/xy'+inum+'.dat'
    with open(nombre,'w') as archivo:
        for i in range(npart):
            archivo.write('{0:10.2f} {1:10.2f}\n'.format( x[i], y
                [i]))
    archivo.closed

    # formatea el nombre de archivo de posiciones
    inum='{0:04d}'.format(it)
    nombre='Datos/vxvy'+inum+'.dat'
    with open(nombre,'w') as archivo:
        for i in range(npart):
            archivo.write('{0:10.2f} {1:10.2f}\n'.format( vx[i],
                vy[i]))
    archivo.closed
```

Lista 9: Almacenamiento de las posiciones y velocidades iniciales.

El bucle principal avanza en el tiempo desde $t = 0$ hasta completar `nt` iteraciones. En cada iteración, se extrae el primer evento de la lista de colisiones `listacol`, que corresponde a la colisión más cercana. Se calcula el tiempo `dt` hasta ese evento y se identifican las partículas involucradas en la colisión mediante el par `ij` $= (i, j)$.

Luego, se eliminan de la lista `listacol` todos los eventos que involucran a las partículas i o j, ya que sus trayectorias cambiarán tras la colisión, y los tiempos previamente calculados ya no son válidos.

El tiempo total de la simulación se actualiza como $t+$`dt`, y se reducen los tiempos restantes de colisión en `listacol` en la misma cantidad. Después, se actualizan las posiciones de todas las partículas usando la función `propaga(dt)`.

Una vez que las posiciones están actualizadas, también se modifican las velocidades de las partículas que han colisionado. Si la colisión es con una pared, se llama a la función `pcolisiona`. De lo contrario, la colisión es entre dos partículas y se utiliza la función `colisiona`.

Después de ajustar las velocidades, se recalculan los nuevos tiempos de colisión de las partículas que participaron en el choque. Para cada partícula, se calcula su nuevo tiempo de colisión con las paredes mediante `tpcol`, y sus tiempos de colisión con el resto de partículas mediante `tcol`. Estos nuevos eventos se insertan en `listacol` en el orden correcto.

```
    for it in range(1, nt+1):

    dt = listacol[0][0] * (1 - tol)
    ij = listacol[0][1]

    listacol = list(filter(lambda x: x[1][0] != ij[0], listacol))
    listacol = list(filter(lambda x: x[1][1] != ij[0], listacol))

    if ij[1] > 0:
        listacol = list(filter(lambda x: x[1][0] != ij[1],
           listacol))
        listacol = list(filter(lambda x: x[1][1] != ij[1],
           listacol))

    t += dt

    limit = range(len(listacol))
    c = [[listacol[i][0] - dt, listacol[i][1]] for i in limit]
    listacol = c


    # actualiza primero las posiciones de las parts.,
    # justo hasta la colision que primero ocurre
    propaga(dt)


    # actualiza vels. de part(s). que ha(n) colisionado si la col
       . es con un muro (pcolisiona)
    # la condicion de colision con un muro es que la "segunda
       particula" tiene indice negativo
    if ij[1]<0:
```

```
28          pcolisiona(ij)
29      # en caso contrario, la col. es entre dos part. (colisiona)
30      else:
31          colisiona(ij)
32
33      # ahora calculamos los tiempos de col. nuevos para las nuevas
            trayectorias
34      # de las particulas que colisionarion,
35      # las funciones tcol y tpcol ademas recolocaran ordenadamente
            esos t en listacol
36
37      # primera particula
38      i=ij[0]
39      # nuevos tiempos de col. de la 1a particula que acaba de
            colisionar
40      tpcol(i)    # con la pared
41
42      for j in range(i):
43          tcol(j,i)   # para todos los pares de particulas (i,j)
                con j<i
44
45      for j in range(i+1,npart):
46          tcol(i,j)   # para todos los pares de particulas (i,j)
                con j>i
47
48      # segunda particula, solo si no es un muro (y por tanto,
            tiene indice positivo)
49      if ij[1]>0:
50          i=ij[1]
51          # nuevos tiempos de col. de la 2a particula que acaba de
                colisionar
52          tpcol(i)    # con la pared
53
54          for j in range(i):
55              tcol(j,i)   # para todos los pares de particulas (i,j
                    ) con j<i
56
57          for j in range(i+1,npart):
58              tcol(i,j)   # para todos los pares de particulas (i,j
                    ) con j>i
```

Lista 10: Simulación.

El siguiente bloque guarda iterativamente las posiciones y velocidades de las partículas en cada instante. Para ello, se comprueba si la iteración actual `it` es múltiplo de `utermo`. Si esta condición se cumple, se calcula el índice de archivo `ia` y se muestra por pantalla el número de iteración actual y el número de archivo correspondiente.

De la misma forma que en el estado inicial, se guardan las posiciones y velocidades de las partículas en el archivo `Datos/xy****.dat` y `Datos/vxvy****.dat`, respectivamente.

A continuación, se estima la temperatura del sistema como la energía cinética media de las partículas:

$$T = \frac{1}{\mathtt{N}} \sum_{i=1}^{\mathtt{N}} (v_x^2 + v_y^2)$$

Finalmente, se determina el parámetro `a2` cuya cuantía es una medida de la desviación respecto a una distribución gaussiana de velocidades.

```
    ##  Escribe pos. y vels. iterativamente
    if (it%utermo==0):
        ia=it/utermo
        print ("######## it: ########", it) # imprime it (n. de
           cols.) #opcional
        print ("######## no. archivo: ########", ia) # n. de
           archivo #opcional
        inum='{0:04d}'.format(int(ia))
        nombre='Datos/xy'+inum+'.dat'
        with open(nombre,'w') as archivo:
            for i in range(npart):
                archivo.write('{0:10.2f} {1:10.2f}\n'.format( x[i
                   ], y[i]))
        archivo.closed

        #system('gnuplot -persist estado.gnuplot')
        #    remove('tmp.gp')

        inum='{0:04d}'.format(int(ia))
        nombre='Datos/vxvy'+inum+'.dat'
        with open(nombre,'w') as archivo:
            for i in range(npart):
                archivo.write('{0:10.2f} {1:10.2f}\n'.format( vx[
                   i], vy[i]))
        archivo.closed

        # bucle de medicion de T y a2

        temp[int(ia)]=0.
        a2[int(ia)]=0.
        for i in range(npart):
            vv=vx[i]*vx[i]+vy[i]*vy[i]
            temp[int(ia)]=temp[int(ia)]+vv
            a2[int(ia)]=a2[int(ia)]+vv*vv
        temp[int(ia)]=temp[int(ia)]/npart
        a2[int(ia)]=a2[int(ia)]/(temp[int(ia)]*temp[int(ia)]*
           npart)
        a2[int(ia)]=(a2[int(ia)]-2.0)*0.5
```

Lista 11: Estado del sistema.

El código finaliza registrando la temperatura del sistema y el parámetro a_2 en el archivo `temp.dat`. Estos valores permiten analizar la evolución del sistema y verificar si se ha alcanzado el equilibrio.

```
    # Escribe, al final de la simulacion, un archivo acumulativo
       de T y a2
    nombre='temp.dat'
    with open(nombre,'w') as archivo:
        for i in range(1, int(nt/utermo)):
            archivo.write('{0:10d} {1:10.6f} {2:10.6f}\n'.format(
                i, temp[i], a2[i]))
    archivo.closed
```

Lista 12: Almacenamiento de T y a_2.

2. Resultados

Al ejecutar el código, se observó que la temperatura del sistema ($T = 1{,}523588$) se mantiene constante a lo largo de toda la simulación. Este comportamiento es coherente con el hecho de que el coeficiente de restitución se fijó a la unidad al inicio del código. Bajo estas condiciones, la energía cinética total se conserva en cada colisión, por lo que es esperable que la temperatura del sistema (definida como la energía cinética media de las partículas) permanezca invariante con el tiempo.

No obstante, el parámetro a_2 presentaba fluctuaciones y no convergía a cero, lo que indica que el sistema aún no ha alcanzado el equilibrio. En la figura 2, puede observarse que, al representar la distribución de velocidades correspondiente a los últimos estados del sistema, esta aún no se ajusta completamente a la distribución de Maxwell-Boltzmann, en concordancia con el comportamiento del parámetro a_2.

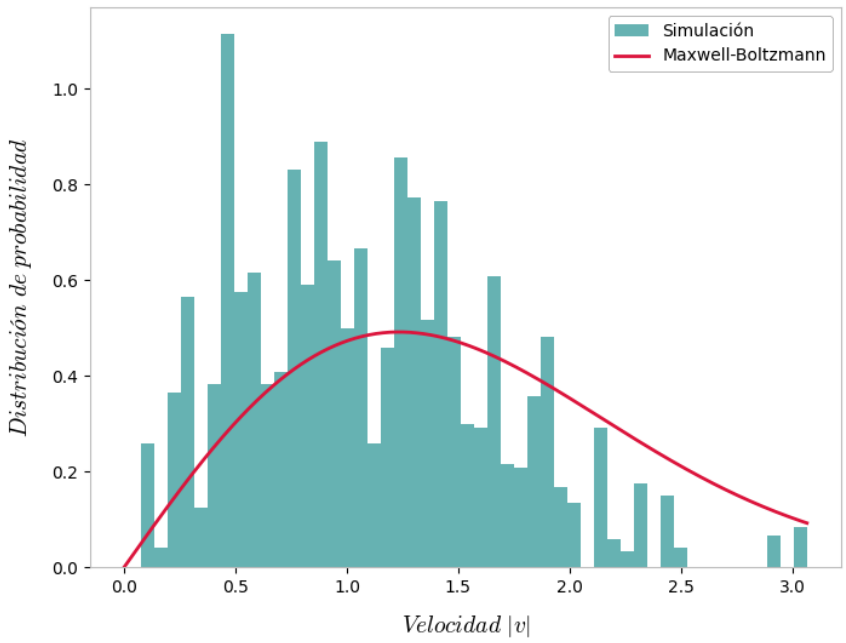

Figura 1: *Comparación entre la distribución de velocidades simulada y la de Maxwell-Boltzmann.*