Reporte 9:
Interacciones entre Partículas

Jorge Torres

24 de abril de 2022

Capítulo 1

Partículas con Carga y con Masa

1.1. Objetivo

En esta práctica se trabaja con un modelo simplificado de los fenómenos de atracción y repulsión de cargas eléctricas, además del fenómeno de atracción gravitacional. El modelo consiste en una cantidad n de partículas que cuentan con una carga eléctrica, las cargas del mismo signo se repelen, mientras que las de signo opuesto se atraen. Además, cada partícula cuenta con una masa, lo que genera una fuerza de atracción gravitacional entre ellas. El objetivo es estudiar la distribución de velocidades de las partículas y verificar gráficamente que esté presente una relación entre los tres factores: la velocidad, la magnitud de la carga, y la masa de las partículas.

1.2. Desarrollo

El desarrollo está basado en el código implementado por E. Schaeffer para simular las partículas únicamente con carga eléctrica [1]. Con el código 1.1 se crean 25 partículas, normal y aleatoriamente distribuídas con coordenadas (x, y) en un plano, además se les asigna una carga c y una masa m distribuídas de la misma manera. Los códigos completos se pueden encontrar en el repositorio en GitHub de J. Torres [2].

Código 1.1: Creación de Partículas

```
n = 25
x = np.random.normal(size = n)
y = np.random.normal(size = n)
c = np.random.normal(size = n)
m = np.random.normal(size = n)
```

El código 1.2 asegura que las posiciones de las partículas oscilen entre 0 y 1, que sus cargas oscilen entre -1 y 1, y que sus masas sean números enteros entre 0 y 10, exceptuando masas nulas.

Código 1.2: Distribución de Partículas, Cargas y Masas

```
xmax = max(x)
xmin = min(x)
x = (x - xmin) / (xmax - xmin)
ymax = max(y)
ymin = min(y)
y = (y - ymin) / (ymax - ymin)
cmax = max(c)
cmin = min(c)
c = 2 * (c - cmin) / (cmax - cmin) - 1
mmax = max(m)
mmin = min(m)
m = 10 * ((m - mmin) / (mmax - mmin) + 0.1)
m = np.round(m).astype(int)
```

Con el código 1.3 se guardan los datos de las partículas en un archivo de tipo CSV para su posterior uso en la simulación.

Código 1.3: Guardado de Datos de Partículas

```
v = [[0]]*n
g = np.round(5 * c).astype(int)
```

```
p = pd.DataFrame({'x': x, 'y': y, 'c': c, 'm':m, 'g':g, 'v':v})
p.to_csv('values.csv')
```

El código 1.4 es utilizado para la asignación de colores a las partículas con cargas, creando un gradiente de color que va de azul (partículas con carga más negativa), a rojo (partículas con carga más positiva).

Código 1.4: Gradiente de Color para Partículas con Carga

```
paso = 256 // 10
niveles = [i/256 for i in range(0, 256, paso)]
colores = [(niveles[i], 0, niveles[-(i + 1)]) for i in range(len(niveles))]
palette = LinearSegmentedColormap.from_list('tonos', colores, N = len(colores))
```

Para poder estudiar los efectos que tienen las fuerzas en cuestión (cargas y gravedad), en la velocidad de las partículas, se han realizado tres códigos separados, cuya única diferencia es la función de fuerza aplicada a las partículas. Se ha creado uno en donde sólo influyen las fuerzas de atracción y repulsión, otro en donde únicamente influye la fuerza de gravedad, y otro más donde influyen ambas fuerzas. Para asegurar que las condiciones iniciales son iguales para todos los experimentos, se utiliza el archivo CSV creado en el código 1.3.

1.2.1. Función de Fuerza - Partículas con Carga

En el código 1.5 se define la función de fuerza que actúa sobre las partículas con carga, de tal manera que la siguiente posición de la partícula depende directamente de la carga que tiene, e inversamente de la distancia entre sus vecinos cercanos. Partículas con mismo signo se repelen, mientras que partículas con diferente signo se atraen. La constante `eps` sirve para evitar un error de cálculo si dos partículas llegasen a estar en la misma posición al mismo tiempo.

Código 1.5: Fuerza Aplicada a Partículas con Carga

```
eps = 0.001
def fuerza(i, shared):
    p = shared.data
    n = shared.count
    pi = p.iloc[i]
    xi = pi.x
    yi = pi.y
    ci = pi.c
    fx, fy = 0, 0
    for j in range(n):
        pj = p.iloc[j]
        cj = pj.c
        dire = (-1)**(1 + (ci * cj < 0))
        dx = xi - pj.x
        dy = yi - pj.y
        factor = dire * fabs(ci - cj) / (sqrt(dx**2 + dy**2) + eps)
        fx -= dx * factor
        fy -= dy * factor
    return (fx, fy)
```

1.2.2. Función de Fuerza - Partículas con Masa

Para efectos de simplificación y a manera de experimentación, en esta práctica se ha modificado ligeramente la Ley de Gravitación Universal, de tal manera que la constante gravitacional es mucho mayor a la real, y la fuerza ya no depende inversamente del cuadrado de la distancia entre dos masas, sino que depende inversamente de la distancia por sí sola. Esto se observa en la en la ecuación 1.1 y se aplica con el código 1.6.

$$F = G\frac{(M_1)(M_2)}{R}, \tag{1.1}$$

donde F es la fuerza que actúa sobre las partículas, G es la constante gravitacional, que en este universo es igual a 0.025 (mucho mayor que la real), M_1 y M_2 son las masas de dos partículas vecinas y R es la distancia entre ellas.

Código 1.6: Fuerza Aplicada a Partículas con Masa

```
eps = 0.001
G = 0.025
def fuerza(i, shared):
    p = shared.data
```

```
    n = shared.count
    pi = p.iloc[i]
    xi = pi.x
    yi = pi.y
    mi = pi.m
    fx, fy = 0, 0
    for j in range(n):
        pj = p.iloc[j]
        mj = pj.m
        dx = xi - pj.x
        dy = yi - pj.y
        factor = G * ((mi * mj) / (sqrt(dx**2 + dy**2) + eps))
        fx -= dx * factor
        fy -= dy * factor
    return (fx, fy)
```

1.2.3. Función de Fuerza - Partículas con Carga y Masa

Para observar los efectos de ambas fuerzas en las partículas, éstas simplemente se suman al calcular la posición siguiente, de tal forma que la fuerza total es la suma de fuerzas de atracción, repulsión y de gravedad, como se observa en el código 1.7.

Código 1.7: Fuerza Aplicada a Partículas con Carga y Masa

```
eps = 0.001
G = 0.025
def fuerza(i, shared):
    p = shared.data
    n = shared.count
    pi = p.iloc[i]
    xi = pi.x
    yi = pi.y
    ci = pi.c
    mi = pi.m
    fx1, fy1 = 0, 0
    fx2, fy2 = 0, 0
    for j in range(n):
        pj = p.iloc[j]
        cj = pj.c
        mj = pj.m
        dire = (-1)**(1 + (ci * cj < 0))
        dx = xi - pj.x
        dy = yi - pj.y
        factor = dire * fabs(ci - cj) / (sqrt(dx**2 + dy**2) + eps)
        factor1 = G * ((mi * mj) / (sqrt(dx**2 + dy**2) + eps))
        fx1 = fx1 - dx * factor
        fy1 = fy1 - dy * factor
        fx2 = fx2 - dx * factor1
        fy2 = fy2 - dy * factor1

    fx = fx1 + fx2
    fy = fy1 + fy2
    return (fx, fy)
```

Las funciones `actualiza()` y `velocidad()` del código 1.8 sirven para actualizar la posición de cada partícula, y para calcular y guardar la velocidad de cada partícula en cada paso de la iteración, respectivamente.

Código 1.8: Actualización de Posición y Cálculo de Velocidad

```
def actualiza(pos, fuerza, de):
    return max(min(pos + de * fuerza, 1), 0)

def velocidad(p, pa):
    ppa = pa.data
    n = pa.count
    for i in range(n):
      p1 = p.iloc[i]
      p2 = ppa.iloc[i]
      x1 = p1.x
      x2 = p2.x
      y1 = p1.y
      y2 = p2.y
      va = p2.v
      v = []
      v.extend(va)
      vel = (sqrt(((x2 - x1)**2) + ((y2 - y1)**2)))
      v.append(vel)
      p['v'][i] = v
```

En el código 1.9 se inician los parámetros de operación utilizando los datos de las partículas creados en el código 1.3 y se establece la cantidad de pasos que dura la iteración del experimento, que consiste en un estado inicial y 59 pasos adicionales, para un total de 60 pasos. Ya que la operación del código se paraleliza, se comparten los datos de las partículas a lo largo de la operación utilizando la instrucción `multiprocessing.Manager.Namespace()`.

Código 1.9: Inicio de Parámetros de Operación

```
if __name__ == "__main__":
    n = 25
    p = pd.read_csv('values.csv')
    x = p['x']
    y = p['y']
    g = p['g']
    m = p['m']
    c = p['c']
    p['v'] = [[0]]*n
    mgr = multiprocessing.Manager()
    ns = mgr.Namespace()
    ns.data = p
    ns.count = n
    tmax = 59
```

Finalmente, con el código 1.10, se inicia el movimiento de las partículas utilizando las respectivas funciones de fuerza, actualizando las posiciones con la función `actualiza()` y guardando las velocidades de cada partícula con la función `velocidad()` para su posterior análisis.

Código 1.10: Movimiento de Partículas

```
    for t in range(tmax):
        with multiprocessing.Pool() as pool:
            f = pool.starmap(fuerza, [(i, ns) for i in range(n)])
            delta = 0.02 / max([max(fabs(fx), fabs(fy)) for (fx, fy) in f])
            p['x'] = pool.starmap(actualiza, zip(p.x, [v[0] for v in f], repeat(delta)))
            p['y'] = pool.starmap(actualiza, zip(p.y, [v[1] for v in f], repeat(delta)))
            velocidad(p, ns)
            ns.data = p
```

1.3. Resultados

Para un mejor entendimiento de los resultados, éstos se han representado de dos maneras. La primera es una visualización a manera de viñetas del movimiento de las partículas para cada iteración del experimento. La segunda es una visualización de las distribuciones de las velocidades de cada partícula a manera de diagramas caja-bigote.

En la figura 1.1 se puede observar el movimiento de las partículas en el experimento donde únicamente actúan las fuerzas de atracción y repulsión. La mayoría de las partículas convergen en un punto central, lo cual indica que,

en promedio, existe una mayor fuerza de atracción entre ellas. Una visualización animada del movimiento puede encontrarse en el repositorio en GitHub de J. Torres.

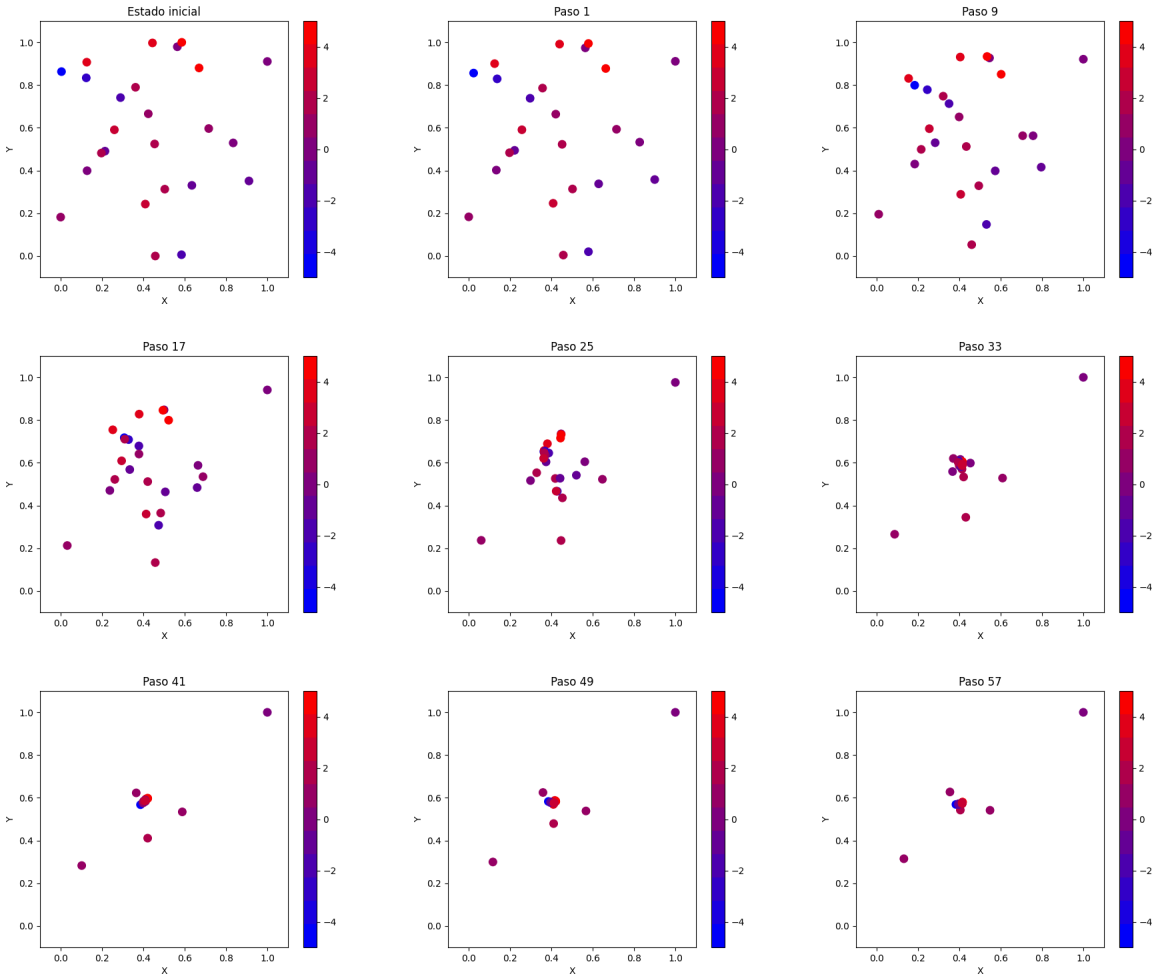


Figura 1.1: Movimiento de partículas con carga.

La figura 1.2 muestra el movimiento de las partículas al actuar sobre ellas la fuerza de gravedad previamente establecida. En este caso, el tamaño de la partícula es indicador de la cantidad de masa que tiene. Se puede observar que la convergencia ocurre a mayor velocidad que las partículas con carga, lo cual puede indicar que, en este universo, la fuerza de gravedad es mayor que las de atracción y repulsión. Una visualización animada del movimiento puede encontrarse en el repositorio en GitHub de J. Torres.

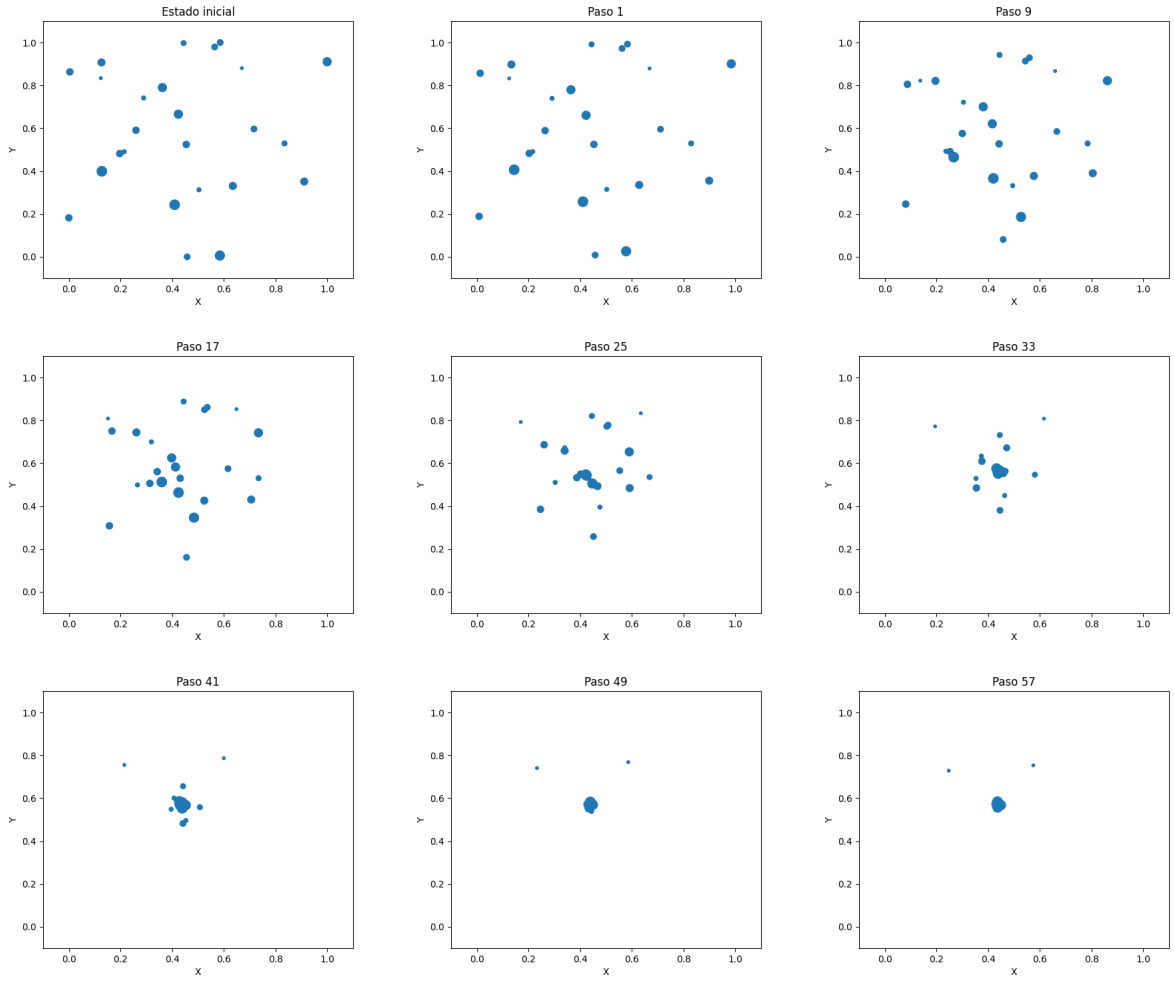


Figura 1.2: Movimiento de partículas con masa.

En la figura 1.3 se representa el movimiento de las partículas cuando actúan sobre ellas las fuerzas de atracción, repulsión y de gravedad. Se observan velocidades un tanto mayores a ambos casos previos, por lo que las partículas convergen antes. Esto es de esperarse, ya que existen dos fuerzas que actúan en favor de la atracción de las partículas. Una visualización animada del experimento puede encontrarse en el repositorio en GitHub de J. Torres.

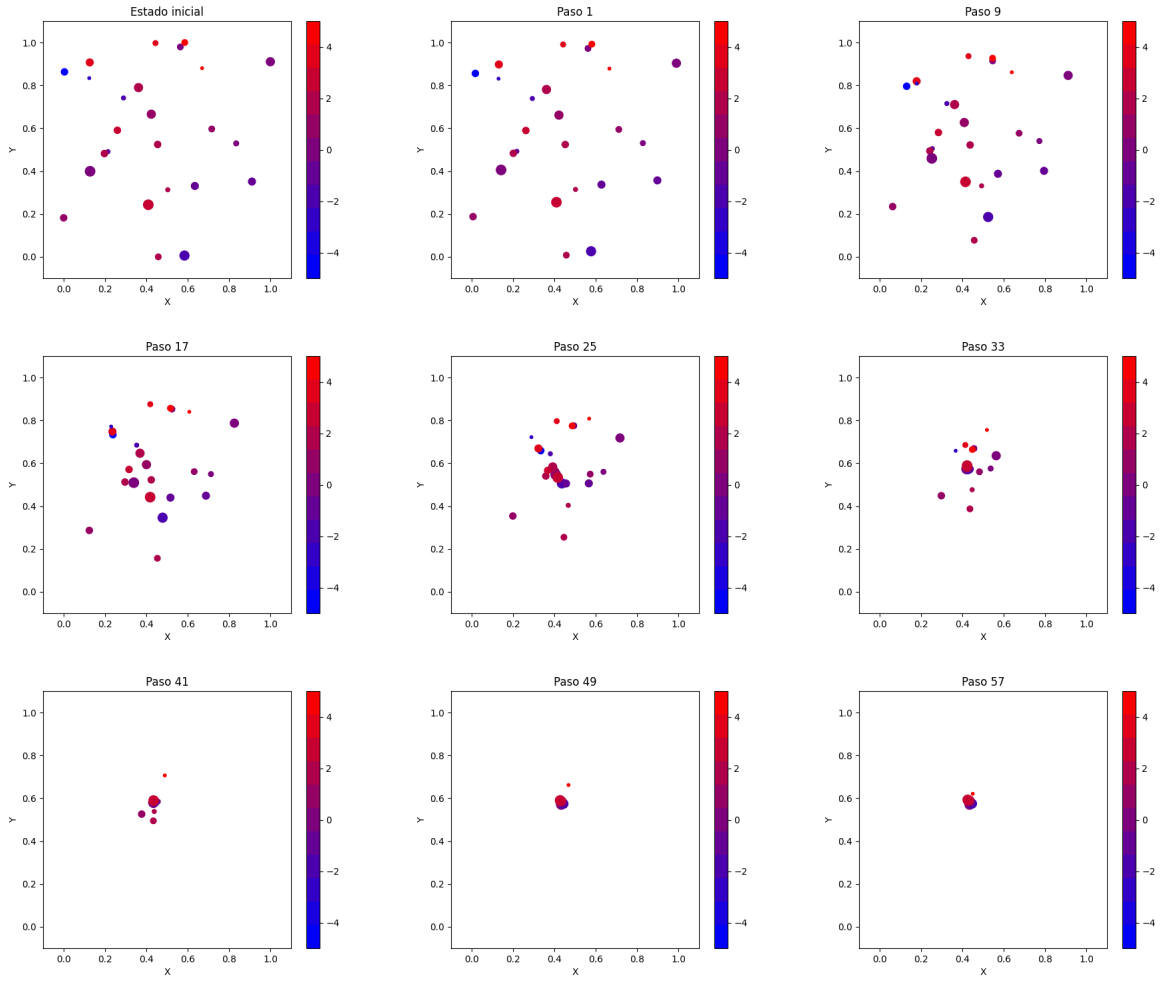
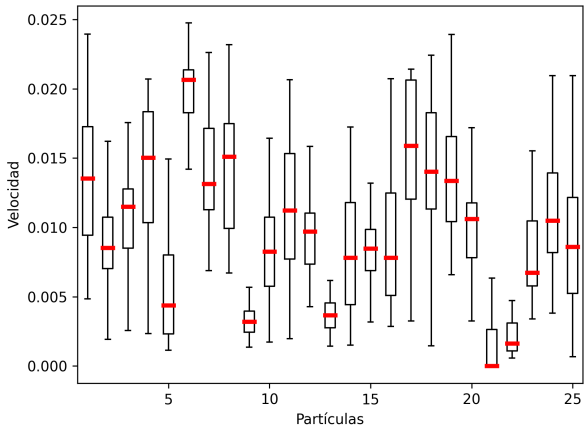
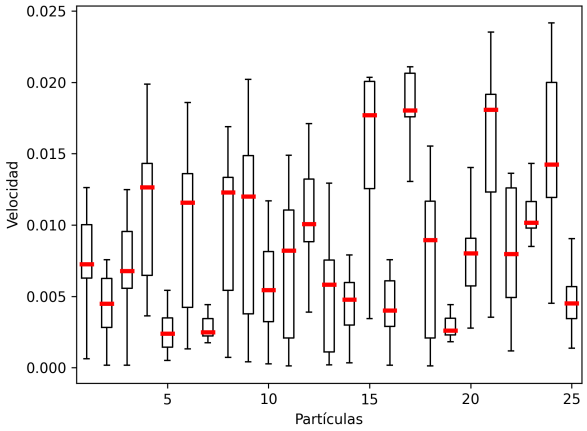


Figura 1.3: Movimiento de partículas con carga y masa.

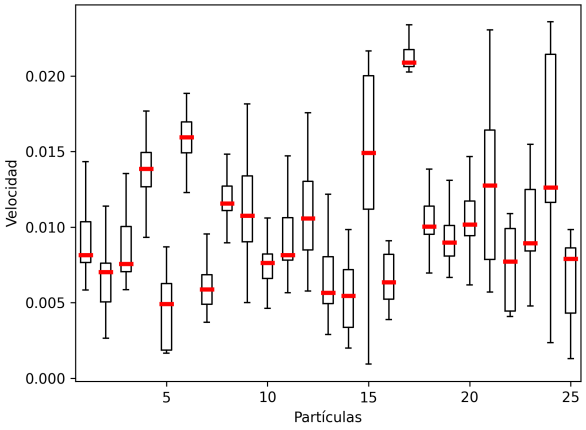
En la figura 1.4 se pueden observar las distribuciones de las velocidades de cada partícula para cada iteración del experimento. En la figura 1.4a, las partículas se mueven debido a las fuerzas de atracción y repulsión, la figura 1.4b muestra las velocidades de las partículas al actuar sobre ellas la fuerza de gravedad y en la figura 1.4c las partículas se mueven debido a la suma de fuerzas de atracción, repulsión y de gravedad.



(a) Velocidades de partículas con carga.



(b) Velocidades de partículas con masa.



(c) Velocidades de partículas con carga y masa.

Figura 1.4: Distribución de velocidades de las partículas para cada tipo de experimento.

Se ha realizado también un análisis estadístico de varianza tipo ANOVA para verificar si existe una diferencia significativa entre las velocidades de una partícula al estar sometida a los diferentes tipos de fuerza. El valor p encontrado por el análisis es mucho menor a 0.05, por lo que se puede concluir que la diferencia sí es estadísticamente significativa.

1.4. Conclusiones

Observando las visualizaciones de los movimientos de las partículas, se puede encontrar una diferencia notable en velocidades. Al observar las distribuciones en los diagramas caja-bigote, esta diferencia se puede apreciar más claramente, habiendo un incremento considerable en velocidad al implementar ambas fuerzas en el movimiento. El análisis de varianza confirma esta diferencia, por lo que se puede concluir que existe una relación entre la fuerza que se aplica y la velocidad de las partículas.

Capítulo 2

Reto 1

2.1. Objetivo

El reto 1 consiste en desarrollar la simulación de dos diferentes tipos de objetos: bolitas duras grandes y partículas frágiles — cuando una partícula es atrapada entre dos bolas, se modifica: si está sola, se fragmenta, pero si hay otras partículas en ese mismo traslape de dos bolas, se pegan todas juntas.

2.2. Desarrollo

El desarrollo del reto está basado en el código descrito por E. Schaeffer para la práctica 9 de su curso de simulación. En primer lugar, con el código 2.1 se crean $m = 7$ bolas y $n = 25$ partículas. Se les asignan posiciones (x, y), y velocidades (vx, vy), así como un tamaño r iniciales. Estos valores se guardan en marcos de datos para su posterior uso en el código.

Código 2.1: Creación de Bolas y Partículas

```
m = 7
vx = (2 * (np.random.uniform(size = m) < 0.5) - 1) * np.random.uniform(low = 0.01, high = 0.04,
    size = m)
vy = (2 * (np.random.uniform(size = m) < 0.5) - 1) * np.random.uniform(low = 0.01, high = 0.04,
    size = m)
x = np.random.uniform(size = m)
y = np.random.uniform(size = m)
r = np.random.uniform(low = 0.05, high = 0.1, size = m)
bolas = pd.DataFrame({'x': x, 'y': y, 'dx': vx, 'dy': vy, 'r': r})
n = 50
vx = (2 * (np.random.uniform(size = n) < 0.5) - 1) * np.random.uniform(low = 0.02, high = 0.05,
    size = n)
vy = (2 * (np.random.uniform(size = n) < 0.5) - 1) * np.random.uniform(low = 0.02, high = 0.05,
    size = n)
x = np.random.uniform(size = n)
y = np.random.uniform(size = n)
print(x)
r = np.random.uniform(low = 0.01, high = 0.03, size = n)
particulas = pd.DataFrame({'x': x, 'y': y, 'dx': vx, 'dy': vy, 'r': r,\
                           'v': [1] * n, 'a': [1] * n})
```

Con el código 2.2 se verifica que las partículas estén en movimiento y, si lo están, se calculan las distancias entre partículas y bolas para saber si están sobrelapadas.

Código 2.2: Verificación de Sobrelapamiento de Partículas y Bolas

```
for t in range(25):

    for i in range(n):
        p = particulas.iloc[i]
        v = p.v

        if v > 0:
            pr = p.r
            px = p.x
            py = p.y
```

```
            conteo = 0
            for k in range(m):
                pk = bolas.iloc[k]
                pkr = pk.r
                pkx = pk.x
                pky = pk.y
                dx = px - pkx
                dy = py - pky
                dr = pkr + pr
                d = sqrt(dx**2 + dy**2)
                if d < dr:
                    conteo += 1
```

En el código 2.3 se realiza la unión de las partículas cuando existen dos o más de ellas sobrelapadas entre dos bolas.

Código 2.3: Unión de Partículas Sobrelapadas

```
            if conteo >= 2:
                conteo2 = 0
                for j in range(n):
                    if i != j:
                        pj = particulas.iloc[j]
                        pjr = pj.r
                        pjx = pj.x
                        pjy = pj.y
                        dx = px - pjx
                        dy = py - pjy
                        dr = pjr + pr
                        d = sqrt(dx**2 + dy**2)
                        if d < dr:
                            conteo2 += 1
                            a1 = np.pi * (pr**2)
                            a2 = np.pi * (pjr**2)
                            a = a1 + a2
                            rt = sqrt(a/np.pi)
                            particulas.at[i, 'r'] = rt
                            particulas.at[j, 'v'] = -1
```

Para fragmentar las partículas que se encuentren solas en un traslape de dos bolas, se implementa el código 2.4. Se actualizan las posiciones, velocidades y tamaños de las partículas.

Código 2.4: Fragmentación de Partículas Sobrelapadas

```
                if conteo2 == 0:
                    v = random()
                    v1 = 1 - v
                    vx1 = p.dx * -1
                    vy1 = p.dy * -1
                    a = np.pi * (pr**2)
                    a1 = a * v
                    a2 = a * v1
                    r1 = sqrt(a1/np.pi)
                    r2 = sqrt(a2/np.pi)
                    particulas.at[i, 'r'] = r1
                    particulas = particulas.append({'x': px, 'y': py, 'dx': vx1, 'dy': vy1,\
                                                    'r': r2, 'v': 1, 'a': 1}, ignore_index=True)
```

El código 2.5 elimina las partículas vacías que quedan después del cálculo y hace un recuento de la cantidad de partículas que quedan. Además, se guardan los datos de bolas y partículas en archivos de tipo CSV para su posterior uso en la visualización del movimiento.

Código 2.5: Eliminacioón de Partículas Vacías y Guardado de Datos

```
    particulas = particulas.loc[particulas['v'] > 0]
    n = particulas.shape[0]
    particulas.to_csv('p_part_{:d}.dat'.format(t), header = False, index = False)
    bolas.to_csv('p_bola_{:d}.dat'.format(t), header = False, index = False)
```

El código 2.6 hace que las bolas reboten de los límites del marco de coordenadas, mientras que el código 2.7 hace lo mismo para las partículas.

Código 2.6: Rebote de las Bolas en los Límites

```
    for i in range(m):
        b = bolas.iloc[i]
        br = b.r
        bx = b.x
        by = b.y
        vx = b.dx
        vy = b.dy
        x = bx + vx
        y = by + vy
        if 0 >= (x-br):
            x = 0 + br
            bolas.at[i, 'dx'] = vx * -1
        elif (x+br) >= 1:
            x = 1 - br
            bolas.at[i, 'dx'] = vx * -1
        if 0 >= (y-br):
            y = 0 + br
            bolas.at[i, 'dy'] = vy * -1
        elif (y+br) >= 1:
            y = 1 - br
            bolas.at[i, 'dy'] = vy * -1

        bolas.at[i, 'x'] = x
        bolas.at[i, 'y'] = y
```

Código 2.7: Rebote de las Partículas en los Límites

```
    for i in range(n):
        b = particulas.iloc[i]
        br = b.r
        bx = b.x
        by = b.y
        vx = b.dx
        vy = b.dy
        x = bx + vx
        y = by + vy
        if 0 >= (x-br):
            x = 0 + br
            particulas.at[i, 'dx'] = vx * -1
        elif (x+br) >= 1:
            x = 1 - br
            particulas.at[i, 'dx'] = vx * -1
        if 0 >= (y-br):
            y = 0 + br
            particulas.at[i, 'dy'] = vy * -1
        elif (y+br) >= 1:
            y = 1 - br
            particulas.at[i, 'dy'] = vy * -1

        particulas.at[i, 'x'] = x
        particulas.at[i, 'y'] = y
```

2.3. Resultados

Los movimientos de las bolas y partículas, así como la unión y fragmentación de las partículas se muestran en una animación de formato GIF contenida en el repositorio en GitHub de J. Torres, y realizada por medio de Gnuplot con los datos obtenidos del código 2.5, introduciendo el código 2.8.

Código 2.8: Animación por Medio de Gnuplot

```
set term gif animate delay 50
set key off
set datafile separator ","
set size square
set xrange [ -0.02 : 1.02] noreverse nowriteback
set yrange [ -0.02 : 1.02 ] noreverse nowriteback
set output 'p9pr.gif'
do for [i=0:24] {
  set title 'Paso '.(i + 1)
```

```
  plot 'p_bola_'.i.'.dat' u 1:2:5 w circles lc rgb "blue" fs solid noborder, \
       'p_part_'.i.'.dat' u 1:2:5 w circles lc rgb "red" fs solid noborder
}
```

2.4. Conclusiones

Por medio de este modelo se puede apreciar la simulación de unión y fragmentación de partículas al estar presente un agente que facilite la interacción entre ellas. En cierto modo, es parecido al comportamiento que tendrían las nanopartículas de algún material al estar en presencia de un agente catalizador o de otras nanopartículas. Sin embargo, el comportamiento es demasiado simple como para describir en su totalidad un sistema de ese tipo, por lo que se necesitarían realizar modificaciones si se requiere una simulación más realista.

Bibliografía

[1] E. Schaeffer. Particles, 2019. URL https://github.com/satuelisa/Simulation/tree/master/Particles.

[2] J. Torres. P9, 2022. URL https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P9.