Práctica 1: simulación de discos empaquetados

Francisco José Clemente García

Curso 2022/23

El siguiente código simula la dinámica de discos en choques inelásticos dentro de un entorno rectangular. El tiempo del sistema está gobernado por el tiempo discreto medido en colisiones (partícula-partícula o partícula-muro).

Las bibliotecas a importar son

```
import math
import numpy as np
import random
import bisect
import time
from operator import itemgetter, attrgetter
from os import system, remove
import pandas as pd
```

Funciones. A continuación se presentan los procedimientos que se emplearán a lo largo del programa principal; se omiten las de escritura que no son utilizadas en el código presentado.

`Propaga`: Toma las posiciones (x, y) de una partícula y las actualiza al paso de una cantidad de tiempo Δt (`dt`).

```
def propaga(dt):
    global vx, vy, x, y
    x = x + vx * dt
    y = y + vy * dt
```

`midedist2`: Dadas dos partículas i y j, calcula el cuadrado de la distancia entre ellas como la distancia entre sus centros menos dos radios, se sirve del Teorema de Pitágoras.

```
def midedist2(i, j):
    global x, y
    dx = x[i] - x[j]
    dy = y[i] - y[j]
    dist2 = (np.power(dx, 2) + np.power(dy, 2)) - 4 * np.power(r, 2)
```

`tcol`: Dadas todas las combinaciones posibles de partículas (i, j), $1 \leq i \neq j \leq n_{\text{part}}$, calcula los tiempos de eventuales colisiones entre partículas dadas sus posiciones así como las trayectorias que describirían sus velocidades.

```
def tcol(i, j):
    global vx, vy, x, y
    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
    if(drdv > 0):
```

```
            vct = float('inf')
        else:
            dist2 = (np.power(dx, 2) + np.power(dy, 2)) - 4 * np.power(r, 2)
            raiz = np.power(drdv, 2) - dist2 * (np.power(dvx, 2) + np.power(dvy, 2))
            if(raiz < 0):
                vct = float('inf')
            else:
                vdt = dist2 / (np.sqrt(raiz) - drdv)
                xicol = x[i] + vx[i] * vdt
                yicol = y[i] + vy[i] * vdt
                xjcol = x[j] + vx[j] * vdt
                yjcol = y[j] + vy[j] * vdt
                if(np.abs(xicol) > lxr):
                    vdt = float('inf')
                elif(np.abs(xjcol) > lxr):
                    vdt = float('inf')
                elif(np.abs(yicol) > lyr):
                    vdt = float('inf')
                elif(np.abs(yjcol) > lyr):
                    vdt = float('inf')
                else:
                    bisect.insort(listacol, [vdt, [i, j]])
```

tpcol: Dado el sistema rectangular por sus paredes $\{-1, -2, -3, -4\}$, calcula los tiempos de colisión entre partícula y muro, conocidas sus posiciones y eventuales trayectorias dadas las velocidades.

```
def tpcol(i):
    global vx, vy, x, y
    if(vx[i] == 0):
        tx = float('inf')
    elif(vx[i] < 0):
        ltx = [-(lxr + x[i]) / vx[i], -1] # pared izq.
    elif(vx[i] > 0):
        ltx = [(lxr - x[i]) / vx[i], -3] # pared dcha.
    if(vy[i] == 0):
        ty = float('inf')
    elif(vy[i] < 0):
        lty = [-(lyr + y[i]) / vy[i], -2] # pared inf.
    elif(vy[i] > 0):
        lty = [(lyr - y[i]) / vy[i], -4] # pared sup.
    ltm = sorted([ltx, lty], key = itemgetter(0))
    vdt = ltm[0][0]
    im = ltm[0][1]
    bisect.insort(listacol, [vdt, [i, im]])
```

pcolisiona: Calcula la velocidad de rebote de las partículas al colisionar con alguno de los muros. `ii[1]` contiene la información de los muros: -1 y -3 son respectivamente izquierdo y derecho, la velocidad en el eje OY se conserva y la horizontal se conserva en valor absoluto pero con signo cambiado. Los muros -2 y -4 son el superior representan los muros inferior y superior, el razonamiento es análogo permutando OY por OX.

Hasta aquí quedaría descrito el caso determinista. Añadimos la componente aleatoria[1] $\phi \sim \mathrm{N}\left(0, \frac{\pi}{1000}\right)$, un error o perturbación en el ángulo de salida. Sólo para este parágrafo, tomemos la siguiente notación: u es el vector de velocidad de incidencia, v es la reflexión de u, y su ángulo con el muro es θ, y w es v con la perturbación ϕ. Además se debe cumplir que $\|u\| = \|v\| = \|w\|$. Dado ϕ, podemos describir dicha perturbación en v como $T_\phi : v \in \mathbb{R}^2 \to T_\phi(v) = w \in \mathbb{R}^2$, que es una

[1]Formalmente ϕ debería estar definido en un dominio de amplitud 2π ($[0, 2\pi[$, $[-\pi, \pi[$, etc.), asumimos que en el práctica es tan improbable que esté definido a una distancia mayor que π de la media, que no habrá problema.)

aplicación $\mathbb{R}$-lineal y tiene por matriz en la base usual

$$\begin{bmatrix} T_\phi(v_1) \\ T_\phi(v_2) \end{bmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{1}$$

Es inmediato ver que describe una *isometría*:

$$\begin{aligned} \|T_\phi(v)\|^2 &= (v_1\cos\phi - v_2\sin\phi)^2 + (v_1\sin\phi + v_2\cos\phi)^2 \\ &= v_1^2\cos^2\phi + v_2^2\sin^2\phi - 2v_1v_2\cos\phi\sin\phi + v_1^2\sin^2\phi + v_2^2\cos^2\phi + 2v_1v_2\sin\phi\cos\phi \\ &= v_1^2\cos^2\phi + v_2^2\sin^2\phi + v_1^2\sin^2\phi + v_2^2\cos^2\phi \\ &= v_1^2 + v_2^2 \\ &= \|v\|^2 \end{aligned} \tag{2}$$

En cuanto a la *orientación* w debería formar un ángulo $\theta + \phi$, veamos que se cumple:

$$\begin{aligned} w &= (\|w\|\cos(\theta+\phi), \|w\|\sin(\theta+\phi)) \\ &= (\|w\|(\cos\theta\cos\phi - \sin\theta\sin\phi), \|w\|(\sin\theta\cos\phi + \cos\theta\sin\phi)) \\ &= (\|w\|\cos\theta\cos\phi - \|w\|\sin\theta\sin\phi, \|w\|\sin\theta\cos\phi + \|w\|\cos\theta\sin\phi) \\ &= (w_1\cos\phi - w_2\sin\phi, w_2\cos\phi + w_1\sin\phi) \\ &= (\Pi_{V_1}T_\phi(v), \Pi_{V_2}T_\phi(v)) = T_\phi(v) \end{aligned} \tag{3}$$

con la que propuesta de T_ϕ satisface las tesis esperada.

```
def pcolisiona(ii):
    global vx, vy, x, y
    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]]
    phi = np.sqrt(np.pi / 1000) * np.random.randn()
    vx[ii] = vx[ii] * np.cos(phi) - vy[ii] * np.sin(phi)
    vy[ii] = vx[ii] * np.sin(phi) + vy[ii] * np.cos(phi)
```

`colisiona`: Dado el par de partículas que chocarán entre sí (i, j) (esto ya se sabe por `tcol`, actualiza sus velocidades después del choque.

```
def colisiona(par):
    global vx, vy, x, y
    i = par[0]
    j = par[1]
    dx = x[i] - x[j]
    dy = y[i] - y[j]
    sigma_norma = np.sqrt(np.power(dx, 2) + np.power(dy, 2))
    sigmax = dx / sigma_norma
    sigmay = dy / sigma_norma
    gsigma = (vx[i] - vx[j]) * sigmax + (vy[i] - vy[j]) * sigmay
    vx[i] = vx[i] - 0.5 * (1 + alfa) * gsigma * sigmax
    vy[i] = vy[i] - 0.5 * (1 + alfa) * gsigma * sigmay
    vx[j] = vx[j] + 0.5 * (1 + alfa) * gsigma * sigmax
    vy[j] = vy[j] + 0.5 * (1 + alfa) * gsigma * sigmay
```

`initialize_random`: Inicializa los centros de las n_{part} partículas de manera que no resulten solapantes.

```
def initialize_random():
    global vx, vy, x, y
    x[0] = random.uniform(-lxr, lxr)
    y[0] = random.uniform(-lyr, lyr)
```

```
        for i in range(1, npart):
            dr = False
            while (dr == False):
                x[i] = random.uniform(-lxr, lxr)
                y[i] = random.uniform(-lyr, lyr)
                for j in range(0, i):
                    dr = (np.power((x[i]-x[j]),2) + np.power((y[i]-y[j]), 2)) > (4*np.power(r,2))
                    if(dr == False):
                        break
        for i in range(npart):
            vx[i] = np.random.randn() * np.power(npart, -0.5)
            vy[i] = np.random.randn() * np.power(npart, -0.5)
```

`calculate_averages`: {}

```
    def calculate_averages(ja):
        global temp, a2, vx, vy, x, y
        temp[ja] = 0.
        a2[ja] = 0.
        for i in range(npart):
            vv = vx[i] * vx[i] + vy[i] * vy[i]
            temp[ja] = temp[ja] + vv
            a2[ja] = a2[ja] + np.power(vv, 2)
        temp[ja] = temp[ja] / npart
        a2[ja] = a2[ja] / (temp[ja] * temp[ja] * npart)
        a2[ja] = (a2[ja] - 2.0) * 0.5
```

El **programa principal** parte de los siguientes parámetros (en mayúsculas) y pide por teclado el resto de variables.

```
LXMIN = 10
LXMAX = 100000
LYMIN = 10
LYMAX = 100000
NU = 0.5
NSIM = 100
```

Donde los cuatro primeros representan la anchura y altura máxima y mínima con la que podemos definir el recinto del sistema. El parámetro `NU` es la fracción de empaquetamiento, *i.e.*, la proporción de área total que está ocupada por los discos. `NSIM` es el número de simulaciones para el cálculo promedio de ejecución con distintas cantidades de partículas. Se piden los valores de anchura `lx`, altura `ly` y el radio general de los discos.

```
while(True):
    lx = float(input("Longitud de la región en el eje OX: "))
    if((lx < LXMIN) or (lx > LXMAX)):
        print(f"Longitud fuera de rago, debe estar en [{LXMIN},{LXMAX}]")
    else:
        break

while(True):
    ly = float(input("Longitud de la región en el eje OY: "))
    if((ly < LYMIN) or (ly > LYMAX)):
        print(f"Longitud fuera de rago, debe estar en [{LYMIN},{LYMAX}]")
    else:
        break

while(True):
    r = float(input("Radio: "))
    if((r < 0) or (r > 0.5 * min(lx,ly))):
        print(f"Radio fuera de rango, debe estar en [{0},{0.5 * min(lx, ly)}]")
    else:
        break
```

La región efectiva por la que podrán moverse los centros es un rectángulo interior, *centrado* en $(0,0)$ y con esquinas $\{(-l_{x,r}, l_{y,r}), (l_{x,r}, l_{y,r}), (l_{x,r}, -l_{y,r}), (-l_{x,r}, -l_{y,r})\}$.

```
lxr = lx * 0.5 - r
lyr = lx * 0.5 - r
```

Pedimos el número de partículas sin que supere el valor `NU` fijado,

```
while(True):
    npart = int(input("Número de partículas: "))
    areadiscos = npart * np.pi * np.power(r, 2)
    if((areadiscos >= NU * lx * ly) or (npart <= 0)):
        print(f"Intorduce otra cantidad, debe estar entre 0 y {np.floor(NU * lx * ly / (np.pi * np.power(r, 2)))}")
    else:
        break
```

Inicializamos un vector de tiempos (cronológicos) , `tiempos`.

Todo lo que sigue viene dentro de un bucle `for k in range(NSIM)`, las realizaciones son independientes del valor de `k`. Como primera instrucción en el bloque dentro del bucle, se declara `inicio = time.time()`.

El tiempo de simulación no vendrá mediado por el tiempo cronológico sino por una cantidad de choques, cuanta mayor velocidad tengan las partículas o más compactadas estén , menor será el tiempo cronológico en cubrir esa cantidad de colisiones. Nos preguntamos cómo podemos *compensar* el tiempo cronológico con el tiempo discretizado en choques [2].

Además, tendremos $\alpha = 1$, que nos indica que el choque será elástico. Se añade un parámetro `tol` para evitar errores numéricos por redondeo de números de punto flotante.

```
nt = 100 * npart # número de pasos temporales (cols.)
print(f"El número de pasos a dar será {nt}")
alfa = 1.0
tol = 1.0e-20
ncp = 1.0 * nt / npart
utermo = 1
```

Se inicializan los *arrays* de posiciones y velocidades,y de tiempos a 0.

```
vx = np.array([0. for i in range(npart)])
vy = np.array([0. for i in range(npart)])
x = np.array([0. for i in range(npart)])
y = np.array([0. for i in range(npart)])
temp = np.array([0. for i in range(nt + 1)])
a2 = np.array([0. for i in range(nt + 1)])
listacol = []
listacol_orden = []
ij = []
```

Se inicializan los *relojes* del proceso y la semilla de aleatoriedad.

```
t = 0.
dt = 0.
it = 0
random.seed()
tiempos = np.array([0. for k in range(NSIM)])
```

Calculamos los tiempos de colisión de cada partícula (primer `for` con las demás (segundo `for`). También de todas las partículas con las paredes.

[2]Se verá en el siguiente apartado, tras describir el código

```
for i in range(npart - 1):
    for j in range(i + 1, npart):
        tcol(i, j)
for i in range(npart):
    tpcol(i)
```

El siguiente proceso iterativo introduce el dinamismo del sistema: hace avanzar el tiempo `it` y a recalcula las futuras colisiones más inmediatas haciendo uso de los procedimientos antes descritos.

```
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):
        listalcol = list(filter(lambda x : x[1][0] != ij[1], listacol))
        listacol = list(filter(lambda x : x[1][1] != ij[1], listacol))
    t = t + dt
    limit = range(len(listacol))
    c=[[listacol[i][0] - dt, listacol[i][1]] for i in limit]
    listacol = c
    propaga(dt)
    if(ij[1] < 0):
        pcolisiona(ij)
    else:
        colisiona(ij) (colisiona)
    i = ij[0]
    tpcol(i)
    for j in range(i):
        tcol(j, i)
    for j in range(i + 1, npart):
        tcol(i, j)
    if(ij[1] > 0):
        i = ij[1]
        tpcol(i)
        for j in range(i):
            tcol(j, i)
        for j in range(i + 1, npart):
            tcol(i, j)
    if(it % utermo == 0):
        ia = int(it / utermo)
        write_micr_state(ia)
        calculate_averages(ia)
```

Se ha realizado una modificación en la generación de velocidades. El tiempo del programa está dado por el número de colisiones, no para un tiempo cronológico dado. A mayor facilidad de colisión (más partículas), menor tiempo cronológico del sistema. Hacemos $v_i^{(x)}, v_i^{(y)} \sim \mathrm{N}\left(0, \frac{1}{n_{\mathrm{part}}}\right)$, esto es, damos más velocidad (más correctamente, hacemos más probable asignar mayores velocidades -en valor absoluto-) a las partículas cuanto menos haya. Así, se produce un efecto compensatorio en en tiempo cronológico.

El bucle sitúa las partículas y asigna velocidades. Se sirve de las funciones programadas para calcular los tiempos entre futuros choques y escoge el más breve. A pesar que el tiempo calculado por el computador es discreto, asumimos que hay la suficiente precisión en los valores de punto flotante como para que la probabilidad de su poner dos colisiones simultáneas es nula, *i.e.*, que para todo par de colisiones en instantes t_{i_1} y t_{i_2}, tendremos $|t_{i_1} - t_{i_2}| > 0$.

Obtenido el choque, se guarda y se simula la dinámica de las partículas en consecuencia. A partir de aquí todos los cálculos se van repitiendo sin más dificulatad de saber encontrar en todo momento la siguiente colisión más próxima en el tiempo, identificar los pares de partículas, o partícula y muro implicados y actualizar. De modo que lo anterior quedaría estructurado en:

```
for k in range(NSIM):
```

```
    ...
    for it in range(1, nt + 1):
        ...
    fin = time.time()
    tiempos[k] = fin-inicio
print(f"El tiempo medio de ejecución fue {np.mean(tiempos)}\
 {npart} partículas.")
```

Nota (Resumen de modificaciones)**.** Las modificaciones sobre el código principal han sido:

1. Se ha distinguido entre *parámetros* y *variables* en el código, dejando los primeros en mayúsculas al inicio y las segundas en minúsculas, ya sean obtenidas por teclado o derivadas del cálculo a lo largo de la ejecución del algoritmo.

2. Se ha cambiado la declaración/introducción de variables, ahora son pedidas por teclado al usuario y tienen un filtro de validación de los datos (no se pueden introducir radios, `R` nulos o negativos, los longitudes están comprendidas entre unos límites prefijados, `LXMAX`, `LXMIN`, `LYMIN` y `LYMAX`.

3. Las instrucciones de la biblioteca `math` se han cambiado a `numpy` (`np`) para eventuales modificaciones que requieran de cálculo tensorial.

4. La generación aleatoria de velocidades en la declaración inicial ya no es $\mathrm{N}(0,1)$ sino que depende proporcionalmente de la cantidad de partículas, $\mathrm{N}\left(\mu = 0, \sigma^2 = \frac{1}{n_{\mathrm{part}}}\right)$.

5. Los discos rebotan en las paredes de manera no determinística, el ángulo de salida es $\theta + \phi$, donde θ es el ángulo bajo hipótesis determinística y $\phi \sim \mathrm{N}\left(\mu = 0, \sigma^2 = \frac{\pi}{1000}\right)$.

6. Dado que el código ahora no busca simular una ejecución del proceso sino encontrar una propiedad en media, quedan comentados todas las instrucciones referidas a la escritura de datos en ficheros, ya que mantenerlas implicaría la escritura `NSIM` (centenares, millares) de veces de éstos.

Simulaciones. Como hemos dicho, vamos a simular $N_{\mathrm{SIM}} = 100$ veces el proceso para distintos valores de $n_{\mathrm{part}} = 5, 10, \ldots, 45, 50$. La siguiente tabla recoge los resultados obtenidos[3]:

Caso no modificado: Para velocidades normales estándar (no modificadas), se obtuvo:

n_{part}	5	10	15	20	25	30	35	40	45	50
$\bar{t}$ (s)	0.0048	0.1533	0.3339	0.5741	0.8932	1.2810	1.7864	2.3360	3.6516	12.8829

Caso modificado: Las velocidades se generan con varianza inversamente proporcional al número de partículas.

n_{part}	5	10	15	20	25	30	35	40	45	50
$\bar{t}$ (s)	0.0046	0.1501	0.3303	0.5963	0.9018	1.3283	1.7646	2.4005	3.0393	3.7069

Los tiempos de cálculo de `Python 3.11` son tan lentos que no permiten hacer el tiempo del sistema independiente del tiempo de ejecución.

[3]La simulación corre en un macOS Ventura 13.4 con un procesador 1.4 GHz Intel Core i5 de cuatro núcleos