

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN  
FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA  
POSGRADO EN CIENCIAS DE LA INGENIERÍA CON ORIENTACIÓN EN NANOTECNOLOGÍA



**UANL**

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN



**FIME**

FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA

PORTAFOLIO DE EVIDENCIAS

DE

JORGE ALEJANDRO TORRES QUINTANILLA

1642654

PARA EL CURSO DE SIMULACIÓN COMPUTACIONAL DE NANOMATERIALES,

DR. VIRGILIO GONZÁLEZ GONZÁLEZ EN COLABORACIÓN CON DRA. ELISA SCHAEFFER ELISA SCHAEFFER.

SEGUNDO SEMESTRE. ENERO - JUNIO 2022.

Práctica												T	P	C
$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$	$t_{10}$	$t_{11}$	$t_{12}$			
7	6	6	6	7	5	6	6	8	6	8	8	79	$3 + 3 + 3 + 1 + 0 + 5 = 15$	94

[HTTPS://GITHUB.COM/FEROXDEITAS/SIMULACION-NANO/TREE/MAIN/TAREAS](https://github.com/FeroxDeitas/SIMULACION-NANO/tree/main/TAREAS)

# Reporte 1: Movimiento Browniano

Jorge Torres

16 de febrero de 2022

## 1. Objetivo

Esta práctica tiene como objetivo el estudiar de manera sistemática el movimiento aleatorio de una partícula en un espacio que va de 1 a 5 dimensiones. Se varía también la cantidad de pasos que dura la caminata, y para cada caminata en cada dimensión, el experimento se repite 30 veces. Los resultados se observan en una sola gráfica con diagramas caja-bigote. Asimismo, se evalúa el tiempo promedio en que se realiza el cómputo de una réplica del experimento.

## 2. Desarrollo

En mi [repositorio](#) de GitHub se puede observar la evolución del código desarrollado. Tomando como base el [código](#) proporcionado por E. Schaeffer [1], se hace una modificación para iterar entre la cantidad de pasos que se realizan en cada dimensión. Como se observa en las siguientes líneas de código, primero se realizan 30 ciclos de 100 pasos en una a cinco dimensiones, después con 1,000 pasos y por último con 10,000 pasos.

```
runs = 30 #replicas
caminatas = [100, 1000, 10000] #pasos
results = [] #almacena las dimensiones

for i in range(3): #itera la cantidad de pasos
    dur = caminatas[i]
    for dim in range(1, 6): #de una a cinco dimensiones
        mayores = []
        for rep in range(runs):#corre el experimento 30 veces en cada dimension
            before = time()*1000
            pos = [0] * dim
            mayor = 0
            for paso in range(dur):
                eje = randint(0, dim - 1)
                if pos[eje] > -100 and pos[eje] < 100:
                    if random() < 0.5:
                        pos[eje] += 1
                    else:
                        pos[eje] -= 1
                else:
                    if pos[eje] == -100:
                        pos[eje] += 1
                    if pos[eje] == 100:
                        pos[eje] -= 1
            mayor = max(mayor, sqrt(sum([p**2 for p in pos])))
            after = time()*1000
            mayores.append(mayor)
            results.append(mayores)
tiempo = after - before
print(tiempo)
```

```
#separar los resultados en tres grupos de caminatas
walks_1 = results[0:5] #caminatas de 100 pasos
walks_2 = results[5:10] #caminatas de 1000 pasos
walks_3 = results[10:15] #caminatas de 10000 pasos
```

En cada ciclo se calcula la distancia máxima euclídea tomada desde el punto de origen utilizando la ecuación (1) y se agrega a la lista de resultados para ser graficados posteriormente,

$$d_{max} = \max\{0, d_e\}, d_e = \sqrt{\sum_{n=1}^m x_n^2} \quad (1)$$

donde  $m$  es la cantidad de ejes.

Adicionalmente, se utiliza el comando `time()` para medir el tiempo que toma realizar los 30 ciclos en cada dimensión, a lo cual se realiza una normalización para conocer el promedio que tarda en ejecutarse un solo ciclo.

### 3. Resultados

En esta sección se discuten los resultados obtenidos de la práctica. Se muestran tanto los datos graficados para las distancias máximas como los tiempos de ejecución del programa realizado.

#### 3.1. Distancia Máxima Euclideana

En la figura 1 se puede observar la agrupación estadística de las distancias máximas recorridas por una partícula con movimiento aleatorio al realizar caminatas de 100 (rojo), 1,000 (verde) y 10,000 (azul) pasos, para espacios cartesianos con 1 a 5 ejes. Cabe resaltar que estos ejes tienen un rango de -100 a 100 unidades, y que la partícula retrocede un paso al llegar al borde.

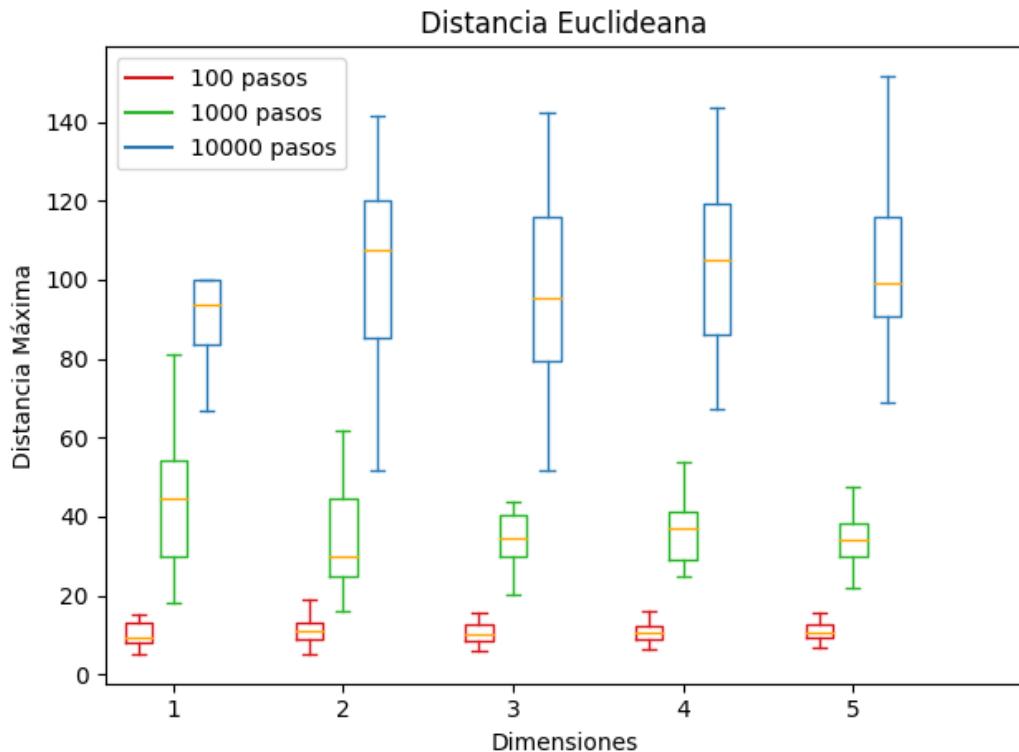


Figura 1: Diagrama caja-bigote donde se observan las estadísticas de distancia máxima euclidean para los grupos de 100, 1,000 y 10,000 pasos en dimensiones de 1 a 5.

Cuadro 1: Tiempo promedio que tarda una repetición de 30 ciclos en realizarse.

Repetición	Tiempo (ms)
1	31.2421875
2	31.2785645
3	37.7722168
4	28.1269531
5	31.2482910
6	31.7312012
7	31.3381348
8	31.3688965
9	31.3549805
10	33.3366699
Promedio	31.8798086

### 3.2. Tiempo de Ejecución

Al realizar un estudio automatizado del tiempo que tarda una repetición en ejecutarse y calcular un promedio al correr el programa varias veces, se puede crear una idea aproximada de lo eficiente que es el código para la tarea en cuestión. Los resultados de tal estudio se muestran en el cuadro 1 y se puede observar que mi computadora tarda  $\approx 32$  ms en completar una repetición de 30 ciclos del experimento, por lo que se estima que tardaría  $\approx 1.06$  ms en completar un ciclo.

## 4. Conclusión

Al observar los diagramas de la figura 1 se puede apreciar cómo el rango de distancia máxima es cada vez más amplio al aumentar la cantidad de pasos, lo cual se debe a que tiene más posibilidades de terminar en puntos diferentes del espacio. Por otro lado, el rango de distancias no aumenta o disminuye significativamente al variar la cantidad de ejes en que se puede mover la partícula.

Tomando en cuenta que el tiempo promedio de ejecución de una repetición es de 32 ms y que se realizan 15 repeticiones (5 grupos de dimensiones por 3 grupos de largo de caminata), se tiene que el programa debería tardar  $\approx 480$  ms en ejecutarse. La realidad es que tarda más debido a que debe realizar y guardar la gráfica en un archivo PNG. Además, existe la posibilidad de paralelizar operaciones al reservar núcleos del CPU para ejecutar el programa, lo cual no se ha implementado en esta ocasión.

## Referencias

- [1] E. Schaeffer. Brownianmotion. *Repositorio, GitHub*, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/BrownianMotion>.

# Reporte 2: Autómata Celular

Jorge Torres

20 de febrero de 2022

## 1. Objetivo

El objetivo de la práctica se centra en diseñar y ejecutar un experimento con por lo menos 30 réplicas para estimar la probabilidad de creación de vida dentro de 200 iteraciones, usando niveles de 10, 15, 20 y 25 para el tamaño de matriz y los niveles 0.2, 0.4, 0.6 y 0.8 para la densidad inicial de vida.

## 2. Desarrollo

Utilizando como base el [código](#) desarrollado por E. Schaeffer [1], para generar un autómata celular, primero se definen los tamaños de matriz, las densidades iniciales de vida, la cantidad de réplicas y las iteraciones que dura el experimento. Estos parámetros se observan en el código 1. En seguida se definen dos funciones, `mapeo` y `paso`, que sirven respectivamente para: 1) mapear la matriz en cuestión y 2) revisar la condición de vida de cada celda individual en la matriz, la cual consiste en tener exactamente tres vecinos vivos (ver código 2). Por último, se comienzan los ciclos `for` para iterar entre los parámetros definidos. En el ciclo donde se llama a la función `paso` se determina si el sistema termina en un estado de vida, para lo cual se lleva un contador que se usa posteriormente para determinar el porcentaje de supervivencia de acuerdo a la ecuación 1,

$$P_s = \frac{C_v}{R} \times 100 \quad (1)$$

donde  $P_s$  es el porcentaje de supervivencia,  $C_v$  es la cantidad de sistemas que lograron sobrevivir y  $R$  es la cantidad de veces que se replicó el experimento. Estas iteraciones se pueden apreciar en el código 3, mientras que el código completo se puede revisar en mi [repositorio](#) de GitHub.

```
dim = [10, 15, 20, 25] #matrix sizes
p = [0.2, 0.4, 0.6, 0.8] #initial life density
runs = 30 #replicas of the experiment
dur = 200 #iterations
```

Código 1: Parámetros

```
def mapeo(pos):
    fila = pos // lado
    columna = pos % lado
    return actual[fila, columna]

def paso(pos):
    fila = pos // lado
    columna = pos % lado
    vecindad = actual[max(0, fila - 1):min(lado, fila + 2),
                      max(0, columna - 1):min(lado, columna + 2)]
    return 1 * (np.sum(vecindad) - actual[fila, columna]) == 3
```

Código 2: Funciones

```

for lado in dim:
    num = lado**2
    for densidad in p:
        contador_viv=0
        for rep in range(runs):
            valores = [1 * (random() < densidad) for i in range(num)]
            actual = np.reshape(valores , (lado , lado))
            assert all([mapeo(x) == valores[x] for x in range(num)])
        for iteracion in range(dur):
            valores = [paso(y) for y in range(num)]
            vivos = sum(valores)
            if vivos == 0:
                break;
            if iteracion == (dur-1):
                contador_viv += 1
            actual = np.reshape(valores , (lado , lado))
        vivieron = ((contador_viv*100)/(runs))
        resultados = { 'Tamano_matriz' : lado ,
                      'P_inicial' : densidad ,
                      'Porcentaje_Supervivencia(%)' : vivieron }
        datos = datos.append(resultados , ignore_index=True)

```

Código 3: Iteración de Parámetros

### 3. Resultados

En la figura 1 se observa el comportamiento del autómata celular conforme se varían el tamaño de matriz y la densidad inicial de vida.

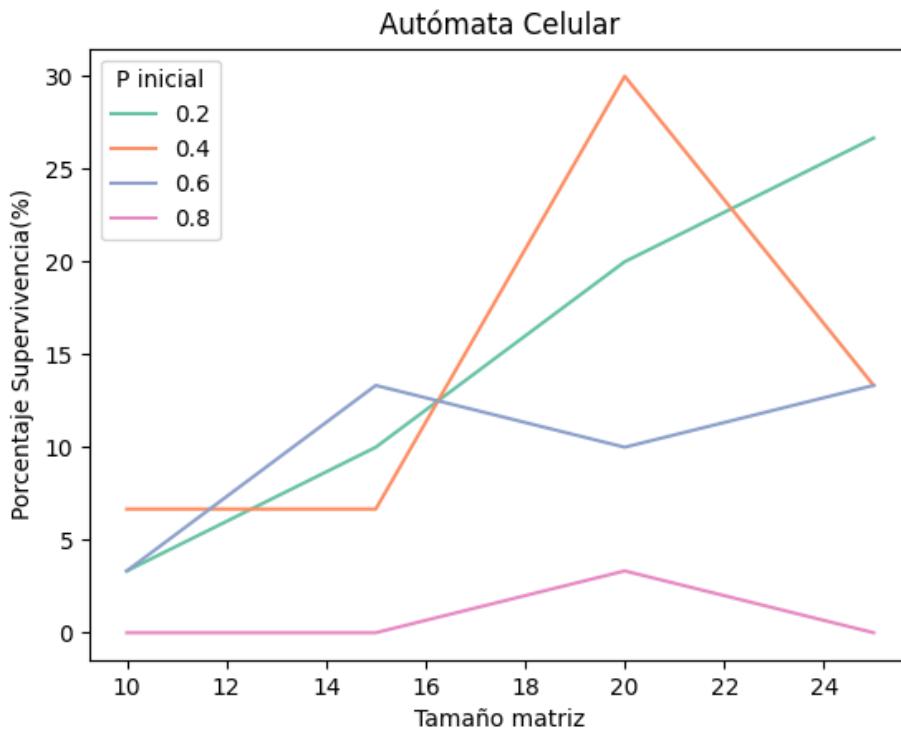


Figura 1: Porcentajes de supervivencia para sistemas con tamaños de matriz 10, 15, 20 y 25, y para densidades iniciales de vida de 0.2, 0.4, 0.6 y 0.8

## 4. Conclusiones

Una observación inicial indica que los sistemas tienen una mayor probabilidad de sobrevivir conforme se aumenta el tamaño de la matriz y para densidades iniciales de vida medias y bajas (0.2, 0.4). Esto puede deberse a que las células tendrían un mayor espacio para proliferar en un área más grande y a que no tendrían tanta competencia al haber menos densidad.

Algo que destaca es la baja o nula probabilidad de supervivencia de los sistemas con una densidad inicial de 0.8, esto debido a la condición de supervivencia de tener exactamente tres vecinos vivos. Podría cambiarse la condición para estimular un mejor o peor crecimiento, pero eso está fuera del rango de estudio de esta práctica.

## Referencias

- [1] E. Schaeffer. Cellularautomata. *Repositorio, GitHub*, 2020. URL <https://github.com/satuelisa/Simulation/tree/master/CellularAutomata>.

# Reporte 3:

## Teoría de Colas

Jorge Torres

26 de febrero de 2022

## 1. Objetivo

El objetivo de la práctica consiste en analizar los tiempos de ejecución de tres algoritmos diferentes que encuentran si un número es primo. Se varían tanto el algoritmo utilizado como el orden inicial de la lista de números de forma independiente y se realiza un análisis estadístico relevante para decidir si la diferencia en tiempos de ejecución es significativa o no. Por último, se muestran resultados de la evaluación en gráficas tipo violín para una mejor interpretación.

## 2. Desarrollo

Basando el desarrollo en el [código](#) implementado por E. Schaeffer [1], primero se definen los parámetros de ejecución de los algoritmos, como se observa en el código 1,

Código 1: Parámetros

```
1 d = 1000
2 h = 5000
3 replicas = 30
4 original = [x for x in range(d, h + 1)]
5 invertido = original[::-1]
6 aleatorio = original.copy()
7 shuffle(aleatorio)
```

donde se crea una lista de números desde 1000 hasta 5000, y se establece que los experimentos tendrán 30 réplicas. Los parámetros `invertido` y `aleatorio` simplemente reordenan la lista de números de forma invertida y aleatoria, respectivamente.

Lo siguiente es definir las tres formas en que se decide si los números son primos o no. Para esto se toman los [algoritmos](#) implementados por E. Schaeffer, y se pueden ver en el código 2.

Código 2: Algoritmos para encontrar números primos

```
1 def primo_1(n):
2     if n < 3:
3         return True
4     for i in range(2, n):
5         if n % i == 0:
6             return False
7     return True
8
9 def primo_2(n):
10    if n < 4:
11        return True
12    if n % 2 == 0:
13        return False
14    for i in range(3, n - 1, 2):
15        if n % i == 0:
16            return False
17    return True
18
19 def primo_3(n):
20    if n < 4:
```

```

21     return True
22 if n % 2 == 0:
23     return False
24 for i in range(3, int(ceil(sqrt(n))), 2):
25     if n % i == 0:
26         return False
27 return True

```

La función `primo_1` simplemente decide que un número es primo si es menor a 3 o si el residuo de la división entre sus predecesores es diferente de cero. La función `primo_2` únicamente revisa entre los números impares de la lista. La tercera función, `primo_3`, utiliza la lógica matemática en la que, para el par de factores  $p$  y  $q$  de  $n$ , el menor de ellos no puede ser mayor a  $\sqrt{n}$ , mejorando así el algoritmo.

El siguiente paso es definir las listas donde se guardan los tiempos de ejecución y comenzar a iterar entre las diversas formas de encontrar los primos y de ordenar la lista de números, como se observa en el código 3.

Código 3: Ejecución de códigos

```

1 if __name__ == "__main__":
2     resultados_1 = {'Algoritmo 1': [], 'Algoritmo 2': [], 'Algoritmo 3': []}
3     resultados_2 = {'Original': [], 'Invertido': [], 'Aleatorio': []}
4     with multiprocessing.Pool(processes = cores-1) as pool:
5         pool.map(primo_1, original)
6         for r in range(replicas):
7             t = (time()*1000)
8             pool.map(primo_1, original)
9             resultados_1['Algoritmo 1'].append((time()*1000)-t)
10            t = (time()*1000)
11            pool.map(primo_2, original)
12            resultados_1['Algoritmo 2'].append((time()*1000)-t)
13            t = (time()*1000)
14            pool.map(primo_3, original)
15            resultados_1['Algoritmo 3'].append((time()*1000)-t)
16            t = (time()*1000)
17            pool.map(primo_3, original)
18            resultados_2['Original'].append((time()*1000) - t)
19            t = (time()*1000)
20            pool.map(primo_3, invertido)
21            resultados_2['Invertido'].append((time()*1000) - t)
22            t = (time()*1000)
23            pool.map(primo_3, aleatorio)
24            resultados_2['Aleatorio'].append((time()*1000) - t)
25    df1 = pd.DataFrame(data = resultados_1)
26    df2 = pd.DataFrame(data = resultados_2)

```

Por último, se realiza un análisis de varianza utilizando una instrucción de la librería estadística de `scipy` (ver código 4), para determinar si la media en cada tipo de variación (algoritmo y orden), representa una diferencia significativa que permita decir que el tiempo de ejecución depende del algoritmo utilizado o del orden inicial de los números.

Código 4: Test de análisis de varianza

```

1 stat1, p1 = f_oneway(resultados_1['Algoritmo 1'],
2                     resultados_1['Algoritmo 2'],
3                     resultados_1['Algoritmo 3'])
4 print('Variando algoritmo\n', 'stat=%.3f, p=%.3f' % (stat1, p1))
5 if p1 > 0.05:
6     print('Estadísticamente no significativa\n')
7 else:
8     print('Estadísticamente significativa\n')
9 stat2, p2 = f_oneway(resultados_2['Original'],
10                     resultados_2['Invertido'],
11                     resultados_2['Aleatorio'])
12 print('Variando orden de numeros\n', 'stat=%.3f, p=%.3f' % (stat2, p2))
13 if p2 > 0.05:
14     print('Estadísticamente no significativa\n')
15 else:
16     print('Estadísticamente significativa\n')

```

El código en su totalidad se puede obtener de mi [repositorio](#) en GitHub.

### 3. Resultados

Los resultados consisten en un test análisis de varianza para determinar si la diferencia entre tiempos de ejecución es estadísticamente significativa, además de un par de gráficas tipo violín para confirmar visualmente los resultados del análisis.

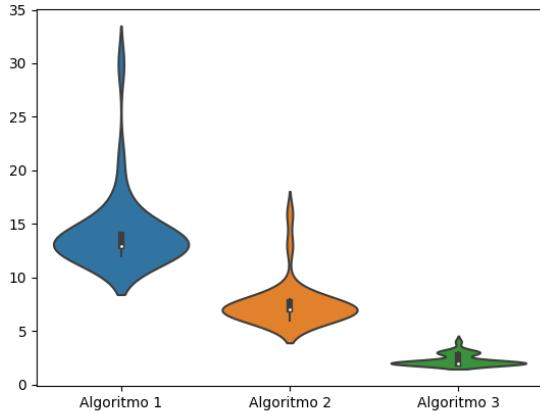
#### 3.1. Análisis de Varianza

Este modelo estadístico permite comprobar la veracidad de la hipótesis nula,  $H_0$ , al determinar si las medias de dos o más muestras pertenecen a la misma densidad de población [2]. En este caso, la hipótesis nula es que las medias de los tiempos pertenecen a la misma densidad de población, y por lo tanto no dependen del algoritmo u orden utilizado al no haber una diferencia significativa. Para concluir si la hipótesis se comprueba o no, se hace referencia al valor  $p$  calculado por el análisis implementado en el código. Específicamente, si  $p > 0,05$ , la diferencia no es significativa, mientras que si  $p < 0,05$ , la diferencia es estadísticamente significativa.

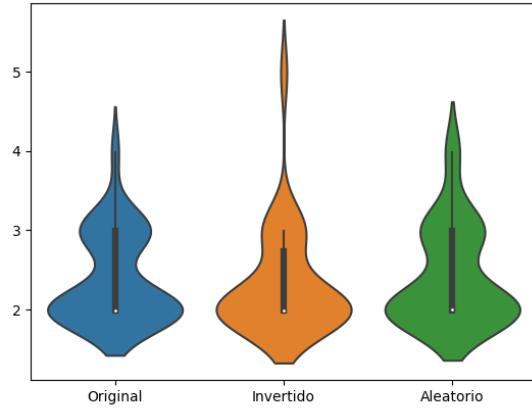
Al variar el algoritmo utilizado, el resultado del análisis indica un valor de  $p$  mucho menor a 0,05. Por otro lado, al variar el orden inicial de los números el resultado indica una  $p$  mayor a 0,05. Esto se puede visualizar mejor al hacer la comparación gráfica con los diagramas tipo violín. La interpretación de estos resultados se discute en la sección 4.

#### 3.2. Diagramas Tipo Violín

Un diagrama tipo violín es similar a uno de caja-bigote en el sentido que muestra el rango intercuartil y la media de una muestra o grupo de muestras. La diferencia radica en que el diagrama tipo violín contiene un diagrama de densidad en sus costados para mostrar la forma de densidad de la muestra de datos [3]. En la figura 1 se pueden observar los diagramas tipo violín de los tiempos de ejecución tanto al variar el algoritmo utilizado (figura 1a) como al variar el orden inicial de los números (figura 1b).



(a) Variación del algoritmo



(b) Variación del orden

Figura 1: Diagramas violín

## 4. Conclusiones

Tomando en cuenta un valor de  $p < 0,05$  como prueba de que la diferencia en tiempos de ejecución es significativa, entonces se tiene que: 1) Hay una clara dependencia entre el algoritmo utilizado y el tiempo que toma encontrar los número primos, y 2) la media de los tiempos de ejecución al variar el orden inicial no parece presentar una diferencia significativa.

Esto puede ser más evidente al observar los diagramas presentados. En la figura 1a se observa claramente la disminución de las medias (representadas por el punto blanco), al utilizar un algoritmo más eficiente. Por otro lado, en la figura 1b las medias se encuentran en posiciones muy similares, además de que las distribuciones para cada orden no son tan distintas.

## Referencias

- [1] E. Schaeffer. Queuingtheory. *Repositorio, GitHub*, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/QueuingTheory>.
- [2] Wikipedia. Analysis of variance, 2022. URL [https://en.wikipedia.org/wiki/Analysis\\_of\\_variance#The\\_F-test](https://en.wikipedia.org/wiki/Analysis_of_variance#The_F-test).
- [3] Wikipedia. Violin plot, 2022. URL [https://en.wikipedia.org/wiki/Violin\\_plot](https://en.wikipedia.org/wiki/Violin_plot).

# Reporte 4:

## Diagramas de Voronoi

Jorge Torres

6 de marzo de 2022

## 1. Objetivo

En esta práctica se genera un conjunto de celdas de Voronoi, también llamadas mosaicos de Dirichlet, con el método computacional descrito por P. J. Green y R. Sibson [1]. Se determina el efecto que tiene el variar la densidad de semillas en cinco niveles ( $k = 25, 50, 75, 100, 125$ ), en la probabilidad de que una segunda grieta llegue a tocar una primera, es decir, fracturando la pieza dos veces con posiciones iniciales generadas independientemente al azar, sobre varias réplicas. Se analizan los resultados y se visualizan en la sección 3.

## 2. Desarrollo

El código que se describe en esta práctica está basado en el [desarrollado](#) por E. Schaeffer [2], y se puede encontrar en su totalidad en el [repositorio](#) de J. Torres en GitHub.

El primer paso consiste en definir los parámetros iniciales de operación, a partir de los cuales se generan las celdas de Voronoi. Estos consisten en un tamaño de matriz de 100, cinco niveles para la densidad inicial de semillas y 200 repeticiones del experimento para cada una de ellas (ver código 1).

Código 1: Parámetros

```
1 n = 100
2 seed = [25, 50, 75, 100, 125]
3 runs = 200
```

En el código 2 se define la función `creacion()`, en donde colocan las semillas en el plano de manera aleatoria, mientras que con la función `celda()` se determinan las distancias euclídeanas más cercanas a cada semilla, definiendo así la celda de Voronoi para dicha semilla.

Código 2: Ubicación de semillas y celdas

```
1 def creacion():
2     semillas = []
3     for s in range(k):
4         while True:
5             x, y = randint(0, n - 1), randint(0, n - 1)
6             if (x, y) not in semillas:
7                 semillas.append((x, y))
8                 break
9     return semillas
10
11 def celda(pos):
12     if pos in semillas:
13         return semillas.index(pos)
14     x, y = pos % n, pos // n
15     cercano = None
16     menor = n * sqrt(2)
17     for i in range(k):
18         (xs, ys) = semillas[i]
19         dx, dy = x - xs, y - ys
20         dist = sqrt(dx**2 + dy**2)
21         if dist < menor:
22             cercano, menor = i, dist
23     return cercano
```

Al tener las celdas definidas, se puede proseguir a determinar, de manera aleatoria, la posición inicial de la grieta con la función `inicio()` descrita en el código 3.

Código 3: Ubicación inicial de la grieta

```

1 def inicio():
2     direccion = randint(0, 3)
3     if direccion == 0:
4         return (0, randint(0, n - 1))
5     elif direccion == 1:
6         return (randint(0, n - 1), 0)
7     elif direccion == 2:
8         return (randint(0, n - 1), n - 1)
9     else:
10        return (n - 1, randint(0, n - 1))

```

Para propagar la grieta se tiene en cuenta que es más probable que se propague a lo largo de una frontera de celda, mientras que en el interior la probabilidad va disminuyendo gradualmente. En el código 4 se propaga una grieta en color negro hasta que toca un borde o hasta que le es demasiado difícil seguir propagándose al interior de una celda.

Código 4: Propagación de la primera grieta

```

1 def propaga_n():
2     prob, dificil = 0.9, 0.8
3     grieta_n = voronoi.copy()
4     g = grieta_n.load()
5     (xn, yn) = inicio()
6     negro = (0, 0, 0)
7     while True:
8         g[xn, yn] = negro
9         frontera, interior = [], []
10        for v in vecinos:
11            (dx, dy) = v
12            vx, vy = xn + dx, yn + dy
13            if vx >= 0 and vx < n and vy >= 0 and vy < n:
14                if g[vx, vy] != negro:
15                    if vor[vx, vy] == vor[xn, yn]:
16                        interior.append(v)
17                    else:
18                        frontera.append(v)
19        elegido = None
20        if len(frontera) > 0:
21            elegido = choice(frontera)
22            prob = 1
23        elif len(interior) > 0:
24            elegido = choice(interior)
25            prob *= dificil
26        if elegido is not None:
27            (dx, dy) = elegido
28            xn, yn = xn + dx, yn + dy
29        else:
30            break
31    return grieta_n

```

Se puede generar una segunda grieta (en color blanco) con la función `inicio()` del código 3. Ésta se propagaría de la misma manera que la primera, con la excepción de que se detiene por completo al hacer contacto con la primera grieta. Este comportamiento se puede observar en el código 5.

Código 5: Propagación de la segunda grieta

```

1 def propaga_b():
2     prob, dificil = 0.9, 0.8
3     grieta_b = propaga_n()
4     g = grieta_b.load()
5     (xb, yb) = inicio()
6     blanco = (255, 255, 255)
7     negro = (0, 0, 0)
8     while True:
9         g[xb, yb] = blanco
10        frontera, interior = [], []
11        for v in vecinos:
12            (dx, dy) = v

```

```

13     vx, vy = xb + dx, yb + dy
14     if vx >= 0 and vx < n and vy >= 0 and vy < n:
15         if g[vx, vy] != blanco:
16             if vor[vx, vy] == vor[xb, yb]:
17                 interior.append(v)
18             else:
19                 frontera.append(v)
20     elegido = None
21     if len(frontera) > 0:
22         elegido = choice(frontera)
23         prob = 1
24     elif len(interior) > 0:
25         elegido = choice(interior)
26         prob *= dificil
27     if elegido is not None:
28         (dx, dy) = elegido
29         xb, yb = xb + dx, yb + dy
30         if g[xb, yb] == negro:
31             tocaron.append(1)
32             break
33     else:
34         break
35 return grieta_b

```

En el código 6 se inicializan las funciones descritas anteriormente. El experimento se repite 200 veces para cada una de las densidades iniciales de semillas. Se lleva un conteo de las veces que se tocaron y se calcula la probabilidad de que se toquen basada en esos resultados utilizando la ecuación 1,

$$P = \frac{N_T}{R} \times 100 \quad (1)$$

donde  $N_T$  es la cantidad de veces que se tocaron y  $R$  es la cantidad de iteraciones.

Código 6: Iteraciones del experimento

```

1 for k in seed:
2     tocaron = []
3     for r in range(runs):
4         semillas = creacion()
5         celdas = [celda(i) for i in range(n * n)]
6         voronoi = Image.new('RGB', (n, n))
7         vor = voronoi.load()
8         c = sns.color_palette("Set3", k).as_hex()
9         for i in range(n * n):
10             vor[i % n, i // n] = ImageColor.getrgb(c[celdas.pop(0)])
11             limite, vecinos = n, []
12             for dx in range(-1, 2):
13                 for dy in range(-1, 2):
14                     if dx != 0 or dy != 0:
15                         vecinos.append((dx, dy))
16             propaga_b()
17     pr = (len(tocaron)/runs)*100
18     resultado.append(pr)

```

### 3. Resultados

Para una mejor visualización de la generación de las celdas y las grietas, en la figura 1 se exponen un par de ejemplos de éstas. Uno donde las grietas no se tocan (figura 1a), y otro donde las grietas se tocan (figura 1b). Por otro lado, en la figura 2 se muestran las probabilidades calculadas de que las grietas se toquen para los cinco distintos niveles de densidad de semillas.

### 4. Conclusiones

Observando la gráfica de la figura 2, podría aparentar que la probabilidad de que las grietas se toquen aumenta conforme se aumenta también la densidad de semillas. Debido a la naturaleza aleatoria del sistema, sería difícil decir con certeza el por qué esto es así, sin embargo podría deberse a que la cantidad de fronteras aumenta con la cantidad de semillas, además de que el tamaño promedio de celda disminuye, lo cual sería propicio para que las



(a) Las grietas no se tocan.

(b) Las grietas se tocan.

Figura 1: Ejemplos de grietas.

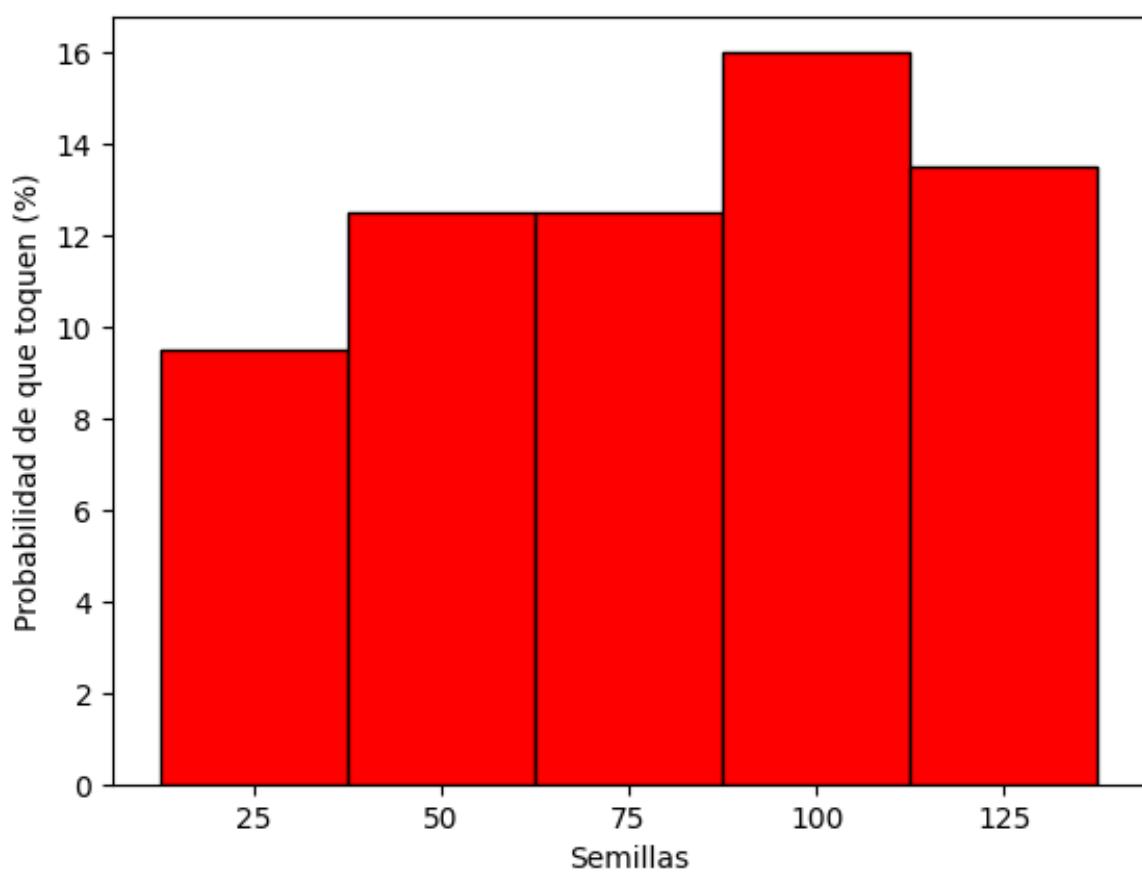


Figura 2: Probabilidades de que las grietas se toquen para los cinco niveles de densidad de semillas.

grietas coincidan en un lugar. Sin embargo, la probabilidad disminuye ligeramente en el último nivel, lo cual hace pensar que existe un límite para la propagación de las grietas. La determinación de dicho límite se encuentra fuera del rango de estudio de ésta práctica.

## Referencias

- [1] P. J. Green and R. Sibson. Computing dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978. doi: <https://doi.org/10.1093/comjnl/21.2.168>.
- [2] E. Schaeffer. Voronoidiagrams. *Repositorio, GitHub*, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/VoronoiDiagrams>.

Reporte 5:  
Método Monte-Carlo

Jorge Torres

13 de marzo de 2022

# Capítulo 1

## Estimación de una Integral Definida

### 1.1. Objetivo

El objetivo de esta actividad es el estudiar estadísticamente la convergencia de la precisión del estimado de una integral definida (ecuación 1.1), con el método Monte Carlo, comparándolo con el valor producido por [Wolfram Alpha](#), en términos de 1) el error absoluto, 2) el error cuadrado y 3) la cantidad de decimales correctos, aumentando el tamaño de muestra en tres niveles.

$$\int_3^7 \frac{1}{\exp(x) + \exp(-x)} dx \quad (1.1)$$

### 1.2. Desarrollo

El desarrollo de la actividad está basado en el [código](#) implementado por E. Schaeffer [2]. En el código 1.1 se establecen los parámetros para la ejecución del programa, que consisten en 1) el valor estimado por Wolfram Alpha, `wolfram`, 2) el rango en que se define la integral, `desde` y `hasta`, 3) la cantidad de números pseudoaleatorios que se producen con la distribución definida por la integral, `pedazo`, y 4) la lista de niveles en que se varía la cantidad de iteraciones del experimento, `cuantos`.

Código 1.1: Parámetros

```
1 wolfram = 0.048834111126049311
2 desde = 3
3 hasta = 7
4 pedazo = 50000
5 cuantos = [500, 5000, 50000]
```

Se define la normalización de la integral por medio de la función `g(x)` dada por la ecuación 1.2 y se vectoriza la distribución resultante en el código 1.2.

$$\frac{2}{\pi} \int_{-\infty}^{\infty} \frac{1}{\exp(x) + \exp(-x)} dx \quad (1.2)$$

Código 1.2: Normalización de la Integral

```
1 def g(x):
2     return (2 / (pi * (exp(x) + exp(-x))))
3
4 vg = np.vectorize(g)
5 X = np.arange(-8, 8, 0.05)
6 Y = vg(X)
```

Importando una `receta` y utilizándola en conjunto con la función `parte()` del código 1.3, se genera una cantidad de números aleatorios con una distribución de acuerdo a la integral normalizada, visualizada en la figura 1.1.

Código 1.3: Generación de Números Aleatorios

```
1 generador = GeneralRandom(np.asarray(X), np.asarray(Y))
2
3 def parte(replica):
```

```

4     V = generador.random(pedazo)[0]
5     return ((V >= desde) & (V <= hasta)).sum()

```

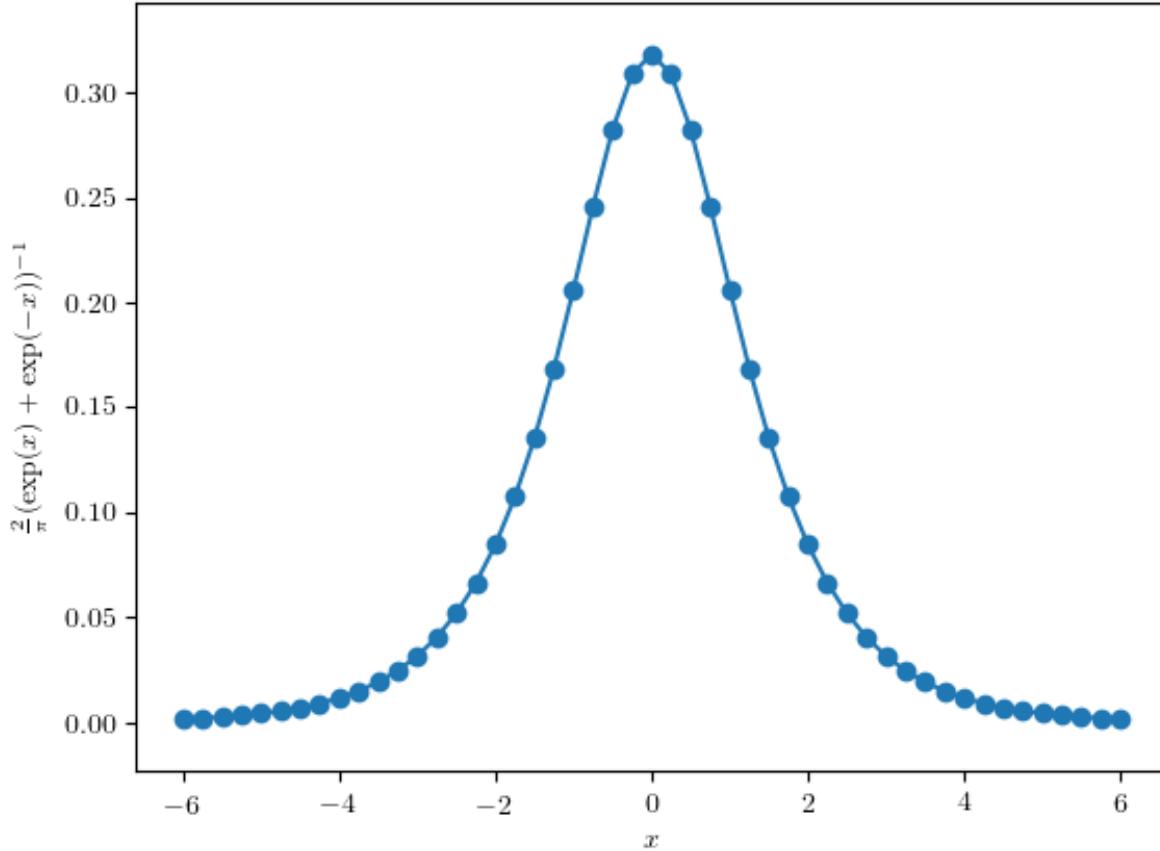


Figura 1.1: Distribución de la integral normalizada. Obtenida de [P5 - Simulación](#)

Para determinar la cantidad de decimales correctos entre el estimado de la integral y el valor de Wolfram Alpha, se define la función `compare_strings` en el código 1.4, que compara ambos valores carácter por carácter hasta que el decimal es diferente.

Código 1.4: Comparación de Decimales

```

1 def compare_strings(a, b):
2     a = str(a)
3     b = str(b)
4
5     if a is None or b is None:
6         return 0
7
8     size = min(len(a), len(b))
9     count = 0
10
11    for i in range(size):
12        if a[i] == b[i]:
13            count += 1
14        else:
15            break
16    return count

```

Por último, por medio del código 1.5 se ejecutan las iteraciones del experimento y se calculan los valores de los errores para poder comparar las estimaciones de la integral definida dependiendo de la cantidad de iteraciones. El desarrollo completo del código se puede revisar en el [repositorio](#) en GitHub de J. Torres [3].

Código 1.5: Estimación de la Integral y Cálculo de Errores

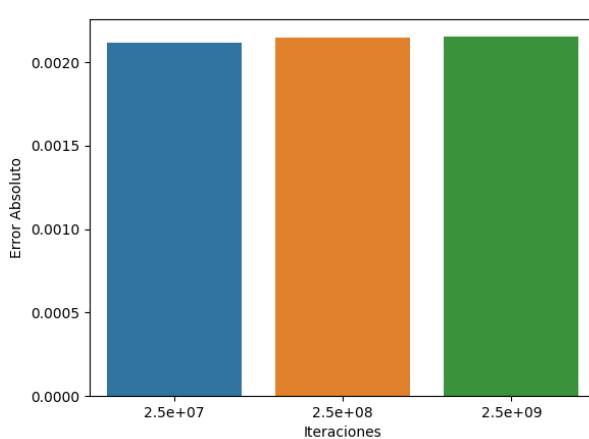
```

1 if __name__ == "__main__":
2     with multiprocessing.Pool() as pool:
3         for c in cuantos:
4             p = c * pedazo
5             puntos.append('{:.1e}'.format(p))
6             montecarlo = pool.map(parte, range(c))
7             integral = sum(montecarlo) / p
8             valor = (pi / 2) * integral
9             ae.append(abs(valor - wolfram))
10            se.append(((valor - wolfram)**2))
11            dec.append(compare_strings(wolfram, valor) - 2)
12    resultados = {'Iteraciones': puntos,
13                  'Error Absoluto': ae,
14                  'Error Cuadrado': se,
15                  'Decimales Correctos': dec}
16 df = pd.DataFrame(resultados)

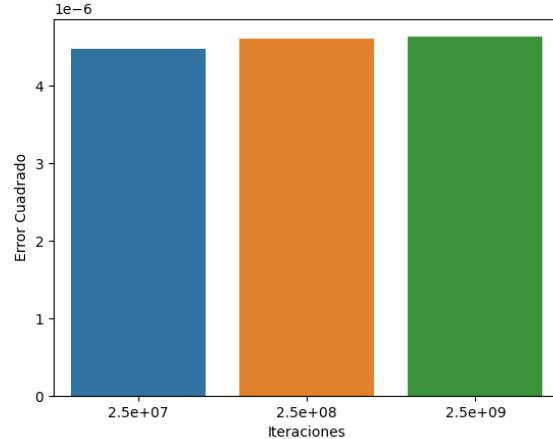
```

### 1.3. Resultados

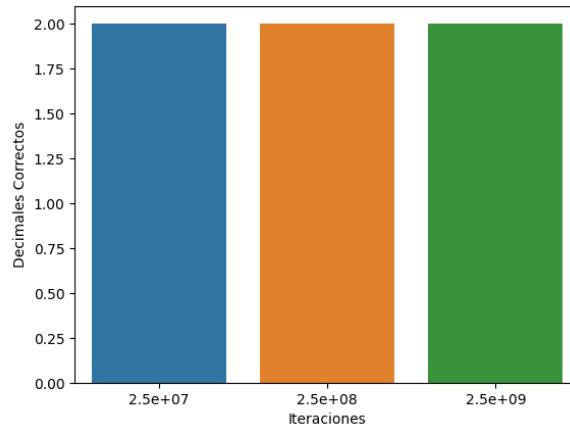
En los diagramas de la figura 1.2 se puede apreciar la variación en los valores de error absoluto (figura 1.2a), error cuadrado (figura 1.2b), y cantidad de decimales correctos (figura 1.2c), al comparar entre los valores de la integral definida estimados por Wolfram Alpha y por el código desarrollado en la actividad.



(a) Error absoluto.



(b) Error cuadrado.



(c) Decimales correctos.

Figura 1.2: Errores determinados entre la estimación de Wolfram Alpha y la calculada por el código desarrollado.

## 1.4. Conclusiones

Como se puede apreciar en los diagramas de la sección 1.3, aumentar la cantidad de iteraciones de  $2,5 \times 10^7$  hasta  $2,5 \times 10^9$  no presenta una diferencia apreciable en ningún tipo de error. Lo que es más, tanto el error absoluto como el error cuadrado son mínimos y no representarían un error estrictamente significativo. Sin embargo, se puede ver que la cantidad de decimales correctos no excede un máximo de dos, por lo que el rango de error comienza a crecer en las milésimas. En muchas aplicaciones, este rango podría causar diferencias bastante apreciables en cálculos que impliquen más precisión.

# Capítulo 2

## Estimación de $\pi$

### 2.1. Objetivo

El primer reto de esta actividad consiste en utilizar el método Monte-Carlo para estimar el valor de  $\pi$  a partir de una serie de puntos, determinando si las coordenadas de estos puntos se encuentran dentro o fuera del área de un círculo de radio  $r$ .

### 2.2. Desarrollo

El código 2.1 es tomado directamente del método descrito por W. Kurt [1], y en él se utiliza la relación entre el área de un círculo y el cuadrado que lo circunscribe para aproximar el valor de  $\pi$  a partir de una serie de puntos de números generados aleatoriamente dentro de una distribución normal. Más específicamente, se suma la cantidad de puntos que queda dentro del círculo, se divide entre la cantidad total de puntos, y se multiplica por 4 para obtener la aproximación.

Código 2.1: Cálculo de  $\pi$  por Método Monte-Carlo

```
1 runs <- 1000000
2 xs <- runif(runs,min=-0.5,max=0.5)
3 ys <- runif(runs,min=-0.5,max=0.5)
4 in.circle <- xs^2 + ys^2 <= 0.5^2
5 mc.pi <- (sum(in.circle)/runs)*4
6 plot(xs,ys,pch='.',col=ifelse(in.circle,"blue","grey"))
7 ,xlab='',ylab='',asp=1,
8 main=paste("MC Approximation of Pi =",mc.pi))
```

### 2.3. Resultados

En la figura 2.1 se visualiza de manera clara la serie de puntos ubicados aleatoriamente, que caen tanto dentro como fuera del círculo. También se puede apreciar la aproximación calculada,  $\pi = 3,141956$ .

### 2.4. Conclusiones

Inmediatamente queda claro que éste método es útil para aproximar valores de procesos que son complejos de calcular analíticamente con una precisión aceptable, ya que se ha podido estimar  $\pi$  correctamente en un rango tres decimales. Es de esperar que la precisión aumente con la cantidad de puntos que se generan, pero en esta actividad no se han realizado pruebas para confirmarlo.

**MC Approximation of Pi = 3.141956**

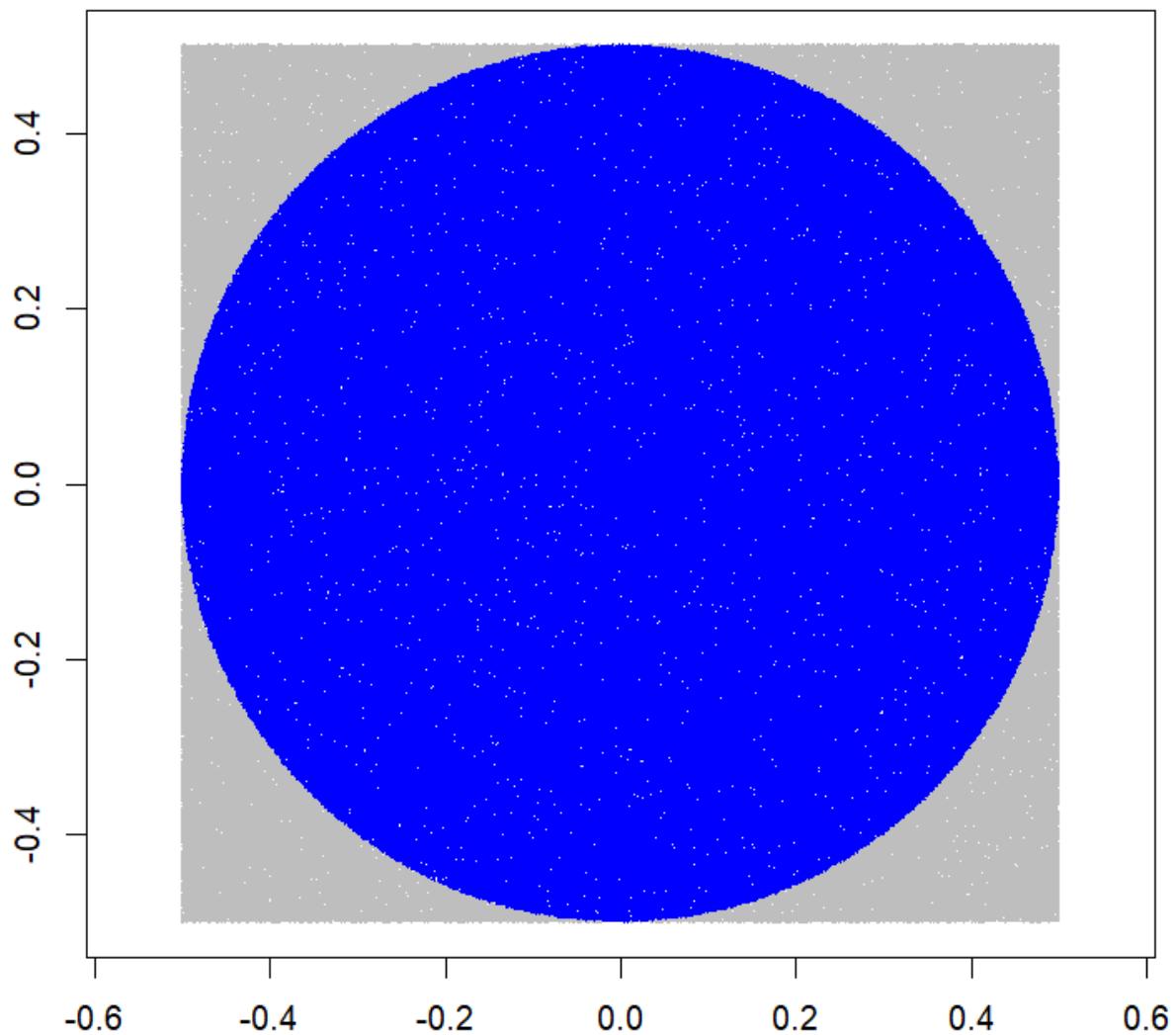


Figura 2.1: Círculo formado por los puntos aleatorios.

# Bibliografía

- [1] W. Kurt. Count bayesie, 2015. URL <https://www.countbayesie.com/blog/2015/3/3/6-amazing-trick-with-monte-carlo-simulations>.
- [2] E. Schaeffer. Montecarlo, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/MonteCarlo>.
- [3] J. Torres. P5, 2022. URL <https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P5>.

# Reporte 6:

## Sistema Multiagente

Jorge Torres

20 de marzo de 2022

### 1. Objetivo

En esta práctica se implementa un sistema multiagente aplicado en epidemiología a través del modelo SIR, que consiste en definir tres tipos de agentes: susceptibles, infectados y recuperados. El objetivo es el de analizar el efecto que tiene el contener o reducir la velocidad de movimiento de los agentes infectados, tanto en la magnitud de la epidemia (la altura del pico en la curva del porcentaje de infectados por iteración), como en su velocidad (la iteración en la cual se llega por la primera vez al valor pico).

### 2. Desarrollo

Basando el desarrollo de la práctica en el [código](#) desarrollado por E. Schaeffer [1], en primer lugar, se definen los parámetros iniciales con los cuales actúa el sistema, como se observa en el código 1. Estos se conforman por 1) el largo de cada lado del sistema,  $l$ , 2) la cantidad de agentes que se crean,  $n$ , 3) la probabilidad inicial de infectados,  $p_i$ , 4) la probabilidad de que cada agente infectado se recupere,  $p_r$ , 5) la velocidad inicial de los agentes,  $v$ , 6) el umbral de cercanía para considerar una infección,  $r$ , 7) la cantidad de pasos que hacen los agentes,  $t_{max}$ , y 8) la cantidad de veces que se repite el experimento,  $runs$ . El [desarrollo](#) completo se puede observar en el repositorio en GitHub de J. Torres [2]

Código 1: Parámetros Iniciales

```
1 l = 1.5
2 n = 50
3 pi = 0.05
4 pr = 0.02
5 v = 1 / 30
6 r = 0.1
7 tmax = 100
8 runs = 100
```

Con la función `contagiados()` del código 2 se decide qué agentes son los próximos en infectarse de acuerdo a la probabilidad,  $p_c$ , descrita en la ecuación 1,

$$p_c = \begin{cases} 0, & \text{si } d(i, j) \geq r, \\ \frac{r-d}{r}, & \text{para cualquier otro caso,} \end{cases} \quad (1)$$

donde  $d$  es la distancia euclídea entre un par,  $(i, j)$ , de agentes y  $r$  es el umbral de cercanía.

Código 2: Nuevos Agentes Infectados

```
1 def contagios():
2     for i in range(n):
3         a1 = agentes.iloc[i]
4         if a1.estado == 'I':
5             for j in range(n):
6                 a2 = agentes.iloc[j]
7                 if a2.estado == 'S':
8                     d = sqrt((a1.x - a2.x)**2 + (a1.y - a2.y)**2)
9                     if d < r:
10                         if random() < (r - d) / r:
11                             contagios[j] = True
12
13 return contagios
```

Utilizando el código 3, se crean los agentes, se distribuyen uniformemente a través del espacio y se decide su estado inicial de infección utilizando la probabilidad inicial,  $\pi_i$ . Además, para cada agente creado, se decide de manera aleatoria un diferencial de movimiento que representa su velocidad, dado por los términos  $dx$  y  $dy$ .

Código 3: Creación de Agentes

```

1 agentes = pd.DataFrame()
2 agentes['x'] = [uniform(0, 1) for i in range(n)]
3 agentes['y'] = [uniform(0, 1) for i in range(n)]
4 agentes['estado'] = ['S' if random() > pi else 'I' for i in range(n)]
5 agentes['dx'] = [uniform(-v, v) for i in range(n)]
6 agentes['dy'] = [uniform(-v, v) for i in range(n)]

```

Con las líneas del código 4 se lleva un conteo de la cantidad de infectados para cada paso iterado, y se manda a llamar a la función `contagiados()` del código 2 para decidir los nuevos agentes infectados. Si la cantidad de infectados llega a cero, el programa se detiene.

Código 4: Conteo de Infectados

```

1 epidemia = []
2     for tiempo in range(tmax):
3         conteos = agentes.estado.value_counts()
4         infectados = conteos.get('I', 0)
5         epidemia.append(infectados)
6         contagios = [False for i in range(n)]
7         if infectados == 0:
8             break
9         contagios = contagiados()

```

En seguida, se decide si un agente cambia de estado a infectado (por medio de la función `contagiados()`, descrita previamente), o a recuperado por medio de la probabilidad de recuperación,  $pr$ . En el código 5 se observan éstas funciones.

Código 5: Decisión de Agentes Infectados y Recuperados

```

1     for i in range(n):
2         a = agentes.iloc[i]
3         if contagios[i]:
4             agentes.at[i, 'estado'] = 'I'
5         elif a.estado == 'I':
6             if random() < pr:
7                 agentes.at[i, 'estado'] = 'R'

```

Por último, se trasladan todos los agentes a su nueva posición dada por los diferenciales  $dx$  y  $dy$ , establecidos en el código 3. Las líneas 3 a 6 simplemente establecen que los agentes que lleguen a un límite del plano reaparecen en el límite opuesto, con la misma velocidad y dirección. Con las líneas 9 y 10 se crea una lista de la cantidad máxima de infectados y la iteración en la que se llegó por primera vez a esa cantidad, para todas las repeticiones del experimento. Esto se observa en el código 6.

Código 6: Movimiento de los Agentes

```

1     x = a.x + a.dx
2     y = a.y + a.dy
3     x = x if x < l else x - l
4     y = y if y < l else y - l
5     x = x if x > 0 else x + l
6     y = y if y > 0 else y + l
7     agentes.at[i, 'x'] = x
8     agentes.at[i, 'y'] = y
9     maximos['Maximos Infectados'].append(max(epidemia))
10    maximos['Posicion'].append(epidemia.index(max(epidemia)) + 1)

```

## 2.1. Reducción de Velocidad de Agentes Infectados

Para hacer una comparación entre el modelo epidemiológico con agentes infectados a velocidad normal y uno en donde éstos reducen su velocidad a la mitad, se tienen que realizar algunas modificaciones. Las líneas 5 y 6 del código 3 se modifican para reducir la velocidad de los agentes inicialmente infectados a la mitad, como se observa en el código 7.

Código 7: Modificación del Código 3

```

1 agentes['dx'] = [uniform(-v/2, v/2) if agentes.at[i, 'estado'] == 'I'\
2     else uniform(-v, v) for i in range(n)]
3 agentes['dy'] = [uniform(-v/2, v/2) if agentes.at[i, 'estado'] == 'I'\
4     else uniform(-v, v) for i in range(n)]

```

Por otro lado, al código 5 se agregan dos líneas para reducir la velocidad de los nuevos agentes infectados, de tal manera que resulta en el código 8.

Código 8: Modificación del Código 5

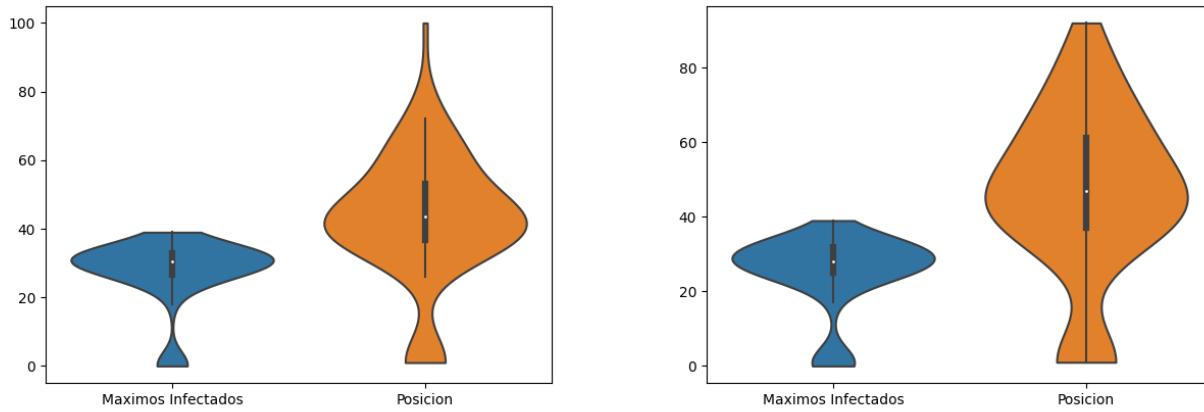
```

1 for i in range(n):
2     a = agentes.iloc[i]
3     if contagios[i]:
4         agentes.at[i, 'dx'] /= 2
5         agentes.at[1, 'dy'] /= 2
6         agentes.at[i, 'estado'] = 'I'
7     elif a.estado == 'I':
8         if random() < pr:
9             agentes.at[i, 'estado'] = 'R'

```

### 3. Resultados

En los diagramas de la figura 1 se puede observar una comparación entre las distribuciones de los picos de infectados y las iteraciones en que se llega por primera vez a esos picos, para una epidemia con velocidad normal de agentes (figura 1a) y una en donde los agentes infectados reducen su velocidad a la mitad (figura 1b).



(a) Distribución de epidemia a velocidad normal.

(b) Distribución de epidemia con velocidad reducida.

Figura 1: Distribuciones de máxima cantidad de infectados y posición del pico para cada tipo de epidemia.

### 4. Conclusiones

Al analizar la comparación de la figura 1, se puede observar que, al reducir la velocidad de agentes infectados, no hay una diferencia relevante en cuanto a la máxima cantidad de infectados, ambas epidemias llegando a picos de aproximadamente 30. Esto puede deberse a la cantidad limitada de agentes totales y el hecho de que los agentes recuperados ya no pueden infectarse ni propagar la infección.

Por otro lado, sí se observa una diferencia significativa en cuanto al comportamiento en el tiempo de la epidemia, observando que la velocidad de infección se reduce al hacerlo también la velocidad de los agentes infectados, propagándose más lentamente.

## Referencias

- [1] E. Schaeffer. Multiagent, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/MultiAgent>.
- [2] J. Torres. P6, 2022. URL <https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P6>.

# Reporte 7: Búsqueda Local

Jorge Torres

27 de marzo de 2022

## 1. Objetivo

La actividad consiste en maximizar una variante de una función bidimensional,  $g(x, y)$ , por medio de una búsqueda local aleatoria con restricciones en los ejes. La posición actual es un par  $x, y$  y se realizan dos movimientos aleatorios,  $\Delta x$  y  $\Delta y$ , cuyas combinaciones posibles proveen ocho posiciones vecino, de los cuales aquella que logra el mayor valor para  $g$  es seleccionado. El resultado final es un compendio de los valores máximos encontrados para 10, 20 y 30 réplicas simultáneas de 100, 1000 y 10,000 pasos de la búsqueda. Además, en la sección 3 se visualizan algunos de los pasos que se realizaron para la búsqueda de 10 réplicas en una proyección plana de la función.

## 2. Desarrollo

El desarrollo de la práctica está basado en el [código](#) implementado por E. Schaeffer [1], con el cual realiza una búsqueda de valores mínimos para una función de una sola variable. En ésta actividad, se selecciona la ecuación 1 como función a evaluar por la particular cantidad de valores máximos y mínimos locales que presenta, lo que podría representar una dificultad para encontrar un máximo total. La ecuación se define en el código 1.

$$g(x, y) = \cos^2 x + 0,3x + \sin^2 y - 0,6y \quad (1)$$

Código 1: Función a Evaluar

```
1 def g(x, y):
2     px = (cos(x))**2 + 0.3 * x
3     py = (sin(y))**2 - 0.6 * y
4     return px + py
```

En el código 2 se definen algunos parámetros con los que se evalúa la función, como el rango de los ejes, la resolución con que se le grafica y el valor máximo que pueden tomar los diferenciales de movimiento,  $\Delta x$  y  $\Delta y$ .

Código 2: Parámetros de Evaluación

```
1 low = -10
2 high = -low
3 step = 0.05
4 p = np.arange(low, high, step)
5 n = len(p)
6 z = np.zeros((n, n), dtype=float)
7 valores=[]
8 paso = 0.5
```

Para graficar la función en una proyección plana, se necesita evaluar la función en una cantidad finita de puntos, dada por la instrucción `p`, dentro del rango previamente definido, como se observa en el código 3.

Código 3: Puntos para Graficar la Función

```
1 for i in range(n):
2     x = p[i]
3     for j in range(n):
4         y = p[n - j - 1] # voltear
5         z[i, j] = g(x, y)
6         valores.append(g(x, y))
```

Por medio del código 4 se definen las coordenadas de los agentes o puntos que realizarán la búsqueda. Además, también se definen las coordenadas iniciales para el mejor valor entre los agentes creados.

Código 4: Creación de Coordenadas para Búsqueda

```

1 tmax=10
2 for a in range(10, 31, 10):
3
4     resultados = pd.DataFrame()
5     for tiem in range(2, 5):
6         agentes = pd.DataFrame()
7         agentes['x'] = [uniform(low, high) for i in range(a)]
8         agentes['y'] = [uniform(low, high) for i in range(a)]
9         agentes['best'] = [min(valores) for i in range(a)]
10        bestx = agentes['x'][0]
11        besty = agentes['y'][0]
12        best = g(bestx, besty)

```

En el código 5, utilizando los valores de `tmax` y `tiem` definidos anteriormente, se consigue la cantidad de pasos que darán los agentes. Los diferenciales de paso se deciden de manera aleatoria y se agregan a la lista.

Código 5: Diferenciales de Paso

```

1     for tiempo in range(tmax**tiem):
2         agentes['dx'] = [uniform(0, paso) for i in range(a)]
3         agentes['dy'] = [uniform(0, paso) for i in range(a)]

```

Para que cada agente realice movimientos en cada una de las ocho posibles direcciones, se implementa el código 6. Además, se evalúa la función en las ocho direcciones para posteriormente poder determinar cuál presenta un valor mayor. Las líneas 9 a 16 impiden que cualquiera de los agentes realice un movimiento que se encuentre fuera del rango evaluado.

Código 6: Movimiento de los Agentes

```

1     for i in range(a):
2         r = agentes.iloc[i]
3
4         xl = r.x - r.dx
5         xr = r.x + r.dx
6         yd = r.y - r.dy
7         yu = r.y + r.dy
8
9         if xl < low+step:
10            xl = r.x
11         if xr > high-step:
12            xr = r.x
13         if yd < low+step:
14            yd = r.y
15         if yu > high-step:
16            yu = r.y
17
18         g1 = g(xl, yu)
19         g2 = g(r.x, yu)
20         g3 = g(xr, yu)
21         g4 = g(xl, r.y)
22         g5 = g(xr, r.y)
23         g6 = g(xl, yd)
24         g7 = g(r.x, yd)
25         g8 = g(xr, yd)
26         lista = [g1,g2,g3,g4,g5,g6,g7,g8]

```

Por último, se decide la dirección en que la función toma un valor menor y se reescriben las coordenadas de los agentes. Además, se toman las coordenadas del mayor de los valores evaluados de la función y se guardan para poder graficar el punto posteriormente, como se aprecia en el código 7.

Código 7: Maximización de la Función

```

1 mayor = lista.index(max(lista))+1
2
3 if mayor == 1:
4     agentes.at[i, 'x'] = xl
5     agentes.at[i, 'y'] = yu
6 elif mayor ==2:

```

```

7         agentes.at[i, 'x'] = r.x
8         agentes.at[i, 'y'] = yu
9     elif mayor ==3:
10        agentes.at[i, 'x'] = xr
11        agentes.at[i, 'y'] = yu
12    elif mayor ==4:
13        agentes.at[i, 'x'] = xl
14        agentes.at[i, 'y'] = r.y
15    elif mayor ==5:
16        agentes.at[i, 'x'] = xr
17        agentes.at[i, 'y'] = r.y
18    elif mayor ==6:
19        agentes.at[i, 'x'] = xl
20        agentes.at[i, 'y'] = yd
21    elif mayor ==7:
22        agentes.at[i, 'x'] = r.x
23        agentes.at[i, 'y'] = yd
24    elif mayor ==8:
25        agentes.at[i, 'x'] = xr
26        agentes.at[i, 'y'] = yd
27
28    mejor = g(r.x, r.y)
29    if mejor > best:
30        best = g(r.x, r.y)
31        bestx = r.x
32        besty = r.y
33
34    if mejor > r.best:
35        agentes.at[i, 'best'] = mejor

```

El desarrollo completo del código puede encontrarse en el repositorio de J. Torres en GitHub [2].

### 3. Resultados

En su repositorio de GitHub, J. Torres presenta una animación tipo GIF del proceso completo que realizan 10 agentes en 1000 pasos para encontrar los valores máximos de la función evaluada, mientras que en la figura 1 se pueden observar algunos de estos pasos. La proyección plana está codificada por colores, en donde el azul oscuro representa los menores valores y el amarillo brillante representa los mayores valores que toma la función al ser evaluada en el rango definido. El eje horizontal representa los valores de las  $x$ , mientras que el eje vertical representa los valores de las  $y$ . Cada uno de los puntos rojos que se observan es un agente que realiza un movimiento,  $\Delta x$  y  $\Delta y$ , en cada paso de la iteración, mientras que el punto verde es el máximo valor encontrado en dicho paso.

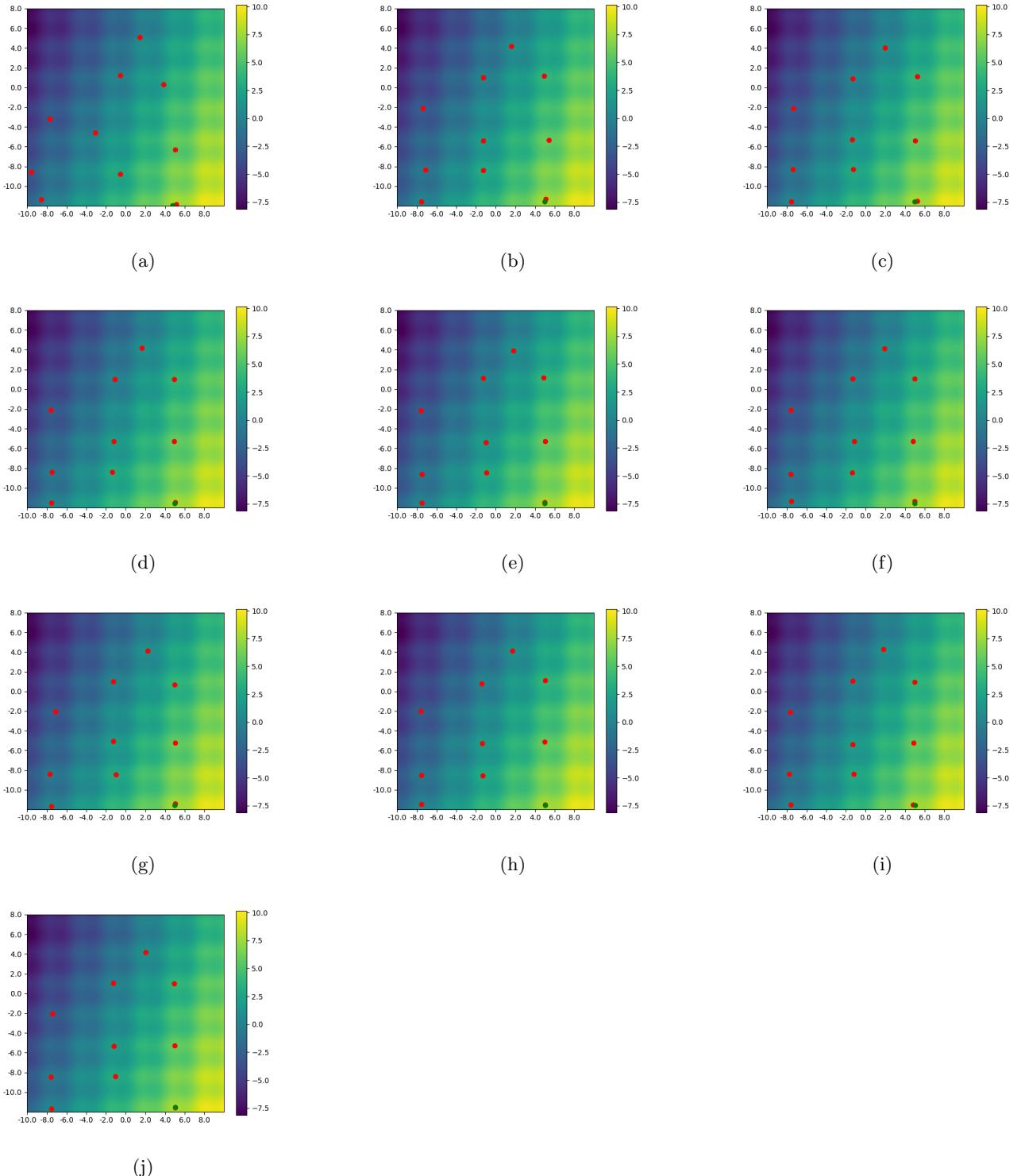


Figura 1: Movimientos de agentes y maximización de la función.

## 4. Conclusiones

Como se puede observar en la figura 1, ésta técnica es bastante versátil y logra encontrar los valores máximos (o mínimos) de una función. Sin embargo, presenta algunas dificultades al tratarse de una función oscilatoria, pues los puntos pueden caer en un ciclo en donde no pueden salir de una cresta o un valle y, por lo tanto, no encuentran un máximo o mínimo absoluto.

## Referencias

- [1] E. Schaeffer. Localsearch, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/LocalSearch>.
- [2] J. Torres. P7, 2022. URL <https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P7>.

# Reporte 8:

## Modelo de Urnas

Jorge Torres

4 de abril de 2022

### 1. Objetivo

En esta práctica se realiza la simulación de un sistema de coalescencia y fragmentación de partículas, en donde una cantidad  $n$  de partículas forma inicialmente una cantidad  $k$  de cúmulos de diversos tamaños. En cada paso de la iteración, un cúmulo puede unirse a otro o puede fragmentarse en dos cúmulos más pequeños, no necesariamente del mismo tamaño. El objetivo consiste en estudiar estadísticamente el efecto que tiene la tasa  $n/k$  en el porcentaje de partículas que se lograrían filtrar si se tuviese un filtro de un tamaño determinado. En esta ocasión, se utilizan tres cantidades de partículas en combinación con tres cantidades de cúmulos iniciales, dando un total de 9 combinaciones para la tasa  $n/k$ .

### 2. Desarrollo

El desarrollo de la presente práctica está basado en el [código](#) implementado por E. Schaeffer [1], en donde realiza una simulación de unión y fragmentación de partículas. El [desarrollo](#) completo puede encontrarse en el repositorio en GitHub de J. Torres [2].

En primera instancia, se definen los parámetros iniciales con los que opera la simulación. Éstos consisten de una lista con las tres cantidades de cúmulos iniciales, `clusters`; una lista con las tres cantidades de partículas, `particles`; el tamaño del filtro, en términos de la cantidad mínima de partículas que debe tener un cúmulo para que se quede alojado en el mismo, `filtersize`; la cantidad de iteraciones que se realizan para cada combinación de partículas y cúmulos, `runs`; y la cantidad de pasos que dura cada iteración para cada combinación, `dur`. Los parámetros se pueden ver implementados en el código 1.

Código 1: Parámetros de Operación

```
1 clusters = [2500, 5000, 10000]
2 particles = [250000, 500000, 1000000]
3 filtersize = 100
4 runs = 100
5 dur = 50
```

En el código 2 se definen las probabilidades de que los cúmulos se rompan o se unan con las funciones `rotura()` y `union()`, respectivamente. La probabilidad de rotura depende de manera sigmoidal de un tamaño crítico de cúmulo,  $c$ , y de un factor arbitrario para suavizar la curva,  $d$ . El tamaño crítico está dado por la media de tamaños de los cúmulos en cada iteración, mientras que el factor de suavización se determina con la desviación estándar de los tamaños de cúmulos. Por otro lado, la probabilidad de unión depende únicamente del tamaño crítico de manera exponencial negativa. De esta forma, los cúmulos más pequeños tienden a unirse, mientras que los más grandes tienden a fracturarse.

Código 2: Probabilidades de Rotura y Unión

```
1 def rotura(x, c, d):
2     return 1 / (1 + exp((c - x) / d))
3
4 def union(x, c):
5     return exp(-x / c)
6
7 c = np.median(cumulos)
8 d = np.std(cumulos) / 4
```

En seguida, en el código 3 se definen las funciones `romperse()` y `unirse()`. La función `romperse()` decide qué cúmulos se romperán en el siguiente paso y qué tamaños tendrán los dos cúmulos resultantes de la rotura, mientras que la función `unirse()` crea dos grupos de cúmulos para posteriormente decidir cuáles se unirán con cuáles en el próximo paso de la iteración. Estas funciones dependen respectivamente de las probabilidades de rotura y unión descritas en el código 2.

Código 3: Acciones de Rotura y Unión de Cúmulos

```

1 def romperse(tam, cuantos):
2     if tam == 1:
3         return [tam] * cuantos
4     res = []
5     for cumulo in range(cuantos):
6         if random() < rotura(tam, c, d):
7             primera = randint(1, tam - 1)
8             segunda = tam - primera
9             assert primera > 0
10            assert segunda > 0
11            assert primera + segunda == tam
12            res += [primera, segunda]
13        else:
14            res.append(tam)
15    assert sum(res) == tam * cuantos
16    return res
17
18 def unirse(tam, cuantos):
19     res = []
20     for cumulo in range(cuantos):
21         if random() < union(tam, c):
22             res.append(-tam)
23         else:
24             res.append(tam)
25     return res

```

El siguiente paso, observado en el código 4, es comenzar la iteración entre las cantidades iniciales de cúmulos y las cantidades de partículas. También se comienzan las 100 repeticiones de cada combinación. Para cada repetición se crea nuevamente una cantidad  $k$  de cúmulos, de tal manera que la distribución de tamaños siguen una distribución normal, no existen cúmulos vacíos, en total suman a la cantidad  $n$  de partículas y todos los tamaños son números enteros.

Código 4: Inicio de Iteraciones y Creación de Cúmulos

```

1 for k in clusters:
2     resultados = pd.DataFrame()
3     for n in particles:
4         filtrados = []
5         for r in range(runs):
6             filtro = 0
7             orig = np.random.normal(size = k)
8             cumulos = orig - min(orig)
9             cumulos += 1
10            cumulos = cumulos / sum(cumulos)
11            cumulos *= n
12            cumulos = np.round(cumulos).astype(int)
13            diferencia = n - sum(cumulos)
14            cambio = 1 if diferencia > 0 else -1
15            while diferencia != 0:
16                p = randint(0, k - 1)
17                if cambio > 0 or (cambio < 0 and cumulos[p] > 0):
18                    cumulos[p] += cambio
19                    diferencia -= cambio
20            assert all(cumulos != 0)
21            assert sum(cumulos) == n

```

En el código 5, se inician los 50 pasos de la iteración, en donde los cúmulos son agrupados por tamaños y la cantidad de cúmulos que existen con esos tamaños. Es aquí donde se comienzan a fracturar y unir los cúmulos con las funciones definidas en el código 3. Para cada paso de la iteración, primero se decide qué cúmulos van a romperse y se rompen; después, se decide qué cumulos van a unirse y se unen de manera aleatoria unos con otros. Estos pasos crean una nueva distribución de cantidades y tamaños de cúmulos para cada paso de la iteración.

Código 5: Fractura y Unión de Cúmulos en la Iteración

```

1  for paso in range(dur):
2      assert sum(cumulos) == n
3      assert all([c > 0 for c in cumulos])
4      (tams, freqs) = np.unique(cumulos, return_counts = True)
5      cumulos = []
6      assert len(tams) == len(freqs)
7      for i in range(len(tams)):
8          cumulos += romperse(tams[i], freqs[i])
9      assert sum(cumulos) == n
10     assert all([c > 0 for c in cumulos])
11     (tams, freqs) = np.unique(cumulos, return_counts = True)
12     cumulos = []
13     assert len(tams) == len(freqs)
14     for i in range(len(tams)):
15         cumulos += unirse(tams[i], freqs[i])
16     cumulos = np.asarray(cumulos)
17     neg = cumulos < 0
18     a = len(cumulos)
19     juntarse = -1 * np.extract(neg, cumulos)
20     cumulos = np.extract(~neg, cumulos).tolist()
21     assert a == len(juntarse) + len(cumulos)
22     nt = len(juntarse)
23     if nt > 1:
24         shuffle(juntarse)
25     j = juntarse.tolist()
26     while len(j) > 1:
27         cumulos.append(j.pop(0) + j.pop(0))
28     if len(j) > 0:
29         cumulos.append(j.pop(0))
30     assert len(j) == 0
31     assert sum(cumulos) == n
32     assert all([c != 0 for c in cumulos])

```

Por último, con el código 6, al final de cada iteración se decide la cantidad de cúmulos que son lo suficientemente grandes para quedar en el filtro y se calcula un porcentaje de partículas filtradas. Estos porcentajes se guardan en archivos que se utilizan para su posterior análisis.

Código 6: Cálculo de Porcentajes de Partículas Filtradas

```

1  for p in cumulos:
2      if p >= filtersize:
3          filtro += 1
4      porcentaje = (filtro / sum(cumulos)) * 100
5      filtrados.append(porcentaje)
6
7  resultados['n: ' + str(n) + ', k: ' + str(k)] = pd.DataFrame(filtrados)
8
9  resultados.to_csv('urnas_{:d}.csv'.format(k))

```

### 3. Resultados

A manera de resultados, primero se muestra una visualización de viñetas donde se observa el comportamiento a través del tiempo de una iteración de fracturas y uniones de cúmulos por medio de histogramas. En segunda instancia, se muestran diagramas tipo violín donde se relacionan la tasa de cantidad de partículas a cantidad de cúmulos iniciales, con el porcentaje de partículas filtradas. Además, se explica el análisis estadístico realizado para determinar si existe una diferencia estadísticamente significativa al variar la cantidad de cúmulos iniciales.

#### 3.1. Comportamiento del Sistema a Través del Tiempo

La figura 1 muestra el comportamiento que tiene una iteración de la simulación a través del tiempo. Los parámetros de dicha iteración son una cantidad de partículas,  $n = 250000$  y una cantidad inicial de cúmulos,  $k = 2500$ . Una visualización animada de la misma iteración se puede encontrar en el archivo [GIF](#) del repositorio de J. Torres. Como se puede observar, el sistema tiende tener más cúmulos de tamaños cercanos al tamaño crítico, lo cual es de esperarse, ya que estarían oscilando entre unirse y fracturarse.

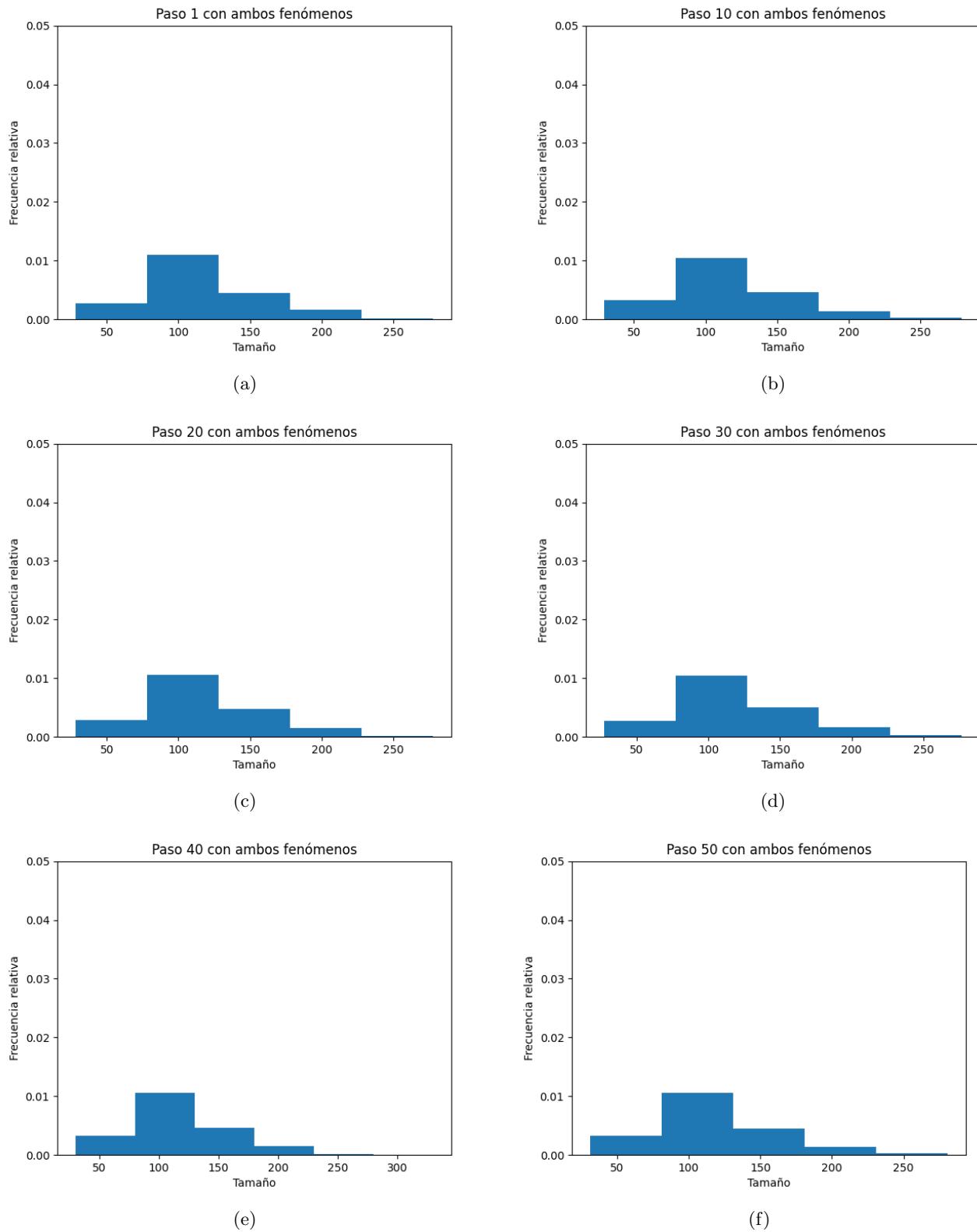


Figura 1: Evolución del sistema a través del tiempo

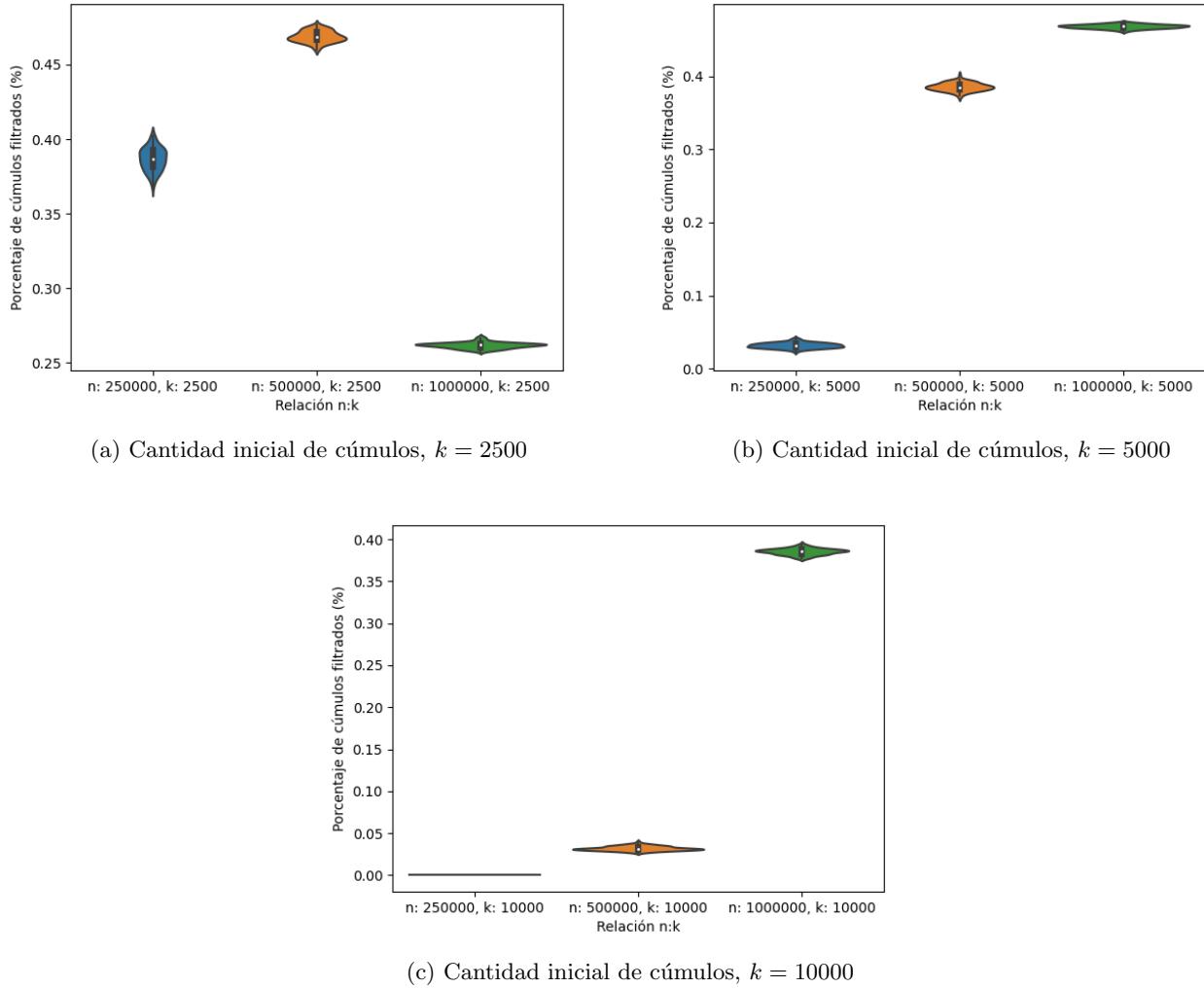


Figura 2: Distribución de porcentajes de partículas filtradas para cada tasa  $n/k$ .

### 3.2. Análisis Estadístico

La figura 2 muestra los diagramas tipo violín de las nueve combinaciones entre cantidad de partículas y cantidad inicial de cúmulos, donde se observan las distribuciones del porcentaje de partículas filtradas para las 100 iteraciones de cada combinación. En la figura 2a se observan los porcentajes para la variación de cantidad de partículas con una  $k = 2500$ , para la figura 2b la cantidad inicial de cúmulos es  $k = 5000$ , y para la figura 2c es de  $k = 10000$ .

Además de los diagramas tipo violín, se realizó un análisis de varianza para determinar si existe una diferencia significativa al variar la cantidad de partículas con las que trabaja el sistema. Del análisis se encontró que los porcentajes de partículas filtradas no pertenecen a la misma distribución al variar la cantidad de partículas, ya que los valores  $p$  son mucho menores al valor de 0,05 que es necesario para deducir lo contrario. Cabe mencionar que no se realizó un análisis para demostrar si los porcentajes pertenecen a la misma distribución al variar la cantidad inicial de cúmulos formados. Sin embargo, al observar los diagramas se puede apreciar una diferencia considerable en las distribuciones, lo cual podría indicar que también existe una relación entre el porcentaje de partículas filtradas y la cantidad inicial de cúmulos.

## 4. Conclusiones

En la práctica se realizó la variación de la tasa entre cantidad de partículas y cantidad inicial de cúmulos,  $n/k$ , para determinar si existe una relación entre ella y el porcentaje de partículas filtradas. Del análisis estadístico se

observa que hay una diferencia significativa al variar la cantidad de partículas, por lo que es muy probable que exista una relación con la misma. Aunque no se realizó el análisis para determinar si también hay relación con la cantidad inicial de cúmulos, es fácil deducir que sí la hay al observar la gran diferencia de distribuciones de los diagramas tipo violín.

## Referencias

- [1] E. Schaeffer. Urnmodel, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/UrnModel>.
- [2] J. Torres. P8, 2022. URL <https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P8>.

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 distribuidas con coordenadas  $(x, y)$  en un plano, además se les asigna una carga  $c$  y una masa  $m$  distribuidas 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

```
1 n = 25
2 x = np.random.normal(size = n)
3 y = np.random.normal(size = n)
4 c = np.random.normal(size = n)
5 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

```
1 xmax = max(x)
2 xmin = min(x)
3 x = (x - xmin) / (xmax - xmin)
4 ymax = max(y)
5 ymin = min(y)
6 y = (y - ymin) / (ymax - ymin)
7 cmax = max(c)
8 cmin = min(c)
9 c = 2 * (c - cmin) / (cmax - cmin) - 1
10 mmax = max(m)
11 mmin = min(m)
12 m = 10 * ((m - mmin) / (mmax - mmin) + 0.1)
13 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

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

```

3 p = pd.DataFrame({'x': x, 'y': y, 'c': c, 'm': m, 'g': g, 'v': v})
4 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

```

1 paso = 256 // 10
2 niveles = [i/256 for i in range(0, 256, paso)]
3 colores = [(niveles[i], 0, niveles[-(i + 1)]) for i in range(len(niveles))]
4 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

```

1 eps = 0.001
2 def fuerza(i, shared):
3     p = shared.data
4     n = shared.count
5     pi = p.iloc[i]
6     xi = pi.x
7     yi = pi.y
8     ci = pi.c
9     fx, fy = 0, 0
10    for j in range(n):
11        pj = p.iloc[j]
12        cj = pj.c
13        dire = (-1)**(1 + (ci * cj < 0))
14        dx = xi - pj.x
15        dy = yi - pj.y
16        factor = dire * abs(ci - cj) / (sqrt(dx**2 + dy**2) + eps)
17        fx -= dx * factor
18        fy -= dy * factor
19    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.

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

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

```

1 eps = 0.001
2 G = 0.025
3 def fuerza(i, shared):
4     p = shared.data

```

```

5  n = shared.count
6  pi = p.iloc[i]
7  xi = pi.x
8  yi = pi.y
9  mi = pi.m
10 fx, fy = 0, 0
11 for j in range(n):
12     pj = p.iloc[j]
13     mj = pj.m
14     dx = xi - pj.x
15     dy = yi - pj.y
16     factor = G * ((mi * mj) / (sqrt(dx**2 + dy**2) + eps))
17     fx -= dx * factor
18     fy -= dy * factor
19 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

```

1 eps = 0.001
2 G = 0.025
3 def fuerza(i, shared):
4     p = shared.data
5     n = shared.count
6     pi = p.iloc[i]
7     xi = pi.x
8     yi = pi.y
9     ci = pi.c
10    mi = pi.m
11    fx1, fy1 = 0, 0
12    fx2, fy2 = 0, 0
13    for j in range(n):
14        pj = p.iloc[j]
15        cj = pj.c
16        mj = pj.m
17        dire = (-1)**(1 + (ci * cj < 0))
18        dx = xi - pj.x
19        dy = yi - pj.y
20        factor = dire * fabs(ci - cj) / (sqrt(dx**2 + dy**2) + eps)
21        factor1 = G * ((mi * mj) / (sqrt(dx**2 + dy**2) + eps))
22        fx1 = fx1 - dx * factor
23        fy1 = fy1 - dy * factor
24        fx2 = fx2 - dx * factor1
25        fy2 = fy2 - dy * factor1
26
27    fx = fx1 + fx2
28    fy = fy1 + fy2
29    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

```

1 def actualiza(pos, fuerza, de):
2     return max(min(pos + de * fuerza, 1), 0)
3
4 def velocidad(p, pa):
5     ppa = pa.data
6     n = pa.count
7     for i in range(n):
8         p1 = p.iloc[i]
9         p2 = ppa.iloc[i]
10        x1 = p1.x
11        x2 = p2.x
12        y1 = p1.y
13        y2 = p2.y
14        va = p2.v
15        v = []
16        v.extend(va)
17        vel = (sqrt(((x2 - x1)**2) + ((y2 - y1)**2)))
18        v.append(vel)
19        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 paralleliza, 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

```

1 if __name__ == "__main__":
2     n = 25
3     p = pd.read_csv('values.csv')
4     x = p['x']
5     y = p['y']
6     g = p['g']
7     m = p['m']
8     c = p['c']
9     p['v'] = [[0]]*n
10    mgr = multiprocessing.Manager()
11    ns = mgr.Namespace()
12    ns.data = p
13    ns.count = n
14    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

```

1 for t in range(tmax):
2     with multiprocessing.Pool() as pool:
3         f = pool.starmap(fuerza, [(i, ns) for i in range(n)])
4         delta = 0.02 / max([max(fabs(fx), fabs(fy)) for (fx, fy) in f])
5         p['x'] = pool.starmap(actualiza, zip(p.x, [v[0] for v in f], repeat(delta)))
6         p['y'] = pool.starmap(actualiza, zip(p.y, [v[1] for v in f], repeat(delta)))
7         velocidad(p, ns)
8         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.

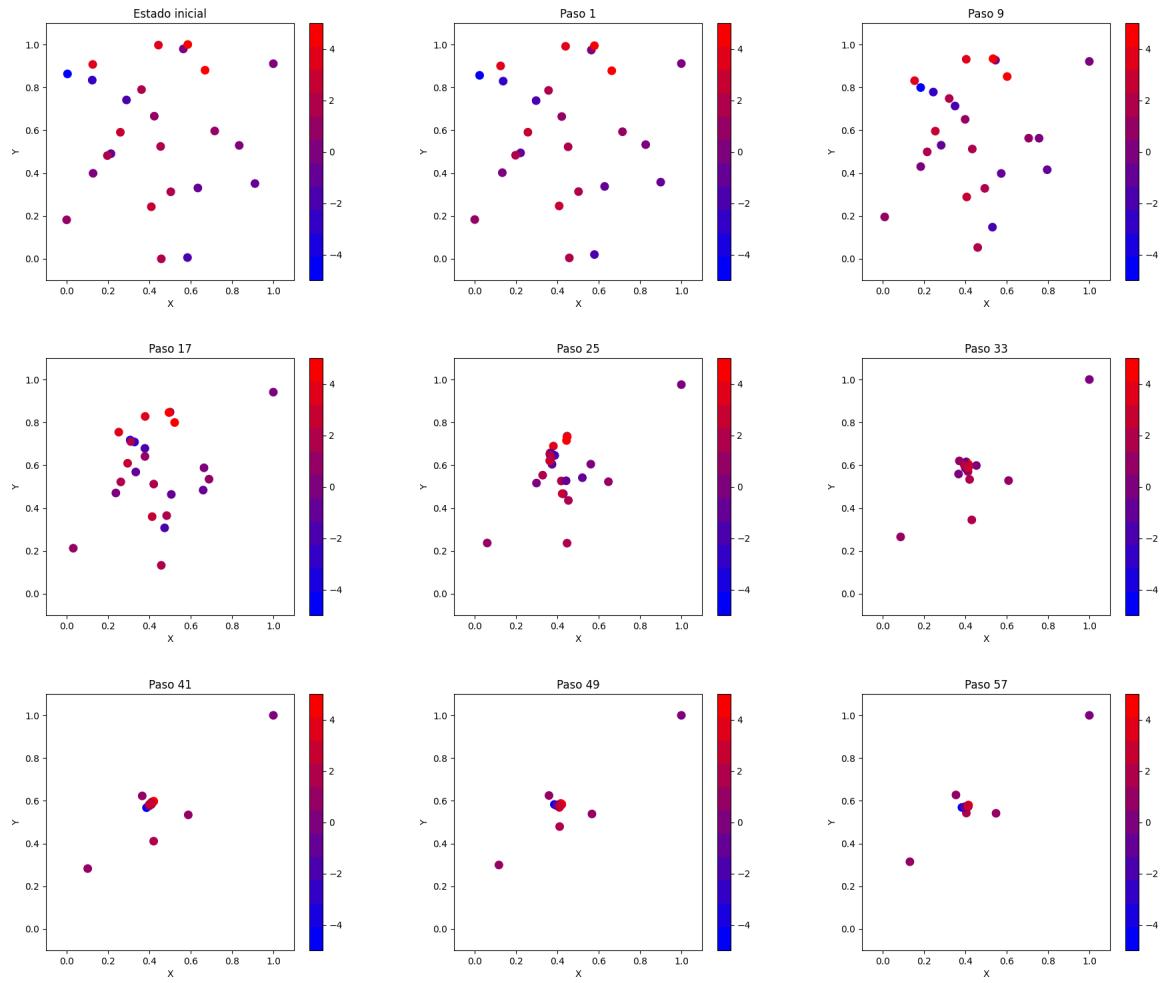


Figura 1.1: Movimiento de partículas con carga.

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.

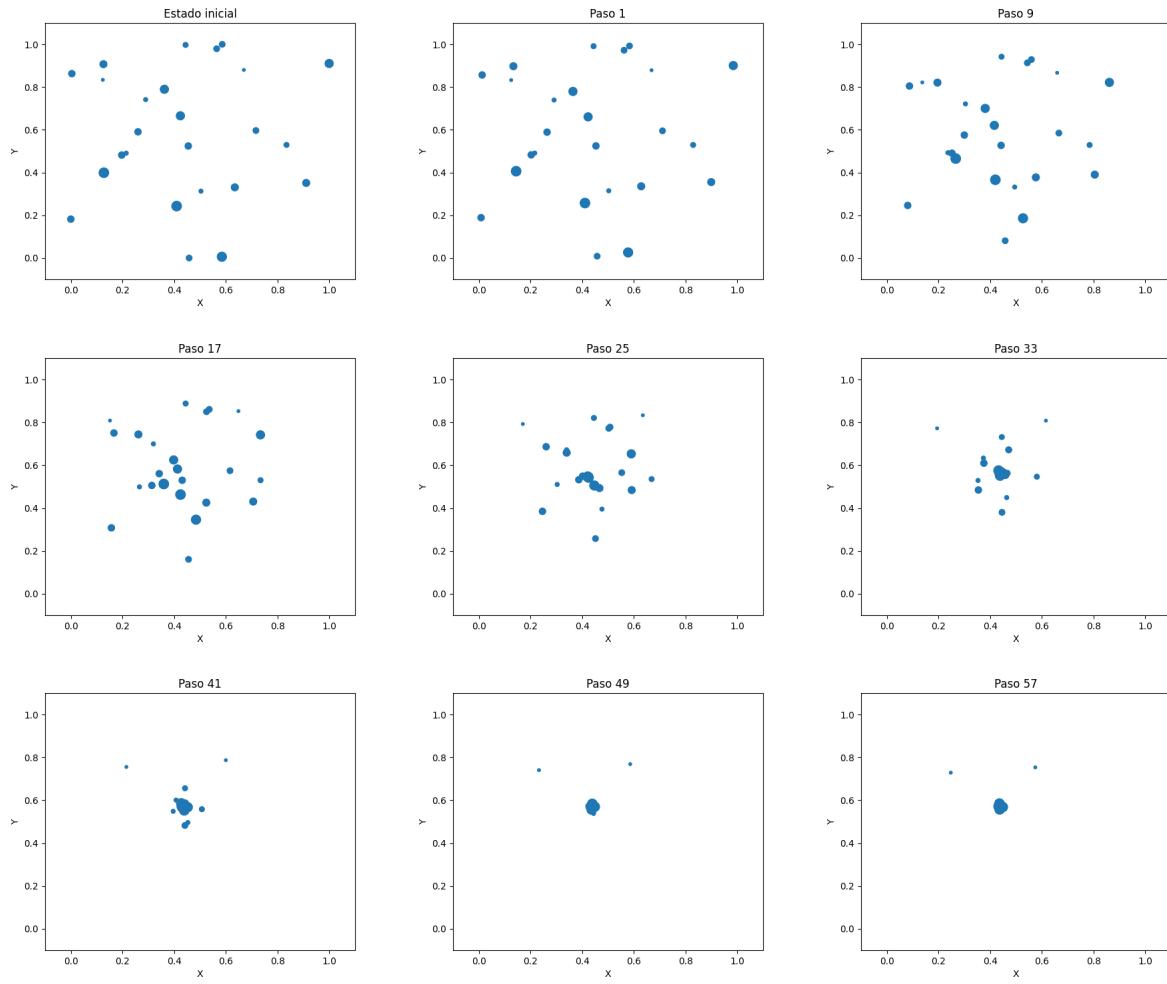


Figura 1.2: Movimiento de partículas con masa.

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.

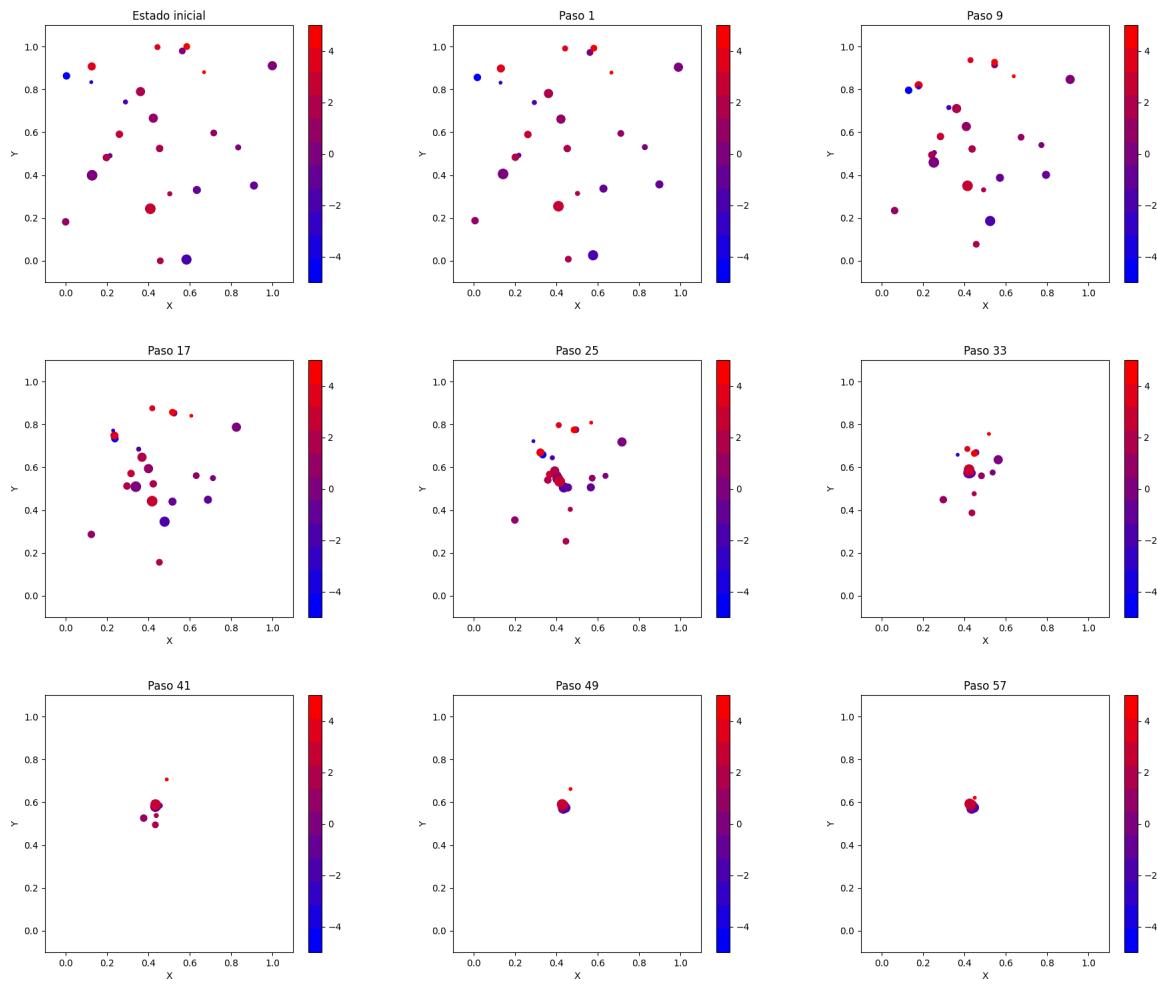
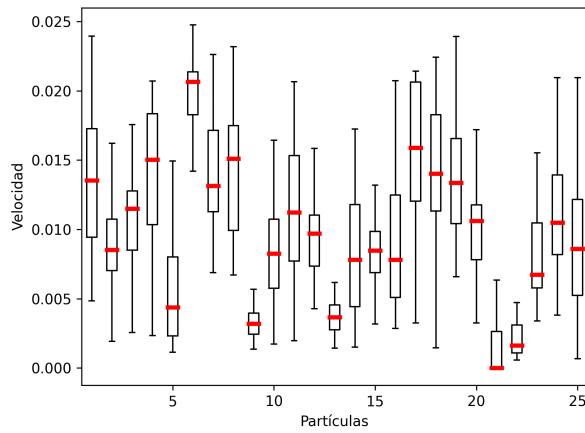
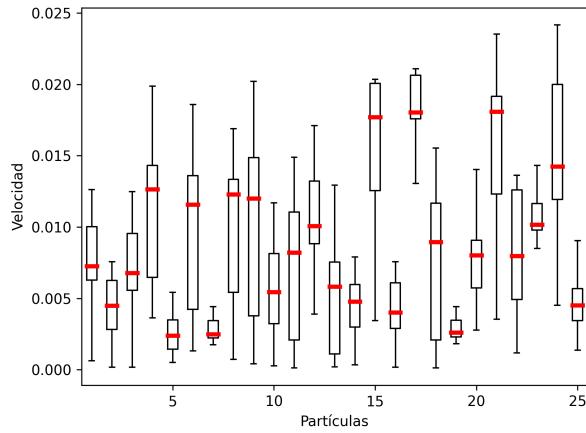


Figura 1.3: Movimiento de partículas con carga y masa.

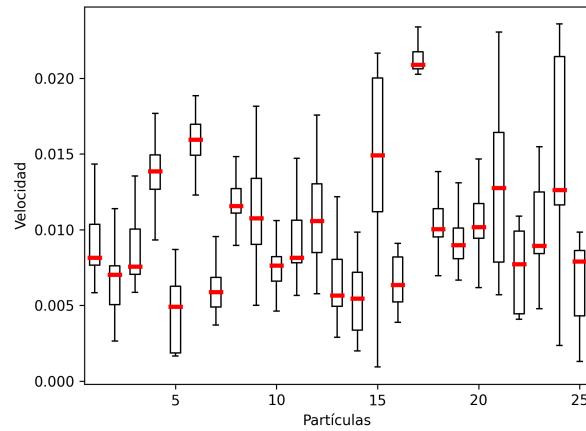
En la figura 1.4 se pueden observar las distribuciones de las velocidades de cada partícula para cada iteración del experimento. En la figura 1.4a, las partículas se mueven debido a las fuerzas de atracción y repulsión, la figura 1.4b muestra las velocidades de las partículas al actuar sobre ellas la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

```
1 m = 7
2 vx = (2 * (np.random.uniform(size = m) < 0.5) - 1) * np.random.uniform(low = 0.01, high = 0.04,
   size = m)
3 vy = (2 * (np.random.uniform(size = m) < 0.5) - 1) * np.random.uniform(low = 0.01, high = 0.04,
   size = m)
4 x = np.random.uniform(size = m)
5 y = np.random.uniform(size = m)
6 r = np.random.uniform(low = 0.05, high = 0.1, size = m)
7 bolas = pd.DataFrame({'x': x, 'y': y, 'dx': vx, 'dy': vy, 'r': r})
8 n = 50
9 vx = (2 * (np.random.uniform(size = n) < 0.5) - 1) * np.random.uniform(low = 0.02, high = 0.05,
   size = n)
10 vy = (2 * (np.random.uniform(size = n) < 0.5) - 1) * np.random.uniform(low = 0.02, high = 0.05,
   size = n)
11 x = np.random.uniform(size = n)
12 y = np.random.uniform(size = n)
13 print(x)
14 r = np.random.uniform(low = 0.01, high = 0.03, size = n)
15 particulas = pd.DataFrame({'x': x, 'y': y, 'dx': vx, 'dy': vy, 'r': r,\n                           '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 sobreapadas.

Código 2.2: Verificación de Sobreaplicamiento de Partículas y Bolas

```
1 for t in range(25):
2
3     for i in range(n):
4         p = particulas.iloc[i]
5         v = p.v
6
7         if v > 0:
8             pr = p.r
9             px = p.x
10            py = p.y
```

```

11     conteo = 0
12     for k in range(m):
13         pk = bolas.iloc[k]
14         pkr = pk.r
15         pkx = pk.x
16         pky = pk.y
17         dx = px - pkx
18         dy = py - pky
19         dr = pkr + pr
20         d = sqrt(dx**2 + dy**2)
21         if d < dr:
22             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

```

1  if conteo >= 2:
2      conteo2 = 0
3      for j in range(n):
4          if i != j:
5              pj = particulas.iloc[j]
6              pjr = pj.r
7              pjx = pj.x
8              pjy = pj.y
9              dx = px - pjx
10             dy = py - pjy
11             dr = pjr + pr
12             d = sqrt(dx**2 + dy**2)
13             if d < dr:
14                 conteo2 += 1
15                 a1 = np.pi * (pr**2)
16                 a2 = np.pi * (pjr**2)
17                 a = a1 + a2
18                 rt = sqrt(a/np.pi)
19                 particulas.at[i, 'r'] = rt
20                 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

```

1  if conteo2 == 0:
2      v = random()
3      v1 = 1 - v
4      vx1 = p.dx * -1
5      vy1 = p.dy * -1
6      a = np.pi * (pr**2)
7      a1 = a * v
8      a2 = a * v1
9      r1 = sqrt(a1/np.pi)
10     r2 = sqrt(a2/np.pi)
11     particulas.at[i, 'r'] = r1
12     particulas = particulas.append({'x': px, 'y': py, 'dx': vx1, 'dy': vy1,\n13                                     '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: Eliminación de Partículas Vacías y Guardado de Datos

```

1  particulas = particulas.loc[particulas['v'] > 0]
2  n = particulas.shape[0]
3  particulas.to_csv('p_part_{:d}.dat'.format(t), header = False, index = False)
4  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

```

1  for i in range(m):
2      b = bolas.iloc[i]
3      br = b.r
4      bx = b.x
5      by = b.y
6      vx = b.dx
7      vy = b.dy
8      x = bx + vx
9      y = by + vy
10     if 0 >= (x-br):
11         x = 0 + br
12         bolas.at[i, 'dx'] = vx * -1
13     elif (x+br) >= 1:
14         x = 1 - br
15         bolas.at[i, 'dx'] = vx * -1
16     if 0 >= (y-br):
17         y = 0 + br
18         bolas.at[i, 'dy'] = vy * -1
19     elif (y+br) >= 1:
20         y = 1 - br
21         bolas.at[i, 'dy'] = vy * -1
22
23     bolas.at[i, 'x'] = x
24     bolas.at[i, 'y'] = y

```

Código 2.7: Rebote de las Partículas en los Límites

```

1  for i in range(n):
2      b = particulas.iloc[i]
3      br = b.r
4      bx = b.x
5      by = b.y
6      vx = b.dx
7      vy = b.dy
8      x = bx + vx
9      y = by + vy
10     if 0 >= (x-br):
11         x = 0 + br
12         particulas.at[i, 'dx'] = vx * -1
13     elif (x+br) >= 1:
14         x = 1 - br
15         particulas.at[i, 'dx'] = vx * -1
16     if 0 >= (y-br):
17         y = 0 + br
18         particulas.at[i, 'dy'] = vy * -1
19     elif (y+br) >= 1:
20         y = 1 - br
21         particulas.at[i, 'dy'] = vy * -1
22
23     particulas.at[i, 'x'] = x
24     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

```

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

```

```
10 plot 'p_bola_'.i.'.dat' u 1:2:5 w circles lc rgb "blue" fs solid noborder, \
11      'p_part_'.i.'.dat' u 1:2:5 w circles lc rgb "red" fs solid noborder
12 }
```

## 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>.

# Reporte 10:

## Algoritmo Genético

Jorge Torres

1 de mayo de 2022

## 1. Objetivo

El problema de la mochila (inglés: knapsack) es un problema clásico de optimización, particularmente de programación entera, donde la tarea consiste en seleccionar un subconjunto de objetos de tal forma que (i) no se exceda la capacidad de la mochila en términos de la suma de los pesos de los objetos incluidos, y que (ii) el valor total de los objetos incluidos sea lo máximo posible. Este problema es pseudo-polinomial, ya que existe un [algoritmo](#) de tabulación que determina la combinación óptima.

Un algoritmo genético representa posibles soluciones a un problema en términos de un genoma, que en este caso va a ser un vector de verdades y falsos, indicando cuáles objetos se van a incluir en la mochila (TRUE o 1 significa que llevamos el objeto, FALSE o 0 que lo descartamos de la selección).

El objetivo de esta actividad consiste en generar tres instancias con tres distintas reglas:

- Instancia 1: el peso y el valor de cada objeto se generan independientemente con una distribución uniforme.
- Instancia 2: el valor de cada objeto se genera independientemente con una distribución exponencial y su peso es inversamente correlacionado con el valor.
- Instancia 3: el peso de cada objeto se genera independientemente con una distribución normal y su valor es (positivamente) correlacionado con el cuadrado del peso, con un ruido normalmente distribuido de baja magnitud.

## 2. Desarrollo

El desarrollo de la actividad está basado en el [código](#) implementado por E. Schaeffer, en el que primero calcula el óptimo teórico del problema de la mochila y después lo compara con los resultados del algoritmo genético [1]. La función `knapsack()` del código 1 calcula el óptimo teórico del problema pseudo-polinomial. El desarrollo completo puede encontrarse en el repositorio en GitHub de J. Torres [2].

Código 1: Solución Óptima

```
1 def knapsack(peso_permitido, pesos, valores):
2     assert len(pesos) == len(valores)
3     peso_total = sum(pesos)
4     valor_total = sum(valores)
5     if peso_total < peso_permitido:
6         return valor_total
7     else:
8         V = dict()
9         for w in range(peso_permitido + 1):
10             V[(w, 0)] = 0
11         for i in range(len(pesos)):
12             peso = pesos[i]
13             valor = valores[i]
14             for w in range(peso_permitido + 1):
15                 cand = V.get((w - peso, i), -float('inf')) + valor
16                 V[(w, i + 1)] = max(V[(w, i)], cand)
17     return max(V.values())
```

Las funciones `factible()` y `objetivo()` del código 2 determinan si el peso de los objetos seleccionados es menor al límite y el valor total de dichos objetos, respectivamente.

Código 2: Peso y Valor Totales de la Selección

```

1 def factible(seleccion, pesos, capacidad):
2     return np.inner(seleccion, pesos) <= capacidad
3
4 def objetivo(seleccion, valores):
5     return np.inner(seleccion, valores)

```

En el código 3 se tienen cuatro funciones. La función `normalizar()` arregla una distribución entre 0 y 1 de un conjunto de datos, que se utiliza para obtener valores aleatorios con esta distribución en subsecuentes funciones. La función `poblacion_inicial()` crea un arreglo bidimensional de  $n$  columnas (objetos) y  $tam$  filas (individuos) con  $n$  valores 0 ó 1 aleatoria e uniformemente distribuidos para cada individuo. La función `mutacion()` selecciona objetos al azar de los individuos y, si el valor actual es 0, lo cambia a 1, creando un individuo diferente. La función `reproduccion()` toma dos individuos, selecciona una posición al azar en sus vectores de objetos, y la utiliza como posición de corte. Después realiza un cruzamiento entre los vectores cortados, creando un nuevo individuo que tiene información de los individuos originales.

Código 3: Funciones Normalizar, Población Inicial, Mutación y Reproducción

```

1 def normalizar(data):
2     menor = min(data)
3     mayor = max(data)
4     rango = mayor - menor
5     data = data - menor # > 0
6     return data / rango # entre 0 y 1
7
8 def poblacion_inicial(n, tam):
9     pobl = np.zeros((tam, n))
10    for i in range(tam):
11        pobl[i] = (np.round(np.random.uniform(size = n))).astype(int)
12    return pobl
13
14 def mutacion(sol, n):
15    pos = randint(0, n - 1)
16    mut = np.copy(sol)
17    mut[pos] = 1 if sol[pos] == 0 else 0
18    return mut
19
20 def reproduccion(x, y, n):
21    pos = randint(2, n - 2)
22    xy = np.concatenate([x[:pos], y[pos:]])
23    yx = np.concatenate([y[:pos], x[pos:]])
24    return (xy, yx)

```

Para realizar observaciones basadas en las instancias mencionadas en la sección 1, se incluyen un total de 6 generadores de pesos y valores aleatorios, cuyos códigos se discuten en las siguientes secciones.

## 2.1. Generadores - Instancia 1

En el código 4, con las funciones `generador_pesos()` y `generador_valores()` se generan independientemente una cantidad  $n$  de pesos y valores de manera aleatoria y uniformemente distribuidos.

Código 4: Generadores de Pesos y Valores de Instancia 1

```

1 def generador_pesos(cuantos, low, high):
2     return np.round(normalizar(np.random.uniform(low=low, high=high, size = cuantos)) * (high -
3     low) + low)
4
5 def generador_valores(pesos, low, high):
6     return np.round(normalizar(np.random.uniform(low = low, high = high, size = pesos)) * (high -
7     low) + low)

```

## 2.2. Generadores - Instancia 2

La función `generador_valores2()` del código 5 crea aleatoriamente una cantidad  $n$  de valores con una distribución exponencial, mientras que la función `generador_pesos2()` obtiene la misma cantidad de pesos cuyo valor numérico está correlacionado con el inverso de los valores.

Código 5: Generadores de Pesos y Valores de Instancia 2

```

1 def generador_pesos2(valores, low, high):
2     cant = 1 / valores
3     return np.round(((normalizar(cant))) * (high - low) + low)
4
5 def generador_valores2(pesos, low, high):
6     cant = np.arange(0, pesos)
7     return np.round(normalizar(expon.pdf(cant)) * (high - low) + low)

```

### 2.3. Generadores - Instancia 3

En la tercera instancia, se necesitan obtener pesos independientemente con una distribución normal, mientras que los valores están positivamente correlacionados con el cuadrado de los pesos, lo cual se implementa con las funciones `generador_pesos3()` y `generador_valores3()` del código 6.

Código 6: Generadores de Pesos y Valores de Instancia 3

```

1 def generador_pesos3(cuantos, low, high):
2     return np.round(normalizar(np.random.normal(size = cuantos)) * (high - low) + low)
3
4 def generador_valores3(pesos, low, high):
5     return np.round((pesos**2) * (high - low) + low)

```

Para las tres instancias se crean  $n = 100$  pesos y valores, el algoritmo se itera en 150 pasos y se realizan 20 repeticiones de cada instancia. Sin embargo, se hacen dos combinaciones distintas variando la probabilidad de mutación, la cantidad de reproducciones o cruzamientos, y la población inicial de individuos. La primera combinación consiste en una probabilidad de mutación  $pm = 0,05$ , una cantidad de cruzamientos  $rep = 50$  y una población inicial  $init = 100$ . La segunda combinación consiste en  $pm = 0,1$ ,  $rep = 100$  e  $init = 50$ . Estos parámetros se pueden observar en el código 7.

Código 7: Parámetros Iniciales

```

1 n = 100
2 tmax = 150
3 iteraciones = 20
4
5 ##### Combinacion 1#####
6
7 pm, rep, init = 0.05, 50, 100
8
9 ##### Combinacion 2#####
10
11 pm, rep, init = 0.1, 100, 50

```

Al iniciar las iteraciones, se generan los pesos y valores, se define la capacidad máxima de la mochila, se calcula el valor óptimo con la función `knapsack()` y se crea la población inicial de individuos, como se observa en el código 8.

Código 8: Inicio de Iteraciones

```

1 for runs in range(iteraciones):
2     pesos = generador_pesos(n, 15, 80)
3     valores = generador_valores(n, 10, 500)
4     capacidad = int(round(sum(pesos) * 0.65))
5     optimo = knapsack(capacidad, pesos, valores)
6     p = poblacion_inicial(n, init)
7     tam = p.shape[0]
8     assert tam == init
9     mejor = None
10    mejores = []

```

En el código 9, se inician las mutaciones y reproducciones de los individuos, creando nuevos individuos que se agregan a la lista total. De esta lista se toman los valores de factibilidad y objetivo obtenidos con las funciones del código 2 y se ordenan los individuos en orden descendente de factibilidad.

Código 9: Mutaciones, Reproducciones y Ordenamiento de Factibilidad

```

1   for t in range(tmax):
2       for i in range(tam): # mutarse con probabilidad pm
3           if random() < pm:
4               p = np.vstack([p, mutacion(p[i], n)])
5       for i in range(rep): # reproducciones
6           padres = sample(range(tam), 2)
7           hijos = reproduccion(p[padres[0]], p[padres[1]], n)
8           p = np.vstack([p, hijos[0], hijos[1]])
9       tam = p.shape[0]
10      d = []
11      for i in range(tam):
12          d.append({'idx': i, 'obj': objetivo(p[i], valores),
13                     'fact': factible(p[i], pesos, capacidad)})
14  d = pd.DataFrame(d).sort_values(by = ['fact', 'obj'], ascending = False)

```

Por último, se toma una cantidad igual a la población inicial de los individuos más factibles y se elimina el resto. Para cada iteración, se agrega a una lista el mejor valor obtenido al final de la iteración y se calcula una proporción comparada con el valor óptimo de acuerdo a la ecuación 1 y se implementa en el código 10,

$$P = \frac{(O - M)}{O} \quad (1)$$

donde  $P$  es la proporción,  $O$  es el valor óptimo obtenido de la función `knapsack()` y  $M$  es el mejor valor obtenido al final de la iteración.

Código 10: Eliminación de Individuos Menos Factibles y Cálculo de Proporción

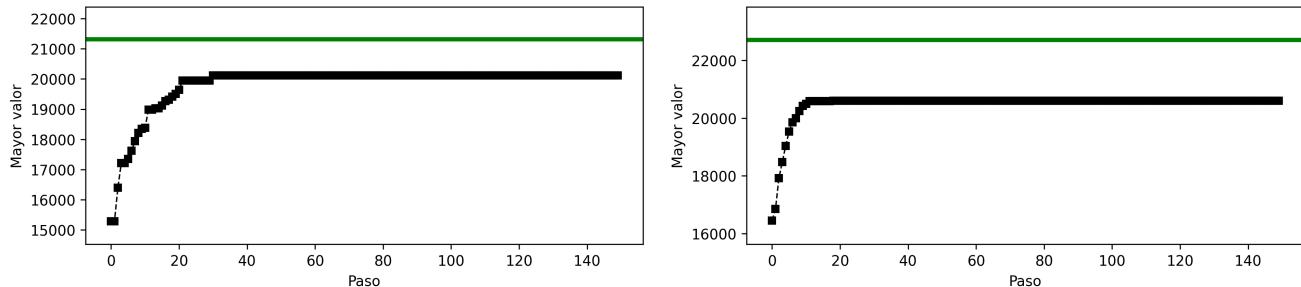
```

1 mantener = np.array(d.idx[:init])
2 p = p[mantener, :]
3 tam = p.shape[0]
4 assert tam == init
5 factibles = d.loc[d факт == True, ]
6 mejor = max(factibles.obj)
7 mejores.append(mejor)
8 resultados1.append((optimo - mejor) / optimo)

```

### 3. Resultados

En la figura 1 se puede ver una de las iteraciones del algoritmo genético para la primera instancia. La figura 1a muestra el desarrollo del algoritmo para la combinación 1, mientras que la figura 1b lo muestra para la combinación 2. Se puede observar cómo para la combinación 1 hay una mejora gradual de los valores hasta que alcanza un máximo y se estanca. Para la combinación 2, la mejora es un poco más rápida, pero la separación entre el valor óptimo y el valor donde se estanca es mayor.

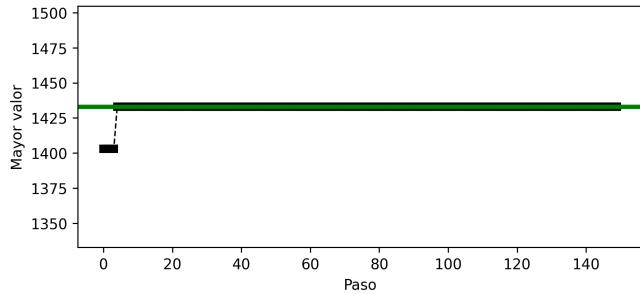


(a) Desarrollo del algoritmo para la combinación 1.

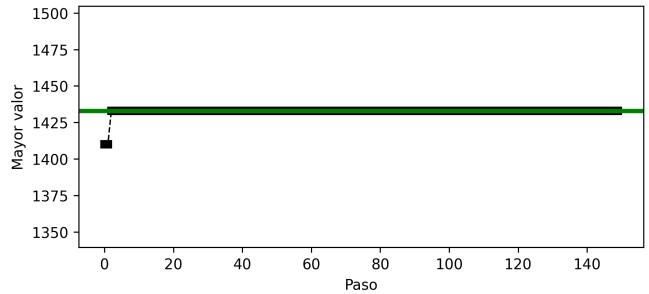
(b) Desarrollo del algoritmo para la combinación 2.

Figura 1: Ejemplo de iteración para la Instancia 1.

La figura 2 muestra una iteración de cada combinación para la segunda instancia. Para ambos casos se puede observar cómo no solamente existe una mejora casi inmediata, sino que el valor máximo logra alcanzar el valor óptimo y quedarse sobre él. Sin embargo, se puede observar que esta instancia trabaja con valores más bajos.



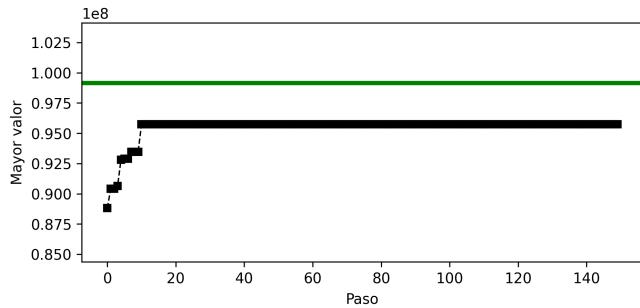
(a) Desarrollo del algoritmo para la combinación 1.



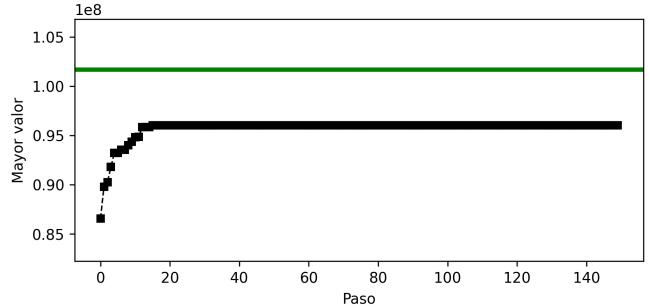
(b) Desarrollo del algoritmo para la combinación 2.

Figura 2: Ejemplo de iteración para la Instancia 2.

En la figura 3 se ven representados ejemplos de las dos combinaciones para la tercera instancia. Esta instancia puede trabajar con valores mucho mayores que las dos anteriores, en el rango de centenas de millones. Contrario al caso de la instancia 1, se puede ver cómo la primera combinación (figura 3a), presenta mejoras en ciertos intervalos y el valor máximo se aproxima más al óptimo, llegando también a estancarse casi al principio de la iteración. Por el otro lado, la combinación 2 (figura 3b), presenta un desarrollo más gradual y la diferencia entre el valor óptimo y el máximo es más grande.

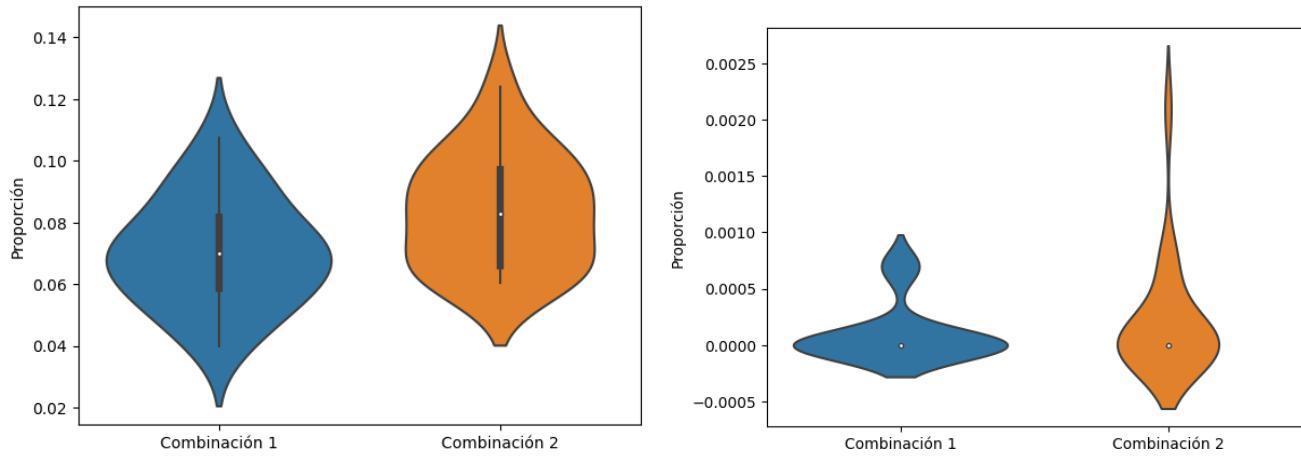


(a) Desarrollo del algoritmo para la combinación 1.

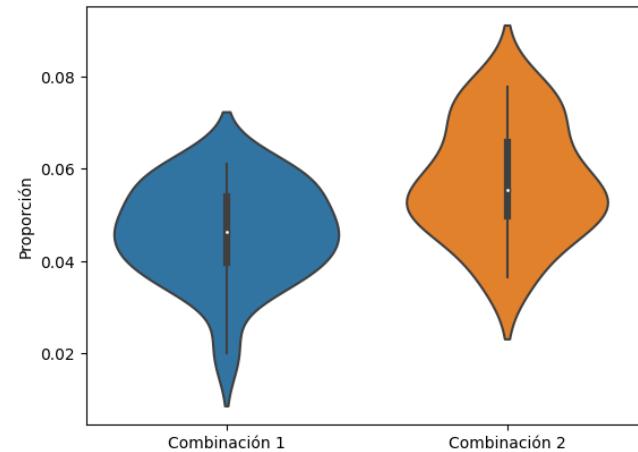


(b) Desarrollo del algoritmo para la combinación 2.

Figura 3: Ejemplo de iteración para la Instancia 3.



(a) Distribución de ambas combinaciones para la instancia 1. (b) Distribución de ambas combinaciones para la instancia 2.



(c) Distribución de ambas combinaciones para la instancia 3.

Figura 4: Distribuciones de las mejores proporciones alcanzadas para cada instancia y cada combinación.

Para una mejor referencia visual, en la figura 4 se han representado las mejores proporciones logradas en las 20 iteraciones de cada instancia y cada combinación en diagramas tipo violín. De esta forma se puede observar la distribución que presentan, así como la media para cada combinación.

Se han realizado también análisis estadísticos de tipo ANOVA entre ambas combinaciones de cada instancia para dilucidar si existe una diferencia estadística al variar la probabilidad de mutación, la cantidad de cruzamientos y el tamaño de población inicial. Para la instancia 1, se ha concluido que sí existe una diferencia estadística, ya que el resultado del análisis arroja un valor  $p$  mucho menor a 0.05. El valor  $p$  encontrado para la instancia 2 es mayor a 0.05, por lo que se podría concluir que no existe diferencia estadísticamente significativa al variar los parámetros mencionados. En la instancia 3 también se encuentra que hay una diferencia estadística al obtener un valor  $p$  menor a 0.05.

## 4. Conclusiones

Este es un ejemplo bastante básico de la forma en que se puede implementar un algoritmo genético para hacer más eficiente la solución de problemas complejos.

De los análisis estadísticos se puede observar que, tanto para la instancia 1 como para la instancia 3, sí existe una diferencia en la calidad de las proporciones entre valor máximo y valor óptimo al variar los parámetros. Sin embargo, para la instancia 2 no existe tal diferencia en calidad. Esto podría deberse a la rapidez con la que el algoritmo llega al valor óptimo, lo cual a su vez podría deberse a la relación exponencial entre valores y pesos.

cuando estos se generan.

## Referencias

- [1] E. Schaeffer. Geneticalgorithm, 2019. URL <https://github.com/satuelisa/Simulation/blob/master/GeneticAlgorithm>.
- [2] J. Torres. P\_10, 2022. URL [https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P\\_10](https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P_10).

# Reporte 11: Frentes de Pareto

Jorge Torres

9 de mayo de 2022

# Capítulo 1

## Cantidad de Funciones Objetivo

### 1.1. Objetivo

En optimización multicriterio, a un mismo conjunto de variables se necesita asignarse valores de tal forma que se optimicen dos o más funciones objetivo, que pueden contradecir una a otra — una mejora en una puede corresponder en una empeora en otra. Además hay que respetar potenciales restricciones, si es que las haya.

El objetivo de la actividad consiste en graficar el porcentaje de soluciones de Pareto como función del número de funciones objetivo para  $k \in [2, 3, 4, 5]$  con diagramas de violín combinados con diagramas de caja-bigote, verificando que diferencias observadas, cuando las haya, sean estadísticamente significativas.

### 1.2. Desarrollo

El desarrollo de la actividad se basa en el [código](#) implementado por E. Schaeffer, donde selecciona las mejores soluciones de un conjunto aleatorio para dos polinomios generados al azar [1]. La implementación completa se puede encontrar en el [repositorio](#) en GitHub de J. Torres [2]. En el código 1.1 se tienen tres funciones. La función `poli` crea una cantidad requerida de expresiones polinomiales aleatoriamente, que después son evaluadas para ciertas soluciones con la función `eval`. La tercera función, `domin.by`, revisa si las soluciones provistas son una mejora o empeora para cada una de las funciones objetivo, seleccionando únicamente las soluciones que son una mejora.

Código 1.1: Creación y Evaluación de Polinomios; Verificación de Soluciones Dominadas

```
1 poli <- function(maxdeg, varcount, termcount) {
2   f <- data.frame(variable=integer(), coef=integer(), degree=integer())
3   for (t in 1:termcount) {
4     var <- sample(1:varcount, 1)
5     deg <- sample(0:maxdeg, 1)
6     f <- rbind(f, c(var, runif(1), deg))
7   }
8   names(f) <- c("variable", "coef", "degree")
9   return(f)
10 }
11
12 eval <- function(pol, vars) {
13   value <- 0.0
14   terms = dim(pol)[1]
15   for (t in 1:terms) {
16     term <- pol[t,]
17     value <- value + term$coef * vars[term$variable]^term$degree
18   }
19   return(value)
20 }
21
22 domin.by <- function(target, challenger) {
23   if (sum(challenger < target) > 0) {
24     return(FALSE)
25   }
26   return(sum(challenger > target) > 0)
27 }
```

Las líneas del código 1.2 establecen los parámetros iniciales de operación, con una cantidad de variables  $vc = 4$ ,

el grado máximo de polinomio  $md = 3$ , y una cantidad de términos  $tc = 5$ . También se tienen una cantidad de funciones objetivo  $k \in [2, 3, 4, 5]$ .

Código 1.2: Parámetros Iniciales

```

1 df = data.frame()
2 vc <- 4
3 md <- 3
4 tc <- 5
5 funciones <- c(2, 3, 4, 5)
6 obj <- list()
7 k = 0

```

Para cada cantidad de funciones objetivo se hacen 20 iteraciones del experimento, lo cual se inicializa en el código 1.3. En estas líneas también se evalúa una cantidad  $n = 200$  soluciones creadas aleatoriamente para todas las funciones objetivo.

Código 1.3: Inicialización de Iteraciones y Evaluación de Soluciones

```

1 for (j in funciones){
2   k = j
3   for (replica in 1:20){
4     for (i in 1:k) {
5       obj[[i]] <- poli(md, vc, tc)
6     }
7     minim <- (runif(k) > 0.5)
8     sign <- (1 + -2 * minim)
9     n <- 200
10    sol <- matrix(runif(vc * n), nrow=n, ncol=vc)
11    val <- matrix(rep(NA, k * n), nrow=n, ncol=k)
12    for (i in 1:n) {
13      for (j in 1:k) {
14        val[i, j] <- eval(obj[[j]], sol[i,])
15      }
16    }
}

```

Para decidir cuáles soluciones son las que dominan y cuáles son las dominadas, se implementa el código 1.4, donde se utiliza un lógica de VERDAD/FALSO según los resultados de la función `domin.by` del código 1.1. Así mismo, se hace un conteo de las soluciones dominadas y no dominadas para calcular el porcentaje de soluciones que dominan, el cual se utiliza posteriormente para hacer las gráficas tipo violín.

Código 1.4: Decisión y Conteo de Soluciones Dominadas y No Dominadas

```

1 mejor1 <- which.max(sign[1] * val[,1])
2 mejor2 <- which.max(sign[2] * val[,2])
3 cual <- c("max", "min")
4 no.dom <- logical()
5 dominadores <- integer()
6 for (i in 1:n) {
7   d <- logical()
8   for (j in 1:n) {
9     d <- c(d, domin.by(sign * val[i,], sign * val[j,]))
10  }
11  cuantos <- sum(d)
12  dominadores <- c(dominadores, cuantos)
13  no.dom <- c(no.dom, sum(d) == 0)
14}
15 frente <- subset(val, no.dom)
16 porcentaje = (length(frente[,1])/n)*100
17 resultado = c(k, replica, porcentaje)
18 df = rbind(df, resultado)
19 names(df) = c("k", "Replica", "Porcentaje")
20 }
21 }

```

### 1.3. Resultado

La figura 1.1 muestra un ejemplo de cómo se ve el frente de soluciones de Pareto con dos funciones objetivo. Resulta bastante difícil graficar ejemplos para sistemas con más de dos funciones, además de que el frente no se podría apreciar tan fácilmente, por lo que no se incluyen de manera visual. Sin embargo, estos sistemas sí se toman

en cuenta al realizar los diagramas de tipo violín, mostrando las distribuciones de los porcentajes de soluciones dominantes, lo cual se observa en la figura 1.2.

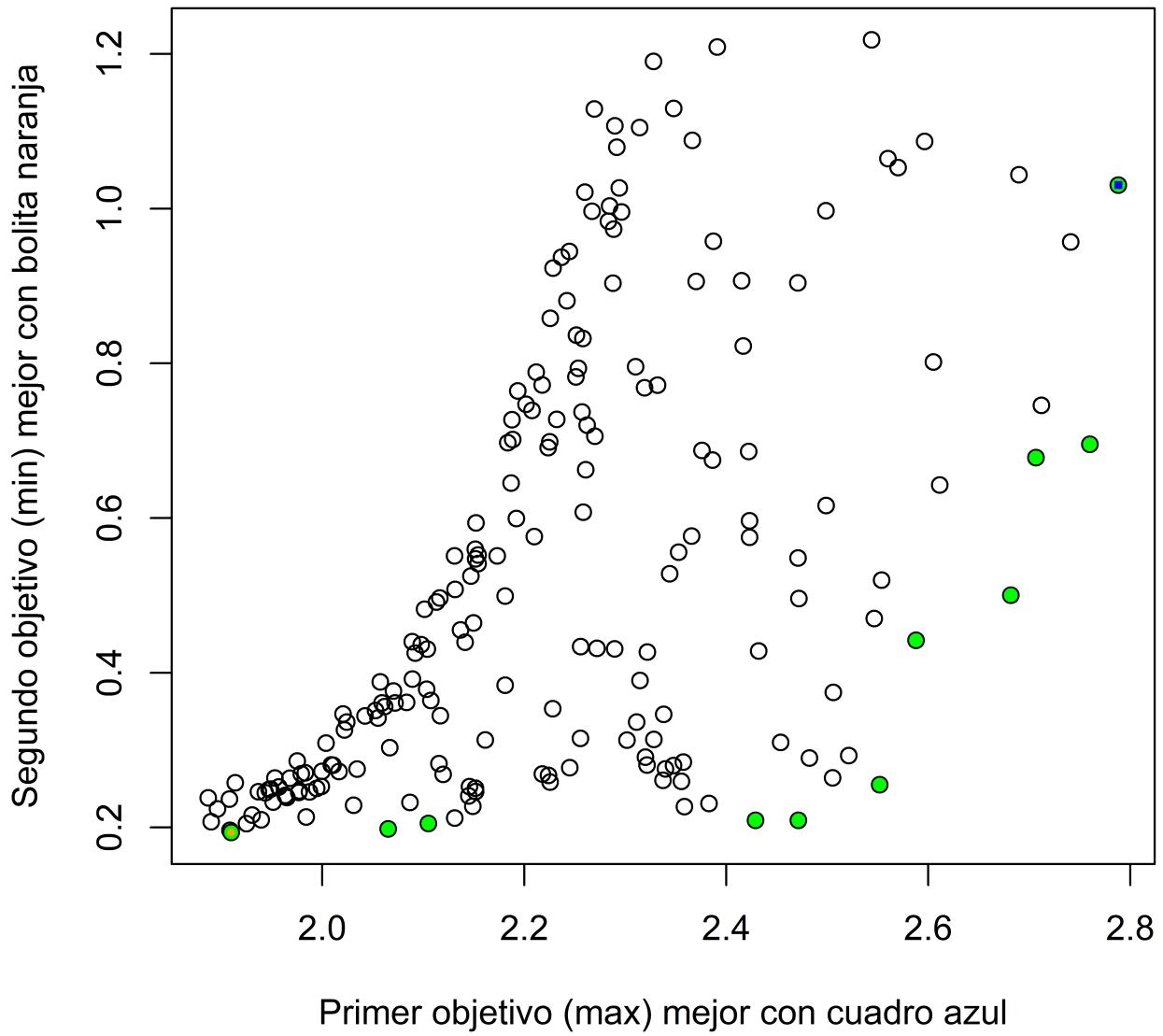


Figura 1.1: Frente de Pareto para un sistema de dos funciones objetivo. Los puntos verdes representan aquellas soluciones que no fueron dominadas por ninguna otra.

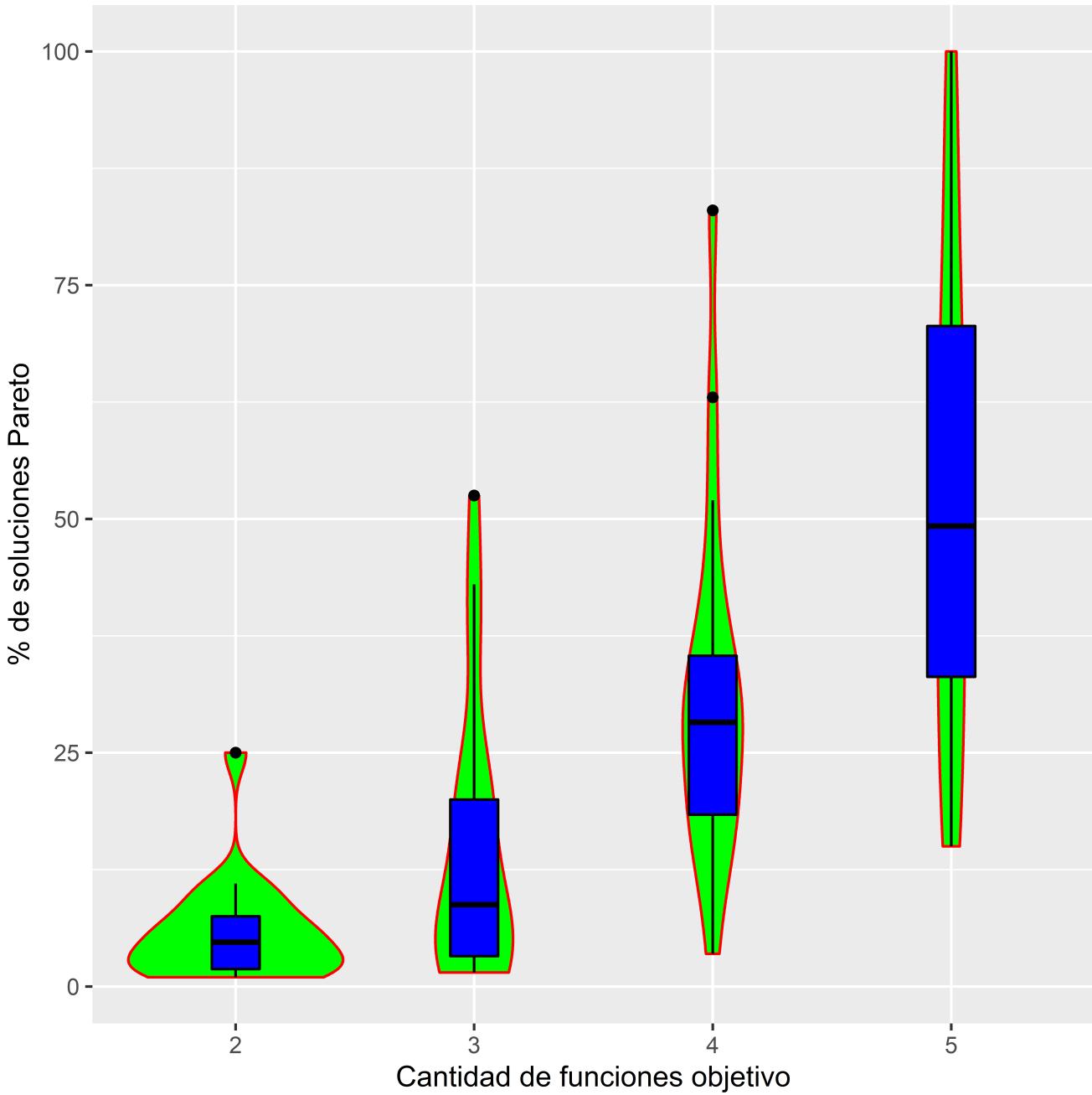


Figura 1.2: Distribuciones de los porcentajes de soluciones dominantes para cada cantidad de funciones objetivo.

Al realizar un análisis de varianza entre los porcentajes arrojados, se encuentra un valor  $p$  mucho menor a 0.05, lo cual prueba que, en efecto, existe una diferencia significativa al aumentar la cantidad de funciones objetivo.

## 1.4. Conclusiones

Como se puede observar del diagrama de la figura 1.2, para sistemas con solamente dos funciones objetivo, la el porcentaje de soluciones que no son dominadas es bastante bajo. En otras palabras, es mucho más encontrar soluciones óptimas para una menor cantidad de funciones. Sin embargo, conforme se aumenta la cantidad de funciones a optimizar, el rango de porcentajes de soluciones no dominadas se vuelve bastante amplio y uniformemente distribuido, teniendo desde  $\sim 10\%$  de soluciones hasta el 100 % de ellas. Esto podría deberse a la forma en que se desean optimizar las funciones. Si todas ellas se quieren minimizar o maximizar, es fácil encontrar soluciones que se encuentren muy cerca del óptimo para todas. Pero si algunas de ellas se maximizan mientras otras se minimizan, es más complicado encontrar soluciones que sean dominadas por otras.

# Capítulo 2

## Reto 1 - Soluciones Diversificadas

### 2.1. Objetivo

El objetivo del primer reto es el seleccionar un subconjunto (cuyo tamaño como un porcentaje del frente original se proporciona como un parámetro) del frente de Pareto, de tal forma que la selección esté diversificada, es decir, que no estén agrupados juntos en una sola zona del frente las soluciones seleccionadas.

### 2.2. Desarrollo

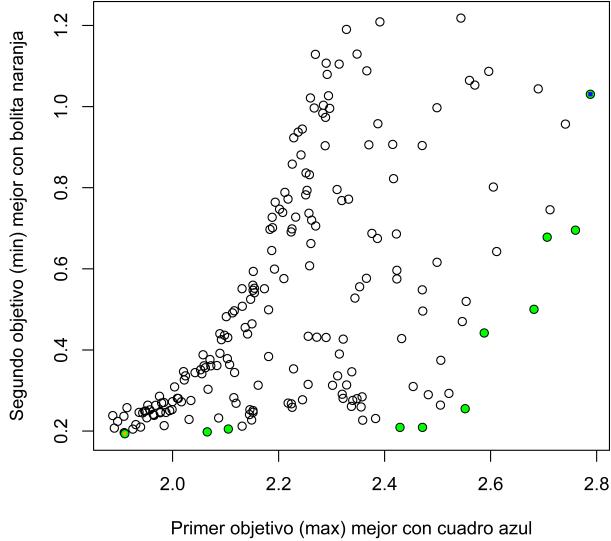
El desarrollo del reto es muy similar a la actividad del capítulo 1. Para simplificación de cálculo y visualización, la cantidad de funciones objetivo se reduce a  $k = 2$ . De la misma forma vista en los códigos 1.1 al 1.4, se crean funciones polinomiales y soluciones de manera aleatoria, se decide si las funciones se maximizan o minimizan y se seleccionan las funciones que dominen a todas las demás del conjunto. La diferencia radica en que ahora se toma un subconjunto de soluciones cuyo tamaño depende de un porcentaje del frente original. Esto se logra mediante la implementación del código 2.1. Estos nuevos puntos se utilizan posteriormente para graficar las soluciones.

Código 2.1: Selección Diversificada de Nuevas Soluciones

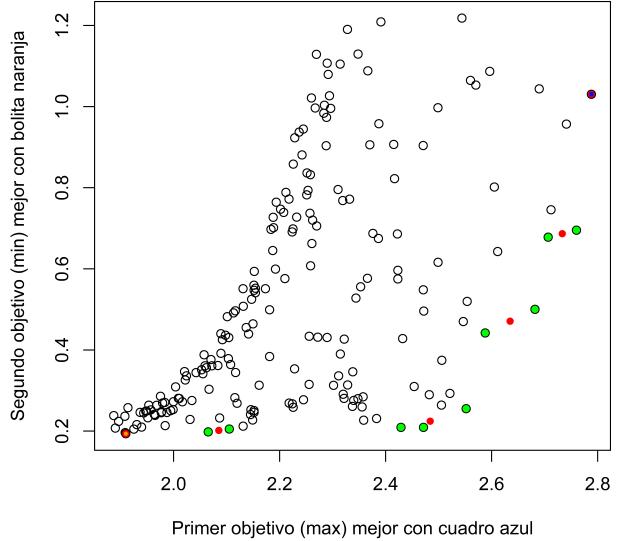
```
1 porcentaje=50
2 dispersos = kmeans(frente, round(dim(frente)[1]*porcentaje/100), iter.max = 1000, nstart = 50,
  algorithm = "Lloyd")
3 dispersos$cluster
4 dispersos$centers
5 mejor1 <- which.max((1 + (-2 * minim[1])) * val[,1])
6 mejor2 <- which.max((1 + (-2 * minim[2])) * val[,2])
```

### 2.3. Resultados

La figura 2.1a muestra el frente de Pareto para las dos funciones creadas aleatoriamente, donde las soluciones dominantes se resaltan en color verde. La primera función se requiere maximizar, mientras la segunda se desea minimizar. Las mejores soluciones para cada función se resaltan con un cuadro azul y una bola naranja, respectivamente. Por otro lado, la figura 2.1b muestra el subconjunto seleccionado de nuevas soluciones, donde éstas se marcan con bolas rojas.



(a) Frente de soluciones de Pareto para el sistema de dos funciones objetivo.



(b) Subconjunto de soluciones dispersas seleccionadas.

Figura 2.1: Frente de Pareto y subconjunto de nuevas soluciones.

Para visualizar la cantidad y frecuencia de estas soluciones dominantes, se implementa un diagrama de tipo violín en conjunto con uno de caja-bigote, como se puede apreciar en la figura 2.2.

## 2.4. Conclusiones

El método de las soluciones dispersas es útil para encontrar nuevas soluciones en la optimización de funciones, las cuales no se encuentran estrictamente dentro de un conjunto dado, pero sí muy cerca de las soluciones previamente encontradas.

## Cantidad de soluciones dominantes

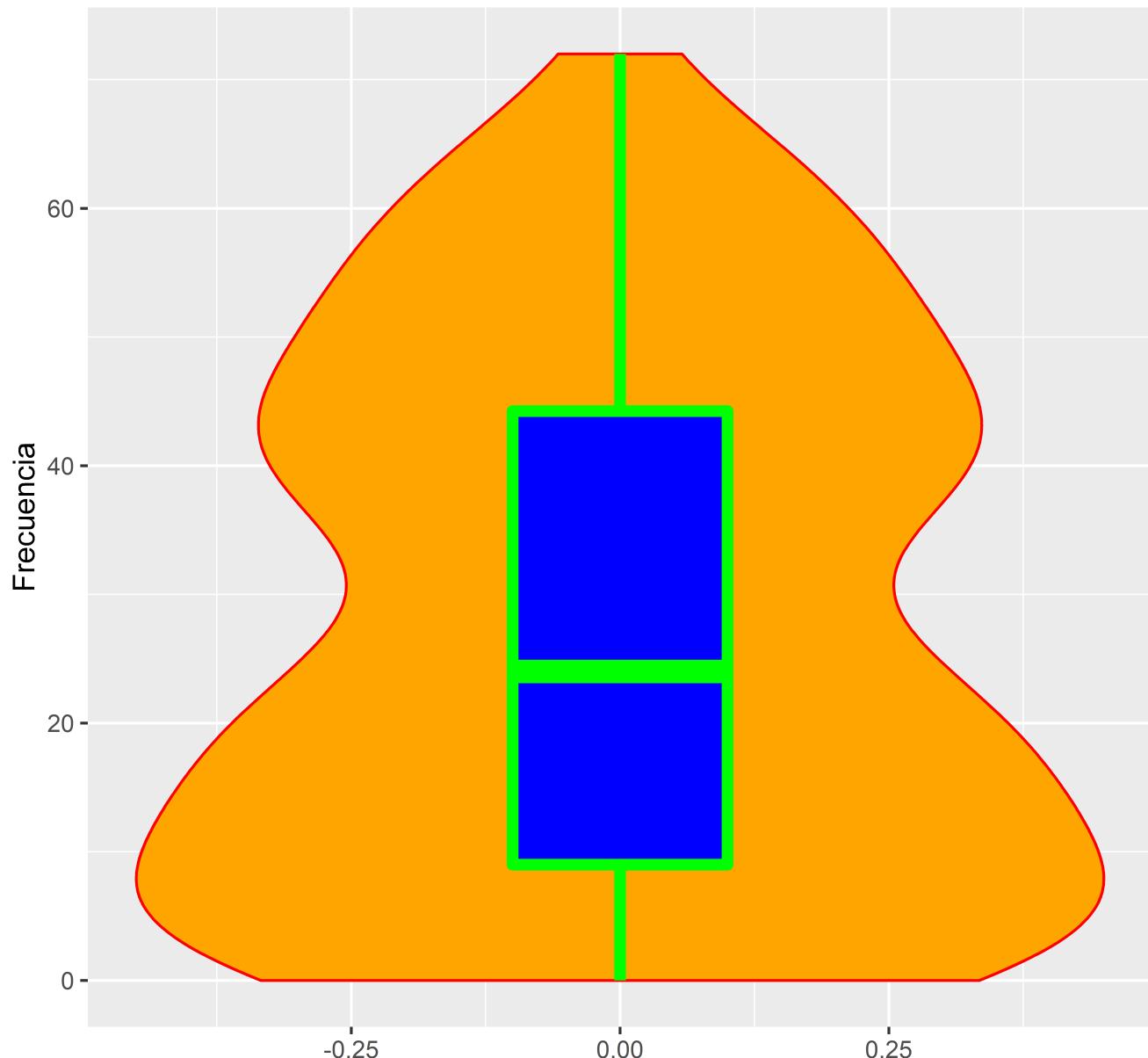


Figura 2.2: Cantidad y frecuencia de soluciones dominantes para el frente de Pareto y el subconjunto de soluciones.

# Bibliografía

- [1] E. Schaeffer. Paretfronts, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/ParetoFronts>.
- [2] J. Torres. P\_11, 2022. URL [https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P\\_11](https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P_11).

# Reporte 12: Red Neuronal

Jorge Torres

16 de mayo de 2022

# Capítulo 1

## Entrenamiento para Números

### 1.1. Objetivo

Esta actividad es una demostración básica de aprendizaje a máquina: se reconocerán dígitos de imágenes pequeñas en blanco y negro con una red neuronal. El objetivo consiste en estudiar de manera sistemática el desempeño de la red neuronal en términos de su puntaje F (F-score en inglés) para los diez dígitos en función de las tres probabilidades asignadas a la generación de los dígitos ( $n, g, b$ ), variando a las tres en un experimento factorial adecuado.

### 1.2. Desarrollo

El desarrollo de la actividad está basado en el [código](#) implementado por E. Schaeffer, con algunas modificaciones para variar las probabilidades de generación de los dígitos y realizar el análisis estadístico [1]. La implementación completa se puede obtener del repositorio en GitHub de J. Torres [3].

Las funciones `binario` y `decimal` del código 1.1 convierten un número decimal en binario y uno binario en decimal, respectivamente.

Código 1.1: Conversiones Binario y Decimal

```
1 binario <- function(d, 1) {
2   b <- rep(FALSE, 1)
3   while (1 > 0 | d > 0) {
4     b[1] <- (d %% 2 == 1)
5     l <- l - 1
6     d <- bitwShiftR(d, 1)
7   }
8   return(b)
9 }
10
11 decimal <- function(bits, 1) {
12   valor <- 0
13   for (pos in 1:1) {
14     valor <- valor + 2^(1 - pos) * bits[pos]
15   }
16   return(valor)
17 }
18
19 df = data.frame()
```

Las líneas del código 1.2 establecen las listas de probabilidades de que los bits del mapa de bits creado sean de color negro, gris o blanco, respectivamente. Este mapa de bits se utilizará para entrenar y probar la red neuronal.

Código 1.2: Probabilidades de Negro, Gris y Blanco

```
1 probn = c(0.98, 0.65, 0.35)
2 probg = c(0.95, 0.75, 0.55)
3 probb = c(0.45, 0.10, 0.005)
```

En el código 1.3 se inician las iteraciones entre las listas de probabilidades y se realizan 12 repeticiones para cada iteración. Se toman los valores de un archivo de texto y las probabilidades respectivas para generar un mapa de bits con una serie de dígitos de manera aleatoria, como se aprecia en la figura 1.1.

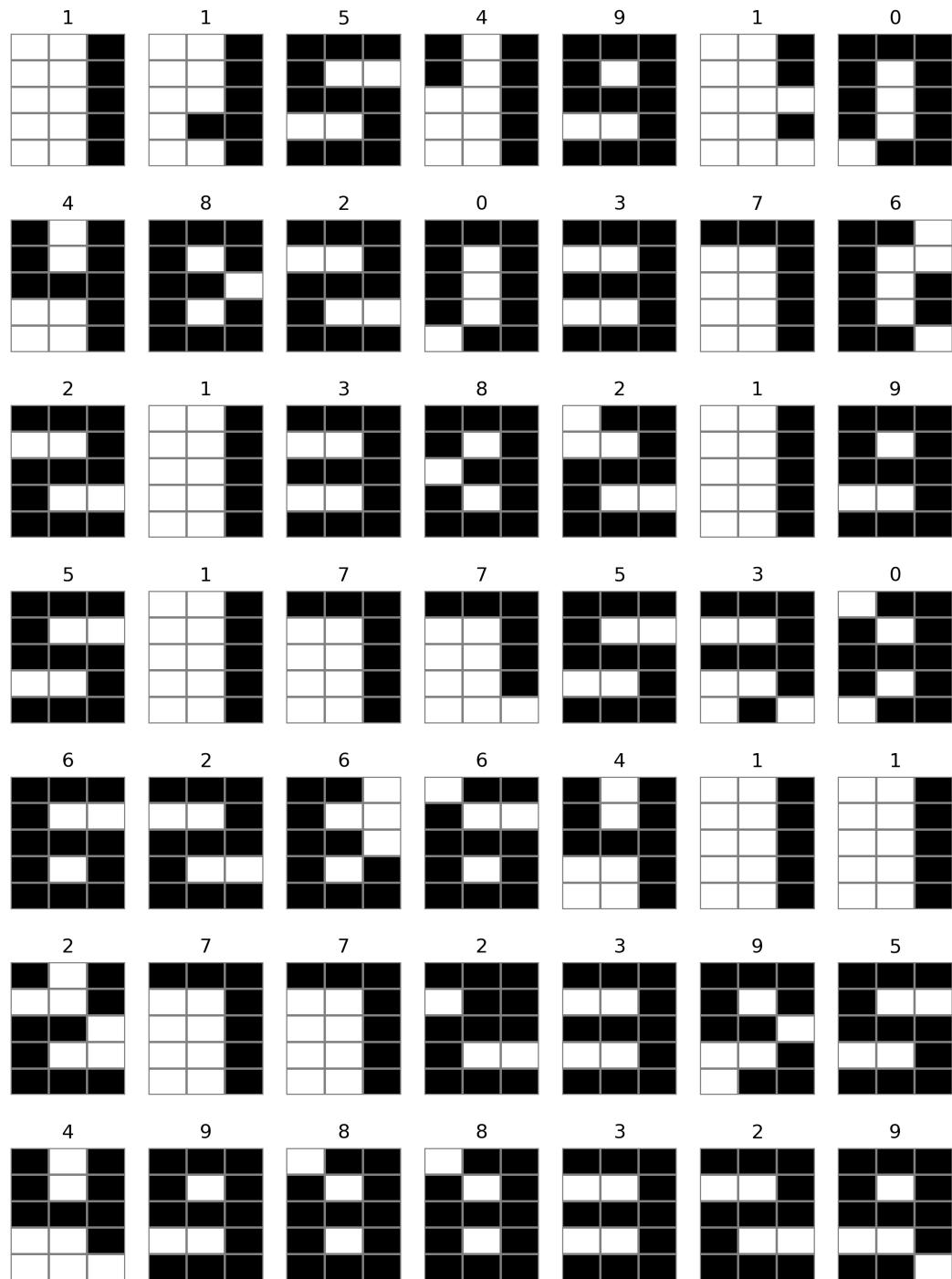


Figura 1.1: Serie de dígitos generados aleatoriamente, tomando las probabilidades del código 1.2.

Código 1.3: Generación del Mapa de Bits de los Dígitos

```

1 for (ne in probn){
2   for (g in probg){
3     for (b in probb){
4       for (replica in 1:12){
5         modelos <- read.csv("digits.txt", sep=" ", header=FALSE, stringsAsFactors=F)
6         modelos[modelos=='n'] <- ne
7         modelos[modelos=='g'] <- g
8         modelos[modelos=='b'] <- b
9
10        r <- 5
11        c <- 3
12        dim <- r * c
13
14        tope <- 9
15        digitos <- 0:tope
16        k <- length(digitos)
17        contadores <- matrix(rep(0, k*(k+1)), nrow=k, ncol=(k+1))
18        rownames(contadores) <- 0:tope
19        colnames(contadores) <- c(0:tope, NA)

```

En el código 1.4 se establecen la tasa de aprendizaje y el factor por el cual irá disminuyendo dicha tasa en la etapa de entrenamiento. También se establece la cantidad de perceptrones necesarios para la red neuronal. La etapa de entrenamiento consiste en tomar dígitos enteros del 0 al 9 de manera aleatoria y comparar el resultado del código 1.3 con el valor real por medio de los perceptrones. En este caso se realizan 5000 pasos de entrenamiento y se ajustan los perceptrones dependiendo de la validez del resultado.

Código 1.4: Etapa de Entrenamiento de los Perceptrones

```

1 tasa <- 0.15
2 tranqui <- 0.99
3 n <- floor(log(k-1, 2)) + 1
4 neuronas <- matrix(runif(n * dim), nrow=n, ncol=dim)
5
6 for (t in 1:5000) {
7   d <- sample(0:tope, 1)
8   pixeles <- runif(dim) < modelos[d + 1,]
9   correcto <- binario(d, n)
10  for (i in 1:n) {
11    w <- neuronas[i,]
12    deseada <- correcto[i]
13    resultado <- sum(w * pixeles) >= 0
14    if (deseada != resultado) {
15      ajuste <- tasa * (deseada - resultado)
16      tasa <- tranqui * tasa
17      neuronas[i,] <- w + ajuste * pixeles
18    }
19  }
20}

```

La etapa de prueba consiste en tomar los perceptrones obtenidos de la etapa de entrenamiento y aplicarlos directamente al mapa de bits. De esta forma, la red neuronal determinará a qué valor pertenece cada dígito generado. Se hace un conteo de todos los resultados obtenidos, el cual se utiliza para su posterior análisis. Esto se implementa en el código 1.5.

Código 1.5: Etapa de Prueba y Obtención de Resultados

```

1 for (t in 1:300) {
2   d <- sample(0:tope, 1)
3   pixeles <- runif(dim) < modelos[d + 1,]
4   correcto <- binario(d, n)
5   salida <- rep(FALSE, n)
6   for (i in 1:n) {
7     w <- neuronas[i,]
8     deseada <- correcto[i]
9     resultado <- sum(w * pixeles) >= 0
10    salida[i] <- resultado
11  }
12  r <- min(decimal(salida, n), k)
13  contadores[d+1, r+1] <- contadores[d+1, r+1] + 1
14}

```

Para obtener el puntaje F y realizar las pruebas estadísticas, se utiliza la ecuación 1.1, obtenida de la lectura en pruebas estadísticas de Shmueli [2], y se aplica para cada grupo de probabilidades por medio del código 1.6.

$$F = 2 \frac{(precision)(recall)}{precision + recall} \quad (1.1)$$

Código 1.6: Obtención del Puntaje F

```

1  precision = diag(contadores) / colSums(contadores[,1:10])
2  recall = diag(contadores) / rowSums(contadores)
3  fscore = (2 * precision * recall) / (precision + recall)
4  result = c(ne, g, b, replica, fscore)
5  df = rbind(df, result)
6 }
7 }
8 }
```

### 1.3. Resultados

Las distribuciones de los puntajes f (F-score) para las 12 repeticiones de cada grupo de probabilidades se muestran en la figura 1.2. Se puede observar cómo aumenta el puntaje para los grupos en los que la probabilidad de bits negros es mayor y la probabilidad de bits blancos es menor. Esto es de esperarse, pues en general se crearían imágenes de los dígitos más nítidas y con menos ruido.

Los puntajes obtenidos no parecen seguir una distribución normal, por lo que se ha realizado una prueba de tipo **Shapiro-Wilk** para comprobar dicho supuesto, mientras que se realiza una prueba de **Levene** para determinar la homogeneidad de varianza entre los grupos. Los resultados se muestran en el cuadro 1.1. Los “outliers” se refieren a la cantidad de valores atípicos en los grupos. Se puede observar que, para la mayoría de grupos, el valor *p* obtenido es menor a 0,05, por lo que no tienen una distribución normal.

Debido a esta distribución, se ha optado por realizar una prueba del tipo **Kruskall-Wallis** para determinar si existe una diferencia significativa en el puntaje F dependiente de las probabilidades iniciales. Los resultados se muestran en el cuadro 1.2. Se obtiene un valor *p* mucho menor a 0,05, por lo que se puede saber que la diferencia en el puntaje F es significativa al variar las probabilidades.

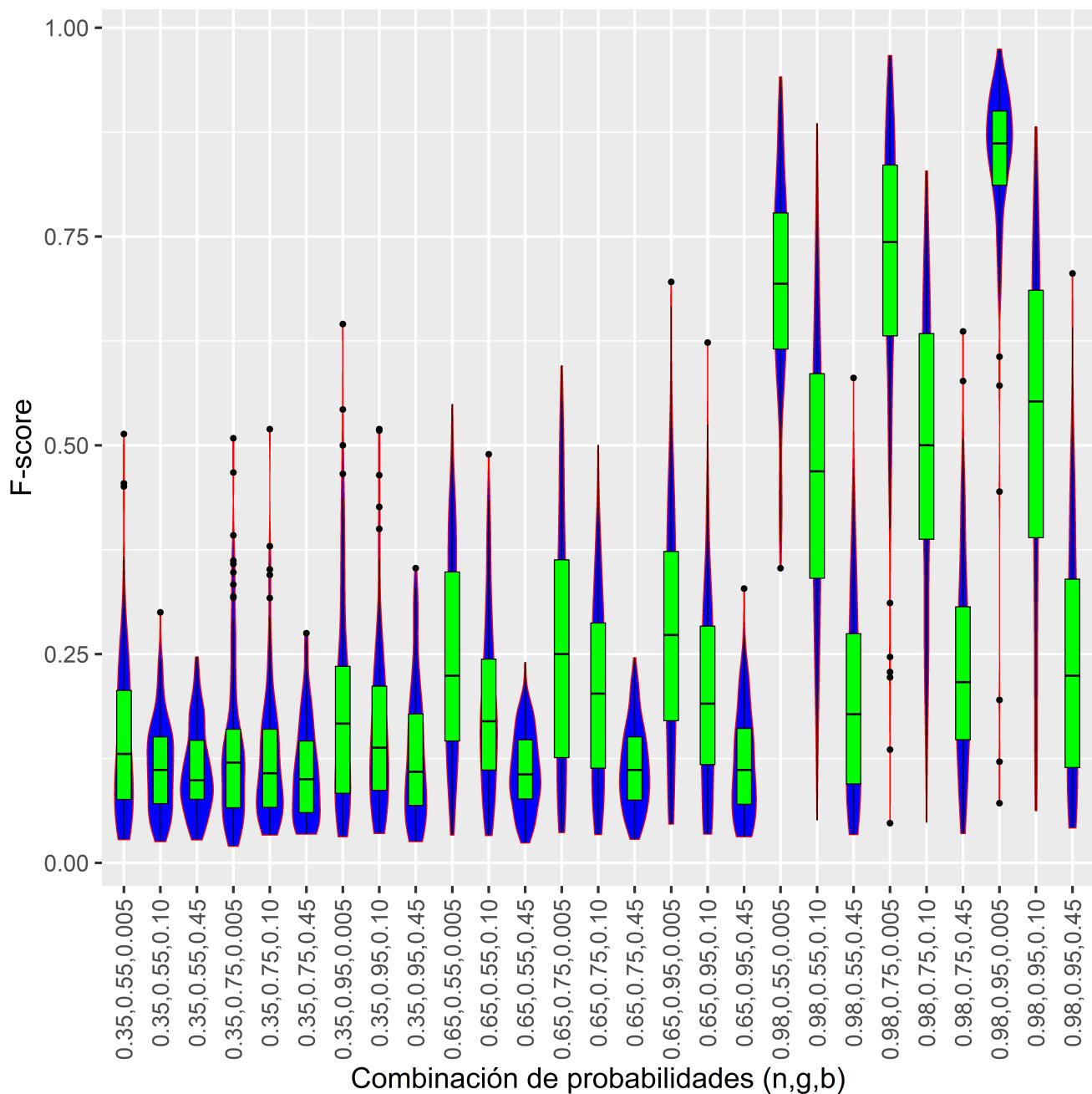


Figura 1.2: Puntajes F obtenidos para todos los grupos de probabilidades.

Cuadro 1.1: Resultados de las pruebas de normalidad y homogeneidad de varianza.

Outliers	47
Normalidad por grupo	0,35/0,55/0,005: $p = 1,72 \times 10^{-6}$ 0,35/0,55/0,10: $p = 1,81 \times 10^{-4}$ 0,35/0,55/0,45: $p = 1,06 \times 10^{-4}$ 0,35/0,75/0,005: $p = 1,62 \times 10^{-7}$ 0,35/0,75/0,10: $p = 2,05 \times 10^{-6}$ 0,35/0,75/0,45: $p = 2,43 \times 10^{-4}$ 0,35/0,95/0,005: $p = 2,35 \times 10^{-6}$ 0,35/0,95/0,10: $p = 4,48 \times 10^{-6}$ 0,35/0,95/0,45: $p = 2,02 \times 10^{-4}$ 0,65/0,55/0,005: $p = 5,30 \times 10^{-5}$ 0,65/0,55/0,10: $p = 2,37 \times 10^{-2}$ 0,65/0,55/0,45: $p = 2,31 \times 10^{-2}$ 0,65/0,75/0,005: $p = 3,42 \times 10^{-3}$ 0,65/0,75/0,10: $p = 3,93 \times 10^{-4}$ 0,65/0,75/0,45: $p = 1,16 \times 10^{-5}$ 0,65/0,95/0,005: $p = 6,09 \times 10^{-3}$ 0,65/0,95/0,10: $p = 8,46 \times 10^{-4}$ 0,65/0,95/0,45: $p = 7,73 \times 10^{-5}$ 0,98/0,55/0,005: $p = 6,68 \times 10^{-4}$ 0,98/0,55/0,10: $p = 2,58 \times 10^{-1}$ 0,98/0,55/0,45: $p = 4,47 \times 10^{-4}$ 0,98/0,75/0,005: $p = 8,90 \times 10^{-6}$ 0,98/0,75/0,10: $p = 1,25 \times 10^{-1}$ 0,98/0,75/0,45: $p = 3,07 \times 10^{-3}$ 0,98/0,95/0,005: $p = 8,79 \times 10^{-15}$ 0,98/0,95/0,10: $p = 2,38 \times 10^{-6}$ 0,98/0,95/0,45: $p = 1,34 \times 10^{-3}$
Homogeneidad de varianza	$p = 3,43 \times 10^{-22}$

Cuadro 1.2: Resultados al aplicar la prueba estadística Kruskal Wallis.

Chi cuadrada	Valor de $p$
1603,7	$2,2 \times 10^{-16}$

## 1.4. Conclusiones

Como se puede observar por la gráfica presentada y los resultados de los análisis estadísticos, es razonable concluir que no solamente existe una diferencia significativa en el puntaje F al variar las probabilidades iniciales, sino que esta diferencia es mucha más apreciable al tener mayores probabilidades de dibujar bits negros y menores probabilidades de dibujar un bit blanco. Esto es debido a que, con tales probabilidades, se obtienen imágenes menos ruidosas, por lo que la red neuronal puede realizar mejores identificaciones.

## Capítulo 2

# Entrenamiento para Números y Símbolos

### 2.1. Objetivo

El objetivo del primer reto consiste en extender y entrenar la red neuronal para que reconozca, además, por lo menos doce símbolos ASCII adicionales, aumentando la resolución de las imágenes a  $5 \times 7$  del original de  $3 \times 5$ .

### 2.2. Desarrollo

El desarrollo para el primer reto es muy similar al implementado en el capítulo 1. La diferencia está en la cantidad de símbolos que se utilizan y, por ende, la cantidad de perceptrones que la red neuronal necesita para las fases de entrenamiento y prueba. En el código 2.1 se listan las probabilidades ( $n, g, b$ ) utilizadas, así como la cantidad de dígitos que ahora debe reconocer la red neuronal.

Código 2.1: Nuevos Parámetros de Operación

```
1 modelos <- read.csv("digits2.txt", sep=" ", header=FALSE, stringsAsFactors=F)
2 modelos[modelos=='n'] <- 0.99
3 modelos[modelos=='g'] <- 0.92
4 modelos[modelos=='b'] <- 0.002
5
6 r <- 7
7 c <- 5
8 dim <- r * c
9
10 n <- 56
11 w <- ceiling(sqrt(n))
12 h <- ceiling(n / w)
13
14 tope <- 21
15 digitos <- 0:tope
16 k <- length(digitos)
17 contadores <- matrix(rep(0, k*(k+1)), nrow=k, ncol=(k+1))
18 rownames(contadores) <- 0:tope
19 colnames(contadores) <- c(0:tope, NA)
20
21 n <- floor(log(k-1, 2)) + 1
22 neuronas <- matrix(runif(n * dim), nrow=n, ncol=dim)
```

Utilizando estos datos, el mapa de bits cuyos símbolos ahora debe reconocer la red neuronal se observa en la figura 2.1.



Figura 2.1: Mapa de bits a reconocer con la inclusión de los nuevos símbolos.

## 2.3. Resultados

Lo que se muestra en el código 2.2 es la matriz de resultados que arroja el programa con los datos establecidos. Las filas representan el valor real de cada símbolo (21 en total), mientras que las columnas representan el valor que la red neuronal interpreta. De esta forma, cada relación fila-columna representa la cantidad de veces que la red neuronal reconoce el símbolo de la fila (real) como el símbolo de la columna (valor interpretado). Un ejemplo bastante extremo sería la intepretación del valor 0; se puede observar cómo, de las 14 pruebas que hizo la red neuronal para este valor, únicamente 3 de ellas fueron correctas, mientras que 11 de ellas las interpreta como un valor 1.

Código 2.2: Matriz de Interpretación

1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	<NA>
2	0	3	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	3	3	0	7	1	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
6	4	0	0	0	0	1	0	0	0	0	0	7	2	0	0	0	0	0	0	0	0	0	0
7	5	0	0	0	0	0	0	2	0	0	0	1	1	11	0	0	0	0	0	0	0	0	2
8	6	7	0	0	1	0	0	0	1	0	2	0	0	0	0	0	0	0	0	0	0	0	0
9	7	0	0	1	0	0	0	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	8	7	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	9	0	2	0	0	0	15	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
12	10	0	0	1	0	0	0	0	0	2	0	11	0	0	0	0	0	0	0	0	0	0	1
13	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	10	0
14	12	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	2
15	13	0	0	0	0	1	0	0	0	0	0	0	6	0	0	0	0	0	0	0	0	0	1
16	14	0	0	0	0	0	0	0	1	0	0	0	0	17	2	0	0	0	0	0	0	0	0
17	15	0	0	0	0	0	0	0	3	0	0	0	0	0	7	0	0	0	0	2	0	0	1
18	16	2	0	0	0	0	0	0	0	0	0	0	0	0	0	3	7	0	0	1	3	0	0
19	17	3	11	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	4
20	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	0	0	0	0	0
21	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	13	0	0	0	0
22	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
23	21	0	8	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1

## 2.4. Conclusiones

En esta ocasión, se puede observar que la red neuronal presenta algunos problemas al intentar reconocer símbolos que son más complejos. Sin embargo, los resultados aún son en su mayoría positivos al poder interpretar la mayoría de ellos como el símbolo que corresponde. Las imágenes no presentan demasiado ruido, pero es posible que se mejore la capacidad de reconocimiento de la red neuronal al incrementar la cantidad de perceptrones y acomodarlos en capas, donde los resultados de uno afecten la entrada de los que le siguen.

# Bibliografía

- [1] E. Schaeffer. Neuralnetwork, 2019. URL <https://github.com/satuelisa/Simulation/tree/master/NeuralNetwork>.
- [2] Boaz Shmueli. Multi-class metrics made simple, part ii: the f1-score, 2019. URL <https://towardsdatascience.com/multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1>.
- [3] J. Torres. P\_12, 2022. URL [https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P\\_12](https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P_12).

# Modelado de nanoindentación de una película delgada de oro

Jorge A. Torres Quintanilla

*Posgrado en Maestría en Ciencias de la Ingeniería con Orientación en Nanotecnología.*

*Facultad de Ingeniería Mecánica y Eléctrica.*

*Universidad Autónoma de Nuevo León.*

---

## Resumen

Debido su alta relación área/volumen, los materiales a escala nanométrica presentan propiedades físicoquímicas que los diferencian del material en bulto. Es por esto que se han desarrollado modelos matemáticos que describan con mayor precisión el comportamiento de las interacciones de los materiales a estas escalas. Los estudios de nanoindentación hacen uso de estos modelos y de la rama de la mecánica de contacto para deducir las propiedades mecánicas y adhesivas (como dureza, ductilidad, fuerza de adhesión), de un sinfín de materiales. En este artículo se realiza la simulación computacional de un nanoindentador de diamante entrando en contacto con una película delgada de oro. Se calculan las profundidades de indentación y se comparan con datos experimentales con el fin de elucidar en la veracidad de dichos modelos. Los resultados muestran diferencias significativas entre los datos reales y los teóricos, pero hay similitudes en la forma del comportamiento, lo cual puede indicar que hay otros fenómenos que no se están tomando en cuenta en la simulación.

*Palabras clave:* Mecánica de contacto, nanoindentación, adhesión, simulación

---

## 1. Introducción

La mecánica de contacto es un área de la tribología que estudia la deformación de sólidos que se tocan entre sí en uno o más puntos. Se hace una distinción entre los esfuerzos que actúan en un objeto de manera perpendicular (normal) y los que son causados por fricción, que tienen una dirección tangencial entre superficies. El estudio de estos esfuerzos y deformaciones es especialmente útil para determinar las propiedades mecánicas y adhesivas que presentan los materiales, como la dureza, módulo elástico y punto de fractura.

En la rama de la nanotecnología, los materiales presentan una alta relación área/volumen comparada al material en bulto, lo cual altera sus propiedades mecánicas y adhesivas. Esto implica que deben proponerse modelos matemáticos que definen con mayor precisión el comportamiento de la materia a tales escalas. Los estudios de nanoindentación son pruebas muy comunes en este ámbi-

to de la ciencia, y se utilizan para determinar un rango de propiedades que presentan los nanomateriales. Estas pruebas consisten en la aplicación de una fuerza al material cuyas propiedades se desean saber por medio de una punta muy fina con geometría y propiedades mecánicas conocidas, usualmente de diamante o un material de alta dureza. Esto se hace con el propósito de causar una indentación en el material estudiado, cuya geometría está directamente relacionada con sus propiedades mecánicas y de adhesión.

En el presente se propone una simulación de los dos modelos más comúnmente utilizados en estudios de nanoindentación, con un enfoque específico en las propiedades de una película delgada de oro. Los resultados se comparan con datos reales obtenidos de pruebas de nanoindentación realizadas por Beake y Smith [1] bajo condiciones similares.

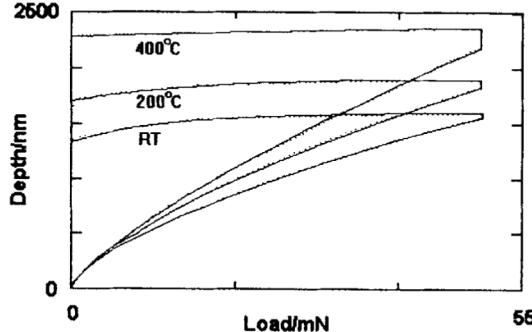


Figura 1: Gráfica carga-profundidad de una película de oro [1].

## 2. Antecedentes

Experimentos como los realizados por Beake y Smith [1] y Dietker *et al.* [2] ya han utilizado la técnica de nanoindentación para medir las propiedades mecánicas de diversos materiales, incluyendo películas delgadas de oro. Los anteriores se basan en el modelo de contacto elástico descrito por Hertz [3] y en la técnica desarrollada por Oliver y Pharr [4], con la cual determinan el módulo elástico y la dureza de diversos materiales por medio de la gráfica carga-profundidad. Un ejemplo de este tipo de gráfica se puede observar en la figura 1.

## 3. Trabajos Relacionados

En su trabajo, Kelchner *et al.* [5] realizan la simulación de una prueba de nanoindentación de una película delgada de oro con dimensiones de  $24 \times 21 \times 16$  nm y un indentador de geometría esférica con un radio de 8 nm. Hacen uso de un modelo de dinámica molecular paralela para simular el comportamiento de 470,000 átomos al entrar en contacto con el indentador, así como de las ecuaciones descritas en el modelo Hertziano. Sus resultados son muy prometedores, ya que no solamente encuentran que el comportamiento simulado se acerca en buena medida al real, sino que pueden determinar puntos de dislocación en la estructura cristalina del oro.

## 4. Modelo Propuesto

En el presente se hace uso de los dos modelos más comúnmente utilizados en el ámbito de mecánica de contacto. El primero de ellos es el descrito por Hertz [3], en

el cual no se toman en cuenta las propiedades adhesivas de los materiales, por lo que la profundidad de indentación está determinada únicamente por la fuerza aplicada  $F$ , el módulo elástico reducido de ambos materiales en contacto  $E^*$ , y la dimensión característica del indentador  $R$ , como se puede apreciar en la ecuación 1.

$$d_{HERTZ} = \left( \frac{9F^2}{16E^{*2}R} \right)^{\frac{1}{3}} \quad (1)$$

El módulo elástico reducido  $E^*$  es una función de los módulos elásticos ( $E_1, E_2$ ), y de las relaciones de Poisson ( $\nu_1, \nu_2$ ), de ambos materiales, como se expresa en la ecuación 2.

$$\frac{1}{E^*} = \frac{1 - \nu_1^2}{E_1} + \frac{1 - \nu_2^2}{E_2} \quad (2)$$

El segundo modelo es el descrito por Johnson, Kendall y Roberts (JKR) [6], en el cual toman en cuenta la energía de adhesión existente entre ambas superficies, por lo que el área de contacto resultante es mayor que la descrita por el modelo Hertziano, y cuya profundidad de indentación depende ahora del trabajo de adhesión ejercido entre ambos materiales  $\Delta\gamma$  y de las presiones ejercidas por la fuerza aplicada ( $p_0, p'_0$ ), como se observa en la ecuación 3.

$$d_{JKR} = \frac{\pi a_{JKR}}{2E^*} (p_0 + 2p'_0) \quad (3)$$

Las presiones  $p_0$  y  $p'_0$  están determinadas por la ecuación 4, donde los términos  $a_{JKR}$  y  $\Delta\gamma$  son el área de contacto descrita por el modelo JKR y el trabajo de adhesión ejercido entre ambas superficies, los cuales se encuentran expresados en las ecuaciones 5 y 6, respectivamente.  $\gamma_1$  y  $\gamma_2$  son las energías de adhesión de cada uno de los materiales en contacto, mientras que  $\gamma_{12}$  es el término de interacción.

$$p_0 = \frac{2a_{JKR}E^*}{\pi R}; p'_0 = -\left( \frac{2\Delta\gamma E^*}{\pi a_{JKR}} \right)^{\frac{1}{2}} \quad (4)$$

$$a_{JKR} = \left\{ \frac{3R}{4E^*} \left[ F + 3\Delta\gamma\pi R + \sqrt{6\Delta\gamma\pi RF + (3\Delta\gamma\pi R)^2} \right] \right\}^{\frac{1}{3}} \quad (5)$$

$$\Delta\gamma = \gamma_1 + \gamma_2 - \gamma_{12} \quad (6)$$

Para los modelos matemáticos presentados, se propone la simulación de un nanoindentador con una punta de diamante tipo Berkovich, de geometría esférica con un radio  $R = 200$  nm. El módulo elástico reducido se obtiene de datos experimentales realizados por Dietiker *et al.* [2], y se toma como  $E^* = 85$  GPa. El trabajo de adhesión entre el diamante y el oro se toma como  $\Delta\gamma = 300$   $mJ/m^2$ , dados los datos obtenidos experimentalmente por Gane *et al.* [7]. Por último, la fuerza aplicada  $F$  se aumenta gradualmente en un rango de 0 - 50 mN, y se calculan las profundidades de indentación descritas por las ecuaciones de ambos modelos. La teoría apunta a una diferencia significativa entre ambos modelos en las profundidades de indentación a cargas muy bajas, mientras que esta diferencia iría disminuyendo conforme se aumenta la carga aplicada.

## 5. Implementación

La simulación se implementa con el software de desarrollo Python v3.10.1 [8], mientras que el código completo se puede encontrar en el repositorio en GitHub de Torres [9].

En el código 1 se toman datos reales de una curva carga-profundidad de experimentos realizados en películas de oro por Beake y Smith [1]. Asimismo, se introducen los parámetros de geometría, módulo elástico y trabajo de adhesión descritos en la sección 4.

Código 1: Datos Experimentales y Parámetros de Operación

```
1 datos = pd.read_csv('DatosReales.txt',
2                      delimiter = ' ')
3 F = (datos.Carga)*(10**-3) #N
4 d_r = datos.Deformacion
5 E = 85*(10**9) #Pa
6 R = 200*(10**-9) #m
7 g = 300*(10**-3) #J/m2
```

Las líneas del código 2 introducen las listas donde se guardan los valores calculados de profundidades de indentación para ambos modelos, así como las diferencias de profundidad entre los datos experimentales y los calculados.

Código 2: Listas de Profundidades de Indentación y Diferencias

```
1 hertz_list = []
2 jkr_list = []
3 diffa = []
4 diffb = abs(d_r - hertz_list)
5 diffc = abs(d_r - jkr_list)
```

Las funciones  $a()$ ,  $d_hertz$  y  $d_jkr$  del código 3 definen las ecuaciones para calcular el área de contacto según el modelo JKR,  $a_{JKR}$ , la profundidad de indentación según el modelo Hertziano,  $d_{HERTZ}$ , y la profundidad de indentación según el modelo JKR,  $d_{JKR}$ , respectivamente.

Código 3: Funciones de Ecuaciones de los Modelos

```
def a(F):
    return (((3*R)/(4*E))*(F + 3*g*pi*R + np
        .sqrt((6*g*pi*R*F) + (3*g*pi*R)**2)))
    **(1/3)

def d_hertz(F):
    return (((9*(F**2))/(16*(E**2)*R))
    **(1/3))*(10**9)

def d_jkr(F, a, p1, p2):
    return (((pi*a)/(2*E))*(p1 + 2*p2))
    *(10**9)
```

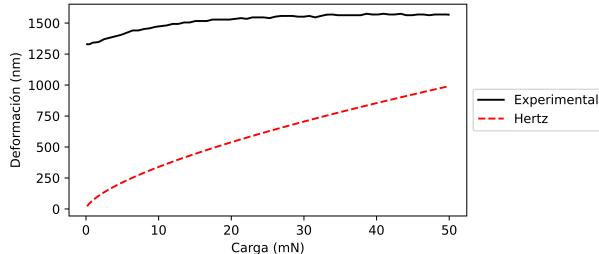
En el código 4 se comienza a introducir la fuerza debido al indentador, y se calculan el área de contacto,  $a_{JKR}$ , los componentes de presión,  $p_0$  y  $p'_0$ , así como las profundidades de indentación,  $d_{HERTZ}$  y  $d_{JKR}$ . Para cada dato de fuerza aplicada se calcula la diferencia entre las profundidades Hertzianas y de JKR y se agregan a la lista.

Código 4: Implementación de Modelos Matemáticos

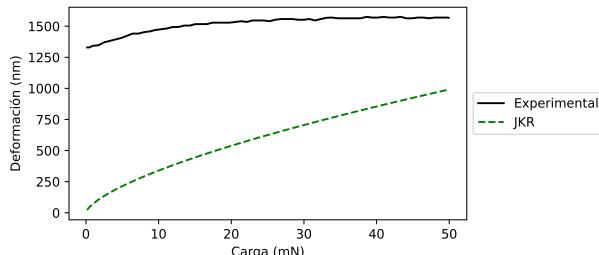
```
for f in F:
    A = a(f)
    p1 = (2*A*E)/(pi*R)
    p2 = -1 * np.sqrt((2*g*E)/(pi*A))
    hertz = d_hertz(f)
    jkr = d_jkr(f, A, p1, p2)
    hertz_list.append(hertz)
    jkr_list.append(jkr)
    diffa.append(abs(jkr - hertz))
```

## 6. Resultados

En la figura 2 se observa una comparación entre las gráficas carga-profundidad de los datos experimentales y los calculados con el modelo Hertziano (figura 2a) y los calculados con el modelo JKR (figura 2b).



(a) Gráficas carga-profundidad de datos experimentales y modelo Hertziano.



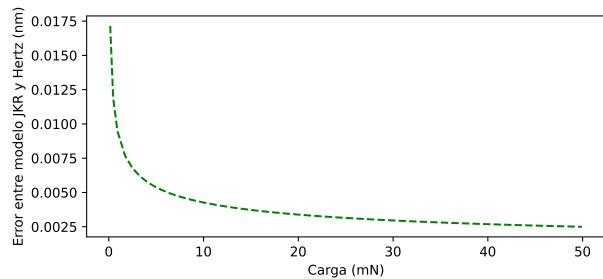
(b) Gráficas carga-profundidad de datos experimentales y modelo JKR.

Figura 2: Comparaciones entre datos experimentales y datos de los modelos Hertziano y JKR.

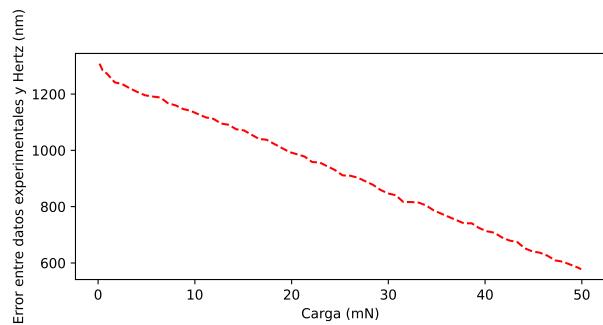
La figura 3 muestra los errores absolutos en profundidades de indentación entre ambos modelos (figura 3a), entre los datos experimentales y el modelo Hertziano (figura 3b), y entre los datos experimentales y el modelo JKR (figura 3c).

### 6.1. Discusión

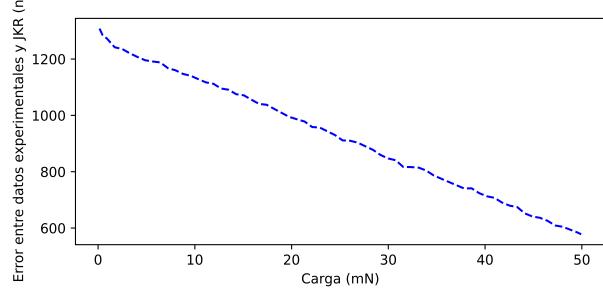
Como se puede apreciar de la figura 2, no parece haber una gran diferencia entre las profundidades de indentación calculadas por ambos modelos, aunque sí que la hay al comparar con los datos experimentales. Sin embargo, parecen seguir una curvatura muy similar. Además, al comparar los errores absolutos entre el modelo Hertziano y el JKR, se puede percibir una gran diferencia a cargas bajas, mientras que esta disminuye a cargas mayores, justo como lo predicen dichos modelos. Lo mismo aplica al comparar cada uno de los modelos con los datos experimentales, excepto que en estas instancias las diferencias son mucho mayores.



(a) Error absoluto entre el modelo Hertziano y el JKR.



(b) Error absoluto entre datos experimentales y el modelo Hertziano



(c) Error absoluto entre datos experimentales y el modelo JKR

Figura 3: Errores absolutos entre los datos experimentales y ambos modelos.

## 7. Conclusión

En el trabajo presente se opta por hacer una simulación muy directa, además de que no se toman en cuenta otros fenómenos como las fuerzas de fricción existentes entre superficies, lo cual puede aumentar el área de contacto, aumentando a su vez la profundidad de indentación. Sin embargo, se comprueban las predicciones hechas por Johnson, Kendal y Roberts , en el que hay un aumento apreciable en cuanto a la profundidad de indentación a comparación con los estimados por Hertz, sobre todo a cargas aplicadas bajas, esto debido a las interacciones de adhesión en las superficies.

### 7.1. Trabajo a Futuro

Debido al tiempo limitado del proyecto, no se ha podido realizar una simulación con un mayor grado de complejidad, en el que se implemente un modelo de dinámica molecular y en el que se tomen en cuenta las interacciones entre átomos de la película delgada de oro y la geometría del indentador de diamante. Además, se podrían tomar en cuenta fuerzas de fricción tangenciales, lo cual acercaría los resultados de la simulación al comportamiento real de los materiales.

## Referencias

- [1] B. D. Beake, J. F. Smith, High-temperature nanoin-dentation testing of fused silica and other materials, Philosophical Magazine A 82 (2002) 2179 – 2186. doi:doi:[10.1080/01418610208235727](https://doi.org/10.1080/01418610208235727).
- [2] M. Dietiker, R. D. Nyilas, C. Solenthaler, R. Spolen-nak, Nanoindentation of single-crystalline gold thin films: Correlating hardness and the onset of plas-ticity, Acta Materialia 56 (2008) 3887 – 3899. doi:doi:[10.1016/J.ACTAMAT.2008.04.032](https://doi.org/10.1016/J.ACTAMAT.2008.04.032).
- [3] H. Hertz, On the contact of elastic solids, 1896.
- [4] W. Oliver, G. Pharr, An improved technique for deter-mining hardness and elastic modulus using load and displacement sensing indentation experiments, Journal of Materials Research 1992 7:6 7 (1992) 1564 – 1583. doi:doi:[10.1557/JMR.1992.1564](https://doi.org/10.1557/JMR.1992.1564).
- [5] C. L. Kelchner, S. Plimpton, J. C. Hamilton, Dislo-ca-tion nucleation and defect structure during surface indentation, Physical Review B - Condensed Mat-ter and Materials Physics 58 (1998) 11085 – 11088. doi:doi:[10.1103/PHYSREVB.58.11085](https://doi.org/10.1103/PHYSREVB.58.11085).
- [6] K. L. Johnson, K. Kendall, A. D. Roberts, Surfa-ce energy and the contact of elastic solids, Proceed-ing-s of the Royal Society of London. A. Mathemat-ical and Physical Sciences 324 (1971) 301 – 313. doi:doi:[10.1098/rspa.1971.0141](https://doi.org/10.1098/rspa.1971.0141).
- [7] N. Gane, P. F. Pfaelzer, D. Tabor, Adhesion be-tween clean surfaces at light loads, Proceedings of the Royal Society of London. A. Mathemat-ical and Physical Sciences 340 (1974) 495 – 517. doi:doi:[10.1098/rspa.1974.0167](https://doi.org/10.1098/rspa.1974.0167).
- [8] Python Software Foundation, The Python Softwa-re Foundation is an organization devoted to advanc-ing open source technology related to the Python programming language, 2022. URL: <https://www.python.org/>.
- [9] J. Torres, Proyecto Integrador, 2022. URL: <https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/Proyecto%20Integrador>.