



UANL

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN



FIME

FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA

Universidad Autónoma de Nuevo León
Facultad de Ingeniería Mecánica y Eléctrica
Maestría en Ciencias de la Ingeniería con Orientación a la Nanotecnología

Portafolio de Evidencias

Simulación Computacional de Nanomateriales

Segundo semestre

enero-junio 2022

Estudiante:

Lagunes Rviera Raul
(2125677)

Docentes:

Dr. Virgilio González
Dra. Elisa Schaeffer

Calificación:

Prácticas:5+5+5+8+8+7+7+9+5+8+7+5=79

Proyecto Final:15

Calificación Final:94

URL GITHUB:

<https://github.com>

UANL

1 de junio del 2022

Práctica 1 Movimiento Browniano

Raul L.

16 de febrero de 2022

1. Introducción

La práctica número 1 estudia el tema Movimiento Browniano, donde se debe estudiar en una caja-bigote el regreso al origen de dicha partícula en 5 dimensiones, variando el número de pasos de la caminata, con 30 repeticiones del experimento para cada combinación [1].

2. Objetivo

El objetivo de la simulación es examinar de manera sistemática los efectos de la dimensión en la distancia Euclídea na máxima del origen del movimiento browniano para cinco dimensiones, variando el número de pasos de la caminata (100,1000,10000 pasos), con 30 repeticiones del experimento para cada combinación y grafica los resultados en una sola figura diagrama caja-bigote con diferentes colores.

3. Código

En el siguiente código se utilizarán secuencias for para realizar todo el objetivo propuesto en una sola ejecución. El código base se sacó del repertorio de la Dra. Elisa Schaeffer se modificó para poder realizar diferentes caminas a cada una de las cinco dimensiones.

Código en python

<https://github.com/satuelisa/Simulation/blob/master/BrownianMotion>

Código creado en Python

```
from random import random, randint
from math import fabs, sqrt
import matplotlib.pyplot as plt
import numpy as np
DIMS=1,2,3,4,5  cuantas dimensiones
caminatas= 100, 1000, 10000
replicas = 30  cuantas veces
CB_100=[] 
CB_1000=[]
CB_10000=[]
for dim in DIMS:
    print("##### Dimencion:",dim,"#####")
    cien=[]
    mil=[]
    diezmill=[]
    for replica in range (replicas):
        pos=[0]* dim
```

```

mayor= 0
re=0
mayores=[]
for s in caminatas:
    for paso in range(s):
        eje= randint(0, dim - 1)
        if random()< 0.5:
            pos[eje] +=1
        else:
            pos[eje]-= 1
        mayor = max ( mayor, sqrt(sum([p**2 for p in pos])))
    mayores.append(mayor)
cien.append(mayores[0])
mil.append(mayores[1])
diezmill.append(mayores[2])
CB_100.append(cien)
CB_1000.append(mil)
CB_10000.append(diezmill)

print("cuantos datos de caja bigote",len(CB_100))
pb=plt.boxplot([CB_100[0],CB_1000[0],CB_10000[0],CB_100[1],
CB_1000[1],CB_10000[1],CB_100[2],CB_1000[2],CB_10000[2],
CB_100[3],CB_1000[3],CB_10000[3],CB_100[4],CB_1000[4],CB_10000[4]])
plt.xticks([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
['100','1000','10000','100','1000','10000',
'100','1000','10000','100','1000','10000','100','1000','10000'])

plt.xlabel('camintas')
plt.ylabel('Distancia máxima')
plt.tick_params(axis='x', rotation=45)
plt.title('Distancia Euclidiana')

colors = ['orange','orange','orange','red','red', 'red',
'green','green','green', 'pink','pink','pink','blue','blue','blue']

for patch, color in zip(pb['boxes'], colors):
    patch.set_color(color)

plt.plot([], c='orange', label='dimencion 1')
plt.plot([], c='red', label='dimencion 2')
plt.plot([], c='green', label='dimencion 3')
plt.plot([], c='pink', label='dimencion 4 ')
plt.plot([], c='blue', label='dimencion 5')

plt.legend()

plt.savefig('figuraPy.png') # mandar a un archivos

print(pos)

```

4. Resultados

En la figura se muestra como se comporta cada dimencion con diferente tamaño de camina, de color amarillo es la dimension uno, de color rojo la dimension dos, de color verde la dimension tres, de color naranja la dimension cuatro y azul la dimension cinco, se puede observar la variacion de la distancia con respecto de la cantidad de pasos que hace en cada recorrido, en los recorridos de 100 pasos la distancia es muy pequena comparada con el recorrido de 10000 pasos el cual el extremadamente grande, en la caja-bigote es el muestreo de 30 repeticiones dando los resultados maximos para crear dicha caja.

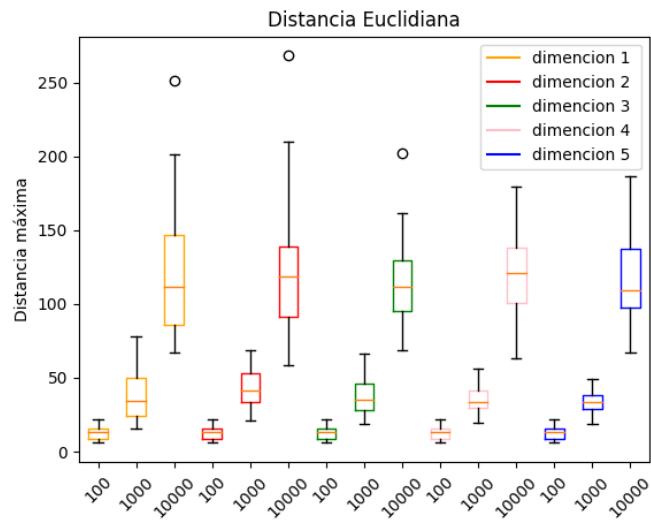


Figura 1: tomado del repositorio de Raul L. del codigo de python <https://github.com/Raullr28/Resultados/blob/main/P1/figuraPy.png>

5. Conclusiones

se realizó la tarea numero 1 correctamente, obteniendo la caja-bigote esperada con las caminas propuestas y las dimensiones deseadas.

Referencias

- [1] E.Schaeffer.Github. Github.movimiento browniano. 2022. URL <https://elisa.dyndns-web.com/teaching/comp/par/p1.html>.

Práctica 2: Autómata Celular

Raul L.

23 de febrero de 2022

1. Introducción

En la segunda práctica trabajamos con autómatas celulares en dos dimensiones, particularmente el famoso juego de la vida. El estado del autómata se representa con una matriz booleana (es decir, contiene ceros y unos). Cada celda es o viva (uno) o muerta (cero). En cada paso, la supervivencia de cada celda (verde) se determina a partir de los valores de sus ocho vecinos (amarillos)[2].

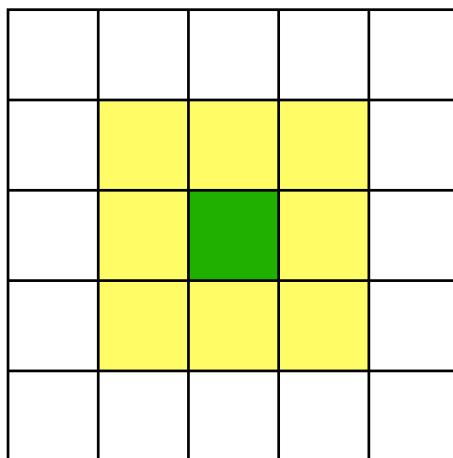


Figura 1: tomado de la práctica 2 <https://satuelisa.github.io/simulation/p2.html>

En los extremos de la matriz, las celdas simplemente tienen menos vecinos. Otra alternativa sería considerar el espacio como un torus — pareciendo una dona — donde el extremo de abajo se reune con el extremo de arriba al igual como los lados izquierdo y derecho, uno con otro.

La regla de supervivencia es sencilla: una celda está viva si exactamente tres vecinos suyos están vivos. Para comenzar, usamos números pseudoaleatorios como el estado inicial[2]

2. Objetivo

El objetivo de la simulación diseña y ejecuta un experimento con por lo menos 30 réplicas para estimar la probabilidad de creación de vida dentro de 50 iteraciones (es decir, que haya celdas vivas al final de la réplica), usando niveles de 10, 15 y 20 para el tamaño de la malla y los niveles 0.2, 0.4, 0.6 y 0.8 para la probabilidad inicial de vida. Visualiza y tabula los hallazgos[2].

3. Código

En el siguiente código se utilizaron secuencias for para realizar todo el objetivo propuesto en una sola ejecución. El código base se sacó del repositorio de Elisa Schaeffer, se modificó, ya que se deben agregar diferentes tamaños de matrices, haciendo el mismo procedimiento con 30 repeticiones en diferentes niveles de probabilidad.

Código en Phyton

<https://github.com/satuelisa/Simulation/blob/master/BrownianMotion>

Código creado en Python

<https://github.com/Raullr28/Resultados/blob/main/P2/PRACTICA2.py>

```
import numpy as np
from random import random
import matplotlib.cm as cm
import matplotlib.pyplot as plt
```

Primero se definieron los datos del código y los valores de las variables:

```
dur = 50
lim = 9
seq = 0
probabilidad=(0.2, 0.4, 0.6, 0.8)
dimension=(10, 15, 20)
repeticiones=(30)
    Datos de matriz
def mapeo(pos,actual):
    fila = pos // dim
    columna = pos % dim
    return actual[fila, columna]
vecinos para cada dato, si vive o muere
def paso(pos):
    fila = pos // dim
    columna = pos % dim
    vecindad = actual[max(0, fila - 1):min(dim, fila + 2),
                      max(0, columna - 1):min(dim, columna + 2)]
    return 1 * (np.sum(vecindad) - actual[fila, columna] == 3)
```

Esta función genera un inicio aleatorio por repeticiones:

```
def nuevos_valores(dim,num):
    valores = [1 * (random() < p) for i in range(num)]
    actual = np.reshape(valores, (dim, dim))
    assert all([mapeo(x,actual) == valores[x] for x in range(num)])
    return(valores, paso, actual)
```

Se llevó a cabo el recorrido para cada probabilidad para las 3 dimensiones, la cual realizó con 30 repeticiones mediante 50 iteraciones. Se inició un nuevo ciclo aleatorio para todos los recorridos con el comando antes mencionado. Para ello, se guardaron los resultados de cada una de las probabilidades antes de graficarlas.

```
if __name__ == "__main__":
    vivieron=[]
    murieron=[]
    for p in probabilidad :
        print("Probabilidad:",p,)
        for dim in dimension:
            print(" dimensión:",dim,)
            num = dim**2
```

```

contador_viv=0
contador_mue=0
for rep in range(repeticiones):
    valores, paso, actual = nuevos_valores(dim,num)
    #genera valores iniciales distintos por rep

    for iteracion in range(dur):
        valores = [paso(x) for x in range(num)]
        vivos = sum(valores)
        if vivos == 0:
            contador_mue += 1
            break; # nadie vivo
        if iteracion == (dur-1):
            contador_viv += 1

        actual = np.reshape(valores, (dim, dim))
print(contador_viv)
print(contador_mue)
contador_viv=((contador_viv*100)/(rep+1))
contador_mue=((contador_mue*100)/(rep+1))
print("contador_viv",contador_viv)
print("contador_mue",contador_mue)
vivieron.append(contador_viv)
murieron.append(contador_mue)
print(vivieron)
print(murieron)

PB02=vivieron[0:3]
PB04=vivieron[3:6]
PB06=vivieron[6:9]
PB08=vivieron[9:12]
print("0.2",PB02)
print("0.4",PB04)
print("0.6",PB06)
print("0.8",PB08)
separacion = np.arange(3)
plt.plot(separacion,PB02,label='Nivel 0.2')
plt.scatter(separacion,PB02)
plt.plot(separacion,PB04,label='Nivel 0.4')
plt.scatter(separacion,PB04)
plt.plot(separacion,PB06,label='Nivel 0.6')
plt.scatter(separacion,PB06)
plt.plot(separacion,PB08,label='Nivel 0.8')
plt.scatter(separacion,PB08)
plt.xticks(separacion , ('10', '15', '20'))
plt.ylabel('supervivencia (%)')
plt.xlabel('Dimensiónes')
plt.title('Supervivencia de población')
plt.legend()
plt.show()

```

4. Resultados

En la figura se muestra como se comporta cada dimensión con un ciclo de 30 repeticiones, el color azul es el nivel 0.2, el color amarillo es el nivel 0.4, el color verde es el nivel 0.6 y el color rojo es el nivel 0.8. Se puede observar tanto en las tablas como en la imagen, que el porcentaje de población fue muy pequeño ya que en muchos de los casos el porcentaje de vivos fue 0 .

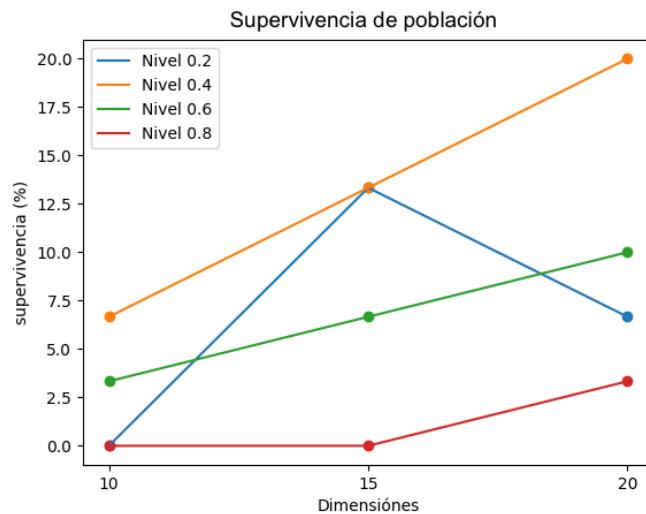


Figura 2: Gráfica tomada del repositorio de Raul L. del código de Python https://github.com/Raullr28/Resultados/blob/main/P2/Figure_1.png

Cuadro 1: Porcentajes obtenidos del valor 0.2 (color azul)

Dimensión	Vivos	Porcentaje de vivos	Muertos	Porcentaje de muertos
10	1	3.33333	29	26.66666
15	7	23.3333	23	76.66666
20	1	3.3333	29	26.66666

Cuadro 2: Porcentajes obtenidos del valor 0.4 (color naranja)

Dimensión	Vivos	Porcentaje de vivos	Muertos	Porcentaje de muertos
10	0	0.0	30	1000.0
15	1	3.3333	29	26.66666
20	6	20	24	80

Cuadro 3: Porcentajes obtenidos del valor 0.6 (color verde)

Dimensión	Vivos	Porcentaje de vivos	Muertos	Porcentaje de muertos
10	0	0.0	30	1000.0
15	2	6.66666	28	93.33333
20	8	26.66666	20	73.33333

Cuadro 4: Porcentajes obtenidos del valor 0.8 (color rojo)

Dimensión	Vivos	Porcentaje de vivos	Muertos	Porcentaje de muertos
10	1	3.3333	29	26.66666
15	0	0.0	30	1000.0
20	0	0.0	30	1000.0

5. Conclusión

Se realizó la segunda tarea correctamente, obteniendo una gráfica donde se observa el comportamiento de los diferentes niveles con un valor de 0.2, 0.4, 0.6 y 0.8 contra el porcentaje de supervivencia, el cual muestra ser un porcentaje muy bajo de población viva al final de todos los recorridos.

Con ello se demuestra que el valor con menos población viva fue el de el nivel más grande (0.8), en este caso se volvió a realizar el experimento con una repetición más grande, lo que generó una población igual de pequeña que con la repetición de 30.

De esta forma, se demostró que los niveles más pequeños tienen más probabilidad de tener una mayor población, al ser comparados con los niveles más grandes, en los cuales no importa si hay más repeticiones o menos repeticiones, su población es casi nula.[1]

Referencias

- [1] Raul. L. Práctica 2: autómata celular. 2022.
- [2] E. Schaaeffer. Práctica 2: autómata celular. 2022. URL <https://satuelisa.github.io/simulation/p2.html>.

Práctica 3: Teoría de colas

Raul L.

1 de marzo de 2022

1. Introducción

La teoría de colas es un área de las matemáticas que estudia el comportamiento de líneas de espera. Los trabajos que están esperando ejecución en un cluster esencialmente forman una línea de espera. Medidas de interés que ayudan caracterizar el comportamiento de una línea de espera incluye, el tiempo total de ejecución. En esta práctica se estudiará el efecto del orden de ejecución de trabajos y el número de núcleos utilizados en esta medida[1].

2. Objetivo

Examinar las diferencias en los tiempos de ejecución variando algunos o todos de los siguientes aspectos:

El orden de los números.

La cantidad de núcleos asignados al cluster.

La variante de la rutina para determinar si un número es primo

Aplicando pruebas estadísticas adecuadas y visualización científica clara e informativa. [1].

3. Código

Con el siguiente código se examinó como las diferencias en los tiempos de ejecución de los diferentes ordenamientos cambian cuando se varía el número de núcleos asignados al clúster, utilizando como datos de entrada diferentes procesos para obtener primos grandes. Además el programa hace un análisis estadístico para determinar si tienen una relación significativa o no, asimismo, este se graficó en barras horizontales .

El código base se sacó del repositorio de Elisa Schaeffer.

Código en Phyton

<https://github.com/satuelisa/Simulation/blob/master/QueuingTheory/fixedshuffle.py>

Código creado en Python

<https://github.com/Raullr28/Resultados/tree/main/P3>

Iniciamos definiendo dos formas de encontrar números, teniendo el método para encontrar números primos se establecen los 3 parámetros de recorrido, las repeticiones y los núcleos presentes.

```

from math import ceil, sqrt
def primo(n):# algoritmo para encontrar los numeros primos
    if n < 3:
        return True
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

def primo2(n1):
    if n1 < 4:
        return True
    if n1 % 2 == 0:
        return False
    for i in range(3, n1 - 1, 2):
        if n1 % i == 0:
            return False
    return True

```

Figura 1: Recorte del código de Raul L. del código de Python <https://github.com/Raullr28/Resultados/tree/main/P3>

Continuamos con un ciclo donde entran en juego los parámetros anteriormente dados, se hace un recorrido con el primer método para encontrar números primos utilizando los 3 parámetros de recorrido, al igual que con el segundo método para encontrar números primos lo cual genera un ciclo completo. El ciclo se repite dependiendo el numero de núcleos de cada computadora.

```

tiempos = {"ot1": [], "it1": [], "at1": []}
tiempos2 = {"ot2": [], "it2": [], "at2": []}
for x in range(1, cores-1):
    with multiprocessing.Pool(processes = x ) as pool:
        for r in range(replicas):
            for r in range(replicas):
                t = time()
                pool.map(primo1, original)
                tiempos["ot1"].append(time() - t)
                t = time()
                pool.map(primo1, invertido)
                tiempos["it1"].append(time() - t)
                t = time()
                pool.map(primo1, aleatorio)
                tiempos["at1"].append(time() - t)
                t = time()
                pool.map(primo2, original)
                tiempos2["ot2"].append(time() - t)
                t = time()
                pool.map(primo2, invertido)
                tiempos2["it2"].append(time() - t)
                t = time()
                pool.map(primo2, aleatorio)
                tiempos2["at2"].append(time() - t)
            for tipo in tiempos:
                print('Con la cantidad de núcleos de: ', x)
                print(describe(tiempos[tipo]),tipo)
                print('')
            for tipo in tiempos2:
                print('Con la cantidad de núcleos de: ', x)
                print(describe(tiempos2[tipo]),tipo)
                print('')

```

Figura 2: Recorte del código de Raul L. del código de Python <https://github.com/Raullr28/Resultados/tree/main/P3>

Se utilizó un sistema estadístico para relacionar si existía una dependencia entre los valores dados en el ciclo.

```
stat, p = pearsonr(tiempos["it1"],tiempos2["it2"])
print('stat=%3f, p=%3f' % (stat, p))
if p > 0.05:
    print('Probably independent')
else:
    print('Probably dependent')

stat, p = pearsonr(tiempos["ot1"],tiempos2["ot2"])
print('stat=%3f, p=%3f' % (stat, p))
if p > 0.05:
    print('Probably independent')
else:
    print('Probably dependent')

stat, p = pearsonr(tiempos["at1"],tiempos["at1"])
print('stat=%3f, p=%3f' % (stat, p))
if p > 0.05:
    print('Probably independent')
else:
    print('Probably dependent')
```

Figura 3: Recorte del código de Raul L. del código de Python <https://github.com/Raullr28/Resultados/tree/main/P3>

4. Resultados

En la figura se muestra el tiempo que transcurrió el código al procesar las diferentes formas en encontrar los números primos, para ello, se varió el número de núcleos.

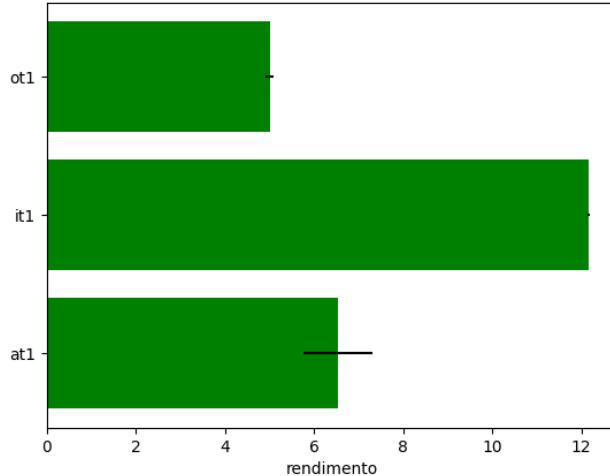


Figura 4: Gráfica tomada del repositorio de Raul L. del código de Python https://github.com/Raullr28/Resultados/blob/main/P3/Figure_1.png

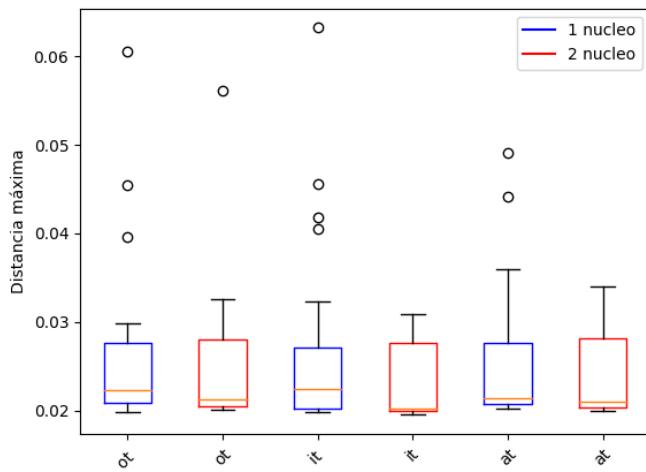


Figura 5: Gráfica tomada del repositorio de Raul L. del código de Python https://github.com/Raullr28/Resultados/blob/main/P3/Figure_2.png

5. Conclusión

Se creó un código que puede variar el número de núcleos para la correcta realización de la práctica 3, así como también se analizaron diferentes formas de encontrar los números primos, para ello, se midió el tiempo en cada procedimiento, utilizando un análisis estadístico, con el cual se obtuvo una relación entre los números de núcleos y el tiempo realizado por cada proceso, lo cual se puede observar en la gráfica de líneas.

Referencias

- [1] E. Schaeffer. Práctica 3: teoría de colas. 2022. URL <https://satuelisa.github.io/simulation/p3.html>.

Práctica 4: diagramas de Voronoi

Raul L.

8 de marzo de 2022

1. Introducción

El tema de esta cuarta práctica tiene su importancia en las matemáticas puras y en las ciencias aplicadas como por ejemplo, en la ciencia de materiales. Se toma de un espacio bidimensional una zona con medidas conocidas que contiene k puntos semillas p_i representados por sus coordenadas (x_i, y_i) lo que se busca es dividir esa zona en regiones llamadas celdas de Voronoi de tal forma que todos los puntos que pertenecen a la región de p_i estén más cerca de esa semilla que de cualquier otra.

El modelo matemático en si es continuo, es decir, las coordenadas son números reales, pero nosotros lo vamos a discretizar en esta práctica [1].

2. Objetivo

Examinar el efecto de la tasa n versus k (por lo menos tres niveles de la densidad de semillas), en la probabilidad de que una segunda grieta llegue a tocar una primera grieta (es decir, fracturando la pieza dos veces con posiciones iniciales generadas independientemente al azar, sobre varias réplicas), visualizando los resultados y aplicando métodos estadísticos [1].

3. Código

En el siguiente código se realizó un estudio cuantificando la probabilidad que hay para que dos grietas se intercepten, estas se muestran de dos diferentes colores (negro y blanco), para ello, el pixel de color rojo muestra el punto exacto de interferencia. Para realizar un concreto estudio, se varió el numero de semillas en 3 diferentes números de semillas dejando fijo el tamaño de la celda, esto garantiza realizar un estudio sistemático eficiente, el cual se representa en forma de gráfica de barras horizontales y verticales, así como también, se muestran las imágenes donde las grietas hicieron contacto con los diferentes números de semilla.

Código en Python

<https://github.com/satuelisa/Simulation/blob/master/VoronoiDiagrams/fracture.py>

Código creado en Python

https://github.com/Raullr28/Resultados/blob/main/P4/codigo_celdas.py

```

1  sem=8, 40, 120
2  porcentaje=[]
3  for k in sem[:]:
4      print(" semillas:",k,)
5      n, semillas = 80, []
6      for s in range(k):
7          while True:
8              x, y = randint(0, n - 1), randint(0, n - 1)
9              if (x, y) not in semillas:
10                  semillas.append((x, y))
11                  break
12
13  celdas = [celda(i) for i in range(n * n)]
14  voronoi = Image.new('RGB', (n, n))
15  vor = voronoi.load()
16  c = sns.color_palette("Set3", k).as_hex()
17  for i in range(n * n):
18      vor[i % n, i // n] = ImageColor.getrgb(c[celdas.pop(0)])
19  limite, vecinos = 10, []# se modiflico para que produzca mas grietas
20  for dx in range(-1, 2):
21      for dy in range(-1, 2):
22          if dx != 0 or dy != 0:
23              vecinos.append((dx, dy))

```

Código 1: Representa la automatización para variar el número de semillas que aparecen.

```

1 def propaga(replica):
2     prob, dificil = 0.9, 0.8
3     grieta = voronoi.copy()
4     g = grieta.load()
5     (x, y) = inicio()
6     largo = 0
7     negro = (0, 0, 0)
8     while True:
9         g[x, y] = negro
10        largo += 1
11        frontera, interior = [], []
12        for v in vecinos:
13            (dx, dy) = v
14            vx, vy = x + dx, y + dy
15            if vx >= 0 and vx < n and vy >= 0 and vy < n: # existe
16                if g[vx, vy] != negro: # no tiene grieta por el momento
17                    if vor[vx, vy] == vor[x, y]: # misma celda
18                        interior.append(v)
19                    else:
20                        frontera.append(v)
21        elegido = None
22        if len(frontera) > 0:
23            elegido = choice(frontera)
24            prob = 1
25        elif len(interior) > 0:
26            elegido = choice(interior)
27            prob *= dificil
28        if elegido is not None:
29            (dx, dy) = elegido
30            x, y = x + dx, y + dy
31        else:
32            break # ya no se propaga
33        if largo >= limite: # aqui decide que imprima las mayores a 80 el limite
34            visual = grieta.resize((10 * n, 10 * n))
35    return (largo, grieta)

```

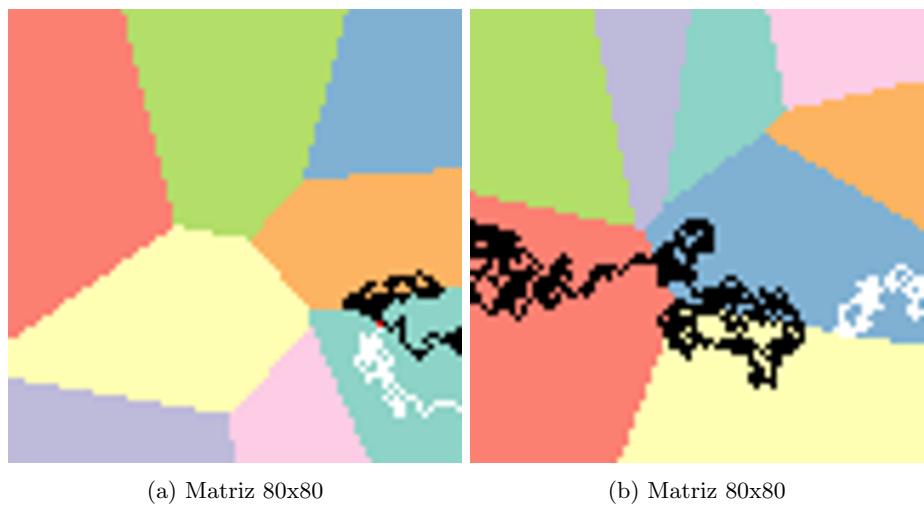
Código 2: Representa un comando para generar la grieta.

```

1 rep=400
2 contacto=[]#guarda cuantas veces chocaron
3 for r in range(rep): # pruebas sin paralelismo
4     largo, nueva_grieta =propaga(r)# para grieta 1 color negro
5     if largo > limite:
6         largo2 =propaga2(r,nueva_grieta,contacto)#para grieta 2 color blanca
7         print(contacto)
8         probabilidad= ((len(contacto))/rep)*100#convierte a porcentaje
9         porcentaje.append(probabilidad)
10        print("Probabilidad de contacto entre grietas es:",probabilidad, "%")

```

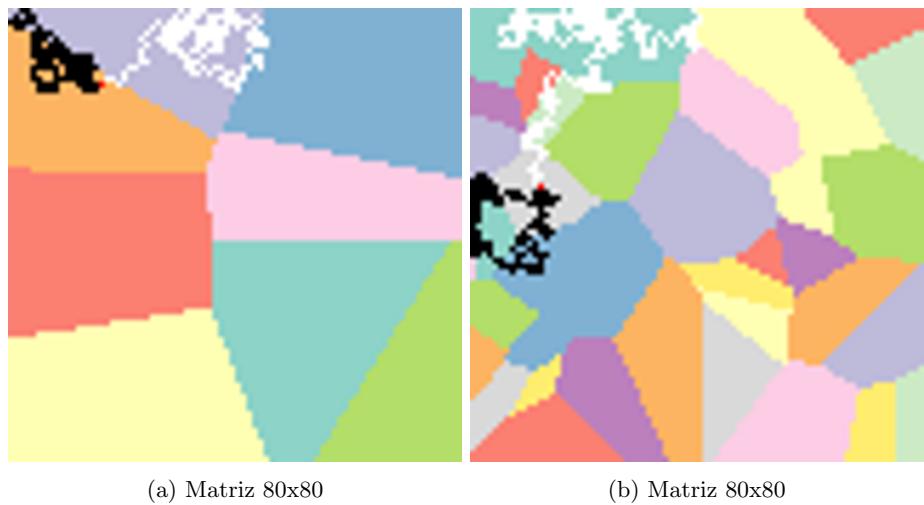
Código 3: Representa el lugar donde hacen contacto las grietas.



(a) Matriz 80x80

(b) Matriz 80x80

Figura 1: imagen de unión de grietas con igual número de semilla.



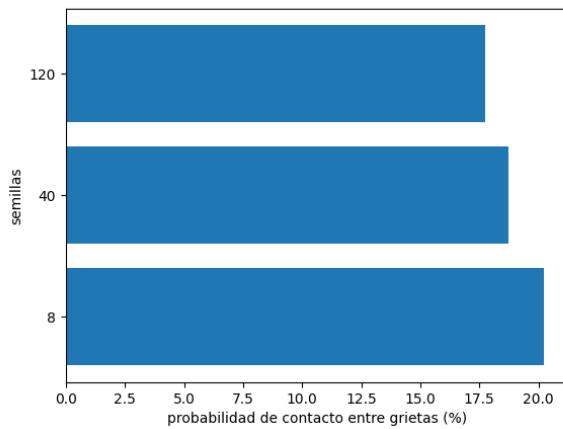
(a) Matriz 80x80

(b) Matriz 80x80

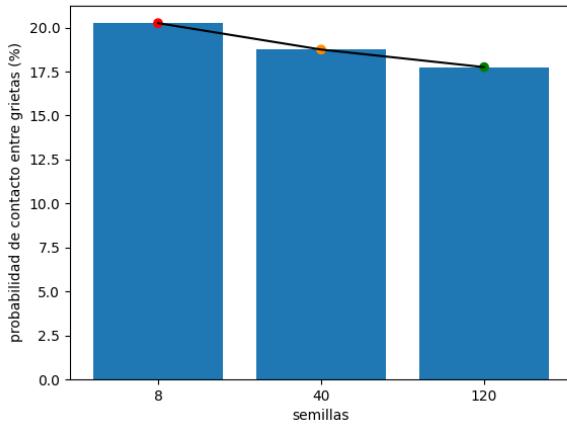
Figura 2: imagen de unión de grietas con diferente número de semilla.

4. Resultados

En las figuras se muestra el porcentaje de intercepción que existió en cada uno de los diferentes números de semillas, para poder dar un resultado estadístico así, se realizó el experimento con un número alto de repeticiones lo cual nos da las gráficas siguientes.



(a)

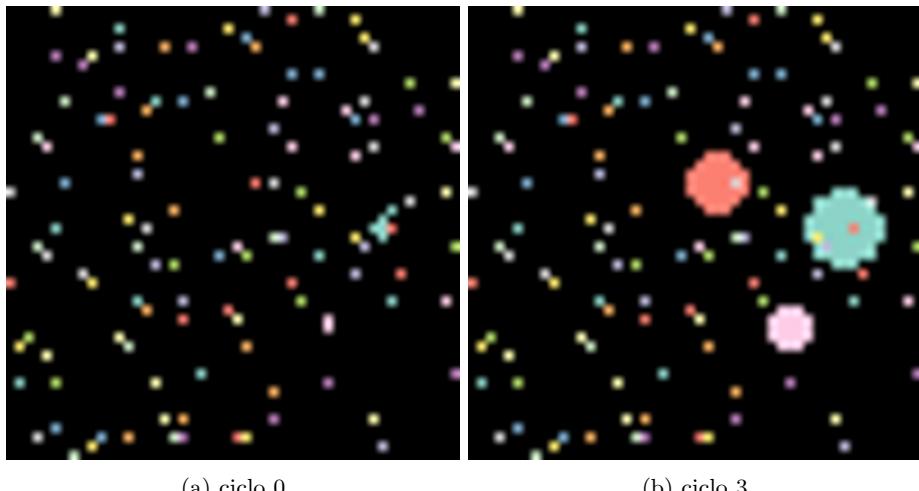


(b)

Figura 3: Grafica estadística de porcentaje.

5. Reto 1

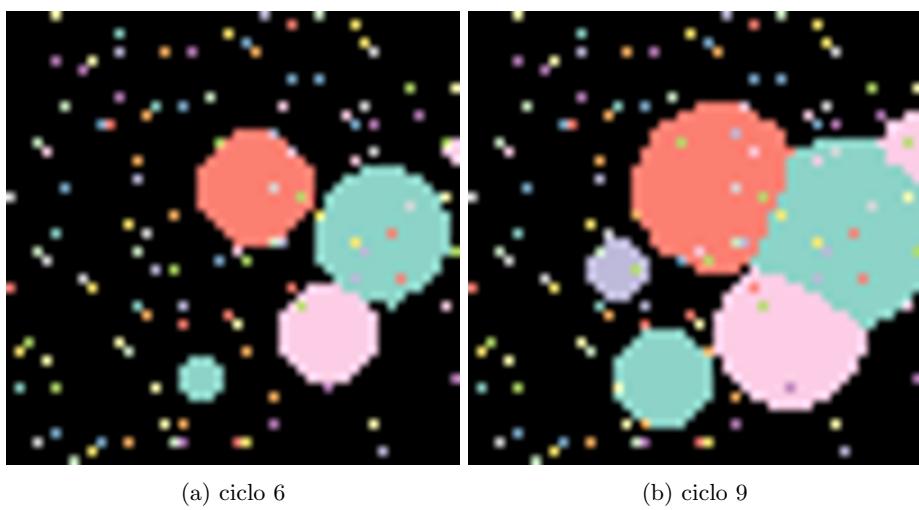
El primer reto es crecer las celdas dinámicamente alrededor de semillas de tal forma que las semillas aparecen al azar en distintas iteraciones y crecen con una tasa exponencialmente distribuida (variable entre núcleos pero constante para un núcleo específico) hasta toparse con las demás celdas, así como se muestra en la animación. Examina los cambios producidos en el fenómeno de propagación de grietas que esta nueva forma de crear las celdas provoca, ya que las semillas resultan en celdas de tamaños distintos según su edad y su tasa, además del efecto de la posición relativa a las demás semillas.



(a) ciclo 0

(b) ciclo 3

Figura 4: Imagen de crecimiento aleatorio de semillas.



(a) ciclo 6

(b) ciclo 9

Figura 5: Imagen de crecimiento aleatorio de semillas.

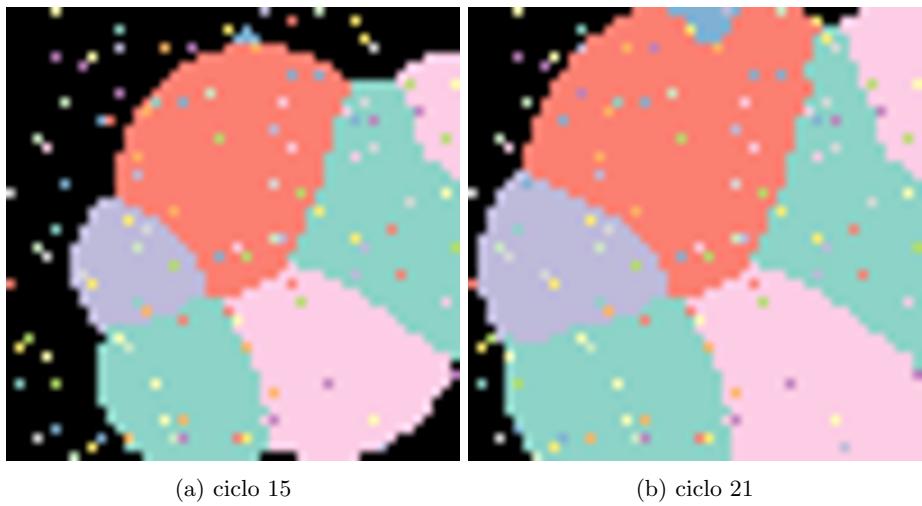


Figura 6: Imagen de crecimiento aleatorio de semillas.

6. Resultados

Se agregó una función donde van creciendo las semillas, cuando crece una semilla se agrega una probabilidad de que otra semilla naciera y generando un análisis de probabilidad donde se determinó el porcentaje que aparezca otra semilla.

7. Conclusión

Se demostró que aumentando el número de repeticiones se puede tener una proporcionalidad en los resultados, demostrando así, un porcentaje alto de intercepción en las grietas con un pequeño número de semillas y por otro lado, muestra un porcentaje más pequeño con un número más pequeño de semillas.

Referencias

- [1] E. Schaeffer. Práctica 4: diagramas de voronoi. 2022. URL <https://satuelisa.github.io/simulation/p4.html>.

Práctica 5: método Monte-Carlo

Raul L.

16 de marzo de 2022

1. Introducción

El método Monte Carlo es idóneo para situaciones en las cuales algún valor o alguna distribución no se conoce y resulta complicado de determinar de manera analítica. Siguiendo los ejemplos de Kurt, paralelicemos algunos casos sencillos en esta práctica. Supongamos que se ocupa conocer el valor de una integral que no se nos antoja resolver para nada, como, por ejemplo

$$\int_3^7 f(x) dx$$

para

$$f(x) = \frac{1}{\exp(x) + \exp(-x)}$$

Por suerte, $2f(x)/\pi$ es una función de distribución válida, ya que

$$\int_{-\infty}^{\infty} \frac{2}{\pi} f(x) dx = 1$$

Este hecho nos permite generar números pseudoaleatorios con la distribución $g(x) = 2 f(x) / \pi$, así estimar

$$\int_3^7 g(x) dx$$

y de ahí normalizar el estimado para que sea

$$\int_3^7 f(x) dx$$

Se puede comparar con el resultado aproximado de Wolfram Alpha, 0.048834, para llegar a una satisfacción que no estemos completamente mal. Se debe notar que cada ejecución dará un resultado distinto ya que es una muestra pseudoaleatoria [?].

2. Objetivo

Estudia estadísticamente la convergencia de la precisión del estimado del integral con en método Monte Carlo, comparando con el valor producido por Wolfram Alpha, en términos del (1) error absoluto, (2) error cuadrado y (3) cantidad de decimales correctos, aumentando el tamaño de muestra [2].

3. Código

Para este código se utilizaron un numero alto de repeticiones repeticiones para cada pedazo, esta cantidad fue con la que se observó que se podía llegar a cuatro decimales de precisión. No se pudo llegar a observar los resultados con más repeticiones por falta de poder de procesamiento.

Código en Python

<https://github.com/satuelisa/Simulation/blob/master/MonteCarlo/rng.py>

Código creado en Python

https://github.com/Raullr28/Resultados/blob/main/P4/codigo_celdas.py

```
vg = np.vectorize(g)
X = np.arange(-8, 8, 0.05) # ampliar y refinar
Y = vg(X) # mayor eficiencia
correcto= 0.04883411112604931084064237
print("correcto",correcto)
desde = 2.96
hasta = 7
pdz = (700,5000,10000,80000,300000,1000000),#5000000)numero de n para estimar valor
repeticiones = 200
result = {"Estimado": [], "abs": [], "cuad": [], "dec": []}
dec = {"cero": [], "uno": [], "dos": [], "tres": [], "cuatro": [], "cinco": []}
ABS, CUAD = [], []
for pedazo in pdz:
    print("##### pedazos:",pedazo,"#####")
    absoluto=[]
    cuadrado=[]
    dec_corr=[]
    for i in range(repeticiones):
        generador = GeneralRandom(np.asarray(X), np.asarray(Y))
        V = generador.random(pedazo)[0]
        montecarlo = ((V >= desde) & (V <= hasta))
        integral = sum(montecarlo) / (pedazo)
        estimado=(pi / 2) * integral
        absoluto.append(abs(correcto - estimado))
        cuadrado.append(((correcto - estimado)**2))
        dec_corr.append(decimales(correcto, estimado))#regresa cuantos decimales hubo semejantes
```

Código 1: Representa la automatización para variar el pedo.

```

def g(x):
    return (2 / (pi * (exp(x) + exp(-x)))))

def decimales(real, obtenido):
    contador=-2 #omite el 0 y el punto . del conteo
    real, obtenido= (str(real)), (str(obtenido))# convierte para leer cada valor
    obtenido=obtenido[:len(real)]# recorte para mismo tamaño
    largo=min([len(real),len(obtenido)])
    for i in range(largo):
        if real[i] == obtenido[i]:
            contador=contador+1
        else:
            break
    return(contador)

```

Código 2: Representación función decimales.

4. Resultados

En las figuras se muestra la probabilidad que existió en cada una de los diferentes partes con un número de repeticiones de 100, esto se realizó para poder dar un resultado estadístico ya que con esto podíamos comparar la probabilidad de encontrar los decimales correctos en cada una de las diferentes partes.

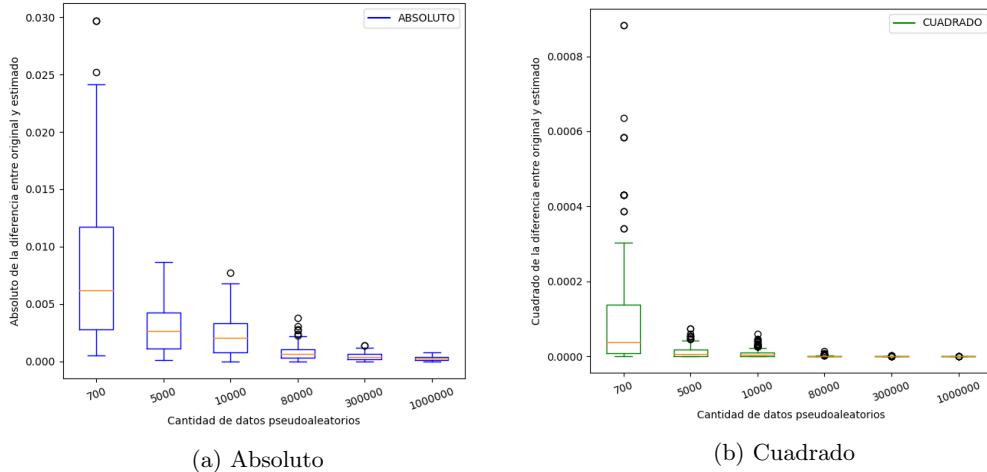


Figura 1: Gráfica comparativa.

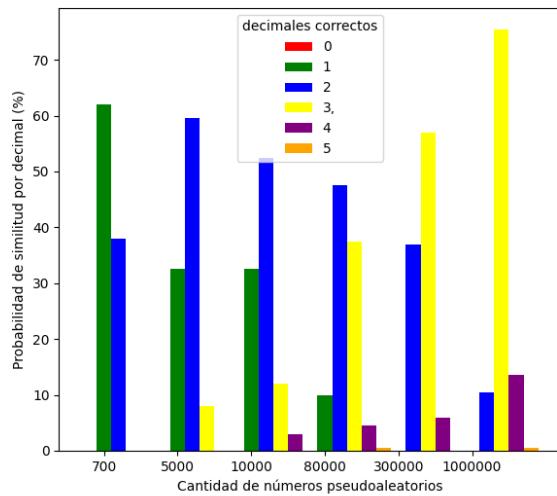


Figura 2: Gráfica estadística de porcentaje.

5. Reto 1

El primer reto es realizar lo mismo para la estimación del valor de Π de Kurt [1].

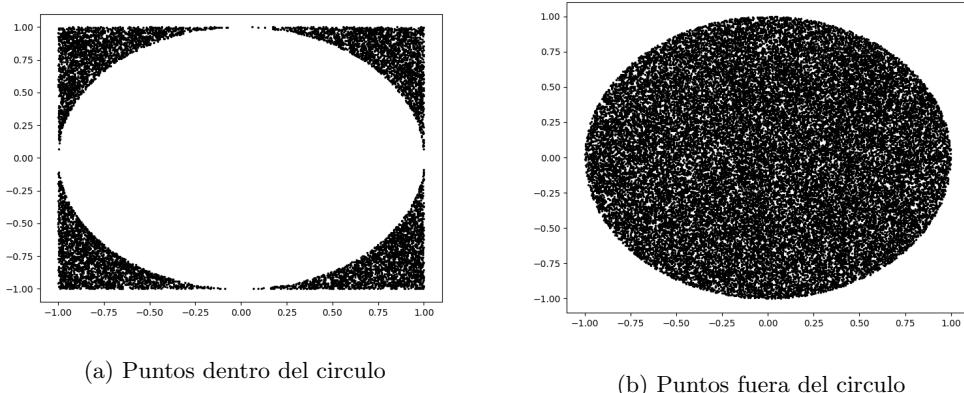


Figura 3: Imagen de crecimiento aleatorio de puntos.

6. Resultados

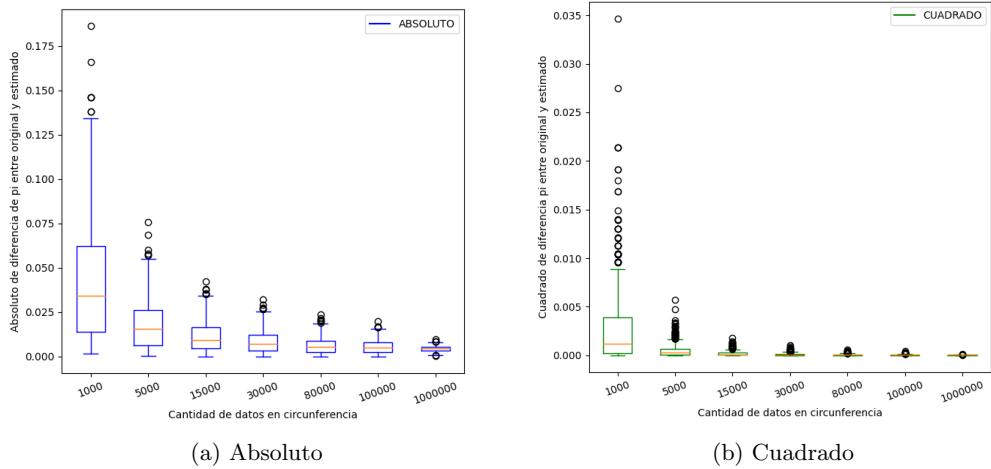


Figura 4: Gráfica comparativa.

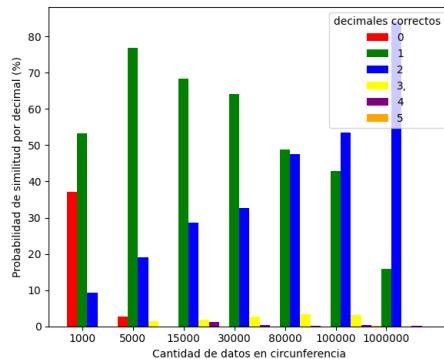


Figura 5: Gráfica estadística de porcentaje.

7. Conclusión

Se demostró que aumentando el numero de repeticiones se podría llegar a tener un porcentaje alto de números decimales correctos.

Referencias

- [1] W. Kurt. 6 neat tricks with monte carlo simulations — count bayesie; probably a probability blog. 2015. URL <https://www.countbayesie.com/blog/2015/3/3/6-amazing-trick-with-monte-carlo-simulations>.
- [2] E. Schaeffer. Práctica 4: diagramas de voronoi. 2022. URL <https://satuelisa.github.io/simulation/p5.html>.

Práctica 5: Sistema Multiagente

Raul L.

22 de marzo de 2022

1. Introducción

Un sistema multiagente es un poco como un autómata celular: hay un conjunto de entidades con estados internos que pueden observar estados de los otros y reaccionar cambiando su propio estado. La diferencia es que un sistema multiagente es un concepto más general y permite que estos agentes se muevan y varíen su vecindad, entre otras cosas. En esta práctica vamos a implementar un sistema multiagente con una aplicación en epidemiología. Los agentes podrán estar en uno de tres estados: susceptibles, infectados o recuperados, esto se conoce como el modelo SIR [1].

2. Objetivo

Estudia el efecto de contención en el sentido de que agentes infectados reduzcan su velocidad de movimiento a la mitad durante su infección. Determina con pruebas estadísticas adecuadas si este cambio produce un efecto significativo en la magnitud de la epidemia (la altura del pico en la curva del porcentaje de infectados por iteración) y en la velocidad de ella la iteración en la cual se llega por la primera vez al valor pico [1].

3. Código

Para este código se utilizó como base el código de la doctora donde se hicieron modificaciones variando la velocidad para comprobar si afectaría en el porcentaje de contagios en el pico como también se agregaron un numero de repeticiones para poder tener una estimación considerable y poder hacer una correcta prueba estadística.

Código en Python

<https://github.com/satuelisa/Simulation/blob/master/MultiAgent/movement.py>

Código creado en Python

https://github.com/Raullr28/Resultados/blob/main/P4/codigo_celdas.py

```

if a.estado == 'I':
    if div != 0:
        x = a.x + (a.dx/div)
        y = a.y + (a.dy/div)
    elif div == 0:
        x = a.x + (a.dx*div)
        y = a.y + (a.dy*div)
    else:
        x = a.x + a.dx
        y = a.y + a.dy

```

Código 1: Representa la automatización para variar la velocidad de los infectados.

```

for div in 1,2,0:
    print("##### velocidad 1/",div,"#####")
    pico, tmp= [], []
    for rep in range(repeticiones):
        agentes = pd.DataFrame()
        agentes['x'] = [uniform(0, 1) for i in range(n)]
        agentes['y'] = [uniform(0, 1) for i in range(n)]
        agentes['dx'] = [uniform(-v, v) for i in range(n)]
        agentes['dy'] = [uniform(-v, v) for i in range(n)]
        agentes['estado'] = ['S' if random() > pi else 'I' for i in range(n)]
        epidemia = []
        for tiempo in range(tmax):
            conteos = agentes.estado.value_counts()
            infectados = conteos.get('I', 0)
            epidemia.append(infectados)
            if infectados == 0:
                break

```

Código 2: Representación función cambio de velocidad.

4. Resultados

En las 3 figuras principales se muestra el comportamiento del pico con respecto a la reducción de la velocidad para demostrar como puede influir la velocidad de propagación en el pico de la pandemia, como prueba estadística se realizaron comparaciones con 30 repeticiones para cada caso de velocidad en graficas de caja-bigote.

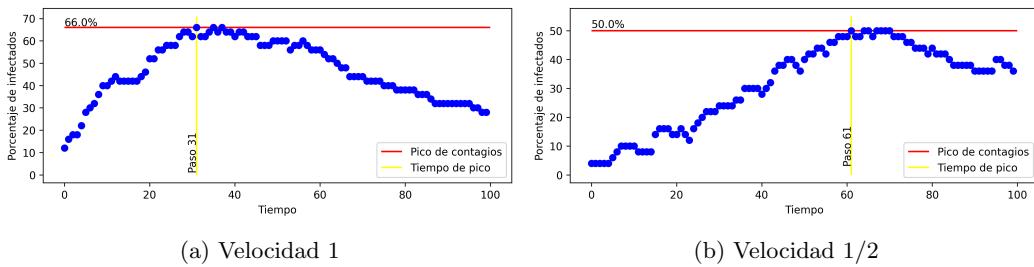


Figura 1: Gráfica de picos a diferentes velocidades.

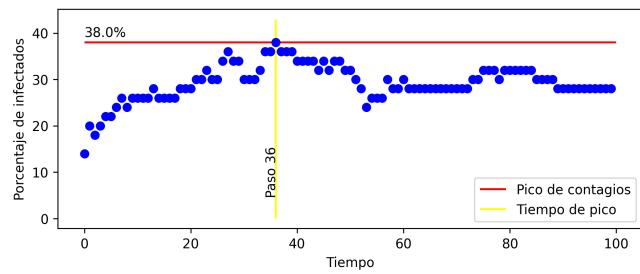


Figura 2: Velocidad nula 0.

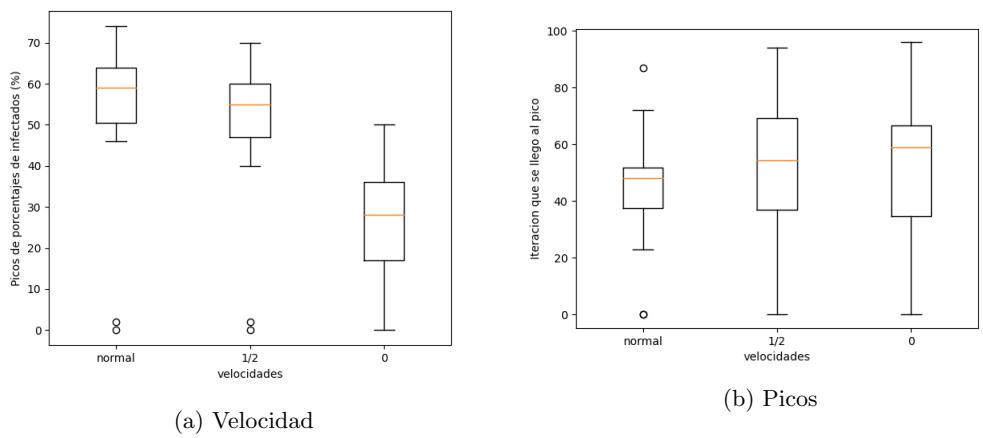
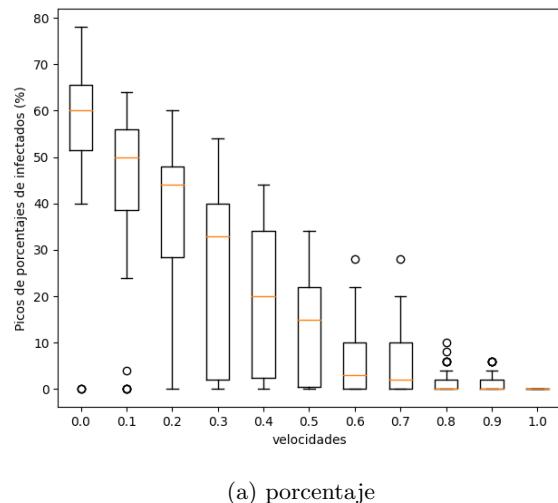


Figura 3: Gráficas Estadísticas.

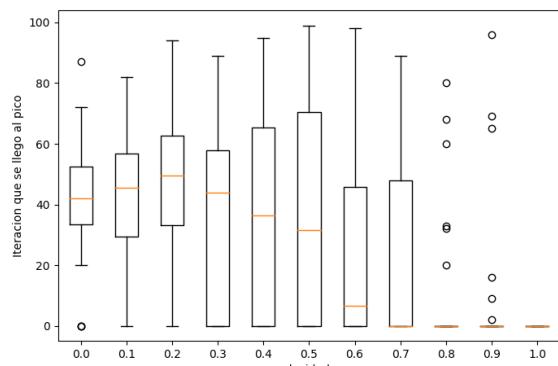
5. Reto 1

En el primer reto, vacuna con probabilidad Pv a los agentes al momento de crearlos de tal forma que están desde el inicio en el estado R y ya no podrán contagiarse ni propagar la infección. Estudia el efecto estadístico del valor de pv en (de cero a uno en pasos de 0.1) el porcentaje máximo de infectados durante la simulación y el momento (iteración) en el cual se alcanza ese máximo..

6. Resultados



(a) porcentaje



(b) Pico

Figura 4: Gráfica comparativa.

7. Conclusión

Se demostró que disminuyendo la velocidad de los puntos como a su vez vacunando, hace que se reduzca la probabilidad de infectados y hace que arde en llegar más rápido el pico.

Referencias

- [1] E. Schaeffer. Práctica 6: sistema multiagente. 2022. URL <https://satuelisa.github.io/simulation/p6.html>.

Práctica 7:Búsqueda local

Raul L.

29 de marzo de 2022

1. Introducción

En la séptima práctica implementamos una optimización heurística sencilla para encontrar máximos locales de funciones, los ejemplos siendo de Womersley [1] — los matemáticamente inclinados pueden consultar su trabajo por métodos de búsqueda más sofisticados, guiados por métodos matemáticos de mayor rigor [2].

Buscamos minimizar la función unidimensional, a partir de un punto seleccionado al azar, realizando movimientos locales. Estando en x , seleccionará al azar un $\Delta x > 0$ pequeño, calculará los valores $f((x \pm \Delta x))$ y seleccionará el menor de los dos como el siguiente valor de x . Esto se repite k veces y aquel x que produjo el menor valor de $f(x)$ se regresa como el resultado. Se realizarán n réplicas, y el menor de ellos es el resultado de la búsqueda en sí. La primera versión es sencilla, ineficiente y con una sola réplica para poder entender el comportamiento de la búsqueda y visualizarla.

2. Objetivo

La tarea se trata de maximizar algún variante de la función bidimensional ejemplo, $g(x, y)$, con restricciones tipo $-3 \leq x, y \leq 3$, con la misma técnica del ejemplo unidimensional. La posición actual es un par x, y y se ocupan dos movimientos aleatorios, Δx y Δy , cuyas combinaciones posibles proveen ocho posiciones vecino, de los cuales aquella que logra el mayor valor para g es seleccionado. Dibujado en tres dimensiones, $g(x, y)$ se ve así:[2].

3. Código

Para este código se utilizó como base el código de la doctora donde se hicieron modificaciones variando la función y los parámetros base como también se crearon los 5 puntos al mismo tiempo y se revisó con una x el mejor valor conocido.

Código en Python

<https://satuelisa.github.io/simulation/p7.html>

Código creado en Python

https://github.com/Raullr28/Resultados/blob/main/P7/practica_7.py

```

def g(x, y):
    return np.exp(-x**2) + np.exp(-y**2)

low = -6
high = -low
step = 0.20
tmax = 500

x = np.arange(low, high, step)
y = np.arange(low, high, step)
x, y = np.meshgrid(x, y)
z = np.exp(-x**2) + np.exp(-y**2)

fig = plt.figure()
ax = plt.axes(projection='3d')
s = ax.plot_surface(x, y, z, cmap=cm.coolwarm, linewidth=0, antialiased=False)
ax.xaxis.set_major_locator(LinearLocator(10))
ax.xaxis.set_major_formatter(FormatStrFormatter('%.01f'))
fig.colorbar(s, shrink=0.5, aspect=5)
plt.savefig("p7_3dinitial.png")
plt.show()

```

Código 1: Representación de la función y parámetros utilizados.

```

for iteracion in range(tmax):#entra a hacer ciclo de la particula
    #mueve particula en x+right y x-left
    deltax = uniform(0, step/5)#movimiento en x
    leftx = currx - deltax
    leftx = low if leftx < low+step else leftx #asegurar que la particula esta dentro
    rightx = currx + deltax
    rightx = high if rightx > high-step else rightx

    deltay = uniform(0, step/5)
    lefty = curry - deltay
    lefty = low if lefty < low+step else lefty
    righty = curry + deltay
    righty = high if righty > high-step else righty

    lista=[(leftx, righty),(currx, righty),(rightx, righty),(leftx, curry),(rightx, curry),(leftx, lefty)]
    v1 = g(leftx, righty)#valores evaluados en g
    v2 = g(currx, righty)
    v3 = g(rightx, righty)
    v4 = g(leftx, curry)
    v5 = g(rightx, curry)
    v6 = g(leftx, lefty)
    v7 = g(currx, lefty)
    v8 = g(rightx, lefty)
    vecinos=[v1, v2, v3, v4, v5, v6, v7, v8]
    vecino_mayor=vecinos.index(max(vecinos))# guarda la posicion del vecino mayor
    [currx, curry]=lista[vecino_mayor]#actualiza particula en posicion nueva
    if g(currx, curry) > g(bestx, besty):#Actualiza si es una mejor posicion
        [bestx, besty] = [currx, curry]

```

Código 2: Representación ciclo de la partícula.

4. Resultados

En la figura principal se muestra la función en 3D creada, en las imágenes siguientes se muestra el comportamiento de los 5 puntos al mismo tiempo marcando con una x el mejor valor en el transcurso de las 500 repeticiones modificándose cada vez que algún punto mejore su valor.

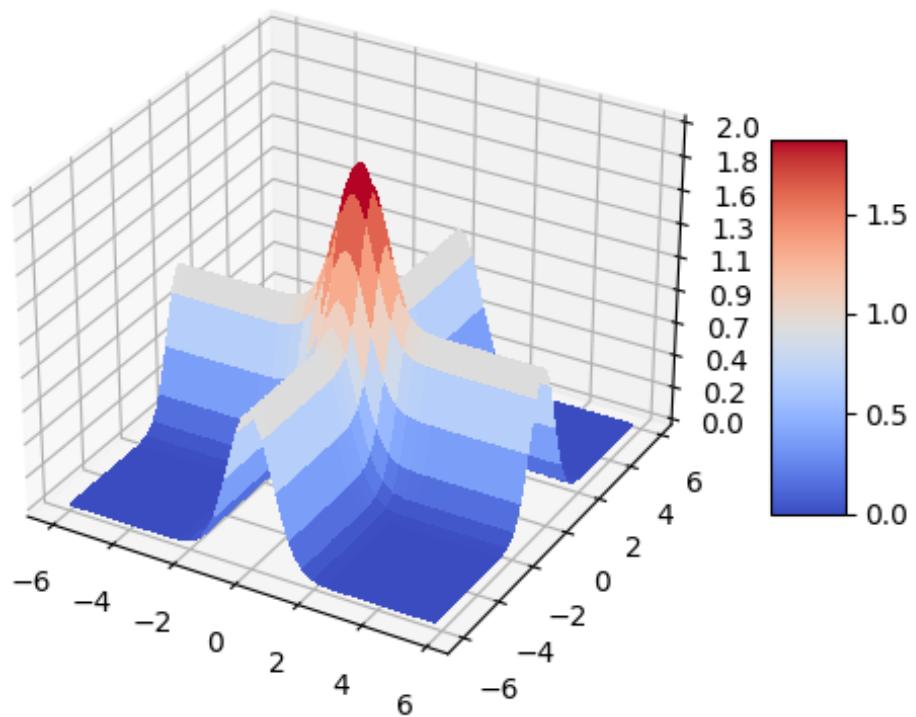


Figura 1: función 3D.

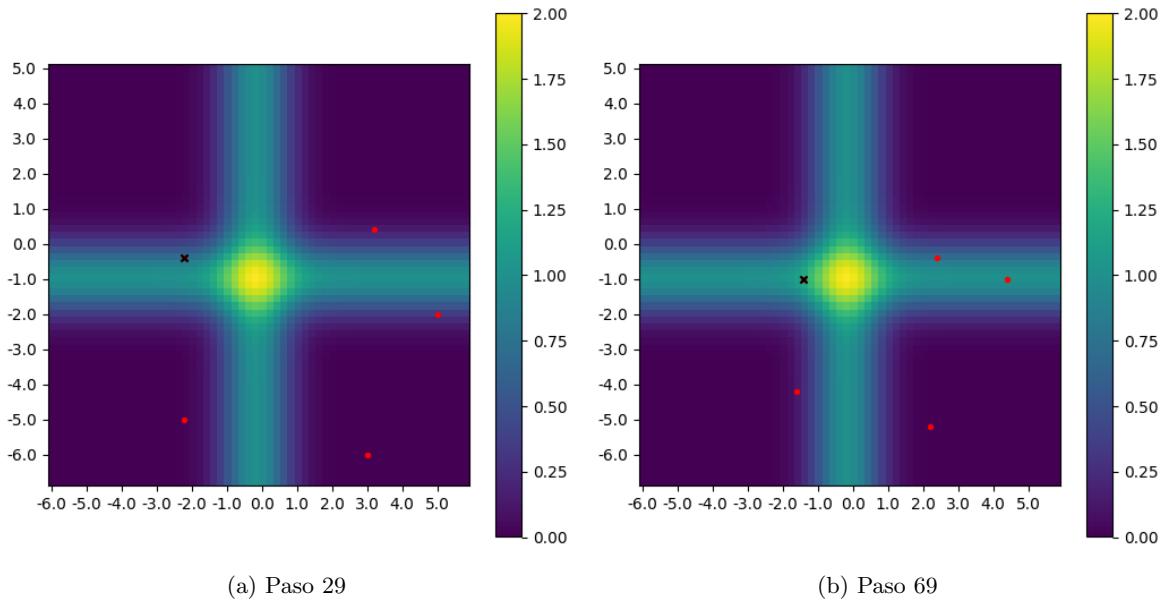


Figura 2: Comportamiento de el mejor punto.

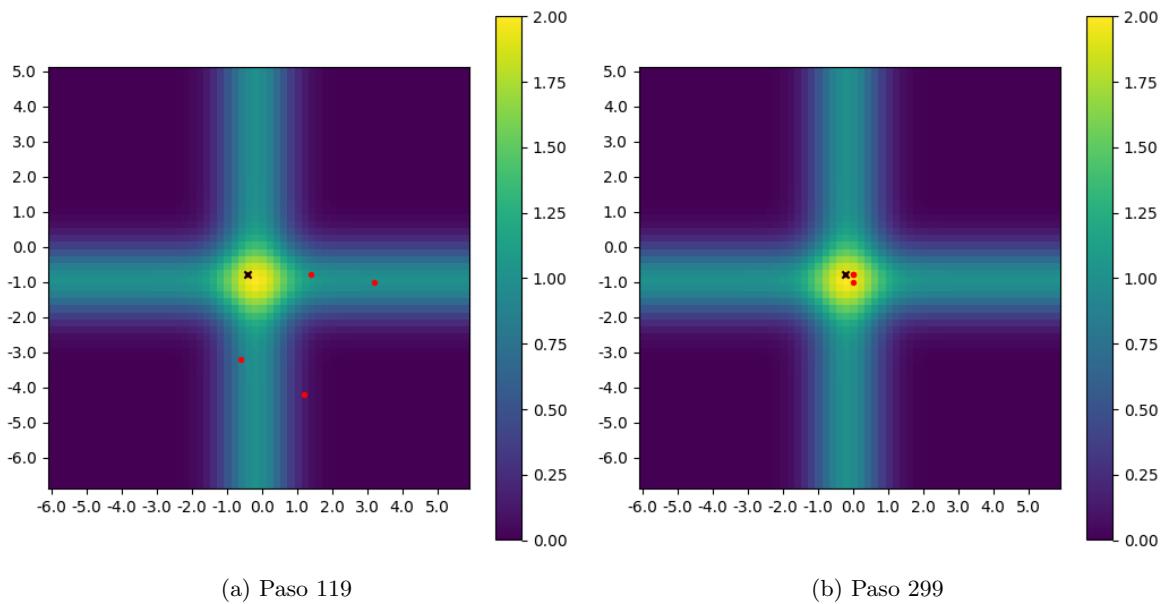


Figura 3: Comportamiento de el mejor punto.

5. Reto 1

El primer reto es cambiar la regla del movimiento de una solución x (un vector de dimensión arbitraria) a la siguiente a la de recocido simulado: para optimizar una función $f(x)$, se genera para la solución actual x un sólo vecino $x=x+\Delta x$ (algún desplazamiento local). Se calcula $\delta = f(x') - f(x)$ (para minimizar; maximizando la resta se hace al revés). Si $\delta > 0$, siempre se acepta al vecino x' como la solución actual ya que representa una mejora. Si $\delta < 0$, se acepta a x' con probabilidad $\exp(\delta/T)$ y rechaza en otro caso. Aquí T es una temperatura que decrece en aquellos pasos donde se acepta una empeora; la reducción se logra multiplicando el valor actual de T con $\xi < 1$, como por ejemplo 0,995. Examina los efectos estadísticos del valor inicial de T y el valor de ξ en la calidad de la solución, es decir, qué tan bajo (para minimizar; alto para maximizar) el mejor valor termina siendo.

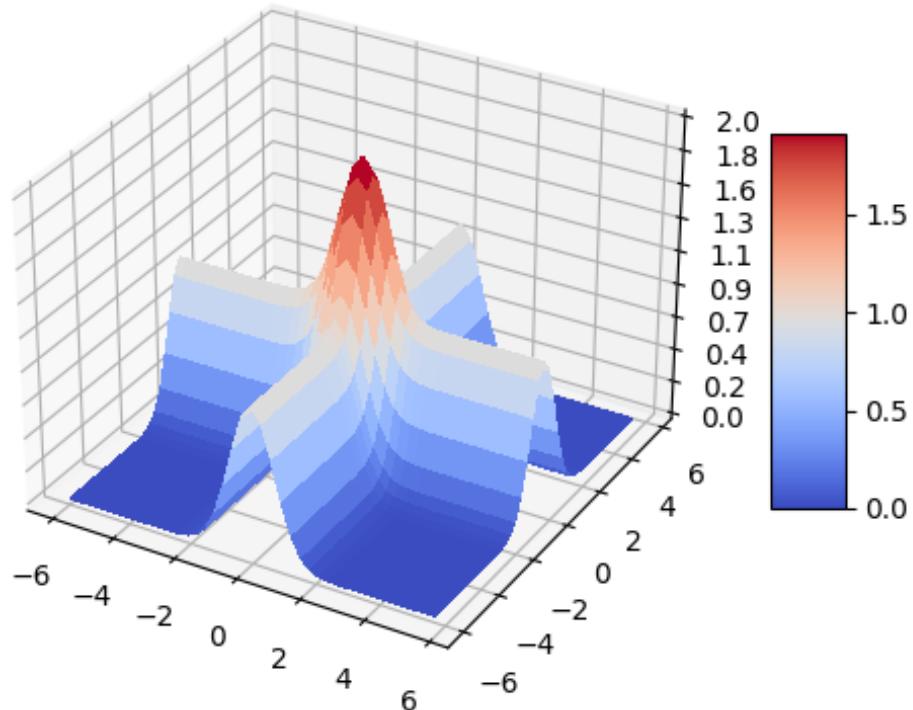


Figura 4: función 3D.

6. Resultados

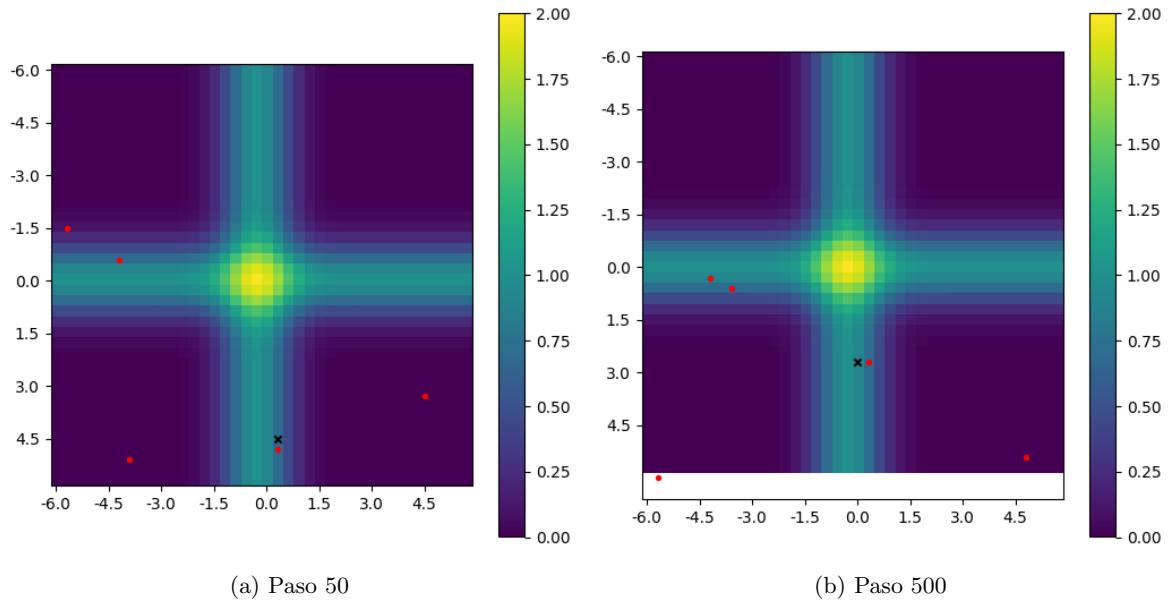


Figura 5: Comportamiento de el mejor punto.

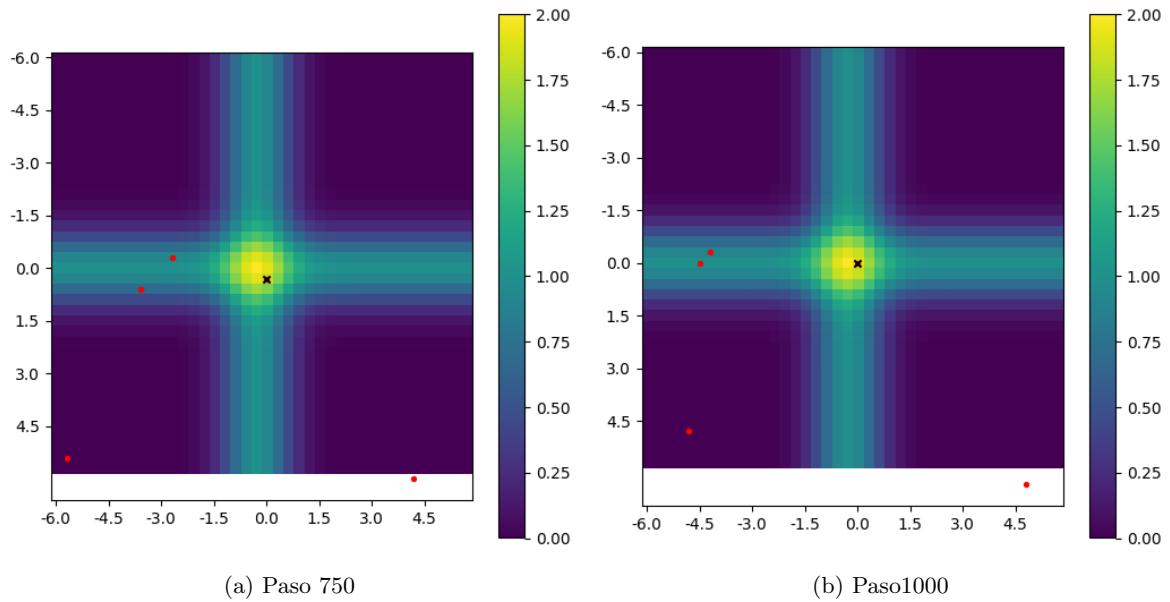


Figura 6: Comportamiento de el mejor punto.

7. Conclusión

Se demostró que en la tarea base le costo menos a los puntos maximizar las posiciones de los puntos para poder encontrar el mejor lugar mientras que en el reto 1 les costo más.

Referencias

- [1] R. Womersley. Local and global optimization. 2008. URL <https://web.maths.unsw.edu.au/~rsw/lgopt.pdf1>.
- [2] E. Schaeffer. Práctica 7: búsqueda local. 2022. URL <https://satuelisa.github.io/simulation/p7.html>.

Práctica 8: modelo de urnas

Raul L.

5 de abril de 2022

1. Introducción

La octava práctica es sobre fenómenos de coalescencia y fragmentación, donde partículas se unen para formar cúmulos y estos cúmulos se pueden volver a descomponer en fragmentos menores. Esto es relevante en muchos campos de química, como por ejemplo en el filtrado de aguas residuales, donde solamente los cúmulos de suficiente tamaño serán capturados por el filtro y hay que buscar formas para facilitar que crezcan los cúmulos de residuos para lograr su filtrado adecuado.

Vamos a suponer que tenemos una cantidad total de n partículas y que al inicio el tamaño de los k cúmulos existentes sigue la distribución normal. Para lograr esto, vamos a crear k valores de la distribución normal estándar (media cero, desviación estándar uno) y luego normalizarlos para convertirlos en enteros positivos que sumen a n [1].

2. Objetivo

Supongamos que cúmulos con c o más partículas (haciendo referencia al tamaño crítico c) son suficientemente grandes para filtrar. Estudia el efecto de la tasa n/k , usando por lo menos cinco valores distintos para ella, el porcentaje de las partículas que se lograría filtrar por iteración.[1].

3. Código

Para este código se utilizó como base el código de la doctora donde se hicieron modificaciones variando el porcentaje de k y n .

Código en Python

<https://github.com/satuelisa/Simulation/blob/master/UrnModel/aggrFrag.py>

Código creado en Python

https://github.com/Raullr28/Resultados/blob/main/P8/practica_8.py

```

densidad=[(100,1000000),(300,5000),
          (55000,1500000),(7000,35000),(30000,10000000)]
replicas=150
data=[]
for k,n in densidad:
    print("##### k,n:",k,n,"#####")
    prom_rep=[]
    antes= time()
    for rep in range(replicas):
        orig = np.random.normal(size = k)
        cumulos = orig - min(orig)
        cumulos += 1 # ahora el menor vale uno
        cumulos = cumulos / sum(cumulos) # ahora suman a uno
        cumulos *= n # ahora suman a n, pero son valores decimales
        cumulos = np.round(cumulos).astype(int) # ahora son enteros
        diferencia = n - sum(cumulos) # por cuanto le hemos fallado
        cambio = 1 if diferencia > 0 else -1

```

Código 1: Representación de la función y parámetros utilizados.

```

duracion = 50
digitos = floor(log(duracion, 10)) + 1
porcen=[]
for paso in range(duracion):
    assert sum(cumulos) == n
    assert all([c > 0 for c in cumulos])
    (tams, freqs) = np.unique(cumulos, return_counts = True)
    cumulos = []
    assert len(tams) == len(freqs)
    for i in range(len(tams)):
        cumulos += romperse(tams[i], freqs[i])

```

Código 2: Representación ciclo de la partícula.

4. Resultados

En una gráfica de violines se graficó los resultados del comportamiento de las diferentes k/n respectivamente para poder ver su comportamiento con un número alto de repeticiones para poder lograr ver mejores los violines.

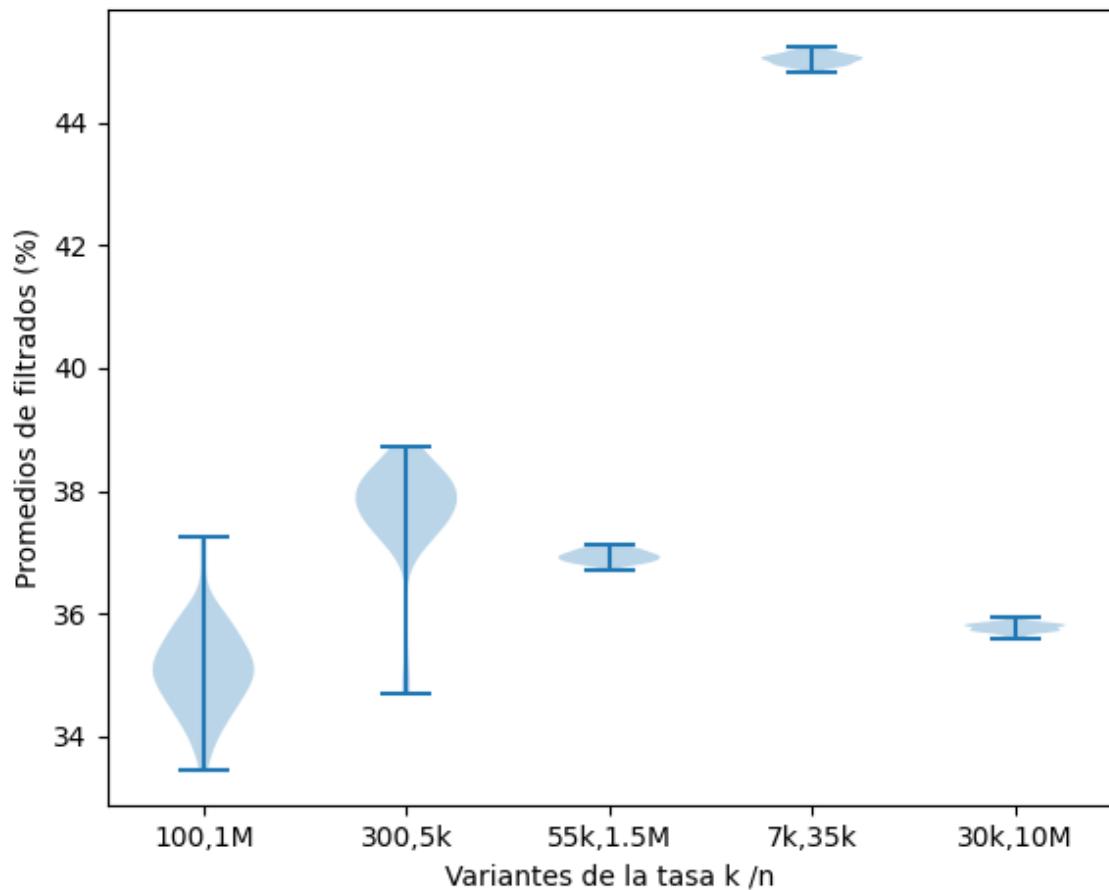


Figura 1: función 3D.

5. Reto 1

Un primer reto, determina cómo el momento idóneo de filtrado depende del valor de c . ¿Qué todo cambia y cómo si c ya no se asigna como la mediana inicial sino a un valor menor o mayor?.

6. Resultados

Para poder llevar a cabo el reto numero uno se varió el dato c , en dos nuevos valores.

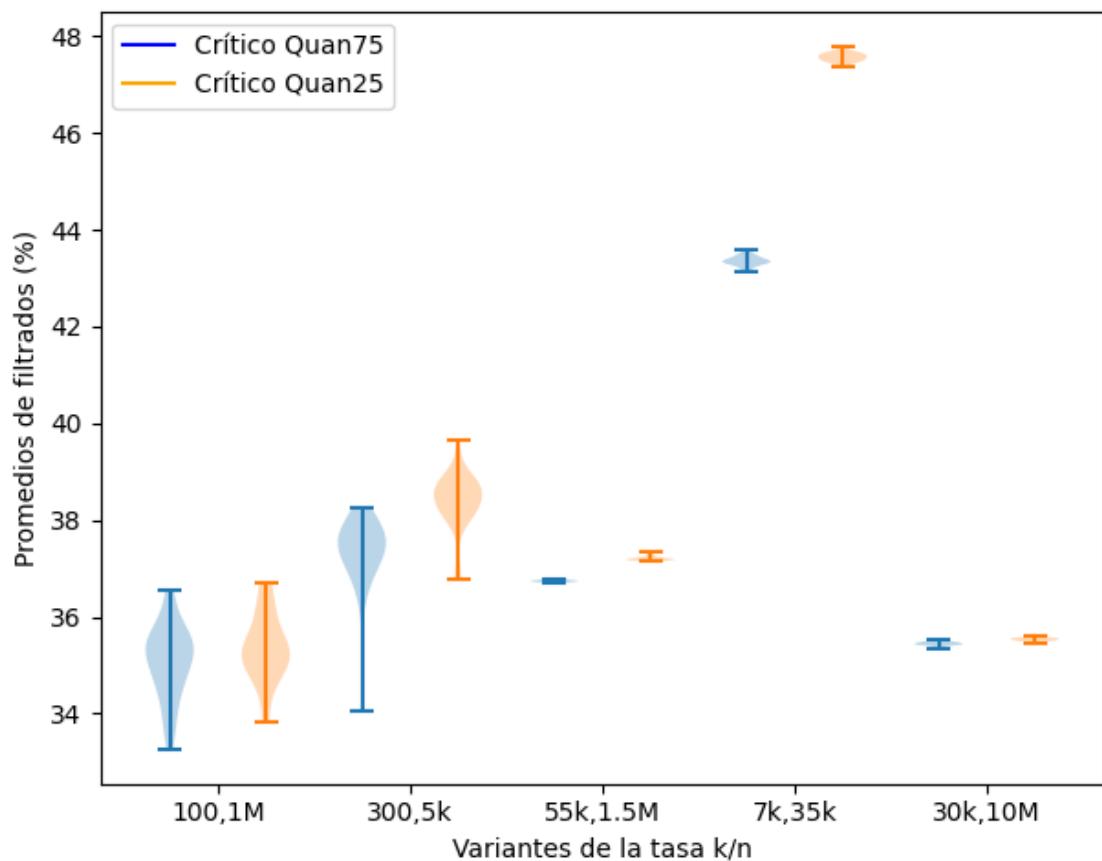


Figura 2: función 3D.

7. Reto 2

Como el segundo reto, estudia el efecto del parámetro suavizante d en el desempeño de filtrado si la meta es recuperar la mayor cantidad posible de partículas en el proceso. ¿En cuál iteración es conveniente realizar el filtrado? Incluye visualizaciones para justificar las conclusiones.

8. Resultados

demostrando tamaños de filtros encontrando el mejor tamaño para filtrar la mejor cantidad de cumulos.

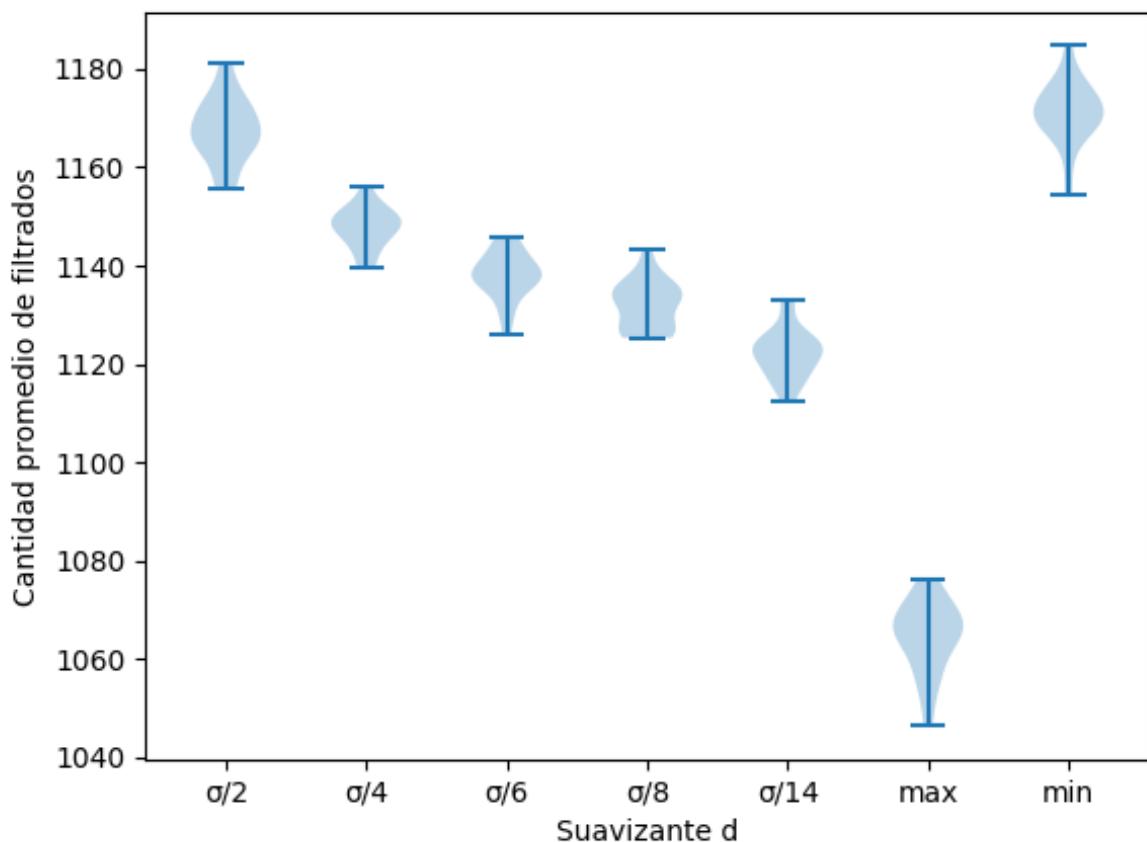


Figura 3: función 3D.

9. Conclusión

Se mostró con gráficas de violín el comportamiento de diferentes k/n , se cambió el valor de nuestra c para poder determinar mejores valores y en nuestro ultimo reto se determinó el mejor valor para nuestro filtro.. .

Referencias

- [1] E. Schaeffer. Práctica 8: modelo de urnas. 2022. URL <https://satuelisa.github.io/simulation/p8.html>.

Práctica 9: interacciones entre partículas

Raul L.

26 de abril de 2022

1. Introducción

En la novena práctica trabajamos con un modelo simplificado para los fenómenos de atracción y repulsión de física (o química, de hecho). Supongamos que contemos con n partículas que habitan un cuadro unitario bidimensional y que cada partícula tiene una carga eléctrica, distribuida independientemente e normalmente al azar entre $[-1, 1]$. Cargas de un mismo signo producirán una repulsión mientras cargas opuestas resultan en una atracción — la magnitud de la fuerza estará proporcional a la diferencia de magnitud de las cargas (mayores diferencias resultando en fuerzas mayores), y además la fuerza será inversamente proporcional a la distancia euclídea entre las partículas (éstas son reglas inventadas de interacción para efectos de demostración). Vamos a comenzar creando y posicionando las partículas, usando la distribución normal (posteriormente normalizada al cuadro unitario) para las coordenadas x, y [1].

2. Objetivo

Agrega a cada partícula una masa y haz que la masa cause fuerzas gravitacionales (atracciones) además de las fuerzas causadas por las cargas. Estudia la distribución de velocidades de las partículas y verifica gráficamente que esté presente una relación entre los tres factores: la velocidad, la magnitud de la carga, y la masa de las partículas. Toma en cuenta que la velocidad también es afectada por las posiciones.[1].

3. Código

Para este código se utilizó como base el código de la doctora donde se hicieron modificaciones.

Código en Python

<https://github.com/satuelisa/Simulation/blob/master/Particles/creation.py>

Código creado en Python

https://github.com/Raullr28/Resultados/blob/main/P9/practica%20_9.py

```

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

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

    fx = fx
    fy = fy
    return (fx, fy)

```

Código 1: Representación de la función y parámetros utilizados.

```

if __name__ == "__main__":
    n = 15
    x = np.random.normal(size = n)
    y = np.random.normal(size = n)
    c = np.random.normal(size = n)
    m = np.random.normal(size = n)
    xmax = max(x)
    xmin = min(x)
    x = (x - xmin) / (xmax - xmin) # de 0 a 1
    ymax = max(y)
    ymin = min(y)
    y = (y - ymin) / (ymax - ymin)
    cmax = max(c)
    cmin = min(c)
    c = 2 * (c - cmin) / (cmax - cmin) - 1 # entre -1 y 1
    masamax = max(m)
    masamin = min(m)
    m = 5 * ((m - masamin) / (masamax - masamin) + 0.1)
    m = np.round(m).astype(int)
    g = np.round(5 * c).astype(int)
    vel = [[0]]*n
    p = pd.DataFrame({'x': x, 'y': y, 'm':m, 'c': c,'vel':vel,'g':g})

    x = p['x']
    y = p['y']
    c = p['c']
    m = p['m']
    g = p['g']

```

Código 2: Representación ciclo de la partícula.

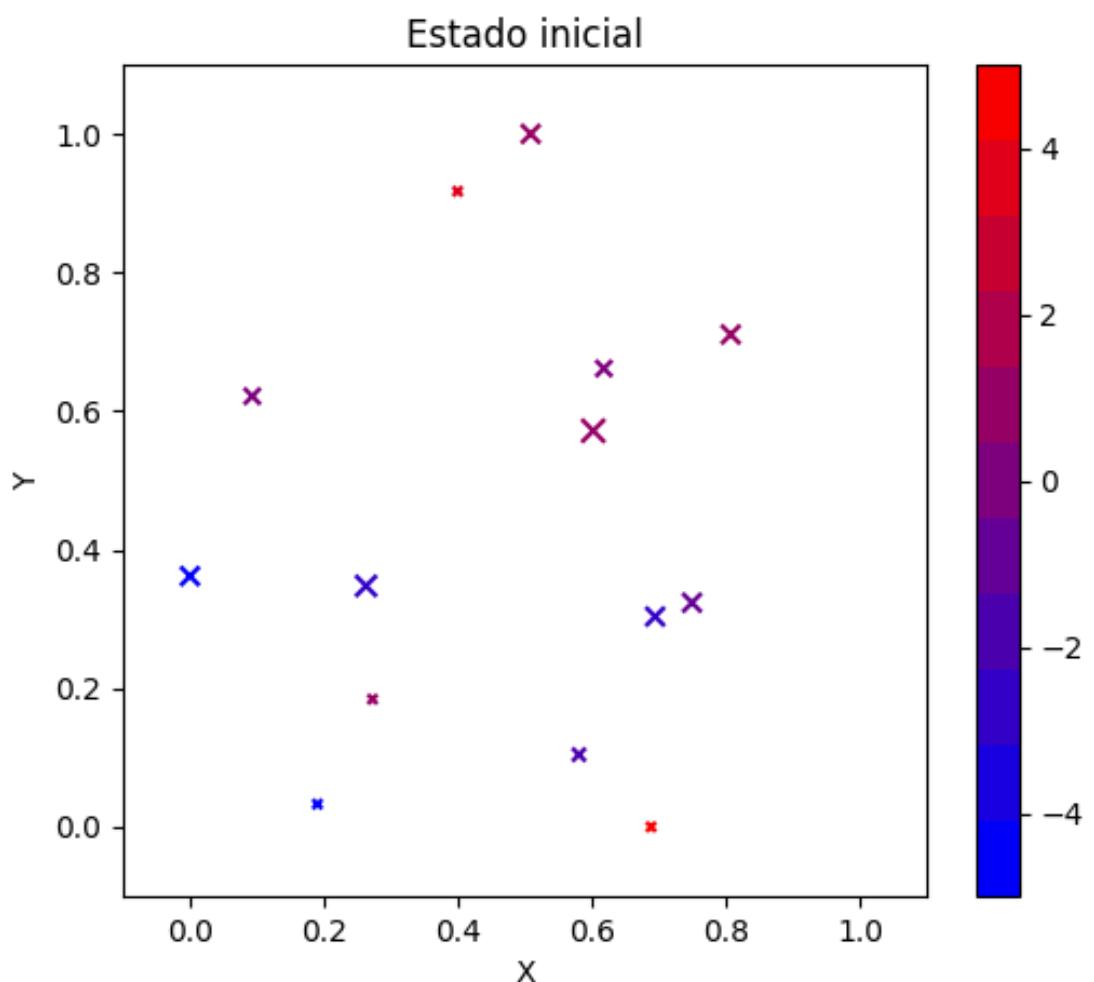


Figura 1: Velocidad sobre tiempo de la partícula.

4. Resultados

En una gráfica de línea se graficó los resultados del comportamiento de las diferentes partículas respectivamente para poder ver su la importancia que toma al agregar el factor de la masa, como también se generaron caja bigotes para poder identificar el comportamiento en su velocidad y carga de la partícula.

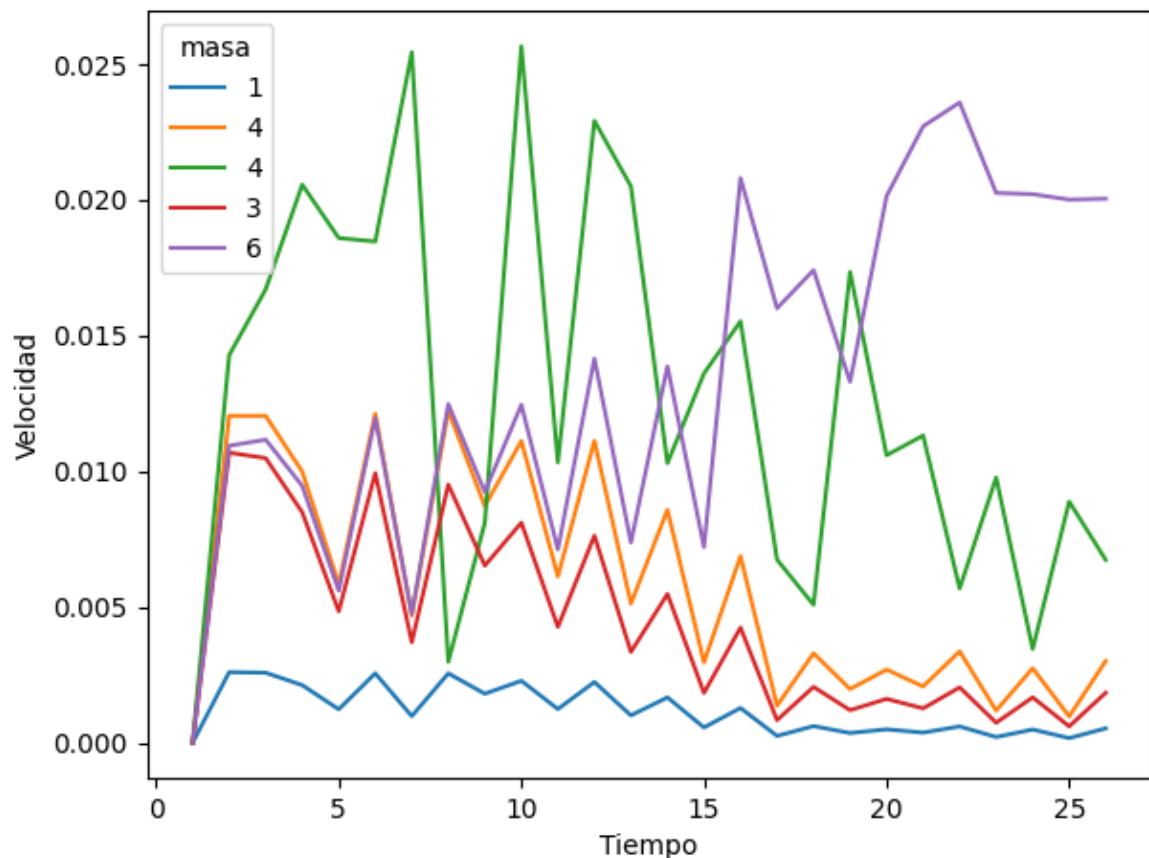


Figura 2: Velocidad sobre tiempo de la partícula.

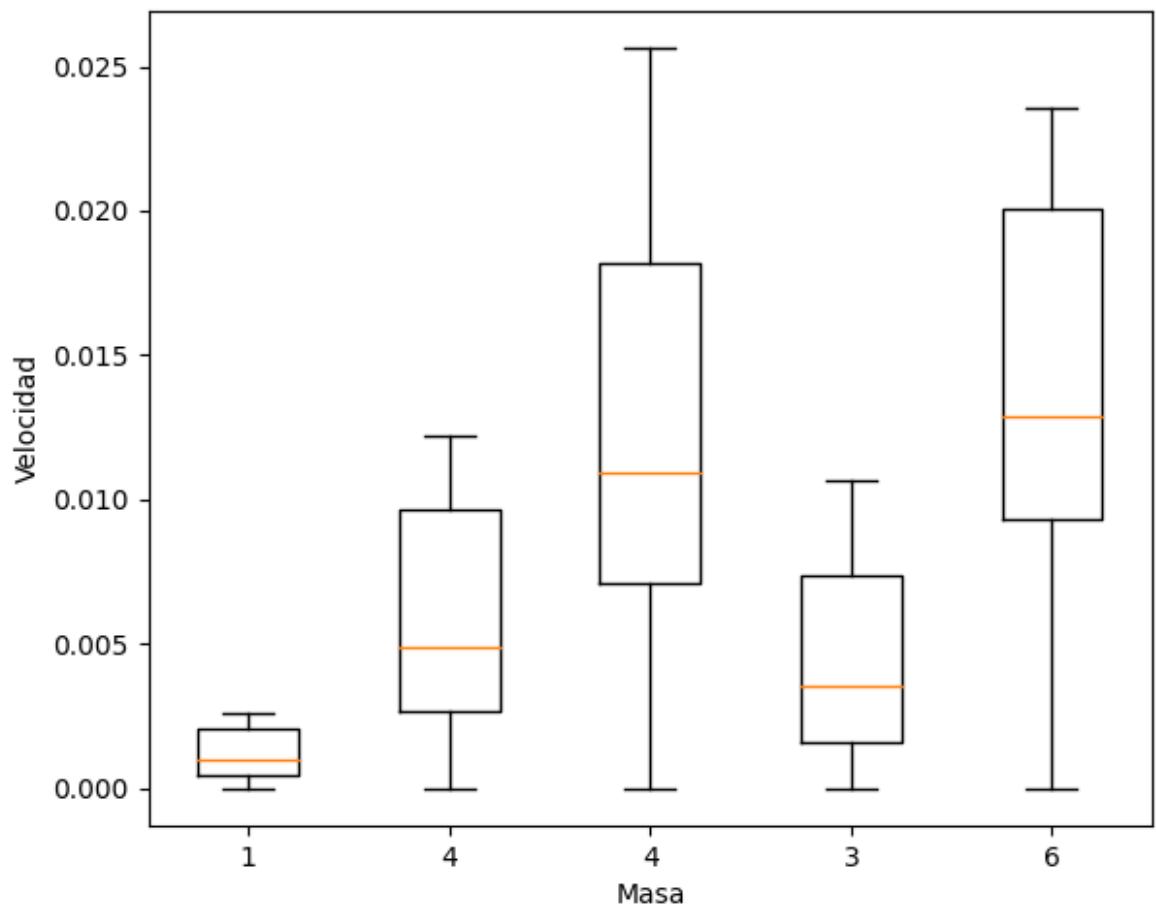


Figura 3: Velocidad sobre masa de la partícula.

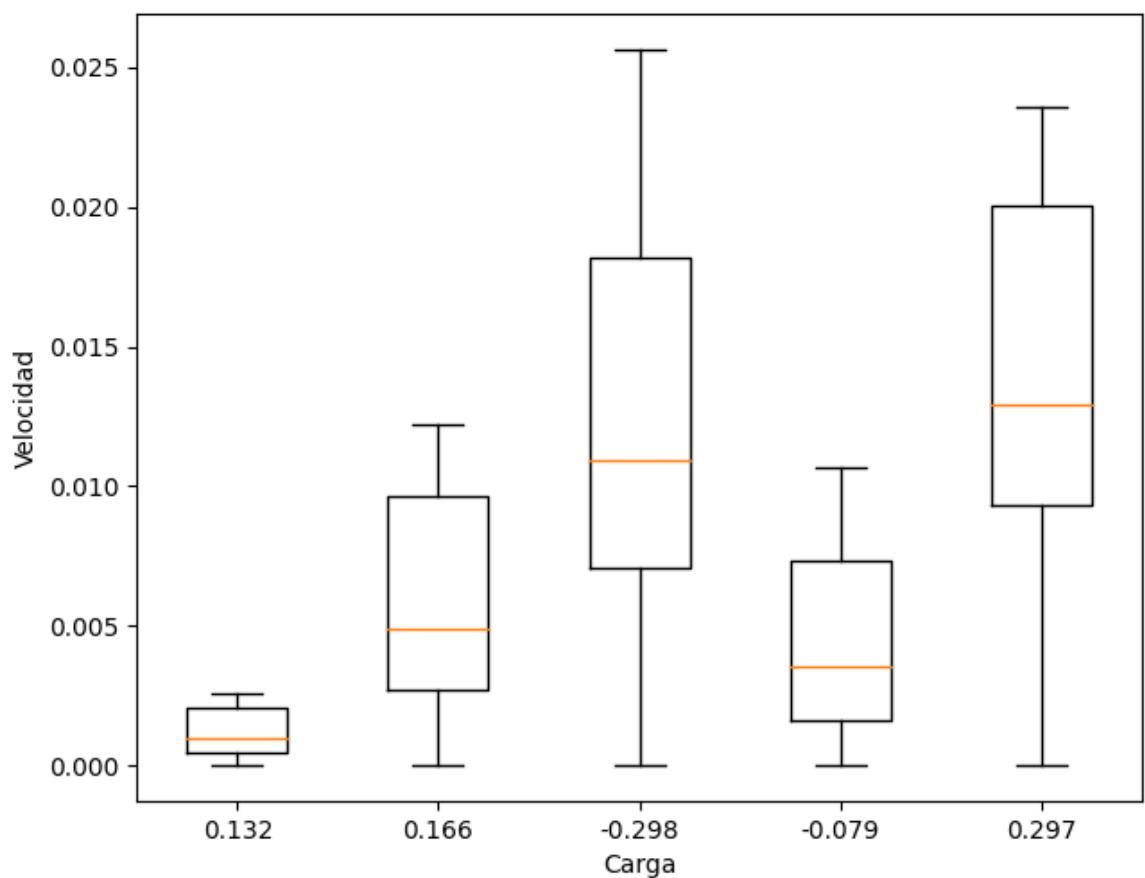


Figura 4: Velocidad sobre carga de la partícula.

5. Conclusión

Se mostró con gráficas de caja bigote que al tomar en cuenta la masa las partículas con más masa tienen hacer más positivas y a tener una mejor velocidad. .

Referencias

- [1] E. Schaeffer. Práctica 9: interacciones entre partículas. 2022. URL <https://satuelisa.github.io/simulation/p9.html>.

Práctica 10: algoritmo genético

Raul L.

3 de mayo de 2022

1. Introducción

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. Aunque el algoritmo pseudo-polinomial sirve solamente para pesos enteros, nos servirá para esta décima práctica, donde probamos la implementación de un algoritmo genético en R de manera paralela. Los algoritmos genéticos se suelen utilizar en casos donde no existe ningún algoritmo exacto eficiente, pero para fines de aprendizaje, nos conviene comparar qué tan cerca a la solución óptima (que nos da el algoritmo pseudo-polinomial) logramos llegar con un algoritmo genético.

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

Preparamos algunas subrutinas necesarias:

verificar si una selección particular respeta la capacidad de la mochila o no; calcular el valor total de los objetos incluidos; generar una selección uniformemente al azar; generar pesos enteros y ordenados al azar (normalmente distribuidos); asignar a cada objeto un valor al azar (normalmente distribuidos alrededor de los pesos pero con desviación estándar uniformemente distribuidos)[1].

2. Objetivo

Genere tres instancias con tres distintas reglas (nueve en total):

el peso y el valor de cada objeto se generan independientemente con una distribución uniforme, el valor de cada objeto se generan independientemente con una distribución exponencial y su peso es inversamente correlacionado con el valor, el peso de cada objeto se generan independientemente con una distribución normal y su valor es (positivamente) correlacionado con el cuadrado del peso, con un ruido normalmente distribuido de baja magnitud. Determina para cada uno de los tres casos si variar la probabilidad de mutación, la cantidad de cruzamientos y el tamaño de la población (dos o tres niveles basta) tienen un efecto (solo o en combinación) estadísticamente significativo (usando por lo menos tres réplicas por instancia) en la calidad de resultado, manteniendo el tiempo de ejecución fijo[1].

3. Código

Para este código se utilizó como base el código de la doctora donde se agregaron las 3 reglas.

Código en Python <https://github.com/satuelisa/Simulation/blob/master/Particles/creation.py>

Código creado en Python

https://github.com/Raullr28/Resultados/tree/main/P_10

```
for regla in range(3):
    print("##### regla:",regla,"#####")
    if regla == 0:
        pesos = pesos1(n, 15, 80)
        valores = valores1(pesos, 10, 500)
    if regla == 1:
        valores = valores2(n, 10, 500)
        pesos = pesos2(valores, 15, 80)
    if regla == 2:
        pesos = pesos3(n, 15, 80)
        valores = valores3(pesos, 10, 500)
```

Código 1: Representación de las 3 reglas generadas.

```
def pesos1(cuantos, low, high):
    return np.round(normalizar(np.random.uniform(size = cuantos)) * (high - low) + low)

def valores1(pesos, low, high):
    n = len(pesos)
    valores = np.empty((n))
    for i in range(n):
        valores[i] = np.random.uniform(pesos[i], random(), 1)
    return normalizar(valores) * (high - low) + low

def pesos2(valores, low, high):
    cuantos=1/valores
    return np.round(normalizar(cuantos) * (high - low) + low)

def valores2(cuantos, low, high):
    valores = expon.rvs(size= cuantos)
    return normalizar(valores) * (high - low) + low

def pesos3(cuantos, low, high):
    return np.round(normalizar(np.random.normal(size = cuantos)) * (high - low) + low)

def valores3(pesos, low, high):
    n = len(pesos)
    valores = np.empty((n))
    magnitud=0.1
    ruido=np.random.normal(loc=5, size = n)
    ruido=ruido*magnitud
    for i in range(n):
        valores[i] = (pesos[i]**2)+ ruido[i]
    return normalizar(valores) * (high - low) + low
```

Código 2: Representación ciclo de la partícula.

```

def poblacion_inicial(n, tam):
    pobl = np.zeros((tam, n))
    for i in range(tam):
        pobl[i] = (np.round(np.random.uniform(size = n))).astype(int)
    return pobl

def mutacion(sol, n):
    pos = randint(0, n - 1)
    mut = np.copy(sol)
    mut[pos] = 1 if sol[pos] == 0 else 0
    return mut

def reproduccion(x, y, n):
    pos = randint(2, n - 2)
    xy = np.concatenate([x[:pos], y[pos:]])
    yx = np.concatenate([y[:pos], x[pos:]])
    return (xy, yx)

```

Código 3: Representación ciclo de la partícula.

4. Resultados

En una gráfica de barras se graficó los resultados del comportamiento del tiempo con las diferentes instancias obtenidas, se agregó una gráfica de violines para saber con que instancia se obtiene un pequeño porcentaje.

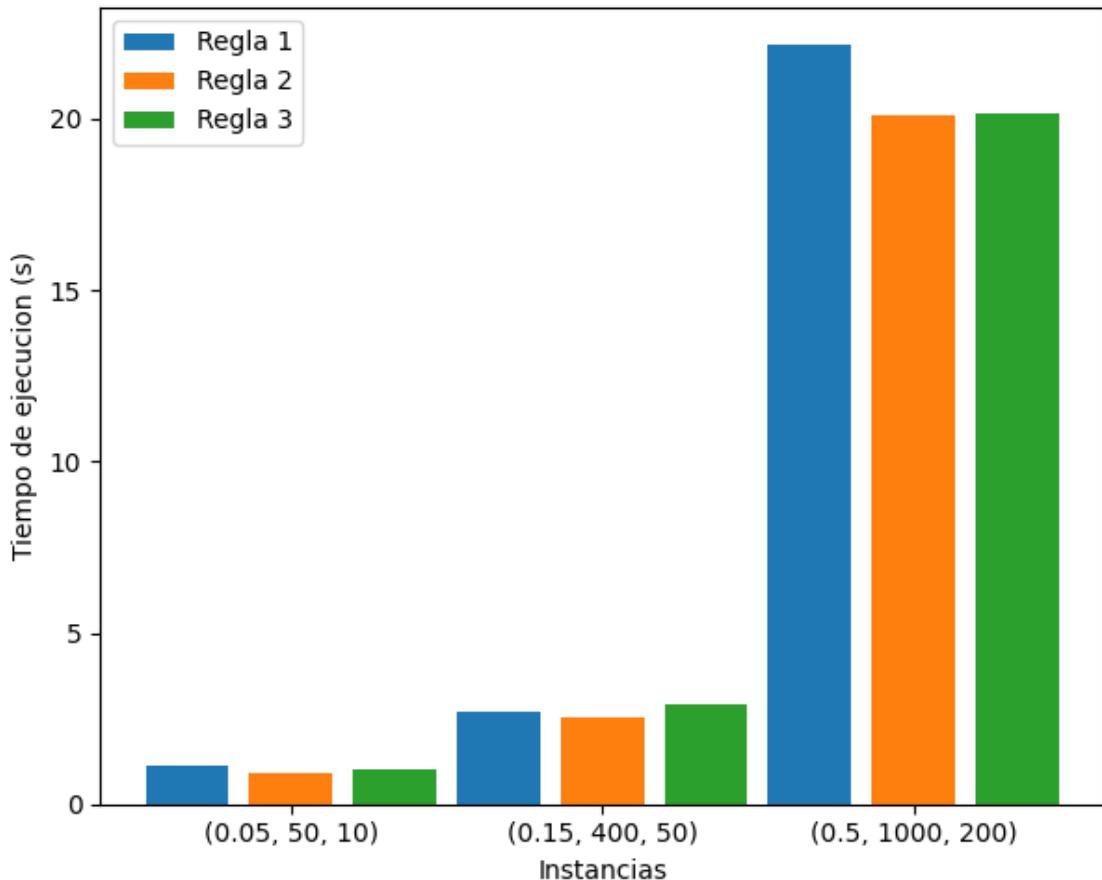


Figura 1: Gráfica de tiempo de ejecución.

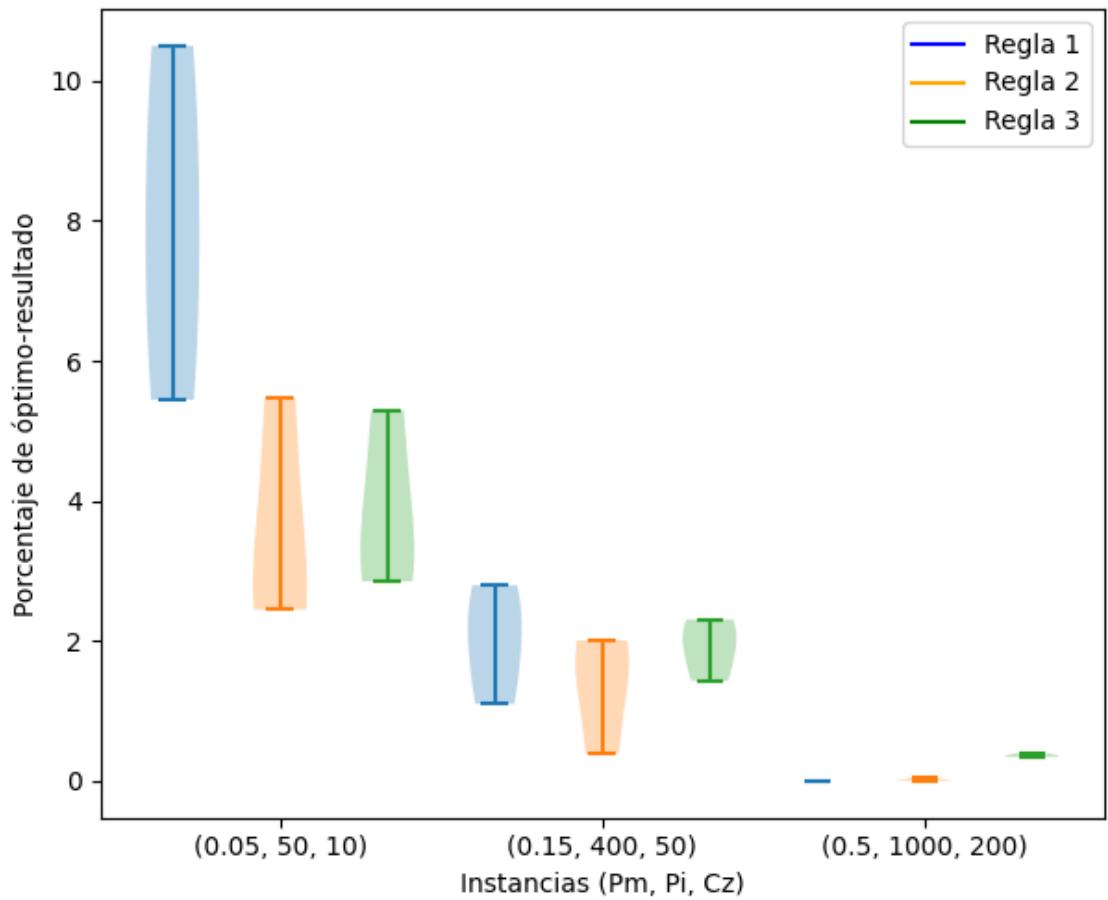


Figura 2: Grafica de Porcentaje.

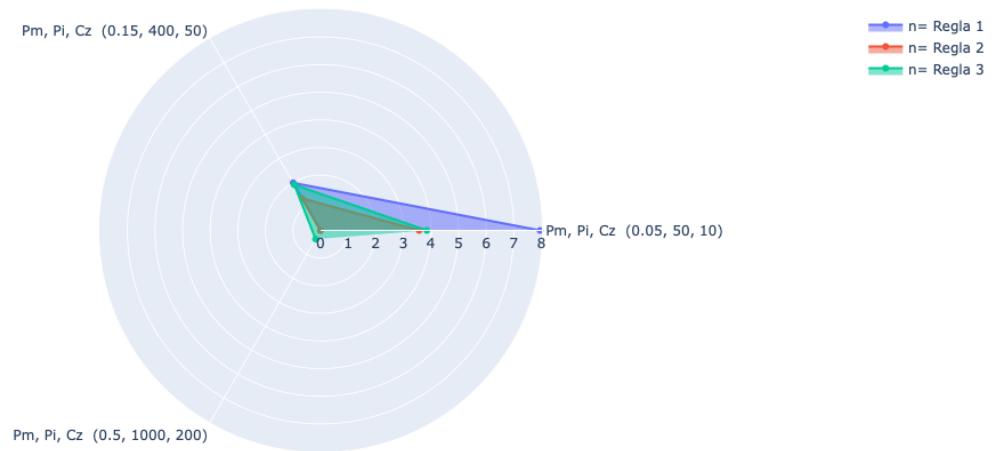


Figura 3: Grafica de reglas.

5. Reto 1

Como un primer reto, cambia la selección de mutación y de los padres para reproducción a que use selección de ruleta: cada solución se selecciona como parente con una probabilidad que es linealmente proporcional a su valor de función objetivo y a su factibilidad, combinando los dos a alguna función que parezca conveniente e inversamente proporcional a alguna combinación de factibilidad y objetivo para la mutación (recomiendo aprovechar el parámetro prob en sample).

Aplicar la selección de ruleta a las fases de cruzamiento y de supervivencia: en vez de quedarnos con las mejores soluciones, cada solución tiene una probabilidad de entrar a la siguiente generación que es proporcional a su valor de la función objetivo, incorporando el sí o no es factible la solución en cuestión, permitiendo que los k mejores entre las factibles entren siempre (donde k es un parámetro). Estudia nuevamente el efecto de este cambio en la calidad de la solución en los tres casos[1].

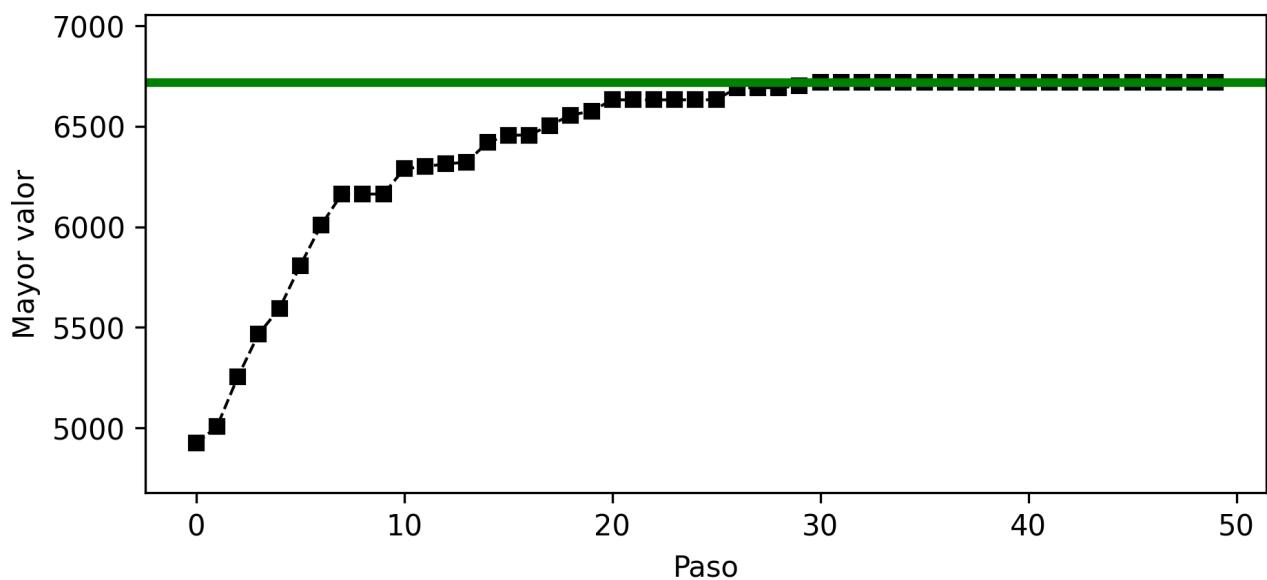


Figura 4: Gráfica de comportamiento regla 1.

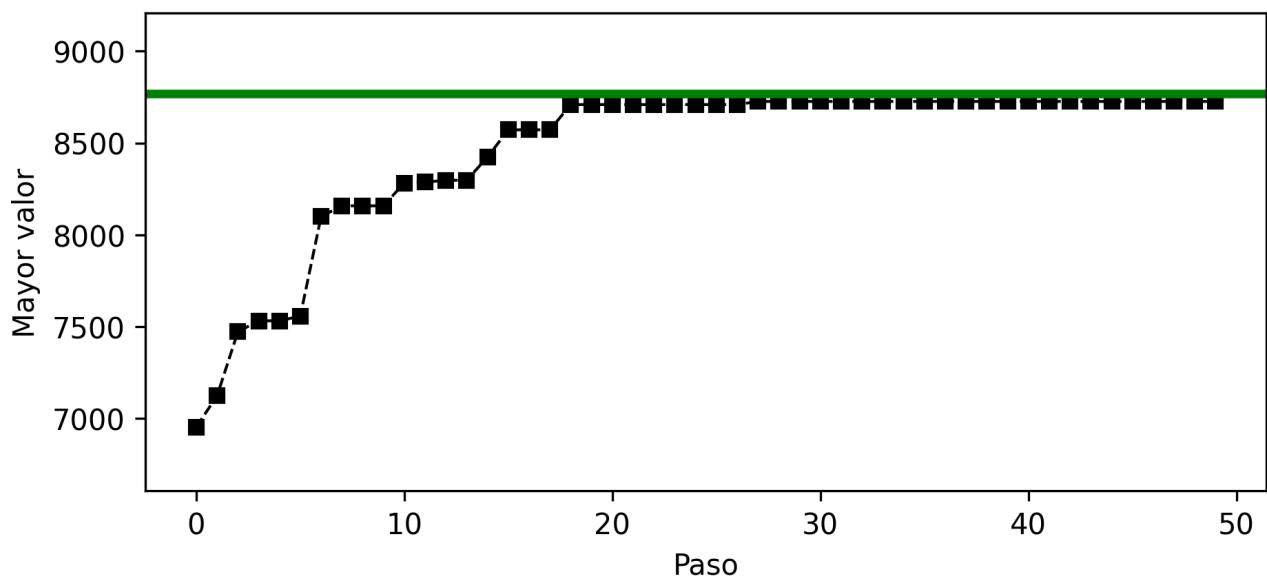


Figura 5: Grafica de comportamiento regla 2 .

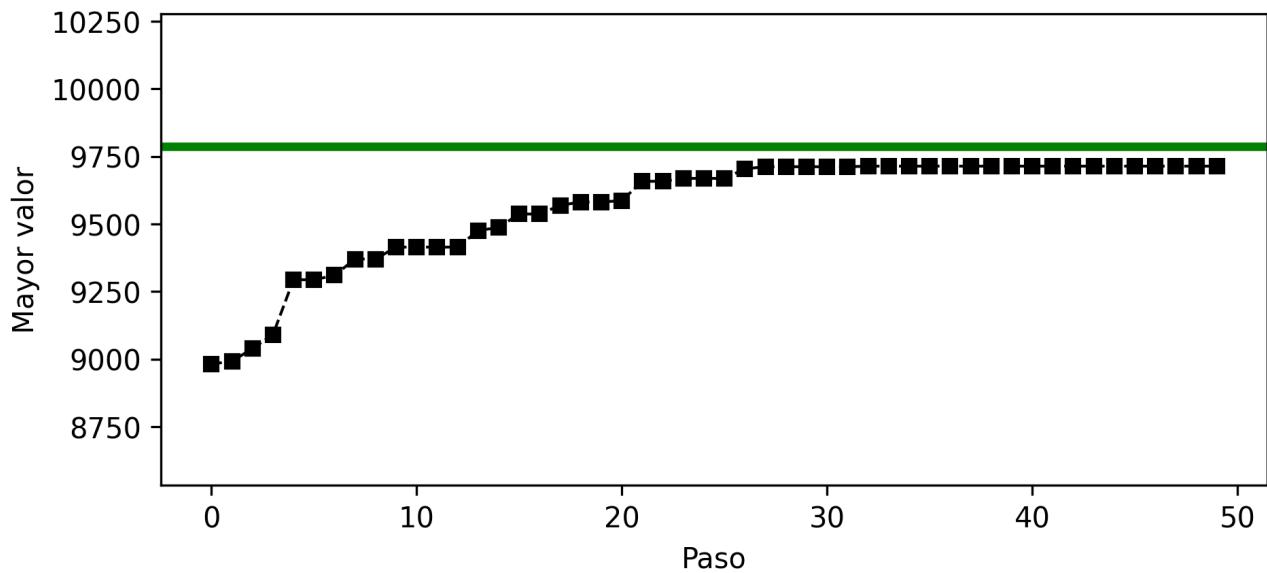


Figura 6: Grafica de comportamiento regla 3.

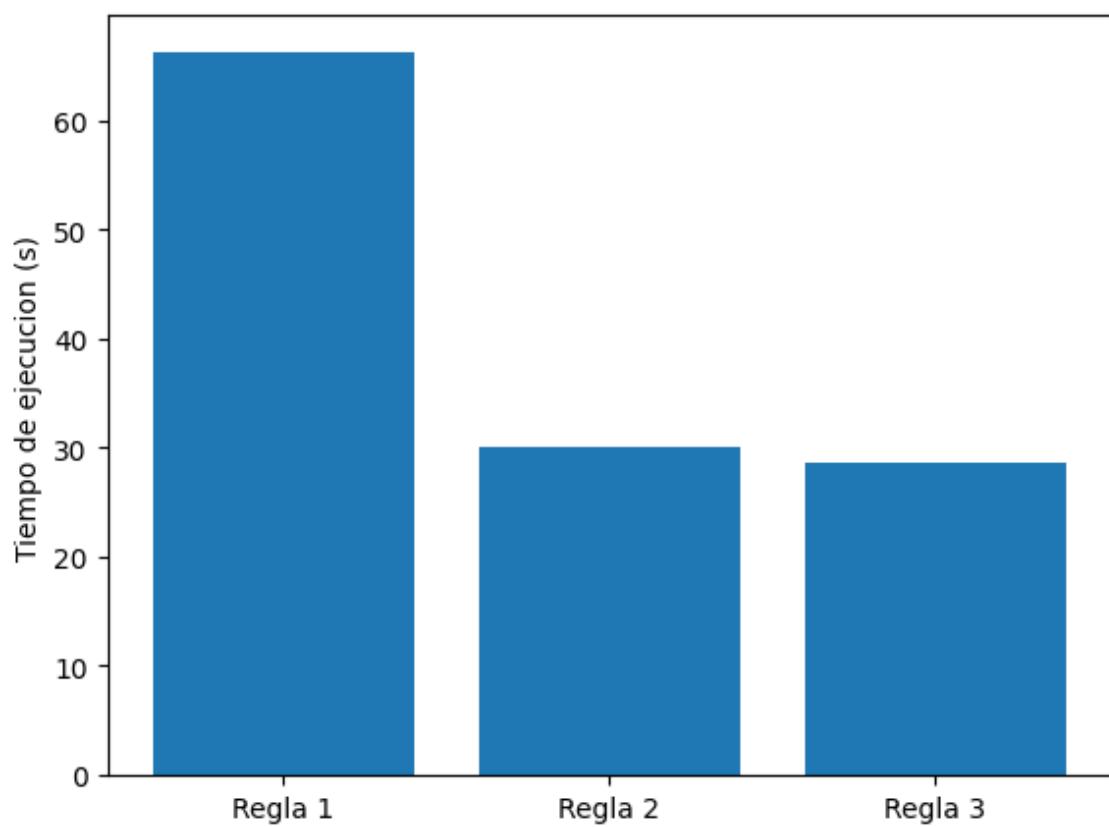


Figura 7: Gráfica de tiempo de ejecución.

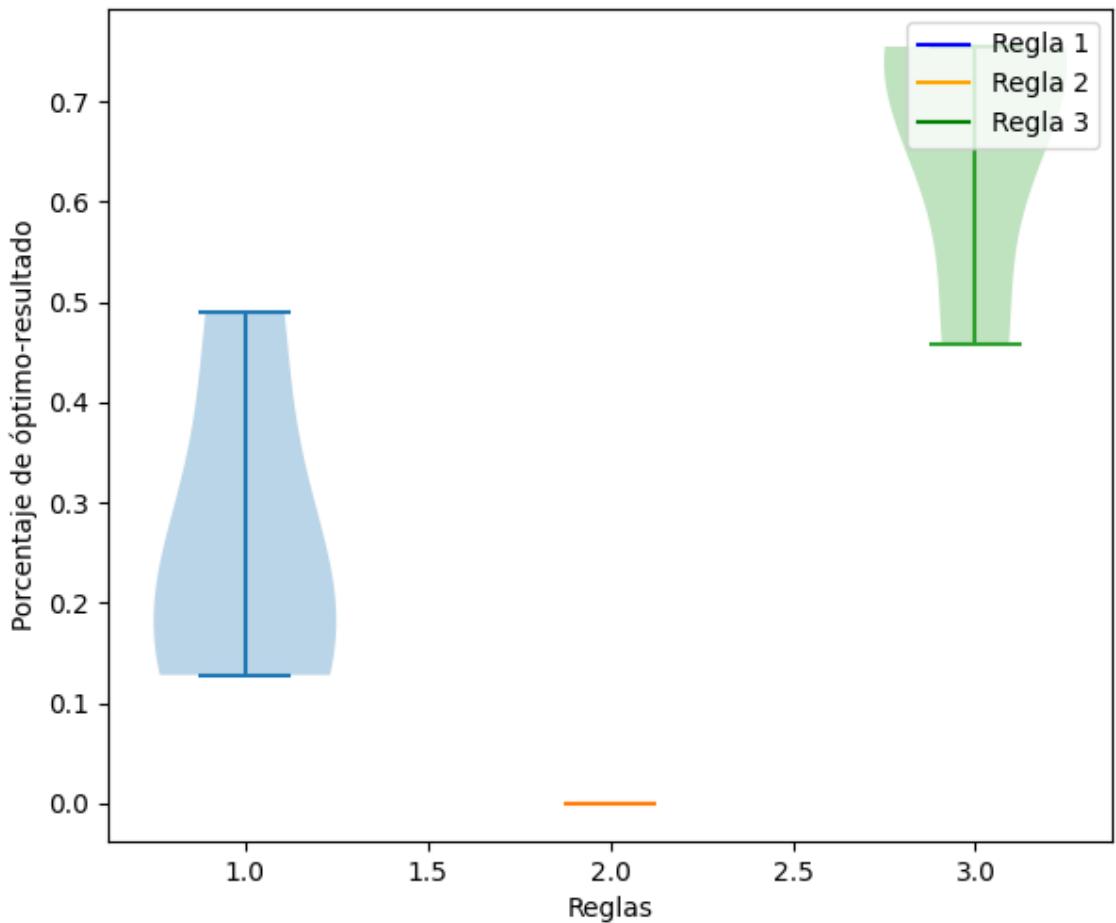


Figura 8: Grafica de Porcentaje.

6. Conclusión

Se mostró con graficas de violín que con las instancias (0.5, 1000, 2000) es el que más se acerca al dato correcto pero las instancias que sobran se demostró que tienen mejor tiempo. .

Referencias

- [1] E. Schaeffer. Práctica 9: interacciones entre partículas. 2022. URL <https://satuelisa.github.io/simulation/p10.html>.

Práctica 11: frentes de Pareto

Raul L.

10 de mayo de 2022

1. Introducción

En optimización multicriterio, a un mismo conjunto de variables ocupa asignarse valores de tal forma que se optimizan 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 haya.

Para estudiar este problema, vamos a primero implementar un generador de polinomios aleatorios. Estos polinomios los utilizaremos como funciones objetivo. Vamos a permitir solamente una variable por término y un término por grado por variable.[1].

2. Objetivo

Grafica el porcentaje de soluciones de Pareto (ojo, no es lo mismo que se grafica en el código ejemplo) como función del número de funciones objetivo para $k[2,3,4,5]$ con diagramas de violín combinados con diagramas de caja-bigote, verificando que diferencias observadas, cuando las haya, sean estadísticamente significativas. Razona en escrito a qué se debe el comportamiento observado.[1].

3. Código

Para este código se utilizó como base el código de la doctora donde se agregaron las 3 reglas.

Código en Python <https://github.com/satuelisa/Simulation/blob/master/ParetoFronts/violin.py>

Código creado en Python

https://github.com/Raullr28/Resultados/blob/main/P_11

```
replicas=50
CV=[]
for k in range(2,6):
    rep=[]
    for r in range(replicas):
        obj = [poli(md, vc, tc) for i in range(k)]
        minim = np.random.rand(k) > 0.5
        n = 250 # cuantas soluciones aleatorias
        sol = np.random.rand(n, vc)
        val = np.zeros((n, k))
        for i in range(n): # evaluamos las soluciones
            for j in range(k):
                val[i, j] = evaluate(obj[j], sol[i])
        sign = [1 + -2 * m for m in minim]
        dom = []
        for i in range(n):
            d = [domin_by(sign * val[i], sign * val[j]) for j in range(n)]
            dom.append(sum(d))
        frente = val[[d == 0 for d in dom], :]
        rep.append((len(frente)*100)/n)
    CV.append(rep)
```

Código 1: Representación de la variación del objetivo.

4. Resultados

En una gráfica de violines podemos ver el comportamiento al variar el objetivo como se demuestra a continuación como también se comprobó con una prueba estadística que las pruebas tienen una relación estilísticamente significativa.

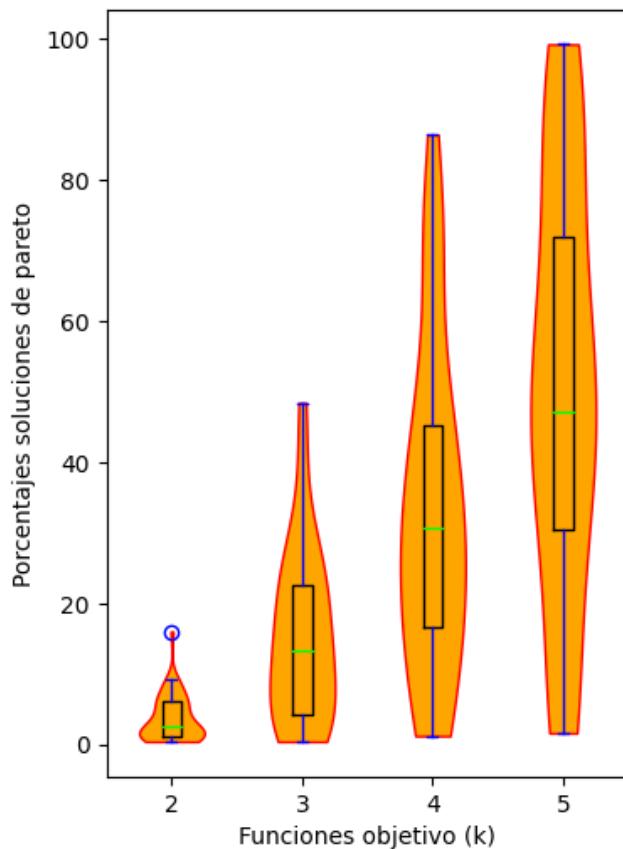


Figura 1: Gráfica de comparación.

El objetivo principal de esta prueba es determinar si existe una diferencia estadística entre las medianas de al menos tres grupos independientes. Hipótesis: La prueba de Kruskal-Wallis tiene las hipótesis nula y alternativa como se analiza a continuación:

La hipótesis nula (H_0): La mediana es la misma para todos los grupos de datos. La hipótesis alternativa: (H_a): La mediana no es igual para todos los grupos de datos. Implementación paso a paso:

```
Resultado | KruskalResult(statistic=87.56143318086774, pvalue=7.315608346059619e-19)
```

5. Reto 1

El primer reto es 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. Graficar los resultados de la selección, indicando con un color cuáles se incluyen en el subconjunto diverso.

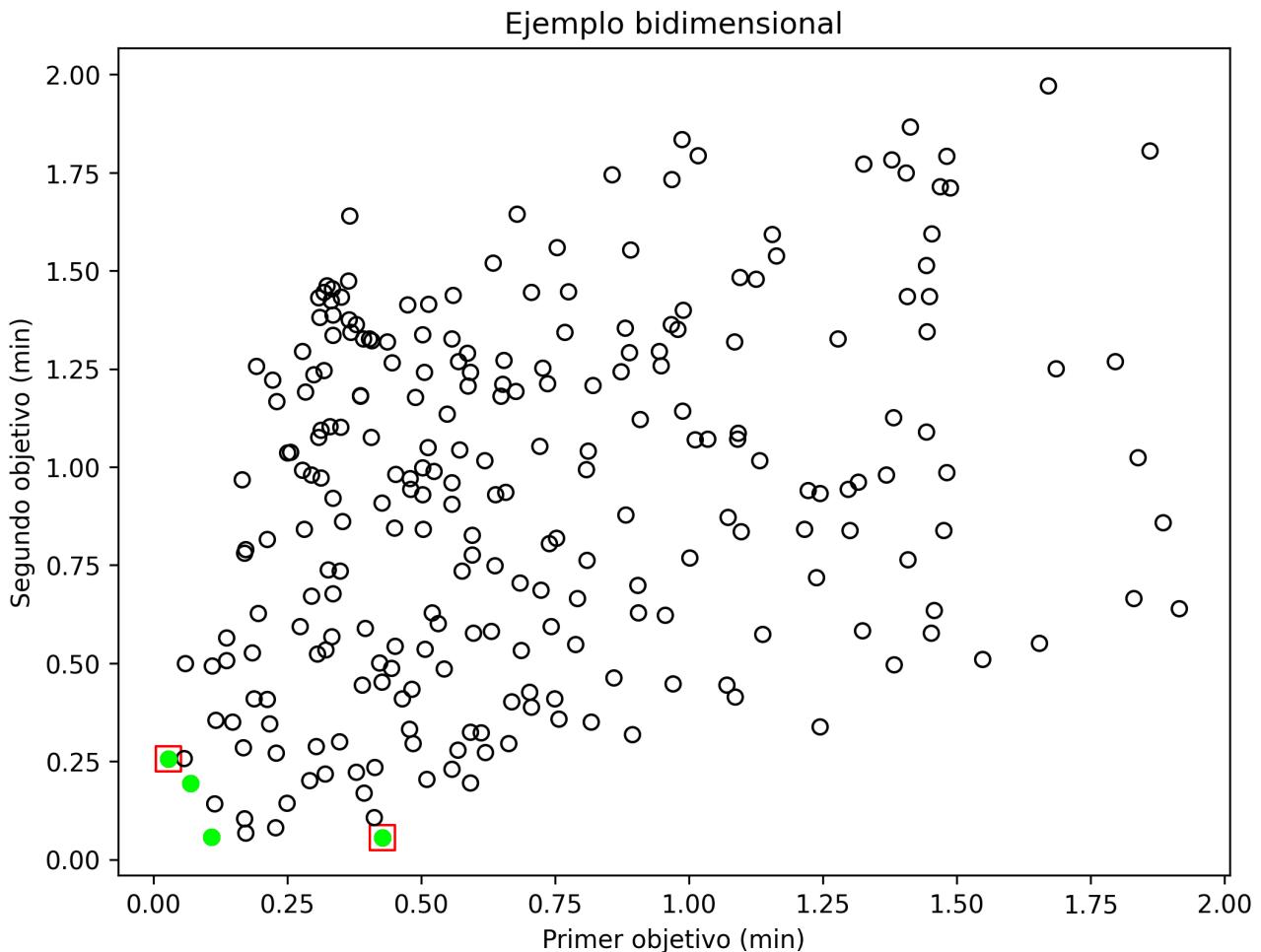


Figura 2: Gráfica de comportamiento regla 1.

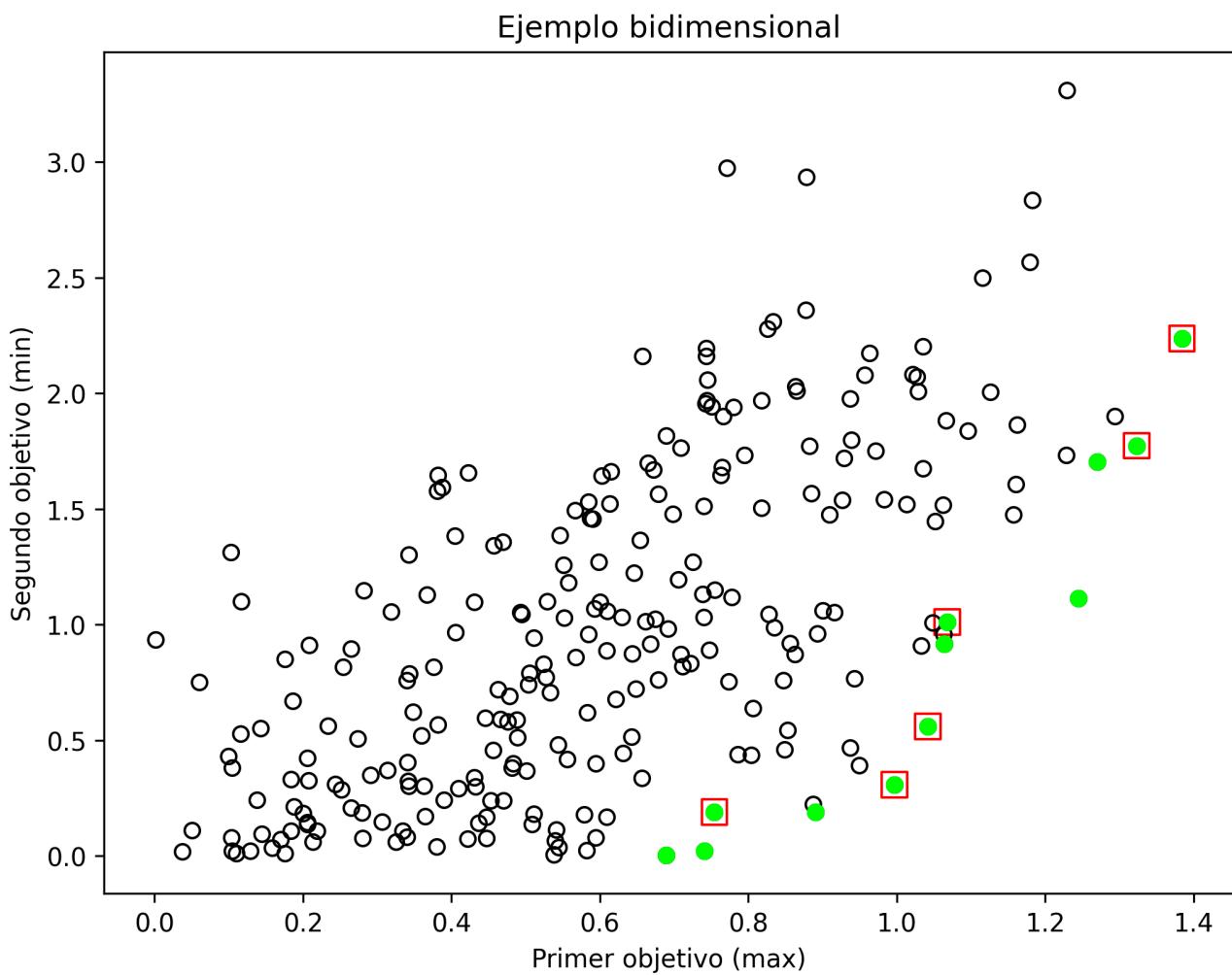


Figura 3: Grafica de comportamiento regla 2 .

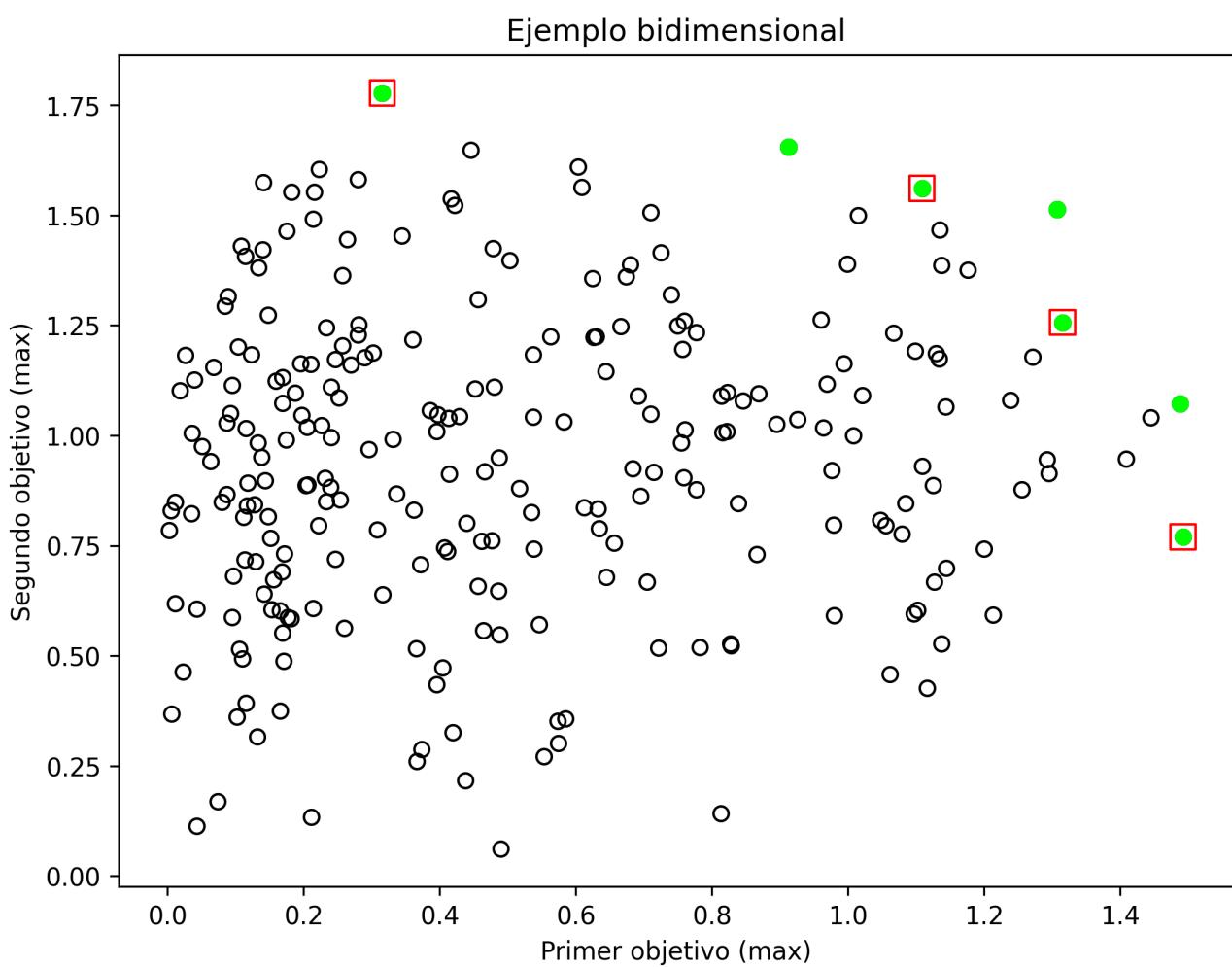


Figura 4: Grafica de comportamiento regla 3.

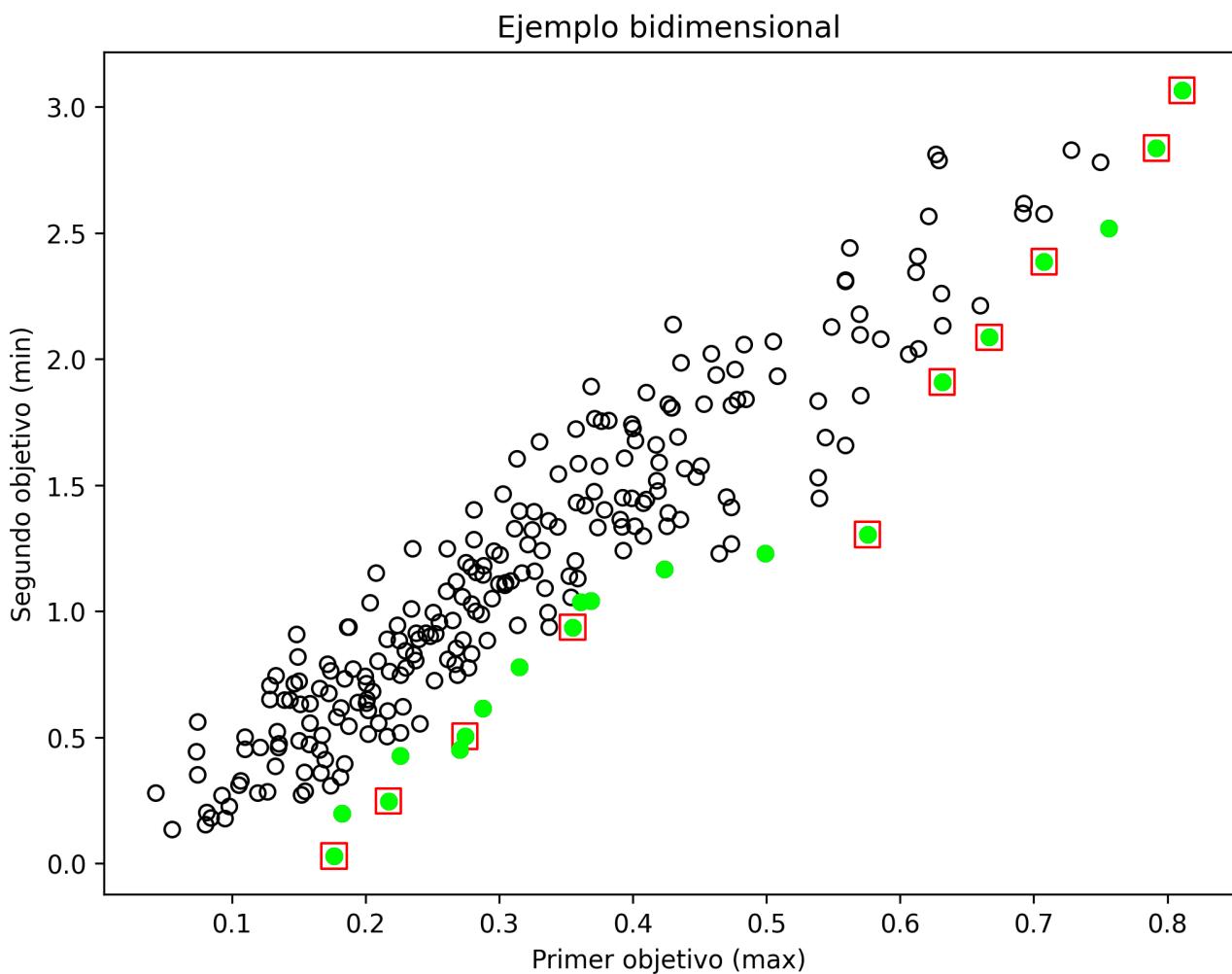


Figura 5: Gráfica de tiempo de ejecución.

6. Conclusión

Se mostró con graficas de violín como va comportamiento del valor del objetivo al aumentar el valor, en el reto 1 calculando la distancia euclidiana podemos encerrar en círculos los más separados del resto de la pared de pareto.

Referencias

- [1] E. Schaeffer. Práctica 11: frentes de pareto. 2022. URL <https://satuelisa.github.io/simulation/p11.html>.

Práctica 12: red neuronal

Raul L.

18 de mayo de 2022

1. Introducción

La última práctica es una demostración básica de aprendizaje a máquina: vamos a reconocer dígitos de imágenes pequeñas en blanco y negro con una red neuronal. El elemento básico de una red neuronal es un perceptrón que esencialmente es un hiperplano (una línea si nos limitamos a dos dimensiones) que busca colocarse en la frontera que separa las entradas verdaderas y las entradas falsas. La dimensión d del perceptrón es el largo del vector x que toma como entrada, y su estado interno se representa con otro vector w que contiene sus pesos. Para responder a una salida proporcionada a ello, el perceptrón calcula el producto interno de xw , es decir

$$\sum_{i=1} = x_i w_i$$

, y si esta suma es positiva, la salida del perceptrón es verdad, en otro caso es falso [1]. Para agarrar la onda con los perceptrones, haremos primero uno más sencillo cuyo jale es identificar $\text{six} > y$ para puntos en dos dimensiones, ya que así es fácil para nosotros visualizar lo que le pasa al perceptrón durante el entrenamiento[2].

2. Objetivo

Estudia 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 (ngb), variando a las tres en un experimento factorial adecuado[2].

3. Código

Para este código se utilizó como base el código de la doctora.

Código en Python <https://github.com/satuelisa/Simulation/blob/master/NeuralNetwork/ann.py>

Código creado en Python

https://github.com/Rauullr28/Resultados/blob/main/P_12

```

n=0.99
g=0.01
b=0.50

fact_des= itertools.product((n,g,b),(n,g,b),(n,g,b))
factor=[]
factor_lab=[]
for i in fact_des:
    factor.append(i)
    factor_lab.append(str(i))

CV=[]
for n, g, b in factor:
    print('#####',n,g,b,'#####')
replicas=[]
repeticiones=10
for rep in range(repeticiones):
    modelos = pd.read_csv('digits.txt', sep=' ', header = None)
    modelos = modelos.replace({'n': n, 'g': g, 'b': b})

```

Código 1: Representación de la función factorial.

```

c = pd.DataFrame(contadores)
c.columns = [str(i) for i in range(k)] + ['NA']
c.index = [str(i) for i in range(k)]

diagonal=[]
for d in range(0,k):
    num=c.iloc[d][d]
    diagonal.append(num)

FP=[]
for e in c.columns[0:-1]:
    suma=sum(c[e])
    diag=diagonal[int(e)]
    FP.append(suma-diag)

FN=[f for f in c['NA']]

PCS= sum(diagonal)/(sum(diagonal)+sum(FP))
RCL=sum(diagonal)/(sum(diagonal)+sum(FN))
F_score= 2*(PCS*RCL)/(PCS+RCL)
replicas.append(F_score)

```

Código 2: Representación FC.

4. Resultados

En una gráfica de violines podemos ver el comportamiento al variar el experimento factorial dando como resultado la mejor combinación para obtener el mejor puntaje .

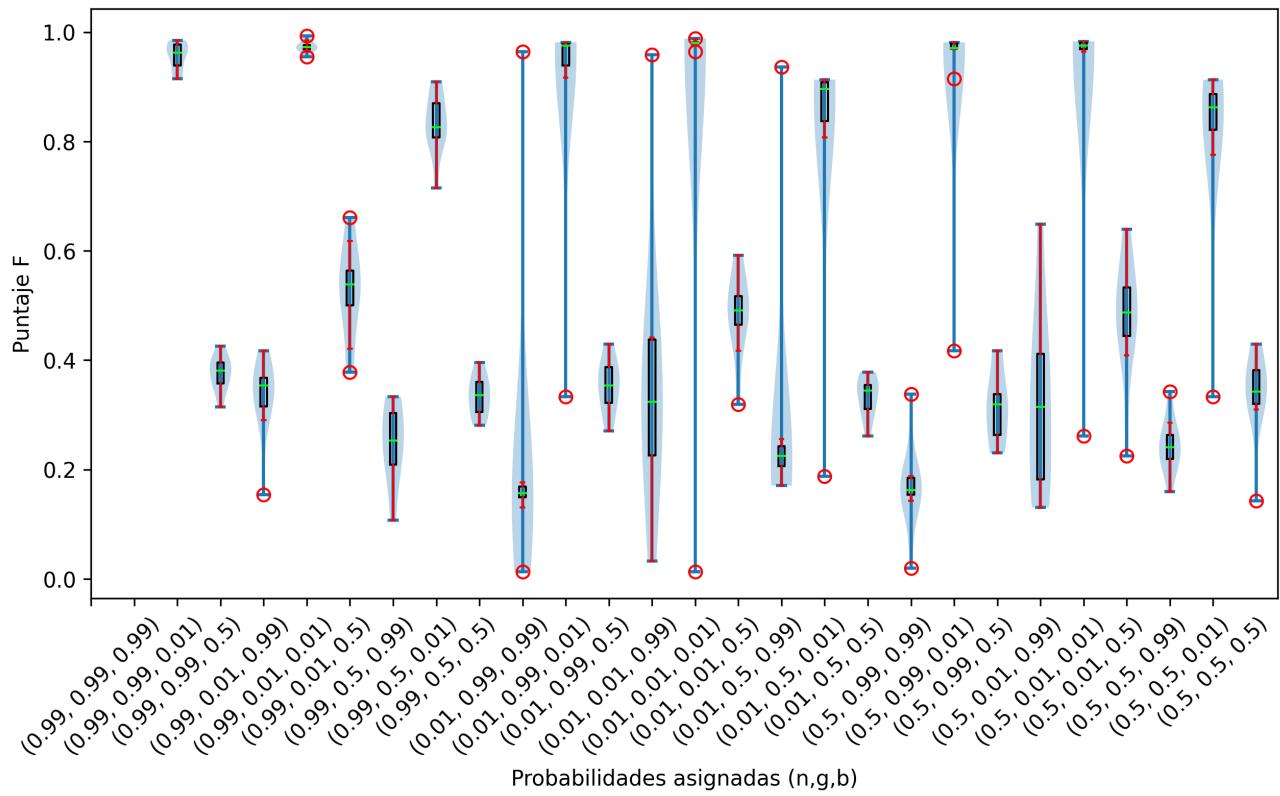


Figura 1: Gráfica de comparación.

5. Reto 1

Como un primer reto, extiende y entrena la red neuronal para que reconozca además por lo menos doce símbolos ASCII adicionales, aumentando la resolución de las imágenes a 5×7 de lo original de 3×5 (modificando las plantillas de los dígitos acorde a este cambio).

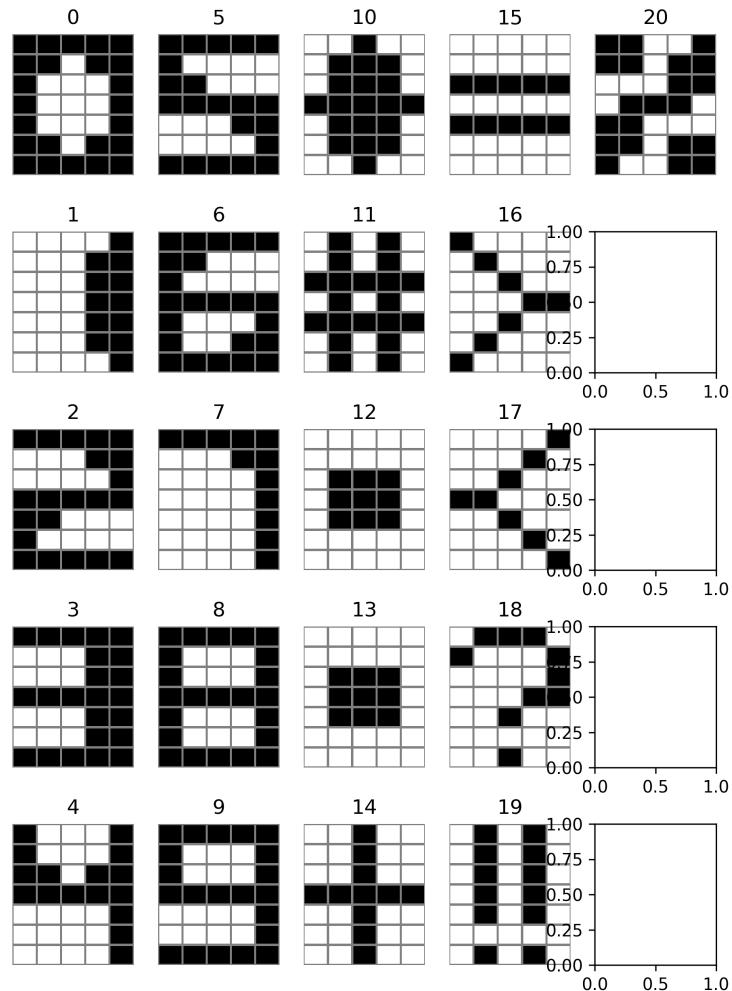


Figura 2: Gráfica de comportamiento.

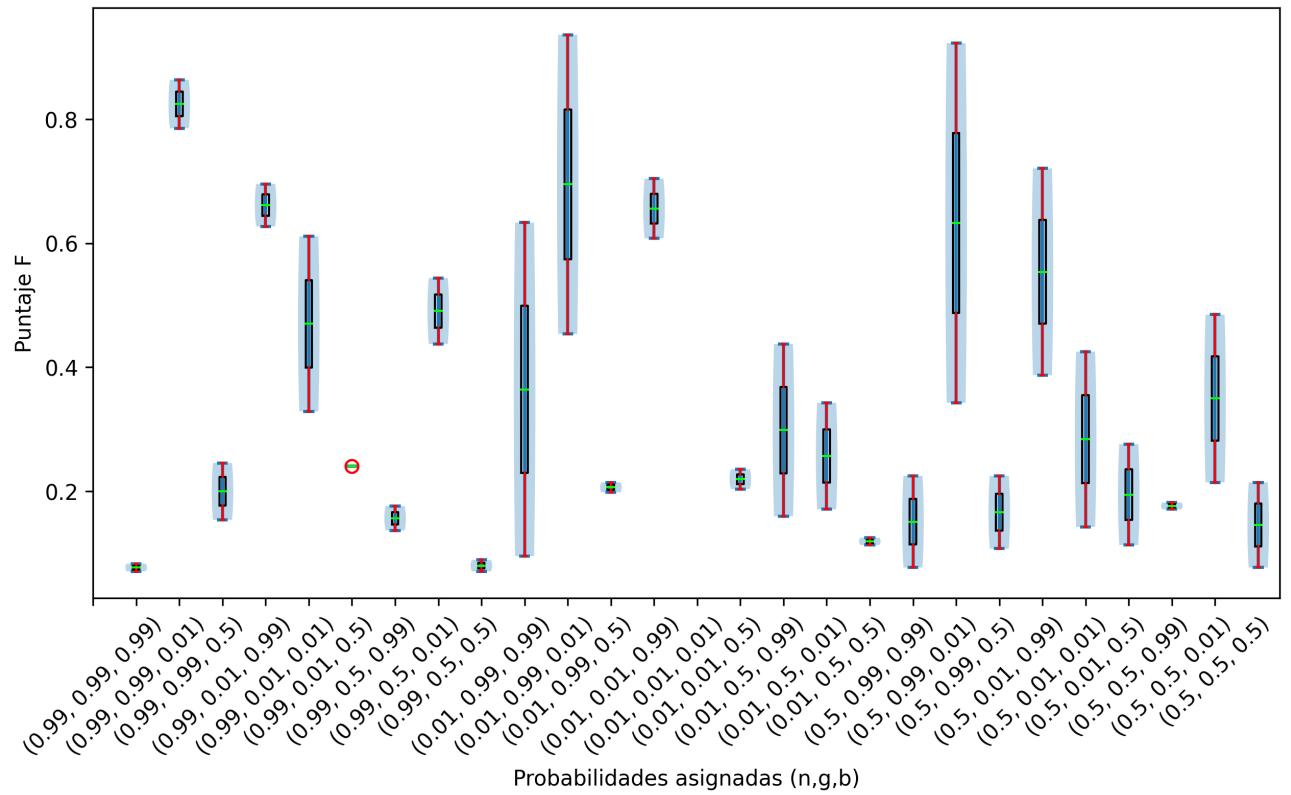


Figura 3: Grafica de comportamiento.

6. Reto 2

En el segundo reto, agrega ruido sal-y-pimienta en las entradas para una combinación ngb con la cual la red desempeña bien; este tipo de ruido se genera cambiando con una probabilidad pr los pixeles a blanco o negro (uniformemente al azar entre las dos opciones). Estudia el efecto de pr en el desempeño de la red (no importa si se hace esto con la red de la tarea base o la red extendida del primer reto).

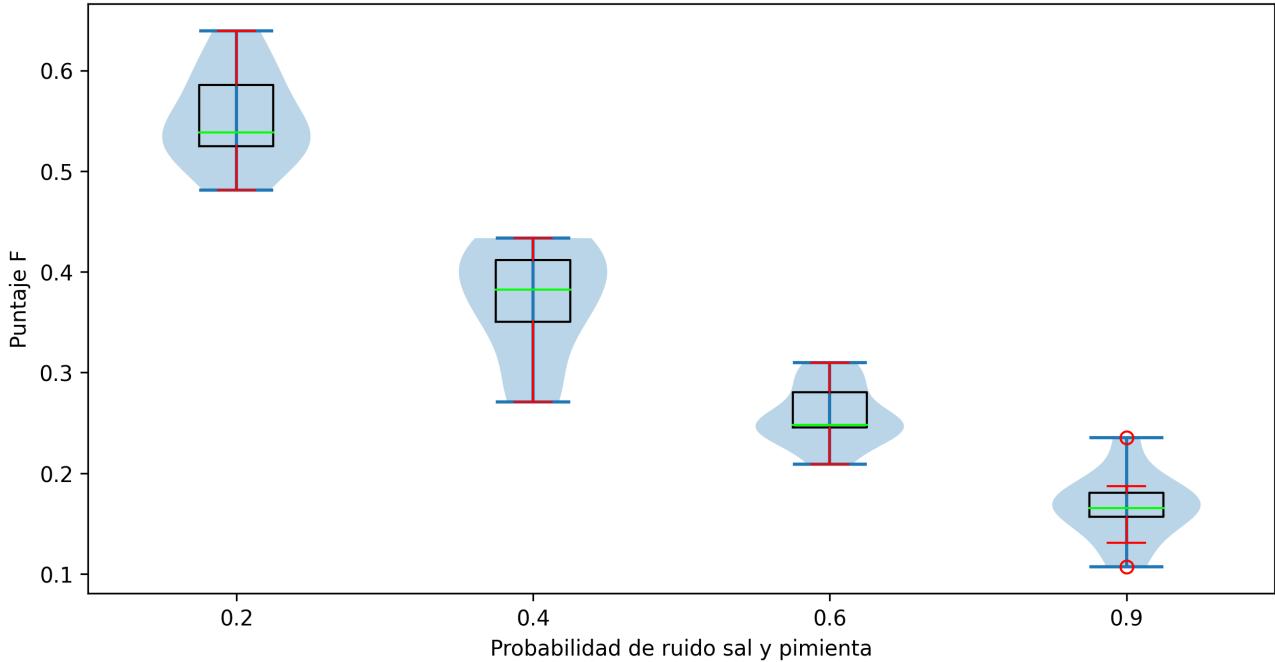


Figura 4: Grafica de comportamiento.

7. Conclusión

Se mostró con graficas de violín como va comportamiento al aumentar la probabilidad del ruido sal y pimienta no puede comprender que dato manejamos y eso hace que baje el porcentaje FC.

Referencias

- [1] K. Gurney. An introduction to neural networks. 1997. URL <https://www.crcpress.com/An-Introduction-to-Neural-Networks/Gurney/p/book/9781857285031>.
- [2] E. Schaeffer. Práctica 11: frentes de pareto. 2022. URL <https://satuelisa.github.io/simulation/p11.html>.

Simulación del dopaje de nanopartículas y comportamiento de flujo eléctrico.

LAGUNES RIVERA R.

Universidad Autónoma de Nuevo León, Facultad de Ingeniería Mecánica y Eléctrica.

Maestría en Ciencias de la Ingeniería con Orientación en Nanotecnología.

Contacto: raul.lagunesr@uanl.edu.mx

Github: [Raullr28](#)

Compiled 1 de junio de 2022

Aquí colocar el resumen

1. INTRODUCCIÓN

El dopaje de materiales con nanopartículas en semiconductores es muy común en el campo de ciencia de materiales en donde un dopaje refiere al proceso en el que se agregan impurezas en un semiconductor puro con la finalidad de cambiar propiedades eléctricas o térmicas, para un dopaje ligero es cuando se agregan una pequeña cantidad de impurezas (1 cada 100 millones de átomos) y un alto dopaje denominado pesado es cuando la impureza ocurre 1 cada 10 mil átomos.

Esta técnica es utilizada para la variación del número de huecos y electrones en un semiconductor lo cual beneficia o impide el flujo eléctrico en el mismo. Un dopaje de material tipo N es el que posee átomos de impureza que admite la aparición de más electrones que huecos. Caso contrario el tipo P que se presenta al tener átomos que permiten mayor formación de huecos sin dejar electrones de sobra lo cual beneficia el flujo de electrones a ocupar dichos huecos de sobra [9].

El dopar semiconductores a nivel nanométrico muestra propiedades mejoradas en comparación a escala a granel, ya que en el material nanocrystalino las propiedades físicas están controladas predominantemente por los límites de grano que por los granos [7]. También se puede alterar las propiedades de nanoestructuras, tanto químicas en su estructura molecular, como físicas en aspectos tales como su conductividad eléctrica, su comportamiento magnético, su resistencia mecánica, entre otras [5], aunado a esto los elementos dopantes pueden generar diferentes tipos de alteración en las características del semiconductor, como variación en la longitud de onda a la que se absorbe radiación [11].

En el presente trabajo se plantea la implementación de una simulación del dopaje de nanopartículas a un material semiconductor, estudiando el flujo eléctrico como varía respecto al porcentaje de impureza agregado y que tanto tamaño o área de cristal se forma de cada material. Primeramente se mencionan antecedentes que hablan de trabajos ya realizados similares a la simulación de dopaje con lo que se validará el proceso si-

mulado, continuando con trabajos relacionados en los que se basó el modelo de este trabajo, después el modelo propuesto e implementación son discutidos para después revisar la experimentación donde son observados los resultados obtenidos y gráficos junto a pruebas estadísticas. Por último se mencionan las conclusiones a las que se llegó en el actual trabajo.

2. ANTECEDENTES

Trabajos ya han sido reportados tal como en la simulación que se discute en el siguiente modelo Born de un sólido en el que los iones están representados por cargas puntuales interactúan tanto electrostáticamente como a través de potenciales de corto alcance. Estos últimos tienen su origen físico en la repulsión de Pauli asociada con la superposición de las distribuciones de densidad de electrones en los iones vecinos. Se obtiene una gráfica de la trayectoria del flujo eléctrico, dopado con mg alúmina a 300 K.

También se observó que entre más grande el área y la mayor duración de la simulación, ninguna de las distribuciones dopantes mostró un comportamiento distinto al de cuando se emplearon los potenciales utilizados en este trabajo. Finalmente, en comparación con los datos experimentales, las conductividades simuladas a bajas temperaturas resultaron más altas. Además de la distribución de dopantes, pueden existir otras causas para la disminución observada en la conductividad a bajas temperaturas, como la presencia de humedad [6].

Otro trabajo parecido se presenta en el cálculo de conductividad del régimen de Hz para silicio ligeramente dopado utilizando una herramienta de simulación desarrollada en Python, donde en dicha simulación se observa un electrón moviéndose con velocidad constante en la cuadrícula 2D, a través de una línea de campos eléctricos desde un punto de rejilla de referencia hasta un punto de rejilla de interés ubicado a lo largo de la trayectoria del electrón. A medida que el electrón se aleja de su punto de origen, quedan atrás los campos electrostáticos de un hueco artificial. Por el contrario, como la fuerza dominante

que actúa sobre los portadores y el flujo de corriente están en la dirección, las condiciones de contorno deben permitir el movimiento del portador sin restricciones y mantener el impulso del conjunto en esta dirección [8].

3. TRABAJOS RELACIONADOS

El trabajo se basa en trabajos relacionados tal como lo es Diagramas de Voronoi que se puede consultar en la pagina de simulación de Schaeffer [4] en el que se simula la cristalización mediante semillas colocadas para formar un material cristalino, lo que se busca es dividir esa zona en regiones llamadas celdas de Voronoi y apartir de estas celdas trabajar distintos fenómenos. Esta simulación pueden ser consultada en el repositorio de Schaeffer [2].

*. Diagramas de Voronoi

El diagrama de Voronoi es una estructura famosa de la geometría computacional, uniendo las regiones de proximidad por un conjunto de sitios en el plano donde la distancia de los puntos se define por su distancia euclídea. Se describe con un número n de semillas o puntos en un plano y cada uno abarca una región que estará delimitada por la distancia euclídea hacia otra semilla, simulando las fronteras de cada zona [10].

4. MODELO PROUESTO

El modelo que se propone para este trabajo es la representación y simulación de un material base siendo dopado por un semiconductor el cual mejora las condiciones de flujo eléctrico, para estudiar la velocidad de mejora en base al porcentaje de dopante y como extra estudiar que tanto material base se va perdiendo (pureza de material) conforme mejora el flujo. Todo el trabajo se propone desarrollar en la herramienta computacional y lenguaje de programación Python [1].

5. IMPLEMENTACIÓN

Esta sección de implementación trata de las líneas de código más importantes para simular el fenómeno del dopaje de un semiconductor. En el código 1 se comienza por la variación del porcentaje de material dopante desde 0,1 hasta 0,9 variándolo en un ciclo `for` y por cada ciclo el material base sera el restante de este porcentaje, para cada variación del dopante se realizan replicas para tener una obtención amplia de datos y poder estimar resultados gráficos.

```
1 dop=(0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9)
2 for dopaje in dop:
3     base = round(1.0 - dopaje,2)
4     for r in range(replicas):
```

Listing 1. Porcentajes de dopaje.

La determinación del porcentaje de semillas para material base y dopante se colocan con la función de `random.choice` el cual decidirá entre el color de la semilla ya sea rojo para base y azul en dopaje. Y por parte del tamaño del cristal se determina contando la cantidad color por material que se encuentra al recorrer la muestra ya con las semillas expandidas y cristalizado (ver código 2).

```
1 col=['red','blue']
2 colors= np.random.choice(col, k, p=[base, dopaje])
3 crystal_b=[]
4 crystal_d=[]
5 for y in range(n):
6     for x in range(n):
```

```
7     px= voronoi.getpixel((x,y))
8     if px==(255,0,0):
9         crystal_b.append(px)
10    if px==(0,0,255):
11        crystal_d.append(px)
```

Listing 2. Colocación de dopaje y tamaño de cristal.

Por último el código 3 muestra el movimiento de la partícula de flujo eléctrico que en cada paso que avanza toma n cantidad de coordenadas que serán posiciones en la muestra y cada posición tendrá un color ya sea del material base o del dopante a lo cual se vuelve a utilizar la función `random.choice` con probabilidad de la cantidad de rojos y azules encontrados, y si es mayor la probabilidad del dopante entonces el flujo o paso de esta partícula avanzará más rápido,

```
1 cnt=0
2 for z in range(n):
3     arr2[z+cnt,z+cnt]=(0,255,0)
4     X=[random.randint(0,n-1) for i in range(n)]
5     Y=[random.randint(0,n-1) for i in range(n)]
6     pix=list(zip(Y, X))
7     cuantos=[]
8     for i in pix:
9         cuantos.append(list(arr[i]))
10    rojos=cuantos.count([255,0,0])
11    azules=cuantos.count([0,0,255])
12    val=np.random.choice([0,1], 1, p=[rojos/n, azules/n])
13    arr2[z+cnt,z+cnt]=arr[z+cnt,z+cnt]
14    cnt=cnt+val
15    if cnt+z >= n-1:
16        break
```

Listing 3. Método para acelerar flujo eléctrico.

finalizando los pasos de avance si el ciclo llega al máximo (final de la muestra).

6. EXPERIMENTACIÓN

La experimentación de este trabajo se simulo en animaciones donde se puede observar el paso del flujo eléctrico (punto verde) a través de la muestra que se dopó a distintos porcentajes representando el dopaje en color azul y el cristal base como color rojo dichas animaciones se pueden consultar en el repositorio de Lagunes [3]. Una imagen de la representación de la muestra puede ser visualizada en la figura 1. El área obtenida de cada

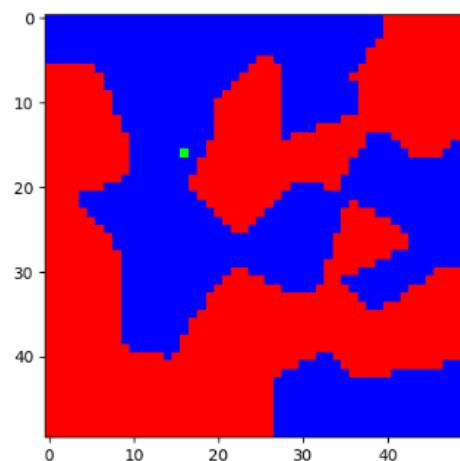


Fig. 1. Nanocristal dopado al 50%.

material tanto base como dopante se interpreta como nanómetros

cuadrados y la partícula verde que representa el flujo eléctrico atraviesa la muestra en diagonal a una velocidad constante siempre y cuando no detecte dopaje, si es así y encuentra dopaje entonces la velocidad de la partícula sera aumentada ya que un dopaje indica mejoramiento en velocidad pero a costa de menos pureza del material base.

* Resultados

Los resultados del presente trabajo se graficaron en distintas representaciones debido a que los fenómenos estudiados fueron tanto la velocidad del flujo eléctrico así como el tamaño del cristal de cada muestra respecto a cada porcentaje del semiconductor. El gráfico 2 muestra como varió el tamaño del cristal del material base respecto se iba variando la probabilidad del dopaje del semiconductor.

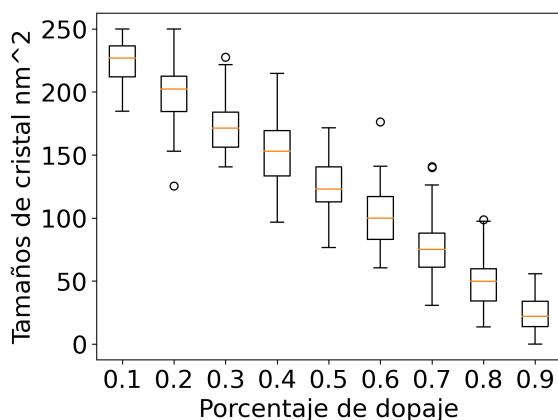


Fig. 2. Gráfico caja bigote del tamaño del cristal base respecto al porcentaje de dopante.

La siguiente figura 3 al igual que la anterior analiza el tamaño del cristal solo que ahora se promediaron los valores recaudados y se realiza una comparativa entre el tamaño de cristal de base (línea roja) y el dopaje (línea azul).

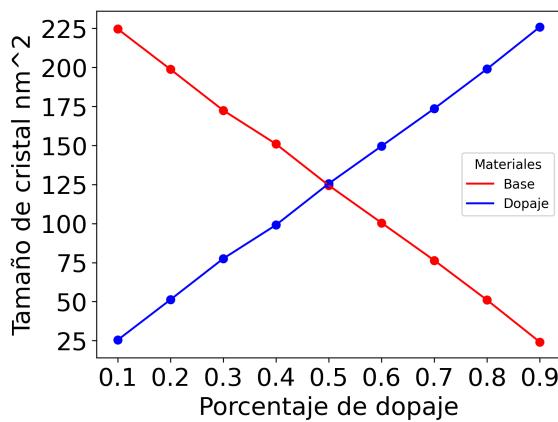


Fig. 3. Comparación tamaños de cristal base y dopante.

Por último la figura 4 muestra diagramas de violín y caja bigote combinados de como se vio afectado el tiempo o pasos en que terminó el flujo eléctrico de atravesar la muestra teniendo un tiempo máximo de 50 pasos.

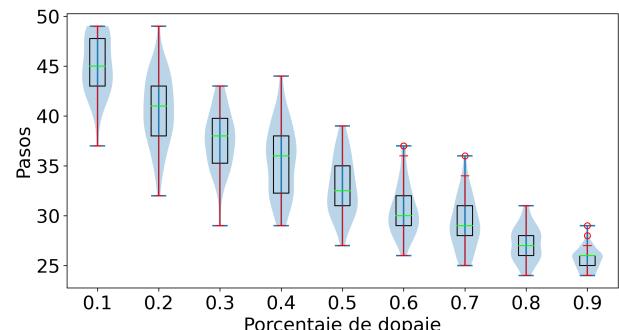


Fig. 4. Velocidad del flujo eléctrico por porcentaje.

Los datos de los gráficos anteriores son mostrados en la tabla 1 que muestra las probabilidades que se variaron en el experimento, los tamaños de cristal entre el material base y dopante y por ultimo el tiempo en pasos que tardó en recorrer la muestra teniendo un máximo de 50.

Cuadro 1. Tabla de tamaños de cristal y velocidad por probabilidad de dopaje.

Probabilidad	Tamaño de cristal			
	Base	Dopaje	Pasos	
0	0.1	227.542	22.458	45.04
1	0.2	196.906	53.094	40.20
2	0.3	170.272	79.728	36.82
3	0.4	154.494	95.506	35.84
4	0.5	129.650	120.350	33.06
5	0.6	98.976	151.024	30.50
6	0.7	76.442	173.558	28.84
7	0.8	49.852	200.148	27.04
8	0.9	27.826	222.174	25.74

La última tabla 2 obtenida, recauda los datos e información estadística de la gráfica 4, en esta se puede encontrar los promedios por porcentajes, así como la varianza, asimetría y curtosis.

Cuadro 2. Tabla estadística.

Dopaje	Promedio	Varianza	Asimetría	Curtosis
0.1	44.3	8.948	-0.561	-0.497
0.2	40.9	12.295	-0.111	-0.746
0.3	37.5	13.642	0.456	0.048
0.4	35.2	9.533	0.327	0.245
0.5	32.4	9.231	0.981	1.269
0.6	31.1	8.354	0.197	-0.158
0.7	28.8	5.171	0.413	-0.277
0.8	27.1	2.173	0.598	1.299
0.9	25.3	1.724	1.398	2.396

7. CONCLUSIONES

En base a los resultados se puede concluir que la velocidad dependerá siempre del porcentaje que sea agregado de dopante ya que el flujo o conductividad será mas rápido aunque la pureza del material tiende a disminuir drásticamente, pero tal como se muestra en la figura 3 ocurre un cruzamiento de las líneas de los tamaños promedios del cristal, así que para mantener una buena proporción de pureza y dopante lo ideal es el 50 % de impurezas en la muestra para tener una buena conducción eléctrica que se observa en la tabla 1 en la probabilidad de 0,5.

*. Trabajo a futuro

Como trabajo a futuro, la simulación puede ser ampliada para que la muestra admita 2 o 3 impurezas más, y aunado a esto realizar una variante de la dimensión de la muestra ya que en este estudio fue fija esa condición, de igual manera se podrían estudiar efectos de como se vería afectada la conducción térmica.

REFERENCIAS

1. Python sitio oficial. URL <https://www.python.org/>. consultado: 2022-05-26.
2. Repositorio github: satuelisa. URL <https://github.com/satuelisa/Simulation>. consultado: 2022-05-24.
3. Repositorio github: Raullr28. URL <https://github.com/Raullr28/Resultados/tree/main>. consultado: 2022-05-28.
4. Diagramas de voronoi. URL <https://satuelisa.github.io/simulation/p4.html>. consultado: 2022-05-20.
5. Ballesteros A. *Síntesis, caracterización y aplicaciones catalíticas de nanoestructuras de carbono y de carbono dopado con nitrógeno*. 2010. URL <https://dialnet.unirioja.es/servlet/tesis?codigo=84950>.
6. Wang B. & Cormack A. Molecular dynamics simulations of mg-doped beta -alumina with potential models fitted for accurate structural response to thermal vibrations. *Solid State Ionics*, 263:9–14, 2014. .
7. Nongjai R., Khan S., Asokan K., Ahmed H. & Khan I. Magnetic and electrical properties of in doped cobalt ferrite nanoparticles. *Journal of applied physics*, 112(8):084321, 2012. .
8. Willis K., Hagness S. & Knezevic I. Terahertz conductivity of doped silicon calculated using the ensemble monte carlo/finite-difference time-domain simulation technique. *Applied Physics Letters*, 96(6):062106, 2010. .
9. Schroder K. *Semiconductor material and device characterization*. John Wiley & Sons, 2015. ISBN 9780471749080.
10. Erwig M. The graph voronoi diagram with applications. *Networks: An International Journal*, 36(3):156–163, 2000. .
11. Santos M. *Dopado de nanopartículas semiconductoras de banda ancha: caracterización estructural y evaluación fotoelectroquímica*. 2014. URL <https://dialnet.unirioja.es/servlet/tesis?codigo=51137>.