Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

Posgrado en Ciencias de la Ingeniería con Orientación en Nanotecnología

Maestría

Portafolio de Evidencias

de

Nestor David Rodríguez Regalado

1460647

curso de Simulación Computacional de Nanomateriales,

con el profesor Virgilio González en colaboración satu elisa schaeffer.

Semestre Enero 2022 - Junio 2022.

Práctica												T	P	C
t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12			
5	5	5	8	6	4	7	8	6	5	4	5	68	$4+4+5+3+0+4=20$	88

https://github.com/NestorZeus/SIMULACION-COMPUTACIONAL-DE-NANOMATERIALES

"P0"

NESTOR RODRIGUEZ

January 2022

Resumen

Practica sencilla y unica de un demo del uso de LATEX en Overleaf.

1. Introducción

Esta practica es un ejemplo de como realizar los reportes de nuestras tareas. Vamos a incluir una ecuacion (1):

$$f(x) = 10\tanh(x) - \int_0^\infty \frac{5}{10+x}\mathrm{d}x. \tag{1}$$

Tambien podemos citar fuentes. Al final se incluye la figura de una orquidea en la figura 1. Vamos a aprender ademas a citar fuentes ejemplo. Se incluye unas tablas 1 con algunos datos y en la figura [**?**] 1 hay una orquidea.

Figura 1: orchid recuperada de `https://www.floresyplantas.net/orquidea-cymbidium/` con licencia CC.

2. Creacion de tablas y cuadros

En esta sección se aprende a crear tablas y cuadros.

Cuadro 1: Cuadro comparativo de reacciones químicas

NP Permanganato de sodio	β	8.2230
NP Oxido de gadolinio	α	236.9102
NP Titanato de Bario	Γ	15.5690
NP Silicato de Calcio	ϵ	89.1691
NP Nanocelulosa	Δ	321.7810
NP Hidrato Cloruro de Magnesio	Ω	101.3010

Cuadro 2: Tablas.

Dato 1	Dato 2	Ω	Dato 3
NP 1	NP 2	π	NP 3
Resultado 1	Resultado 2	α	Resultado 3

2.1. Midiendo en R

Se demostrará las secuencias donde se muestra la tabla de las mediciones en los siguientes parámetros logrando el tamaño en nanometros en medición R.

Cuadro 3: Medidas de tiempo y tamaño en R

Matrices	Datos	Tiempo (s)	Tamaño (nm)	Observaciones
1	9560	< 0.01	8024703	El dato resultado es menor a 1.09
2	120	< 1.20	3780164	El dato resultado es mayor a 1.20
3	2360	1.03	9113822	El dato resultado es menor a 1.03
4	2048	< 2.40	1265499	El dato resultado es mayor a 2.40
5	4096	3.01	4708691	El dato resultado es menor a 1
6	9600	1.02	9410875	El dato resultado es mayor a 1
7	9900	4.20	9782103	El dato resultado es igual a 2.0
8	9703	4.90	9587023	El dato resultado es mayor a 4.0
9	9380	5.10	9032458	El dato resultado es mayor a 5.0
10	9060	5.94	7598013	El dato resultado es mayor a 1.0
11	9799	5.85	8712039	El dato resultado es mayor a 1.0

2.2. Midiendo en Python

Se observa la siguiente tabla en la distribucion de la medicion en los vectores en Python.

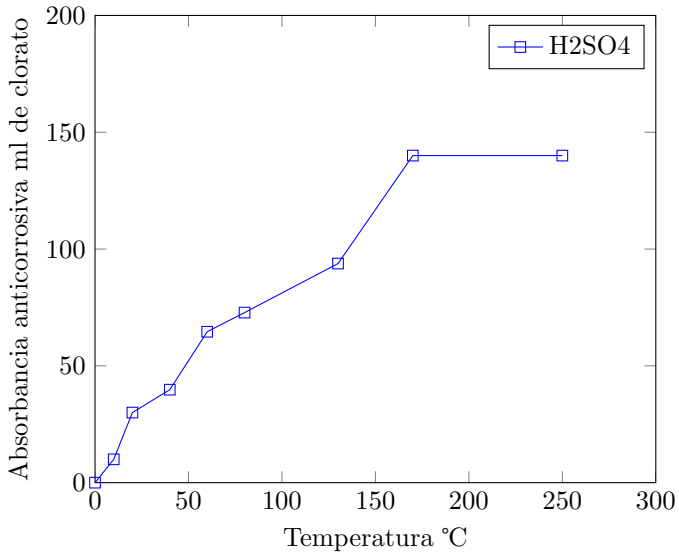
3. Grafica

En esta parte se aprenderá a como utilizar y hacer una grafica en LATEX

Cuadro 4: Mediciones en tiempo, tamaño y real en Python

Datos	Tiempo (s)	Tamaño (nm)	Real
1	9560	< 0.01	8024703
2	120	< 1.20	3780164
3	2360	1.03	9113822
4	2048	< 2.40	1265499
5	4096	3.01	4708691
6	9600	1.02	9410875
7	9900	4.20	9782103
8	9703	4.90	9587023
9	9380	5.10	9032458
10	9060	5.94	7598013
11	9799	5.85	8712039

Absorbancia de la corrosion de H2SO4 con la Temperatura



4. Conclusion

En general todo el desarrollo de este documento me sirvió de práctica que me permitió explorar más a fondo las funciones que tiene el programa, además de familiarizarme con su interfaz, comandos y también obtuve más práctica con este programa. Aún así mismo considero que aún tengo mucho que aprender en cuanto a programación de códigos, dado que tuve muchas dificultades, batalle pero se logro el objetivo para aprender en esta parte para trabajar en Overleaf.

5. Referencias

[**?**] title=, author=Slater, Jay E, journal=Journal of Allergy and Clinical Immunology, volume=94, number=2, pages=139–149, year=1994, publisher=St. Louis, MO: CV Mosby Co., c1971-

"P1" Movimiento Browniano

NESTOR RODRIGUEZ

Febrero 2022

1. Introducción

El primer reto es estudiar de forma sistemática y automatizada el tiempo de ejecución de una caminata (en milisegundos) en términos del largo de la caminata (en pasos) y la dimensión. Para medir el tiempo de una réplica, ejecútala múltiples veces y normaliza con la cantidad de repeticiones para obtener un promedio del tiempo de una réplica individual. El segundo reto es realizar una comparación entre una implementación paralela y otra versión que no aproveche paralelismo en términos del tiempo de ejecución, aplicando alguna prueba estadística adecuada para determinar si la diferencia es significativa.

2. Objetivo

El objetivo de esta práctica es examinar de manera sistemática los efectos de la dimensión en la distancia Euclideana máxima del origen del movimiento Browniano para dimensiones 1, 2, 3, 4 y 5, variando el número de pasos de la caminata (100, 1000 o 10000 pasos), con 30 repeticiones del experimento para cada combinación (1 con 100, 1 con 1000, 1 con 10000, 2 con 100, 2 con 1000, etc.) y graficar todos los resultados en una sola figura con diagramas de caja-bigote.

3. Desarrollo

La definicion de Movimiento Browniano: hace referencia al movimiento de partículas microscopicas que experimentan un movimiento aleatorio debido a fluctuaciones termicas, fenómeno observado por primera vez en 1827 por R. Brown y descrito formalmente en 1905 por A. Einstein. Así que con esto se realizo para generar el código con objetivo de la práctica, esto se encuentran en: mi repositorio en GitHub. Inicie tomando como base el Tomando como base la página código proporcionado por la Dra. Elisa Schaeffer [1], se hace una modificacion para iterar entre la cantidad de pasos que se realizan en cada dimension.

3.1. Ejecución en Python

A continuacion se realiza el siguiente ejercicio en código Python para la realización de la ejecución en los siguientes comandos.

```
from random import random, randint
from math import fabs, sqrt
import matplotlib.pyplot as plt
import numpy as np
from time import time

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])))
            mayores.append(mayor)
            after = time()*1000
        results.append(mayores)
tiempo = after - before
print(tiempo)

#separar los resultados en tres grupos de caminatas
walks_1 = results[0:4]
walks_2 = results[4:8]
walks_3 = results[8:12]

#empezar a graficar

ticks = ['1', '2', '3', '4', '5']

#funcion definir los colores de cajas y bigotes
def set_box_color(bp, color):
    plt.setp(bp['boxes'], color='green')
    plt.setp(bp['whiskers'], color='gray')
    plt.setp(bp['caps'], color='red')
    plt.setp(bp['medians'], color='purple')

def box_plot(data, edge_color, fill_color):
    bp = ax.boxplot(data, patch_artist=True)

    for element in ['boxes', 'whiskers', 'fliers', 'means', 'medians', 'caps']:
        plt.setp(bp[element], color=edge_color)

    for patch in bp['boxes']:
        patch.set(facecolor=fill_color)

    for patch, color in zip(box['boxes'], colors):
```

```
        patch.set_facecolor(color)


plt.figure()

bpl = plt.boxplot(walks_1, positions=np.array(range(len(walks_1)))*6.0-1.0, sym='-1', widths=
bpc = plt.boxplot(walks_2, positions=np.array(range(len(walks_2)))*6.0, sym='-1', widths=1.2)
bpr = plt.boxplot(walks_3, positions=np.array(range(len(walks_3)))*6.0+1.0, sym='-1', widths=
set_box_color(bpl, '#67001f')
set_box_color(bpc, '#1a1a1a')
set_box_color(bpr, '#d6604d')


plt.xticks(range(0, len(ticks)*5, 5), ticks)
plt.ylim(0, len(ticks)*40)
plt.xlim(-3, len(ticks)*5)
plt.title('Distancia Manhattan')
plt.tight_layout()
plt.savefig('DistanciaMan.png')
plt.show()
```

4. Resultados

En esta seccion se muestra la grafica Manhattan en corelacion de dimensiones 1, 2, 3, 4 y 5, variando el número de pasos de la caminata (100, 1000 o 10000 pasos), con 30 repeticiones del experimento para cada combinación (1 con 100, 1 con 1000, 1 con 10000, 2 con 100, 2 con 1000, etc.).

5. Mediciones y Gráfica

Se observa la siguiente tabla en la distribucion de la medicion en los vectores en Python. A partir de las secciones anteriores encontramos que las principales características de este fenómeno son: Las partículas pequeñas tienen mayor velocidad. Las partículas se mueven más rápido en fluidos con poca viscosidad. La energía promedio de las partículas es proporcional a la temperatura. Aumenta la cantidad de pasos, lo cual se debe a que tiene las posibilidades de determinar en ciertos puntos diferentes del espacio.

Cuadro 1: Mediciones para la distancia Euclideana máxima del origen del movimiento Browniano para dimensiones

Medición	Pasos
1, 2, 3,4 y 5	100
1, 2, 3,4 y 5	1000
1, 2, 3,4 y 5	10000

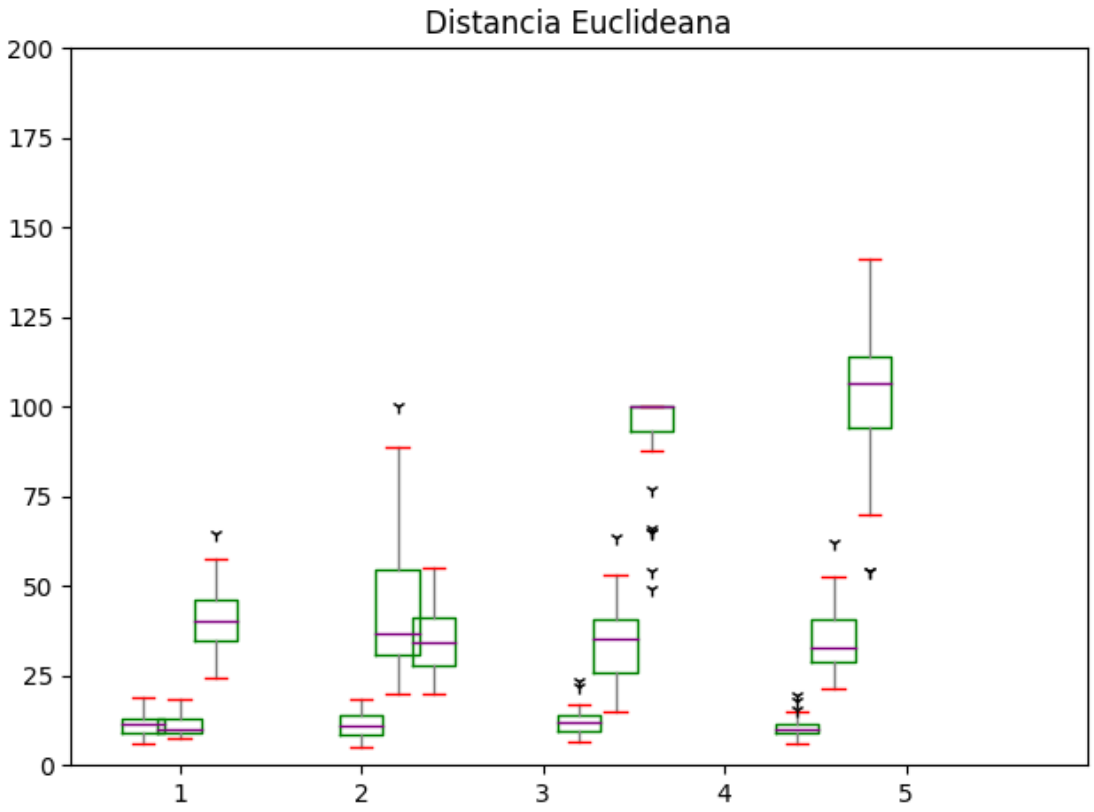


Figura 1: Diagrama caja-bigote en los grupos de 100, 1,000 y 10,000 pasos en dimensiones de 1 a 5 con 30 repeticiones.

6. Conclusion

Se realizo la tarea 1 con éxito para obtener la caja-bigote para las repeticiones del experimento. Al observar los diagramas de la figura 1 se caracteriza en que el rango del máximo que es amplio al aumentar la cantidad de pasos.

Referencias

[1] Dra. Elisa Schaeffer. Brownianmotion. *Repositorio, GitHub*, 2019.

"P2" Autómata Celular

NESTOR

Febrero 2022

1. Introducción

Un autómata celular es una colección de celdas coloreadas en una cuadrícula de forma específica que evoluciona a través de una serie de pasos de tiempo discretos de acuerdo con un conjunto de reglas basadas en los estados de las celdas vecinas. Luego, las reglas se aplican iterativamente durante tantos pasos de tiempo como se desee. Von Neumann fue una de las primeras personas en considerar tal modelo e incorporó un modelo celular. Wolfram realizó estudios exhaustivos de autómatas celulares a partir de la década de 1980, y la investigación fundamental de Wolfram en el campo culminó con la publicación de su libro A New Kind of Science (Wolfram 2002) en el que Wolfram presenta una colección gigantesca de resultados relacionados con autómatas, entre los que se encuentran una serie de nuevos descubrimientos innovadores [1]

2. Objetivo

El objetivo de esta práctica es examinar de manera sistemática con autómatas celulares en dos dimensiones, particularmente el famoso juego de la vida. El estado del autómata se representa con una matríz booleana (es decir, contiene ceros y unos). Cada celda es o viva (uno) o muerta (cero). En los extremos de la matríz, 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 igual como los lados izquiero y derecho uno con otro [2].

3. Desarrollo

La regla de supervivencia es sencilla: una celda está viva si exactamente tres vecinos están vivos. Para comenzar, usamos números pseudoaleatorios como el estado inicial [2]. En mi repositorio de Github tomando como base la página del código proporcionado por la Dra. Elisa Schaeffer [2], las 30 réplicas para estimar la probabilidad de creación de vida dentro de 50 iteraciones (es decir, haya celdas vivas al final de la réplica), usando niveles 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. A continuación se muestra los siguientes comandos:

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

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

```
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)

dimension = (10, 15, 20)
probabilidad = (0.2, 0.4, 0.6, 0.8)
grafico=[]
bp=[]
for dim in dimension:
    print("################ Dimensi n:",dim,"############################")
    num = dim**2
    vpp=[]#vivos por probabilidad
    mpp=[]#muertos por probabilidad
    for p in probabilidad:
        print("######### probabilidad:",p,"################")
        rep = 200
        vivieron=[]
        murieron=[]
        for replicas in range(rep):# ciclo de las 30 r plicas
            valores = [1 * (random() < p) for i in range(num)]
            actual = np.reshape(valores, (dim, dim))
            assert all([mapeo(x) == valores[x]  for x in range(num)])
            dur = 50
            historial = []
            for iteracion in range(dur):
                valores = [paso(x) for x in range(num)]
                vivos = sum(valores)
                historial.append(vivos)
                historial[-4:]
                cuantos = 4
                actual = np.reshape(valores, (dim, dim))
                if vivos == 0:
                    murieron.append(iteracion)#guarda las iteraciones en que murieron todos
                    break; # nadie vivo
                if len(historial) > cuantos and len(set(historial[-cuantos:])) == 1:
                    vivieron.append(iteracion)# guarda las iteraciones llegaron a 50
                    break;
        vpp.append(((len(vivieron))*100)/rep)#guarda el porcentaje que vivieron
        mpp.append(((len(murieron))*100)/rep)
        print("muertes en 30 r plicas:",len(murieron))
        print("vivos en 30 r plicas:",len(vivieron))
    print("vpp:",vpp)
    grafico.append(vpp)
print(grafico)

linedim10=[grafico[0][0],grafico[0][1],grafico[0][2],grafico[0][3]]
linedim15=[grafico[1][0],grafico[1][1],grafico[1][2],grafico[1][3]]
linedim20=[grafico[2][0],grafico[2][1],grafico[2][2],grafico[2][3]]
plt.plot([0,1,2,3],linedim10,label='dimensi n 10')
plt.scatter([0,1,2,3],linedim10)
plt.plot([0,1,2,3],linedim15,label='dimensi n 15')
```

```
plt.scatter([0,1,2,3],linedim15)
plt.plot([0,1,2,3],linedim20,label='dimensi n 20')
plt.scatter([0,1,2,3],linedim20)
plt.xticks([0,1,2,3], ('0.2', '0.4', '0.6', '0.8'))
plt.ylabel('Probabilidad que no muera toda la poblaci n (%)')
plt.xlabel('Probabilidad')
plt.title('Gr fico de porcentaje de supervivencia de poblacion')
plt.legend()
plt.show()
```

4. Resultados

En esta seccion se muestra el gráfico de porcentaje de supervivencia de población con respecto al porcentaje de la probabilidad que no muera toda la población de 200 réplicas para estimar la probabilidad de creación de vida dentro de 50 iteraciones.

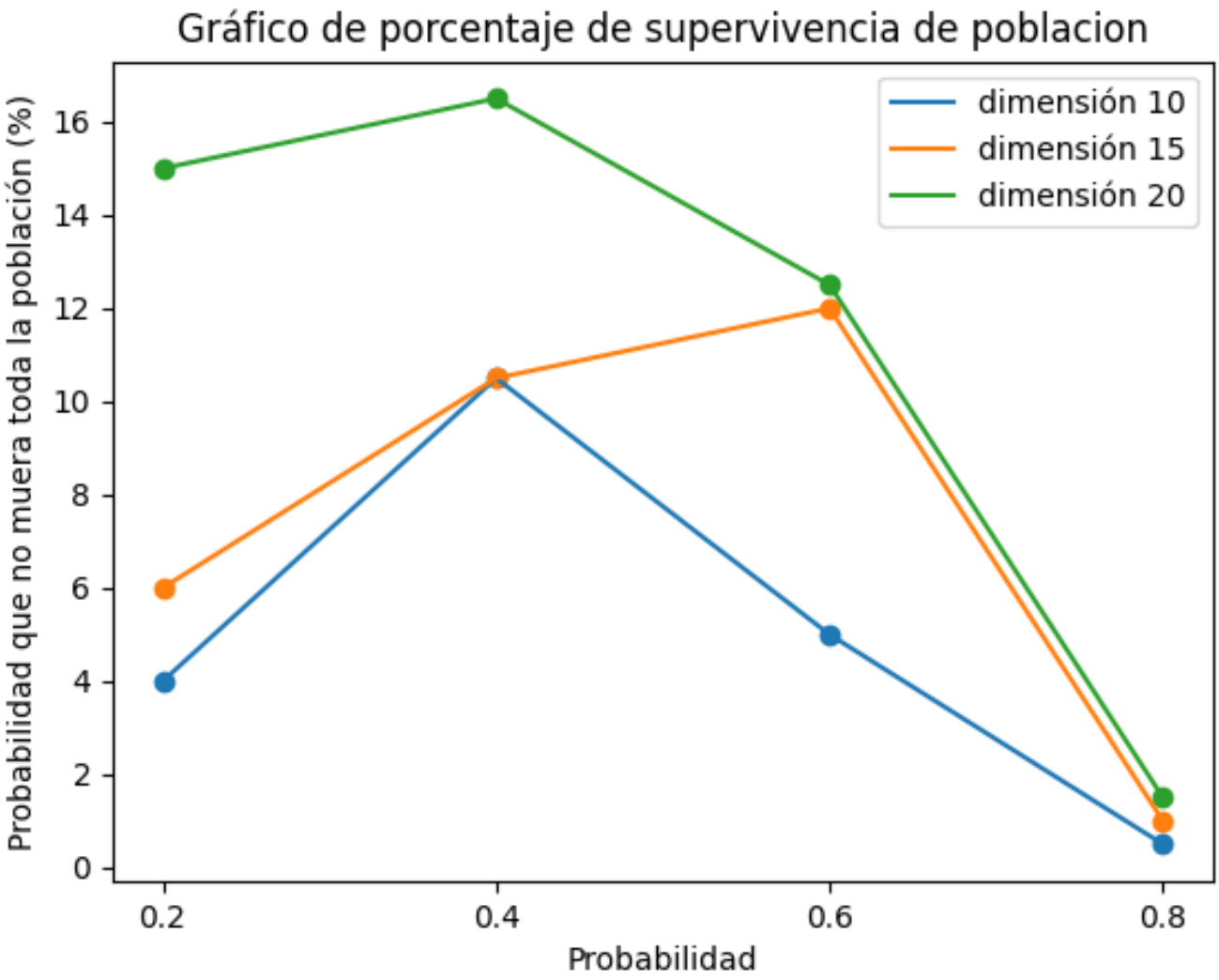


Figura 1: Diagrama puntos y líneas con dimensiones de 10, 15 y 20 con 200 replicas.

5. Mediciones

Para este caso se utilizaron 200 réplicas en lugar de 30 réplicas, para poder visualizar mejor el comportamiento de las dimensiones.

Cuadro 1: Mediciones para la probabilidad en las dimensiones de: 10, 15 y 20

Dimensión	Probabilidad	Muertes	Réplicas	Vivos	Réplicas	Vpp
10	0.2	30	192	30	8	4.0
	0.4	30	179	30	21	5.5
	0.6	30	190	30	10	3.0
	0.8	30	199	30	1	0
15	0.2	30	188	30	12	6.5
	0.4	30	179	30	21	11.5
	0.6	30	176	30	24	6.0
	0.8	30	198	30	2	0.5
20	0.2	30	170	30	30	13.5
	0.4	30	167	30	33	13.5
	0.6	30	175	30	25	16.5
	0.8	30	197	30	3	2.0

6. Conclusion

El comportamiento de los resultados nos muestra que la probabilidad que se utilizó patra este caso es de 200 réplicas ya que se puede representar mejor el diagrama.

Referencias

[1] Steve Weston Rich. Getting started with doparallel and foreach. 2015.

[2] E. Schaeffer. Autómata celular. *Repositorio, GitHub*, 2022.

"P3" Teoría de Colas

NESTOR

Marzo 2022

1. Objetivo

El objetivo de la práctica consiste en analizar los tiempos de ejecución de ("n") algoritmos de diferentes que se encuentran en un número entero dado que es un número primo. Ya que un número primo es un número natural mayor que ("1") que tiene únicamente dos divisores positivos distintos. Ya que que se trata de un número positivo que es imposible de expresar como producto de otros dos números enteros igualmente positivos pero menores que él o en su defecto como un producto de dos números enteros que poseen varias formas.

2. Desarrollo

De acuerdo con el desarrollo en el código implementado por E. Schaeffer [4], primero se definen los parámetros de ejecución de los algoritmos, donde se crea una lista de números desde 1000 hasta 7919. A continuación se muestra los siguientes comandos:

```
from math import ceil, sqrt
from random import shuffle
import multiprocessing
from time import time
from scipy.stats import f_oneway
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

d = 1000
h = 7919
replicas = 30
original = [x for x in range(d, h + 1)]
invertido = original[::-1]
aleatorio = original.copy()
shuffle(aleatorio)
cores = multiprocessing.cpu_count()

def primo_1(n):
if n < 3:
    return True
for i in range(2, n):
    if n % i == 0:
    return False
    return True
```

```
def primo_2(n):
if n < 4:
    return True
if n % 2 == 0:
    return False
for i in range(3, n - 1, 2):
    if n % i == 0:
    return False
    return True

def primo_3(n):
if n < 4:
    return True
if n % 2 == 0:
    return False
for i in range(3, int(ceil(sqrt(n))), 2):
    if n % i == 0:
    return False
    return True

def primo_4(n):
if n < 4:
    return True
if n % 2 == 0:
    return False
for i in range(4, int(ceil(sqrt(n))), 3):
    if n % i == 0:
    return False
    return True

if __name__ == "__main__":
resultados_1 = {'Prueba 1': [], 'Prueba 2': [], 'Prueba 3': [], 'Prueba 4': []}
resultados_2 = {'Resultado 1': [], 'Resultado 2': [], 'Resultado 3': [], 'Resultado 4': []}
with multiprocessing.Pool(processes = cores-1) as pool:
  pool.map(primo_1, original)
  for r in range(replicas):
    t = (time()*1000)
    pool.map(primo_1, original)
    resultados_1['Prueba 1'].append((time()*1000)-t)
    t = (time()*1000)
    pool.map(primo_2, original)
    resultados_1['Prueba 2'].append((time()*1000)-t)
    t = (time()*1000)
    pool.map(primo_3, original)
    resultados_1['Prueba 3'].append((time()*1000)-t)
    t = (time()*1000)
    pool.map(primo_4, original)
    resultados_1['Prueba 4'].append((time()*1000)-t)
    t = (time()*1000)
    pool.map(primo_3, original)
    resultados_2['Resultado 1'].append((time()*1000) - t)
    t = (time()*1000)
    pool.map(primo_3, invertido)
    resultados_2['Resultado 2'].append((time()*1000) - t)
```

```
    t = (time()*1000)
    pool.map(primo_3, aleatorio)
    resultados_2['Resultado 3'].append((time()*1000) - t)
    t = (time()*1000)
    pool.map(primo_3, aleatorio)
    resultados_2['Resultado 4'].append((time()*1000) - t)
df1 = pd.DataFrame(data = resultados_1)
df2 = pd.DataFrame(data = resultados_2)
print(df1, '\n', df2)
stat1, p1 = f_oneway(resultados_1['Prueba 1'],
    resultados_1['Prueba 2'],
    resultados_1['Prueba 3'],
    resultados_1['Prueba 4'])
print('Variando algoritmo\n', 'stat=%.3f, p=%.3f' % (stat1, p1))
if p1 > 0.05: print('Estad sticamente no significativa\n')
else:
print('Estad sticamente significativa\n')
    stat2, p2 = f_oneway(resultados_2['Resultado 1'],
    resultados_2['Resultado 2'],
    resultados_2['Resultado 3'],
    resultados_2['Resultado 4'])
print('Variando orden de numeros\n', 'stat=%.3f, p=%.3f' % (stat2, p2))
if p2 > 0.05:
 print('Estad sticamente no significativa\n')
else:
 print('Estad sticamente significativa\n')

    sns.violinplot(data = df1, scale='count')
    plt.savefig('Prueba.png')
    plt.show()
    sns.violinplot(data = df2, scale='count')
    plt.savefig('Resultado.png')
    plt.show()
```

3. Resultados

Se muestra en las gráficas para visualizar la distribución de los datos y su densidad de probabilidad. Este gráfico es una combinación de un diagrama de cajas y bigotes y un diagrama de densidad girado y colocado a cada lado, para mostrar la forma de distribución de los datos. La barra negra gruesa en el centro representa el intervalo intercuartil, la barra negra fina que se extiende desde ella, representa el 95 por ciento de los intervalos de confianza, y el punto blanco es la mediana. A continuación se muestra las siguiemtes gráficas:

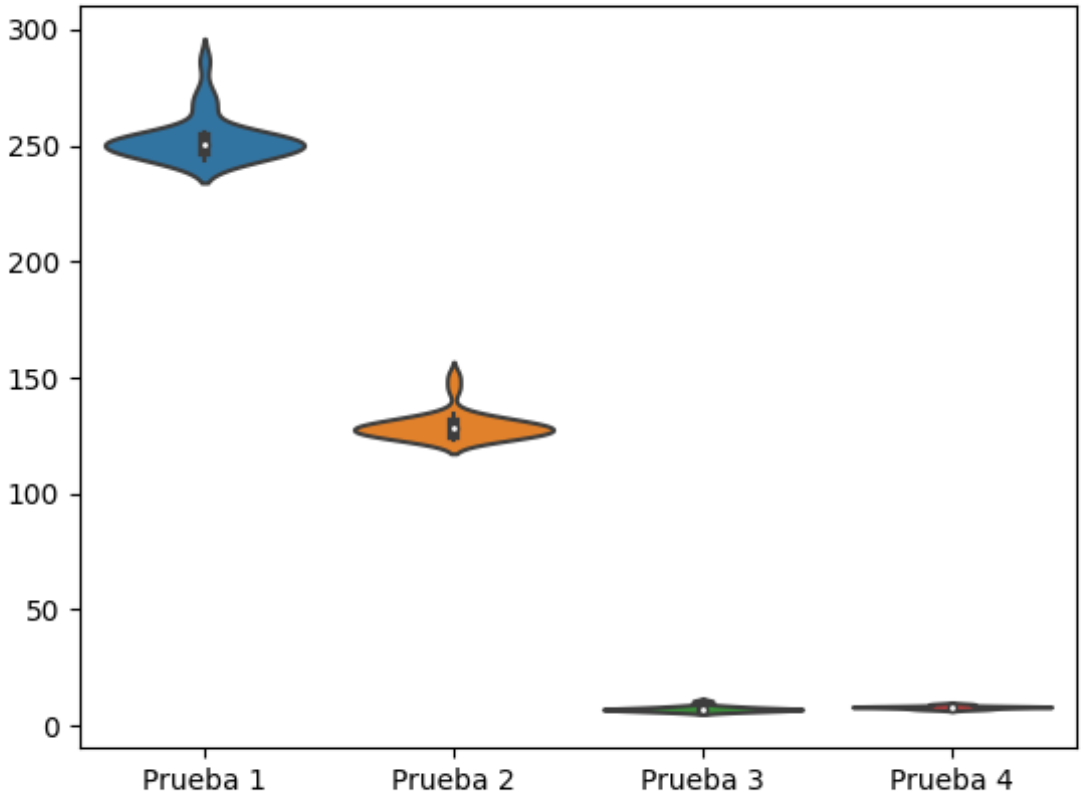


Figura 1: Prueba de Algoritmo

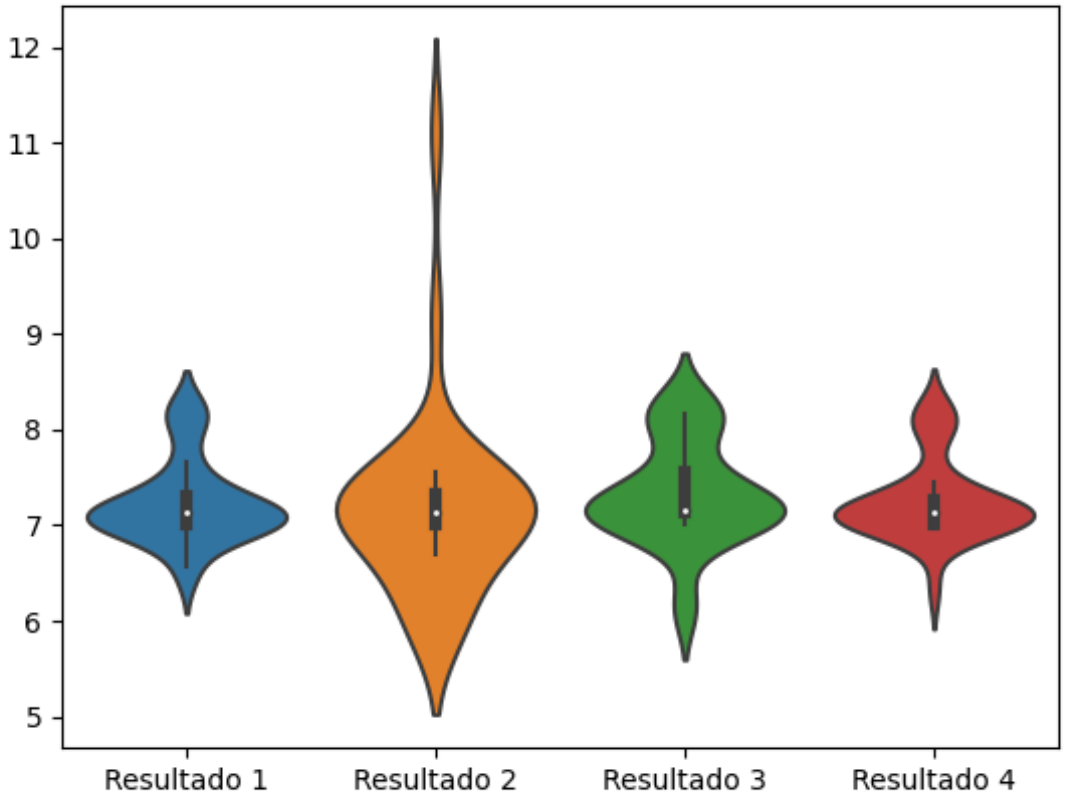


Figura 2: Resultado de Variación

4. Conclusiones

Como puede verse en la tabla, tenemos una densidad más alta entre 100 y 300 para la figura 1 y para la figura 2 es de 5 y 10. Eso es muy significativo porque, como en la descripción nos da un valor medio.

Referencias

[1] J. Hunter. Lines, bars and markers. *Repositorio, GitHub*, 2021.

[2] D. Leyva. Teoría de colas. *Repositorio, GitHub*, 2021.

[3] N. Pérez. Teoría de colas. *Repositorio, GitHub*, 2021.

[4] E. Schaeffer. Queuingtheory. *Repositorio, GitHub*, 2022.

[5] J. Torres. Teoría de colas. *Repositorio, GitHub*, 2022.

[5] [4] [3] [2] [1]

"P4" Diagramas de Voronoi

NESTOR

Marzo 2022

1. Objetivo

El objetivo de la práctica consiste en un espacio bidimensional una zona con medidas conocidas que contiene puntos semilla , representados por sus coordenadas. 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 que estén más cerca de esa semilla que a cualquier otra.

2. Desarrollo

Basando el desarrollo en la codificación implementado por E. Schaeffer [2] y se encuentra todas las instrucciones en el repositorio repositorio de Nestor en GitHub.

Para comenzar se hace primero generar el cuadro de las semillas con las funciones.

Código 1: Función con las semillas

```
def celda(pos):
   if pos in semillas:
   return semillas.index(pos)
   x, y = pos % n, pos // n
   cercano = None
   menor = n * sqrt(2)
   for i in range(k):
   (xs, ys) = semillas[i]
   dx, dy = x - xs, y - ys
   dist = sqrt(dx**2 + dy**2)
   if dist < menor:
   cercano, menor = i, dist
return cercano
```

Posteriormente al tener las celdas se crea generar el cuadro con las funciones. Lo siguiente es definir en una función para las semillas y las grietas implementados por E. Schaeffer, como se muestra a continuación en el código 2.

Código 2: Semillas para encontrar grietas

```
def inicio():
    direccion = randint(0, 3)
    if direccion == 0: # vertical abajo -> arriba
        return (0, randint(0, n - 1))
    elif direccion == 1: # izq. -> der
        return (randint(0, n - 1), 0)
    elif direccion == 2: # der. -> izq.
        return (randint(0, n - 1), n - 1)
    else:
        return (n - 1, randint(0, n - 1))


def propaga(replica,rupturas):
    grieta = voronoi.copy()
    for f in range(rupturas):
        prob, dificil = 0.9, 0.8
        #grieta = voronoi.copy()
        g = grieta.load()
        (x, y) = inicio()
```

```
        largo = 0
        cnt=0
        if f == 0:
            color=(0,0,0)# grieta color negro
        if f == 1:
            color = (0, 0, 255)# color azul de grieta para el segundo ciclo
        while True:
            g[x, y] = color
            largo += 1
            frontera, interior = [], []
```

Para saber el programa de como se genera el cuadro que se llama las funciones y se pueda generar, grietas implementados por E. Schaeffer, se muestra a continuación en el código 3.

Código 3: Ejecución de códigos

```
for k in 5, 80, 250:
   ciclos=ciclos+1
   print("########## semillas",k,"###############")
   for s in range(k):
    while True:
    x, y = randint(0, n - 1), randint(0, n - 1)
    if (x, y) not in semillas:
    semillas.append((x, y))
    break

    celdas = [celda(i) for i in range(n * n)]
    voronoi = Image.new('RGB', (n, n))
    vor = voronoi.load()
    c = sns.color_palette("Set3", k).as_hex()
    for i in range(n * n):
    vor[i % n, i // n] = ImageColor.getrgb(c[celdas.pop(0)])
    limite, vecinos = 1, []
```

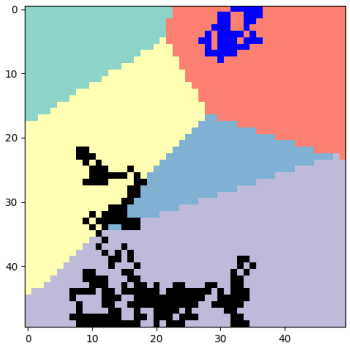
Se realiza para saber las réplicas y en la función de la grieta, se agrega aparte de que haga los ciclos de cincuenta veces, se agrega variables que se conoce como: rupturas. a continuación se muestra el siguiente código.

Código 4: Rupturas

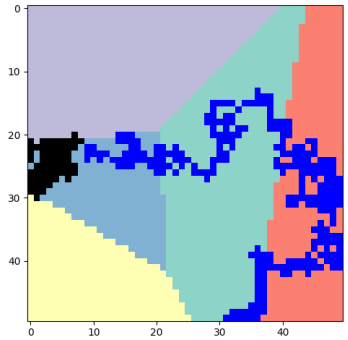
```
rupturas=2# cantidad de grietas
        contacto=[]
        for r in range(300): # replicas
            atraveso = propaga(r,rupturas)
            #print("Por replica:",r, "resultado:",atraveso)
        print("cuantos atravesaron",len(atraveso))
        if ciclos == 1:
            prob["min_sem"].append(((len(atraveso))/300)*100)
        if ciclos == 2:
            prob["med_sem"].append(((len(atraveso))/300)*100)
        if ciclos == 3:
            prob["max_sem"].append(((len(atraveso))/300)*100)
    print("final de probabilidades",prob)
```

3. Resultados

A continuación se muestra las semillas y las grietas, en la figura 1. Donde se da a demostrar que las grietas no se tocan figura 1a, y otro donde las grietas se tocan figura 1b. En la figura 2 se muestra las gietas de las semillas de mayor valor 2a y 2b. Posteriormente, en la figura 3 se muestran el porcentaje de las probabilidades que se calcularon de las grietas de las semillas con las dimensiones. Y finalmente una analisis de los datos multivariados en la figura 4.

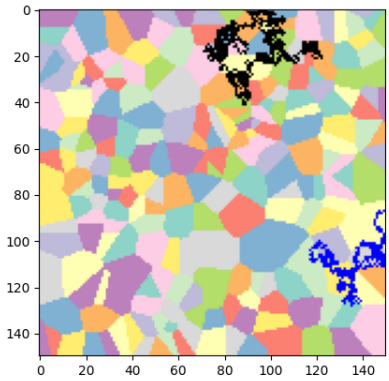


(a) Las grietas no se tocan.

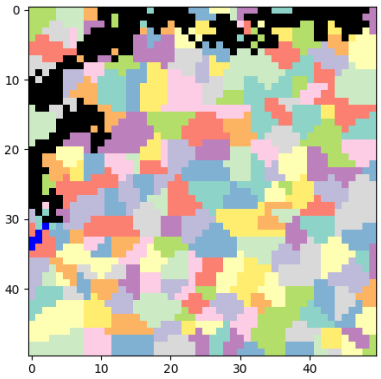


(b) Las grietas se tocan.

Figura 1: Grietas.



(a) Las grietas no se tocan.



(b) Las grietas se tocan.

Figura 2: Semillas de grietas de mayor valor.

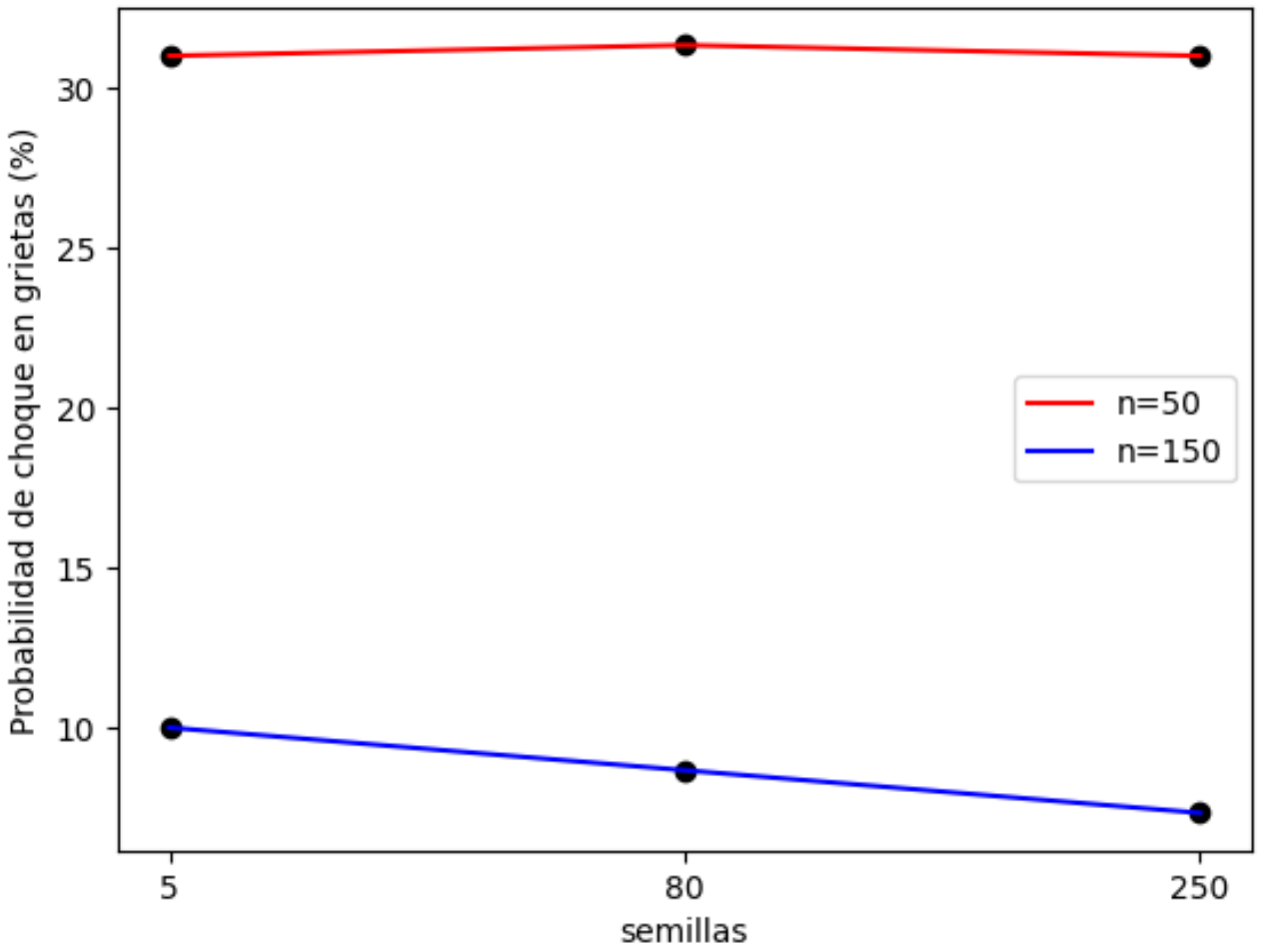


Figura 3: Probabilidad de las grietas

3.1. Radar Chart

Una Gráfica de Radar, también conocida como un diagrama de araña, es una herramienta muy útil para mostrar visualmente los gaps entre el estado actual y el estado ideal. [1]. A continuación se presenta visualmente los gaps existentes entre el estado actual y el estado ideal.

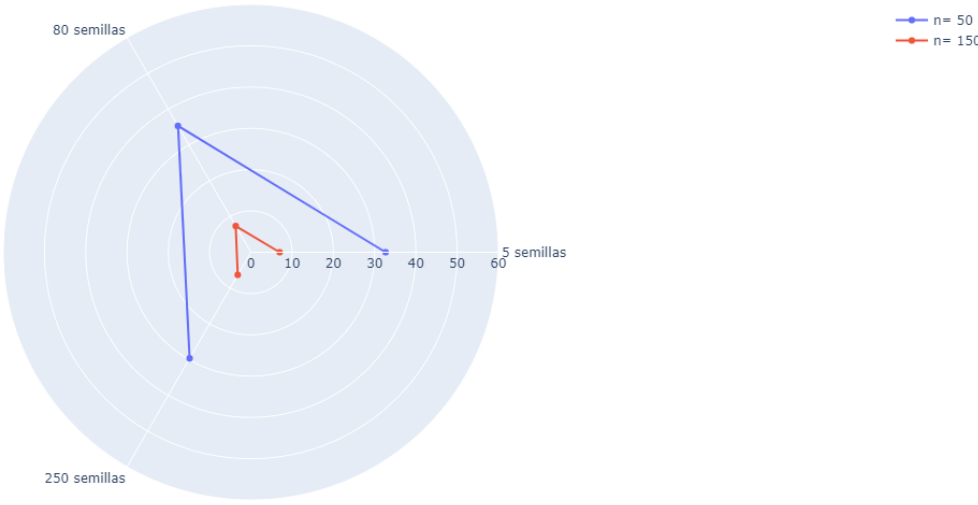


Figura 4: Analisis de los datos multivariados

4. Conclusiones

Se concluye que las fronteras va aumentando con respecto a la cantidad de semillas, ya que las grietas no se aminoran. Sin embargo en la gráfica de radar chart nos muestra como se comporta las semillas dando valores multivariados.

5. Reto 1

En este reto consiste en 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 demas semillas [2].

Para esto se crea una función para la visualización y creación de las semillas de como van creciendo ya que la práctica es lo mismo, este reto consiste si las grietas chocan solo que se modifican la creación de las semillas como se muestra en las siguientes instrucciones:

Código 5: Función con las semillas

```
def union(seeds,im,incr):
  for a in range(len(seeds)):
    x=seeds[a][0]
    y=seeds[a][1]
    r = incr[a]
    color=im.getpixel((x, y))
    img=im.copy()
    image = img
    draw = ImageDraw.Draw(image)
    draw.ellipse((x-r, y-r, x+r, y+r), fill=color)
    for Y in range(n):
```

Luego creamos la posición y color de las semillas como se mjuestra en la siguiente instrucción:

Código 6: Posición y Color de las semillas

```
for s in range(k):
            while True:
                x, y = randint(0, n - 1), randint(0, n - 1)
                if (x, y) not in semillas:
                    semillas.append((x, y))
                    break
        col=[]
        for dato in paleta:
            col.append([(int(i * 255)) for i in dato])
```

Para saber el comportamiento de las semmillas y el crecimiento, se muestra en la siguiente instrucción:

Código 7: Comportamiento y Crecimiento de las semillas

```
for i in range(len(semillas)):
  voronoi.putpixel(semillas[i],(tuple(col[i])))
    #plt.imshow(voronoi)###### IMPRIMIR
    #plt.show()
    #vo=voronoi.copy()
    #visual = vo.resize((10 * n,10 * n))
    #visual.save("ciclo_0_ini.png")
    p=0.4
    aumento=[]
    semi=[]
    for s in range(n):
    #fondo=[]
    #print("######## ciclo:",s)
    if s == 0:
      semi.append(semillas[0])
      semillas.pop(s)
      aumento.append(0)
```

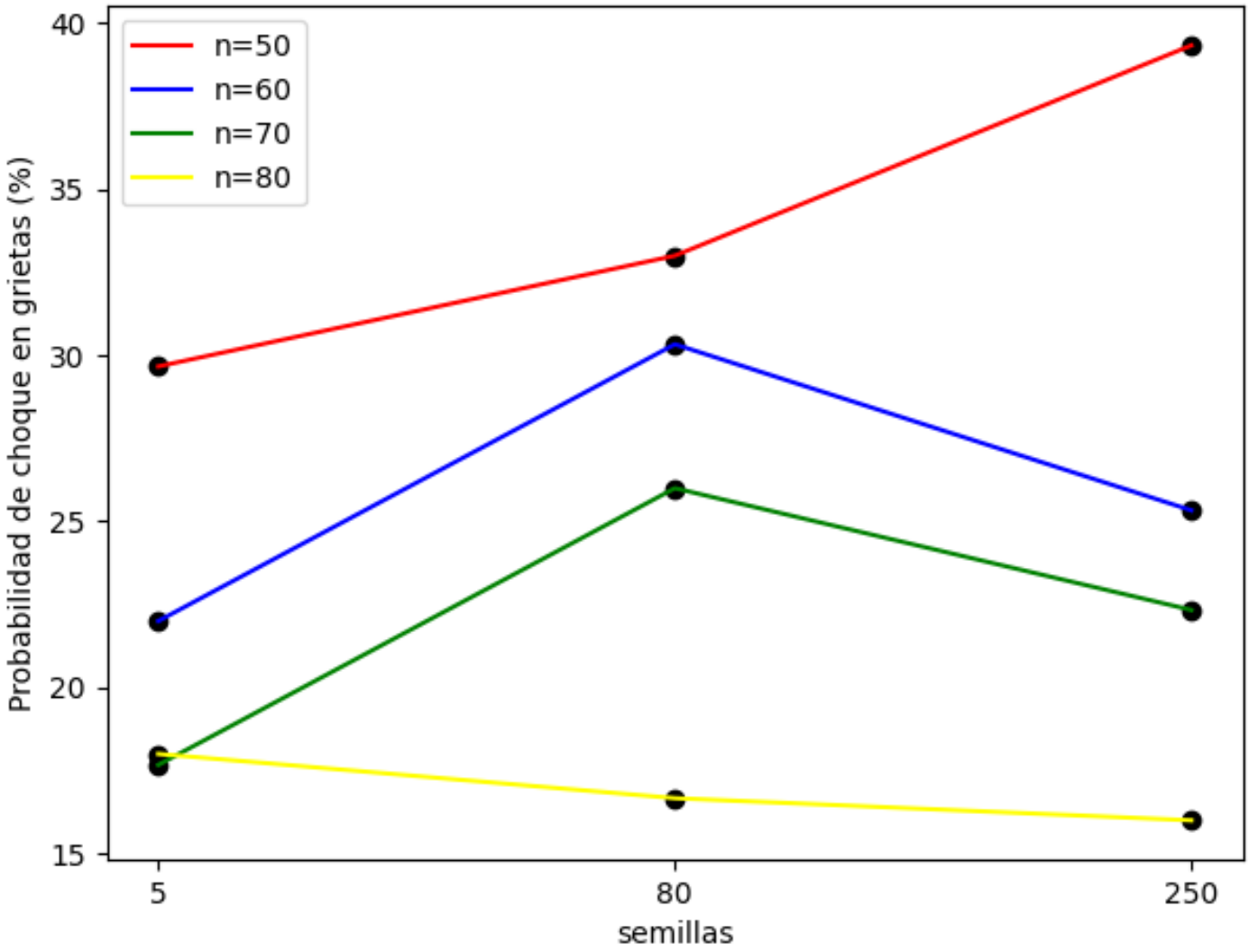
Para obtener los resultados se muestra para las semillas que se crearon, fueron creciendo hasta que se cristalizó con el aumento de más semillas como se muestra a continuación:

Código 8: Obtención del resultado comportamiento del crecimiento de las semillas

```
if s != 0 and ((random.uniform(0, 1)) > p) and len(semillas)>0:
                rnd=random.choice(semillas)
                semi.append(rnd)
                semillas.remove(rnd)
                aumento.append(0)
            aumento=[s+1 for s in aumento]
            voronoi= union(semi,voronoi,aumento)
            #[[fondo.append(voronoi.getpixel((x,y))) for x in range(5)]for y in range(5)]
            #vis=voronoi.copy()
            #visual = vis.resize((10 * n,10 * n))
            #visual.save("ciclo_{:d}.png".format(s))
            #plt.imshow(voronoi)
            #plt.show()
        #plt.imshow(voronoi)####### IMPRIMIR
        #plt.show()################
```

6. Reto 1 resultados

Con el aumento de más semillas se cristalizó y se analizó lo mismo del comportamiento de las grietas. Se encuentra todos los resultados y gifs en el repositorio repositorio de Nestor en GitHub.

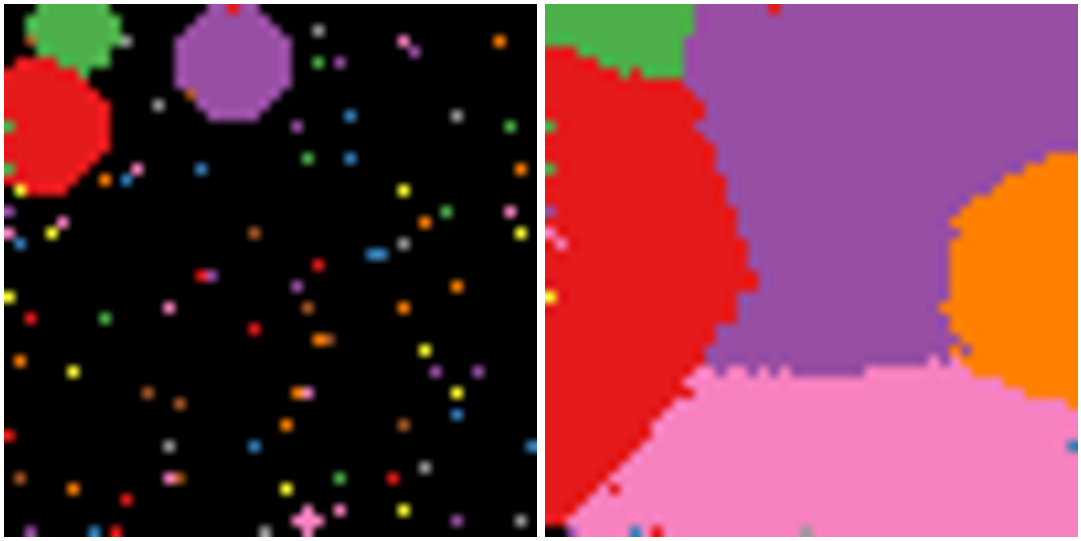

n=50
n=60
n=70
n=80
Probabilidad de choque en grietas (%)
40
35
30
25
20
15
5
80
250
semillas

(a) Las fronteras no se tocan. (b) Las fronteras se tocan.

Figura 3: Intervalo de 10 Semillas de las Grietas

(a) Las fronteras no se tocan. (b) Las fronteras se tocan.

Figura 4: Intervalo de 80 Semillas de las Grietas



(a) Las fronteras no se tocan. (b) Las fronteras se tocan.

Figura 5: Intervalo de 100 Semillas de las Grietas

Referencias

[1] J. Hunter. Lines, bars and markers. *Repositorio, GitHub*, 2021.

[2] E. Schaeffer. Autómata celular. *Repositorio, GitHub*, 2022.

[1]

"P5" Método Monte-Carlo

NESTOR

Marzo 2022

1. Objetivo

El objetivo de esta actividad es el estudiar como es el comportamiento en la situación que se combinan en los conceptos estadísticos como lo es el muestreo aleatorio, con la generación de números aleatorios y la automatización de los cálculos. Es un procedimiento matemático que consiste en la generación numérica de series mediante un muestreo aleatorio de las distribuciones de probabilidad [1].

2. Desarrollo

El desarrollo de la actividad está basado en el código implementado por E. Schaeffer [3]. Basando el desarrollo, en el código [3] se establecen las instrucciones para la ejecución del programa. Se encuentra todas las instrucciones en el repositorio de Nestor en GitHub.

Código 1: Ejecución de Parámetros

```
desde = 2
hasta = 5
pedazo = 50000
cuantos = [500, 5000, 50000]
```

En una receta y ejecutando con la función, se genera una cantidad de números aleatorios con una distribución de acuerdo a la integral normalizada. Esto se define un conjunto de funciones que se utilizan para generar o manipular números aleatorios a través del módulo aleatorio. Las funciones del módulo aleatorio se basan en una función generadora de números pseudoaleatorios random, que genera un número flotante aleatorio entre 0,0 y 1,0 [1]

Código 2: Instrucciones para Generador Random

```
from GeneralRandom import GeneralRandom
generador = GeneralRandom(np.asarray(X), np.asarray(Y))

def parte(replica):
    V = generador.random(pedazo)[0]
    return ((V >= desde) & (V <= hasta)).sum()

def compare_strings(a, b):
    a = str(a)
    b = str(b)

    if a is None or b is None:
        return 0

    size = min(len(a), len(b))
    count = 0

    for i in range(size):
        if a[i] == b[i]:
            count += 1
        else:
            break
    return count
```

¿Qué es el multiprocesamiento de importación en Python? multiprocesamiento es un paquete que admite procesos de generación mediante una API similar al módulo de subprocesamiento. El paquete de multiprocesamiento

ofrece simultaneidad tanto local como remota, eludiendo efectivamente el bloqueo global del intérprete mediante el uso de subprocesos en lugar de subprocesos [1] .

Código 3: Instrucción para calcular iteraciones

```
import multiprocessing
if __name__ == "__main__":
  with multiprocessing.Pool() as pool:
    for c in cuantos:
    p = c * pedazo
    montecarlo = pool.map(parte, range(c))
    integral = sum(montecarlo) / p
```

Se proporciona una serie de funciones y clases útiles para gestionar los subprocesos y las comunicaciones entre ellos. Las iteraciones del análisis y se calculan los valores de los errores para poder comparar las estimaciones de la integral definida dependiendo de la cantidad de iteraciones para representar los resultados.

Código 4: Resultados de iteraciones

```
resultados = {'Iteraciones': puntos,
                      'Error Absoluto': ae,
                      'Error Cuadrado': se,
                      'Decimales Correctos': dec}
        df = pd.DataFrame(resultados)
        sns.barplot(data=df, x='Iteraciones',
                    y='Error Absoluto',
                    dodge=False)
        plt.savefig('AbsErr.png')

        plt.show()
        sns.barplot(data=df, x='Iteraciones',
                    y='Error Cuadrado',
                    dodge=False)
        plt.savefig('SqErr.png')
        plt.show()
        sns.barplot(data=df, x='Iteraciones',
                    y='Decimales Correctos',
                    dodge=False)
        plt.savefig('Decimals.png')
        plt.show()
        print(df)
```

3. Resultados

En los diagramas se representa como es el comportamiento de las variaciones e iteraciones cuando se da valores al momento de estimar un resultado.

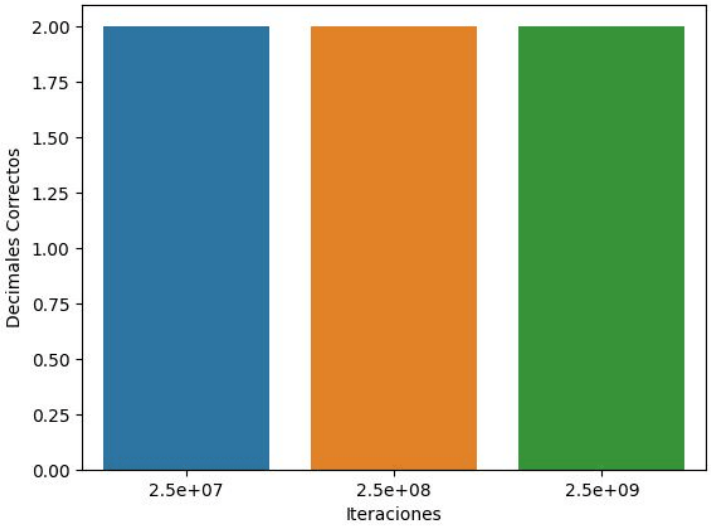


Figura 1: Diagrama decimal.

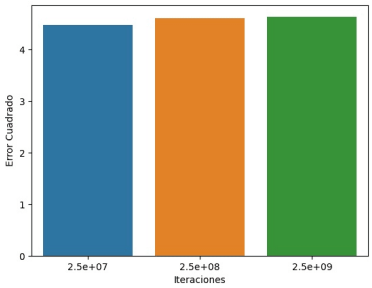
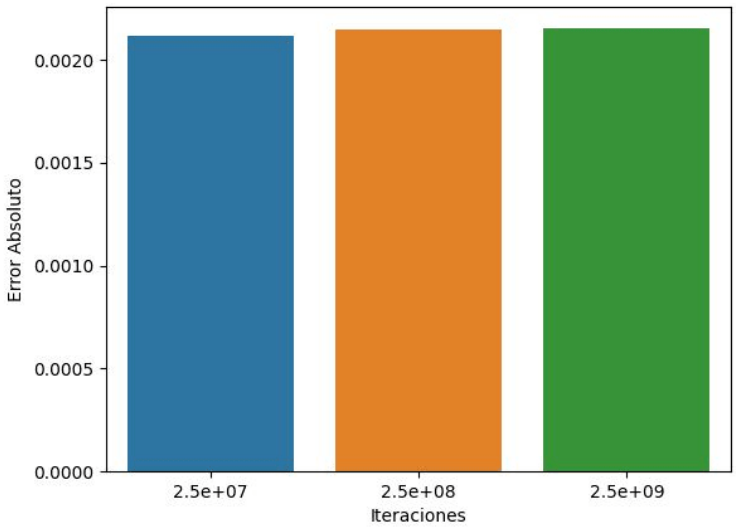


Figura 2: Diagrama Absoluto.



4. Conclusiones

Se concluye que las iteraciones representa una ligera variación va aumentando con respecto a la cantidad de iteraciones. Sin embargo cuando varía esto puede cambiar los valores para ser exactos debido a la ejecución.

[3] [2] [1]

Referencias

[1] J. Hunter. Lines, bars and markers. *Repositorio, GitHub*, 2021.

[2] J.FeroxDeitas. Github,pasos$_a$*leatorios.py.URL*.

E. Schaeffer. Github,montecarlo. URL https://github.com/satuelisa/Simulation/tree/master/MonteCarlo.

"P6" Sistema Multiagente

NESTOR

Marzo 2022

1. Objetivo

El objetivo de esta actividad es el implementar un sistema multiagente con una aplicación en epidemiología. Los agentes podrán estar en uno de tres estados: susceptibles, infectados ó recuperados esto se conoce como el modelo SIR. [2].

2. Desarrollo

El desarrollo de la actividad esta basado en el código implementado por E. Schaeffer [2]. Basando el desarrollo, en el código [2] se establecen las instrucciones para la ejecución del programa. Se encuentra todas las instrucciones en el repositorio de Nestor en GitHub.

Código 1: Ejecución de Parámetros

```
import seaborn as sns
from math import floor, log, sqrt
from random import random, uniform
from time import time
from scipy.stats import f_oneway
import pandas as pd
import matplotlib.pyplot as plt
n = 40
pi = 0.09
pr = 0.05
```

Ejecutando con la función, se genera una cantidad de numeros de contagios aleatorios. Esto se define un conjunto de funciones que se utilizan. Los parámetros serán el número de contagios y la probabilidad de posibles contagios. La probabilidad de contagio será en nuestro caso proporcional a la distancia euclideana entre dos agentes de la siguiente manera de la ecuación:

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

Código 2: Instrucciones para generar parámetros epidemiológicos infectados

```
def contagiados():
 for i in range(n):
  a1 = agentes.iloc[i]
  if n < 1:
  return True
  for i in range(2, n):
  if n % i == 0:
  return False
  if d < r:
  if random() < (r - d) / r:
  contagios[j] = True
  return contagios
```

Para saber el programa de como se genera el cuadro que se llama las funciones y se pueda generar, implementados por E. Schaeffer, se muestra a continuación en el código 2. Los agentes tendrán coordenadas y una dirección y una velocidad. Se ejecutará los parámetros a posicionar. [2]

Código 3: Instrucción para generar agentes

```
for k in range(runs):
    agentes =  pd.DataFrame()
    agentes['x'] = [uniform(0, l) for i in range(n)]
    agentes['y'] = [uniform(0, l) for i in range(n)]
     x = a.x + a.dx
```

Se ejecuta los parámetros para una serie de probabilidades de epidemias y contagios, para representar los resultados.

Código 4: Resultados de epidemias y contagios

```
epidemia = []
    for tiempo in range(tmax):
    conteos = agentes.estado.value_counts()
    infectados = conteos.get('I', 0)
    epidemia.append(infectados)
    contagios = [False for i in range(n)]
```

Se ejecuta los parámetros para una serie de probabilidades de Pico más alto y Caso sospechosos, para representar los resultados.

Código 5: Resultados de Pico más alto y Caso sospechosos

```
elif a.estado == 'I':
    if random() < pr:
    agentes.at[i, 'estado'] = 'R'
    x = a.x + a.dx
    y = a.y + a.dy
    x = x if x < l else x - l
    y = y if y < l else y - l
    x = x if x > 0 else x + l
    y = y if y > 0 else y + l
    agentes.at[i, 'x'] = x
    agentes.at[i, 'y'] = y
    maximos['Pico m s alto'].append(max(epidemia))
    maximos['Caso sospechosos'].append(epidemia.index(max(epidemia)) + 1)
```

3. Resultados

En los diagramas se representa como es el comportamiento de los resultados de infectados, pico de infectados, casos sospechosos y disminución de casos. Las variaciones no hay una estadística perfecta para comparar el impacto de la pandemia. Una combinación de diferentes métricas puede ofrecer una visión más completa del impacto del virus. Estas gráficas representan distintas estadísticas, cada una con sus propias fortalezas y debilidades.

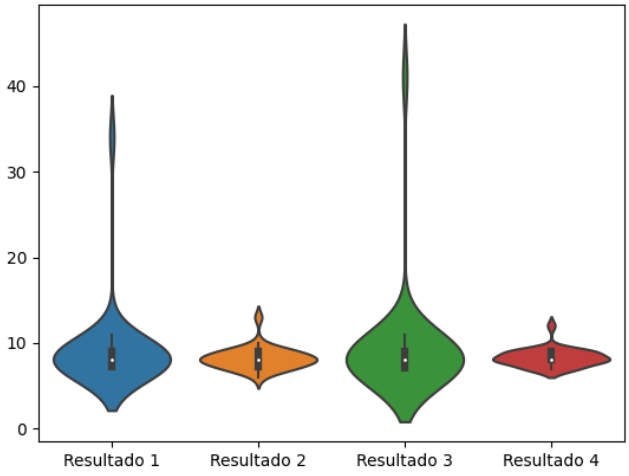


Figura 1: Diagrama de infectados, pico de infectados, casos sospechosos y disminución de casos

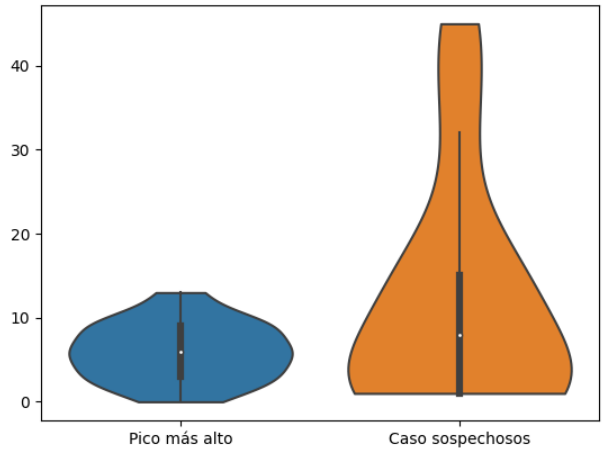


Figura 2: Diagrama de pico más alto infectados y casos sospechosos

4. Conclusiones

Se concluye que se reporta los datos de forma ligeramente distinta y de forma inevitable ya que no han sido diagnosticados. A medida que estos tratan de contener la propagación del virus, independientemente de si se están acercando o ya pasaron el pico de contagios, o si están experimentando un resurgir de contagios.

Referencias

[1] J.FeroxDeitas. Github,p6. URL https://github.com/FeroxDeitas/Simulacion-Nano/blob/main/Tareas/P6.

[2] E. Schaeffer. Sistema multiagente. *Repositorio, GitHub*, 2021. URL https://https://satuelisa.github.io/simulation/p6.html.

[1]

"P7" Búsqueda Local

NESTOR

Marzo 2022

1. Objetivo

El objetivo de la práctica consiste en una variante unidimensional, a partir de un punto seleccionado al azar. Los movimientos locales, por medio de una búsqueda local aleatoria con restricciones en los ejes. Se realizarán réplicas, y el menor de ellos es el resultado de la búsqueda en sí. [1]

2. Desarrollo

Basando el desarrollo en la codificación implementado por E. Schaeffer [1] y se encuentra todas las instrucciones en el repositorio repositorio de Nestor en GitHub.

Para comenzar se hace primero generar la función para graficar con los límites y los pasos. implementados por E. Schaeffer, como se muestra a continuación en el código.

Código 1: Generamos la función

```
import numpy as np
import math as ma
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
def g(x, y):
 return (1 - x/2 + x**5 + y**3) * np.exp(-x**2 - y**2)
 low = -2.7
 high = -low
 step = 0.10
```

Posteriormente generaramos cuantos son los que queremos de los puntos ó muestras con los ciclos ó pasos. Para esta práctica se usará 5 puntos ó muestras con 500 ciclos ó pasos con las posiciones. al tener las celdas se crea generar el cuadro con las funciones que es como inicia el programa como vemos en el código 2.

Código 2: Generamos los puntos ó muestras y los ciclos ó pasos

```
puntos= 5
ciclos= 500
posx = [uniform(low, high) for s in range(puntos)]
posy = [uniform(low, high)for s in range(puntos)]
bestx = posx
besty = posy
```

Se realiza un ciclo for con la funcion movimientos para las posiciones, se muestra a continuación en el código 3. Ya que esto nos muestra que en cada punto va a revisar sus vecinos y va a tomar la mejor posición, por cada punto revisa la posición y revisa sus vecinos y si hay un vecino más grande de mejor posición ese será la nueva posición.

Código 3: Ejecución de códigos para los vecinos y posiciones

```
def vecinos(vecindad):
    pos=[]
    for a in vecindad:
        pos.append((g(a[0],a[1]),a[0],a[1]))
    return(pos)

def movimientos(x, y, bx, by):
  pasos=[]
  listax,listay=[],[]
  listamxy=[]
```

```
    for s in range(puntos):
    mejorx,mejory=bx[0],by[0]
    cx, cy= x[s], y[s]
    Dx=uniform(0, step/3)
    Dy=uniform(0, step/3)
    xi, xd= (cx-Dx), (cx+Dx)
    yi, yd= (cy-Dy), (cy+Dy)
    xi = low if xi < low else xi
    xd = high if xd > high else xd
    yi = low if yi < low else yi
    yd = high if yd > high else yd
    vecindad=[(xi,yd),(cx,yd),(xd,yd),(xi,cy),(xd,cy),(xi,yi),(cx,yi),(xd,yi)]
    datos=vecinos(vecindad)
    pos,cx,cy= (max(datos))
    if pos > g(mejorx,mejory):
    mejorx, mejory=cx,cy
    else:
    mejorx, mejory= bx[0], by[0]
```

3. Resultados

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, Δx y Δy cuyas combinaciones se hacen las posiciones.

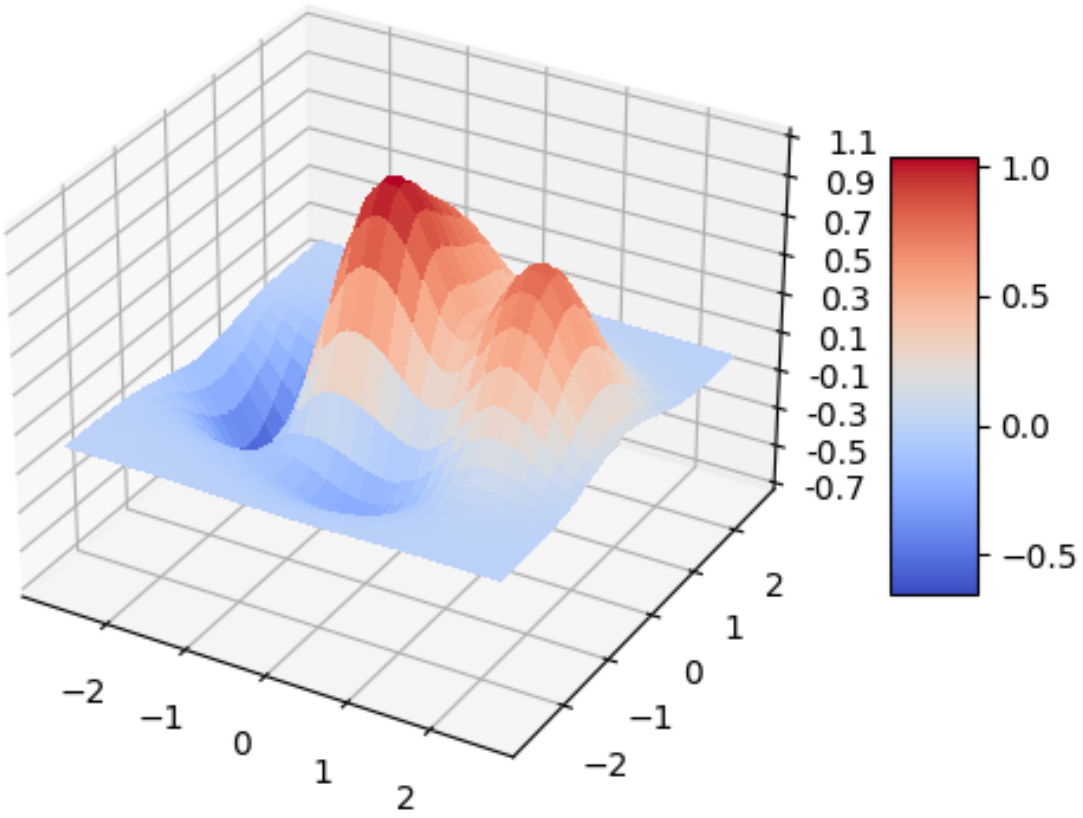


Figura 1: Diagrama en tres dimensiones.

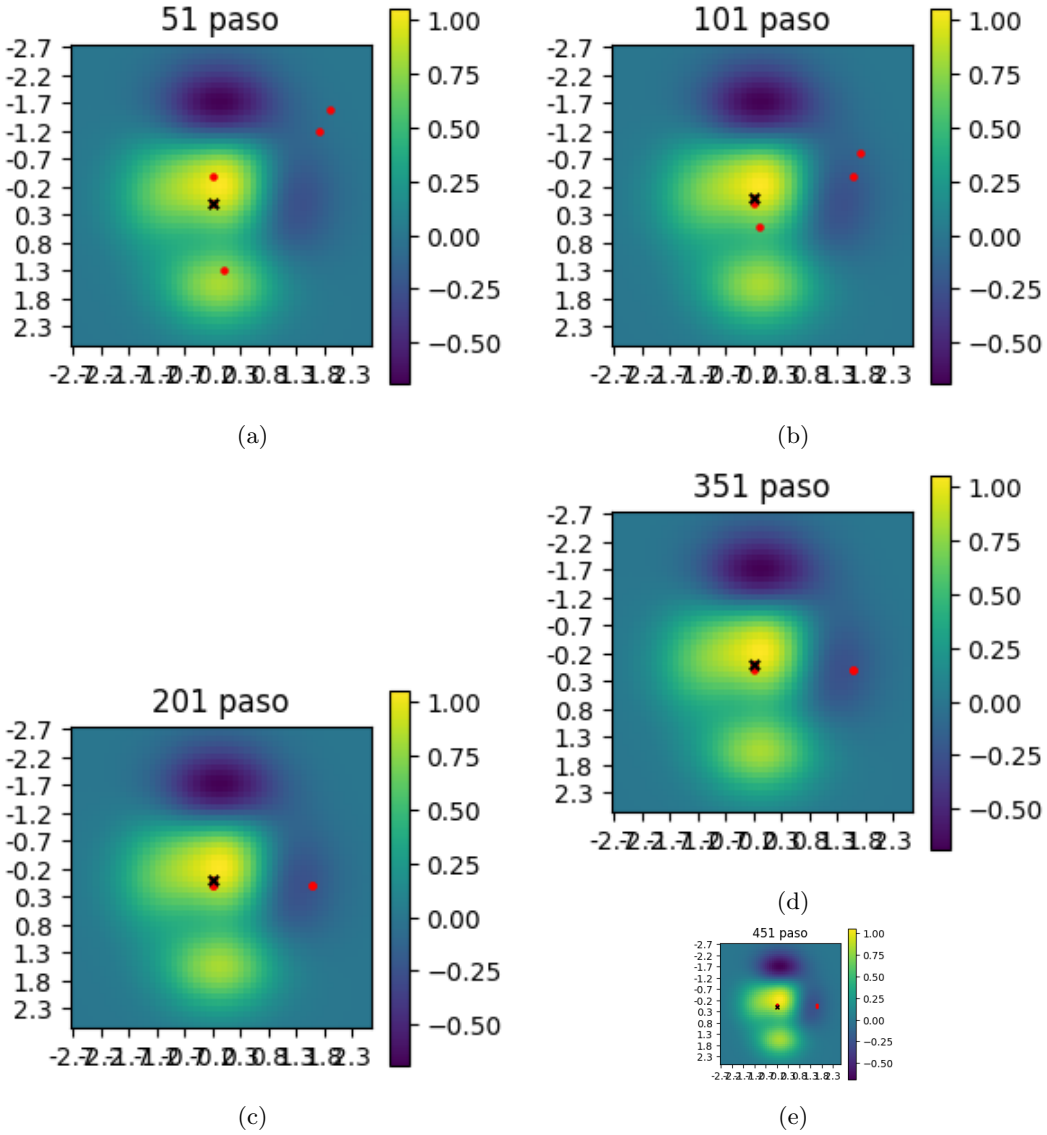


Figura 2: Intervalos de posiciones

4. Conclusiones

Se concluye que los valores máximos y mínimos de una función con los movimientos aleatorios esto nos da la posición de los puntos que se evalúa en los vecinos para comparar si es mejor para ser la nueva posición.

5. Reto 1 Cambio de regla del movimiento

En este reto consiste en cambiar la regla del movimiento de una solución en X (un vector de dimensión arbitraria) a la siguiente a la de recocido simulado: para optimizar una función [1]. 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. A continuación se muestra en las siguientes instrucciones:

Código 4: Temperatura

```
temperatura=100
##temperatura = [100 for i in range(puntos)]
for img in range(ciclos):
    ps,posx,posy,best,temperatura = movimientos(posx, posy, bestx,besty, temperatura)
```

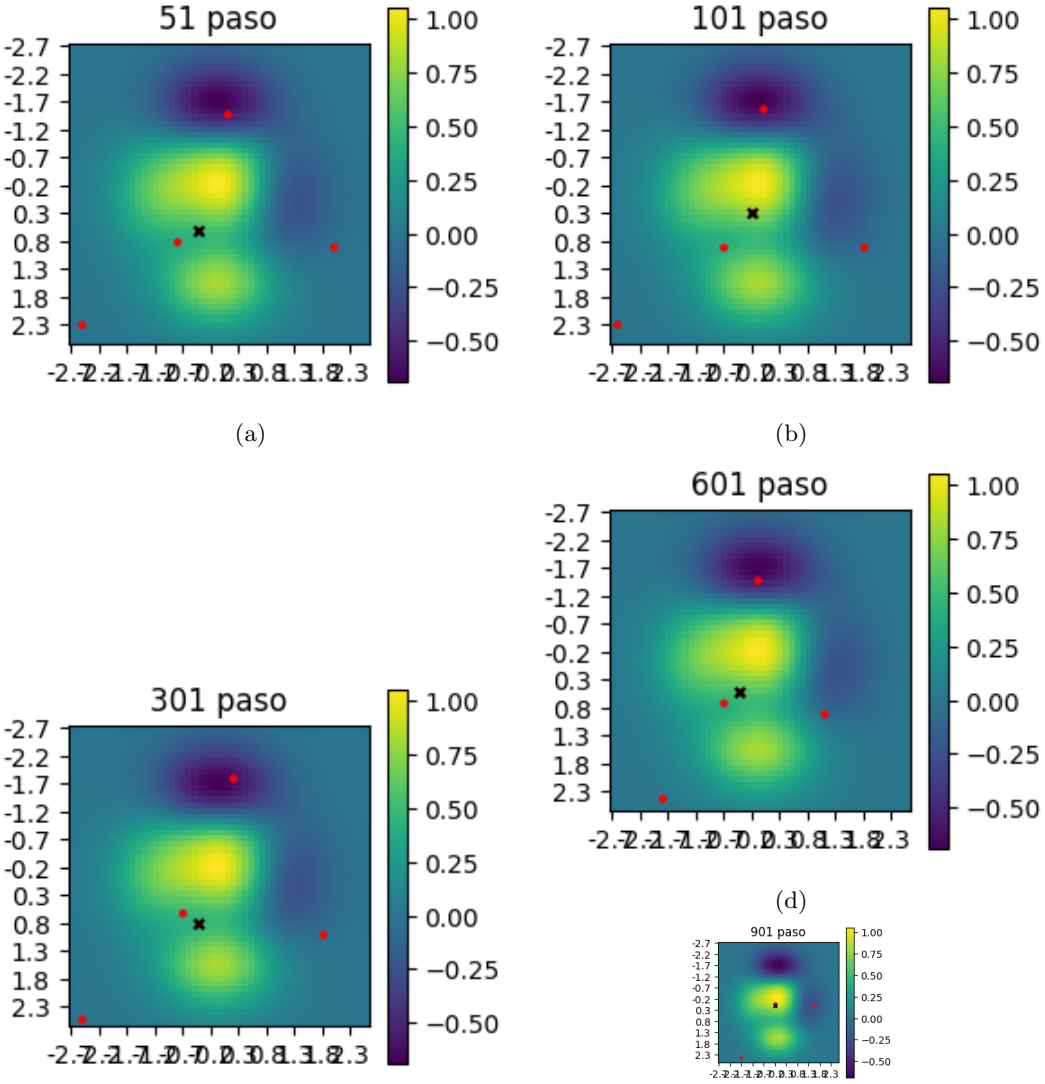
Para este reto la vecindad se vuelve a tomar las mismas decisiones, en cada punto inicial que se revisa su vecindad, cada vecindad se selecciona al azar y se remueve en la lista para que el siguiente ciclo no lo toque.

Código 5: Vecindad

```
vecindad=[(xi,yd),(cx,yd),(xd,yd),(xi,cy),(xd,cy),(xi,yi),(cx,yi),(xd,yi)]
  for k in range(len(vecindad)):
    xnew,ynew = choice(vecindad)
    vecindad.remove((xnew,ynew))
    delta= g(xnew,ynew) - g(cx,cy)
    probabilidad= exp(delta/temp)
    if delta > 0:
    cx, cy=xnew, ynew
    break
    else:
    if random() < (probabilidad):
    cx, cy=xnew, ynew
    temp=(temp*(0.995))
    break
    else:
        cx, cy = cx, cy
```

6. Reto 1 Resultados

Esto se visualiza el comportamiento de la variación respecto a la temperatura. Se encuentra todos los resultados y gifs en el repositorio repositorio de Nestor en GitHub.



(a) (b)

(c) (d)

(e)

Figura 3: Intervalos de Temperatura

7. Reto 2 Comparación de diferencia estadísticamente

En este reto consiste en comparar sí o no hay diferencia estadísticamente significativa entre el método de la tarea base y el método del primer reto en términos de la precisión del resultado obtenido (es decir, la diferencia entre el resultado reportado y el óptimo global) en función del número de iteraciones y el número de réplicas [1]. A continuación se muestra en las siguientes instrucciones:

Código 6: Comparación de iteraciones

```
iteraciones= (100,500,1000)
CB_est2=[]
for f in iteraciones:
    estimados2=[]
    for r in range(replicas):
        posx = [uniform(low, high) for s in range(puntos)]
        posy = [uniform(low, high)for s in range(puntos)]
        bestx = posx
        besty = posy
        temperatura=1000
        for img in range(f):
            ps,posx,posy,best,temperatura = mov_reto1(posx, posy, bestx,besty, temperatura)
            if img==(f-1):
                mejor=max(best)
                estimados2.append(mejor[0])
    CB_est2.append(estimados2)
plt.boxplot([CB_est2[0], CB_est2[1], CB_est2[2]])
plt.xlabel('iteraciones')
plt.ylabel('Posiciones finales maximizadas')
plt.xticks([1,2,3], ['100','500','1000'])
```

8. Reto 2 Resultados

Esto se visualiza la diferencia entre los resultado reportado y la función del número de iteraciones y el número de réplicas estadísticamente. Se encuentra todos los resultados y gifs en el repositorio repositorio de Nestor en GitHub.

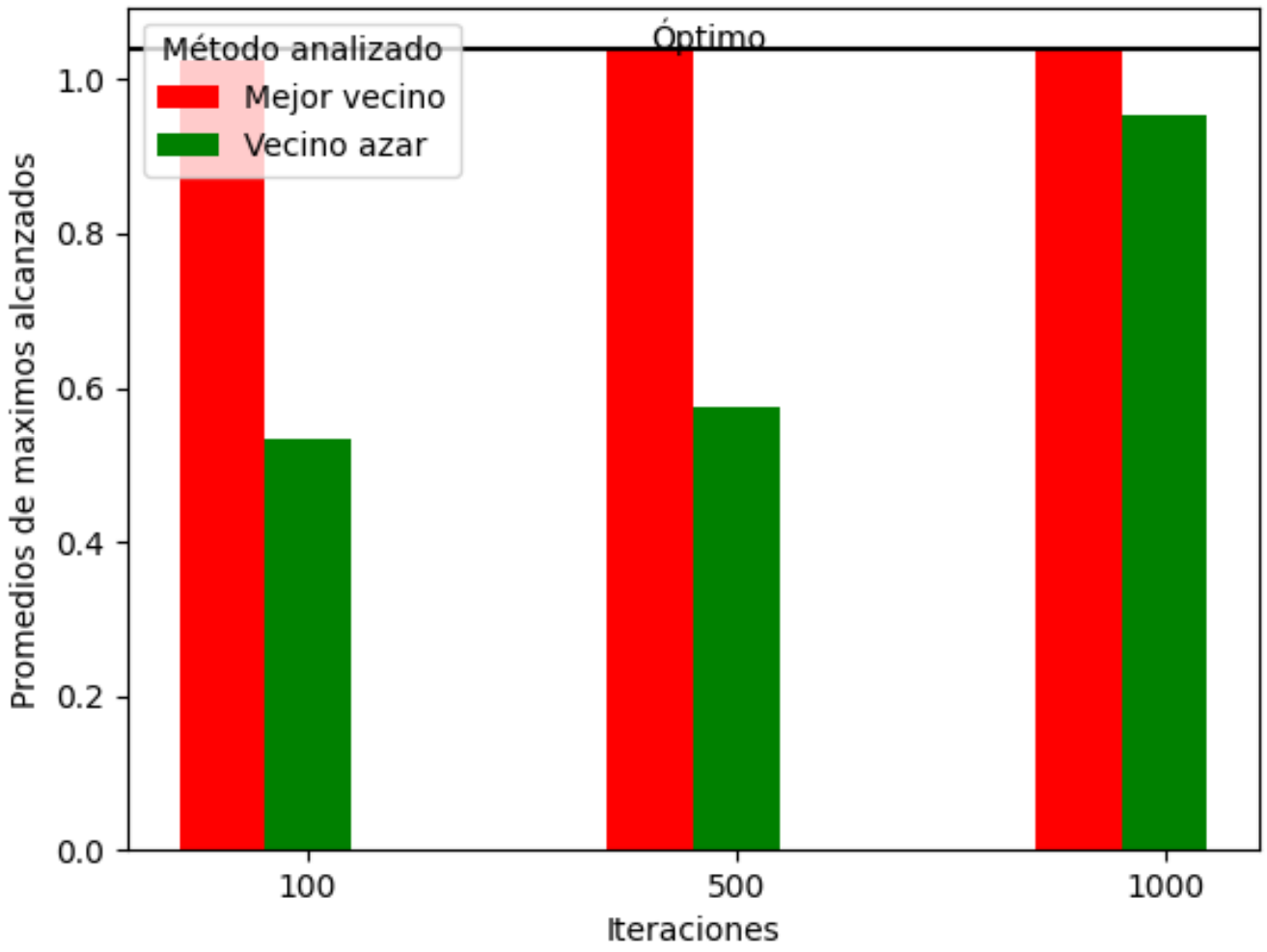


Figura 4: Diagrama de comparación de iteraciones

Referencias

[1] E. Schaeffer. Búsqueda local. *Repositorio, GitHub*, 2022. URL https://github.com/satuelisa/Simulation/blob/master/LocalSearch/minimize1D.py.

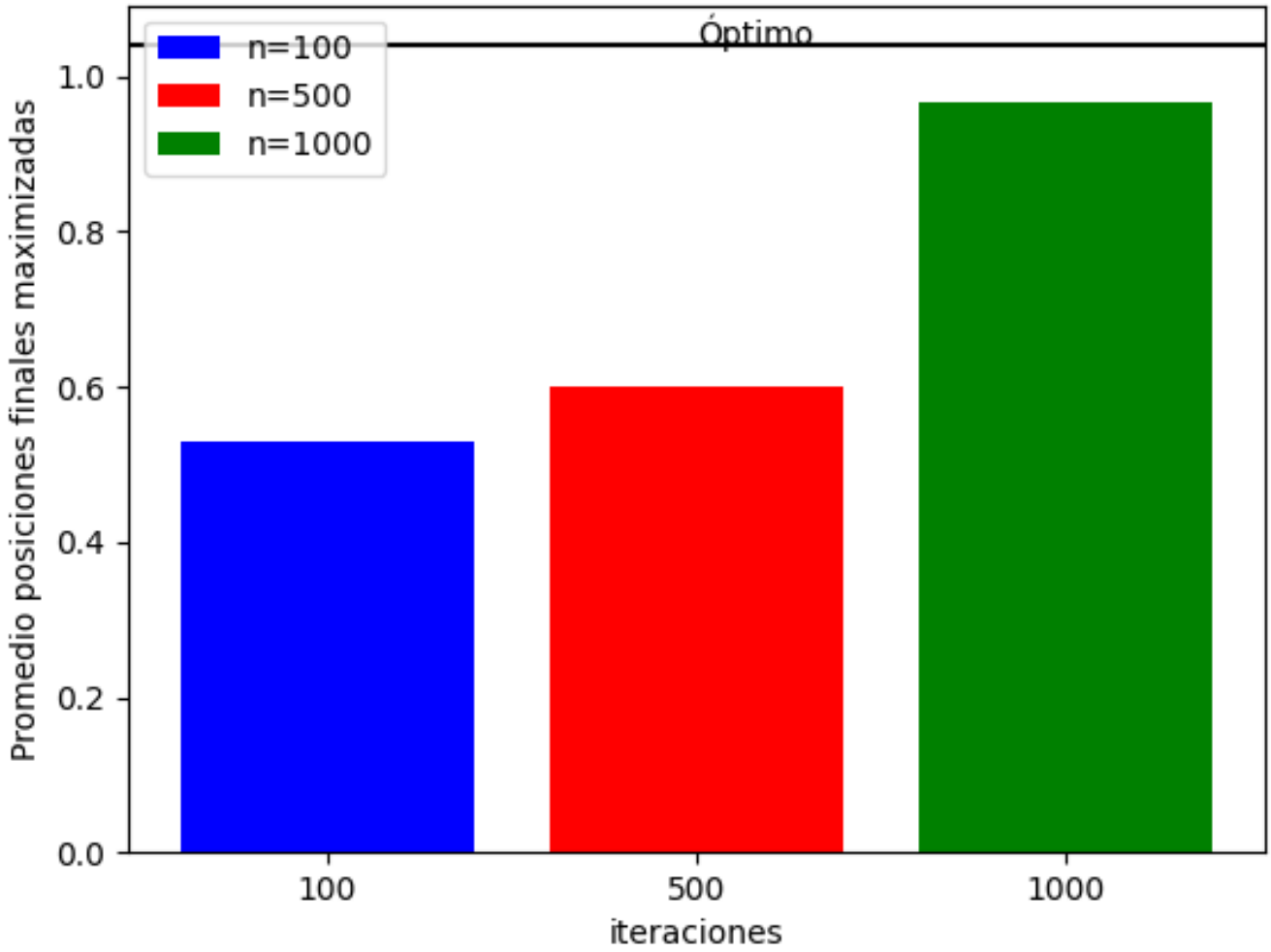


Figura 5: Diagrama promedio de posiciones de vecinos al azar

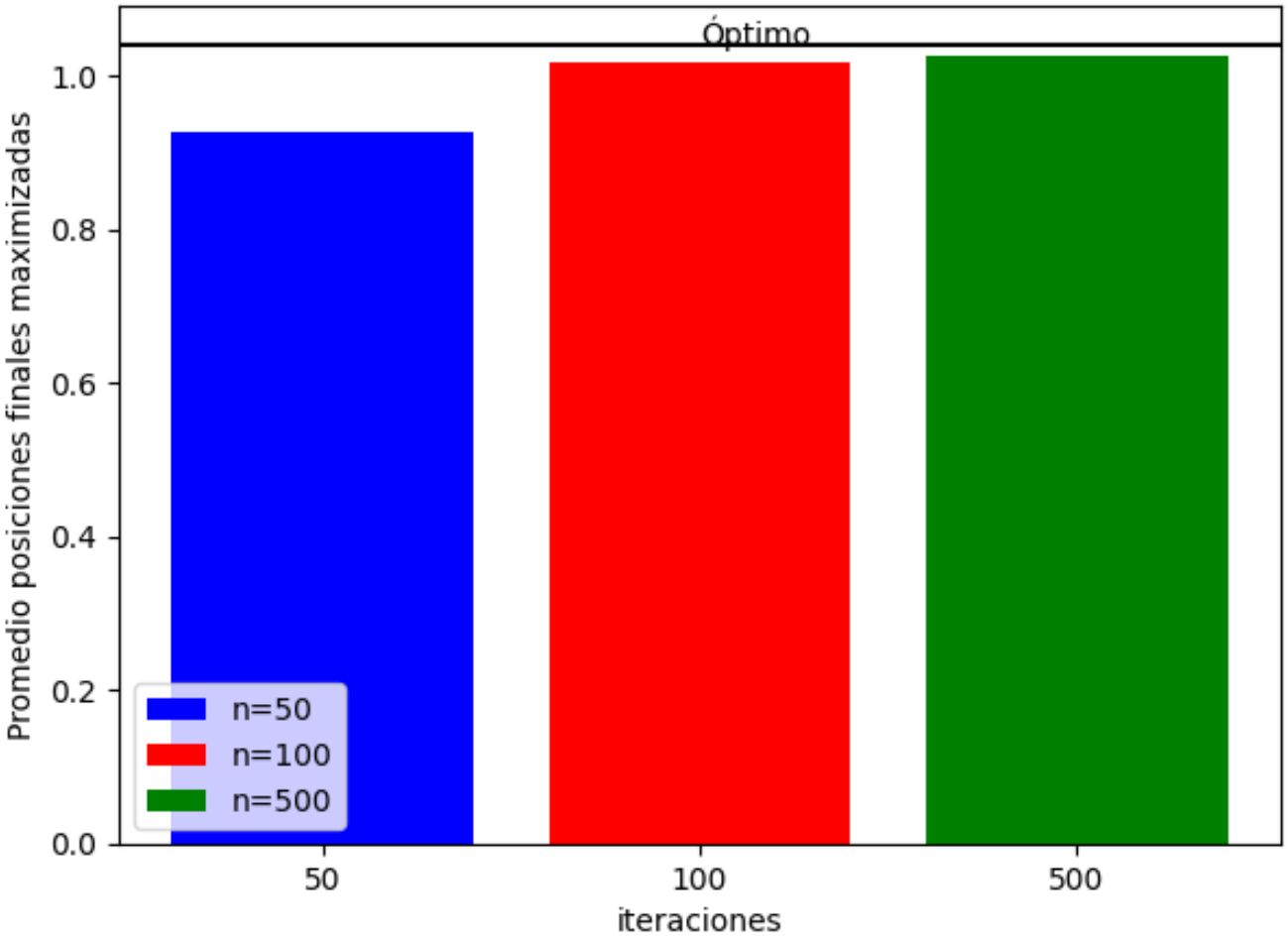


Figura 6: Diagrama promedio de posiciones de vecinos máximos

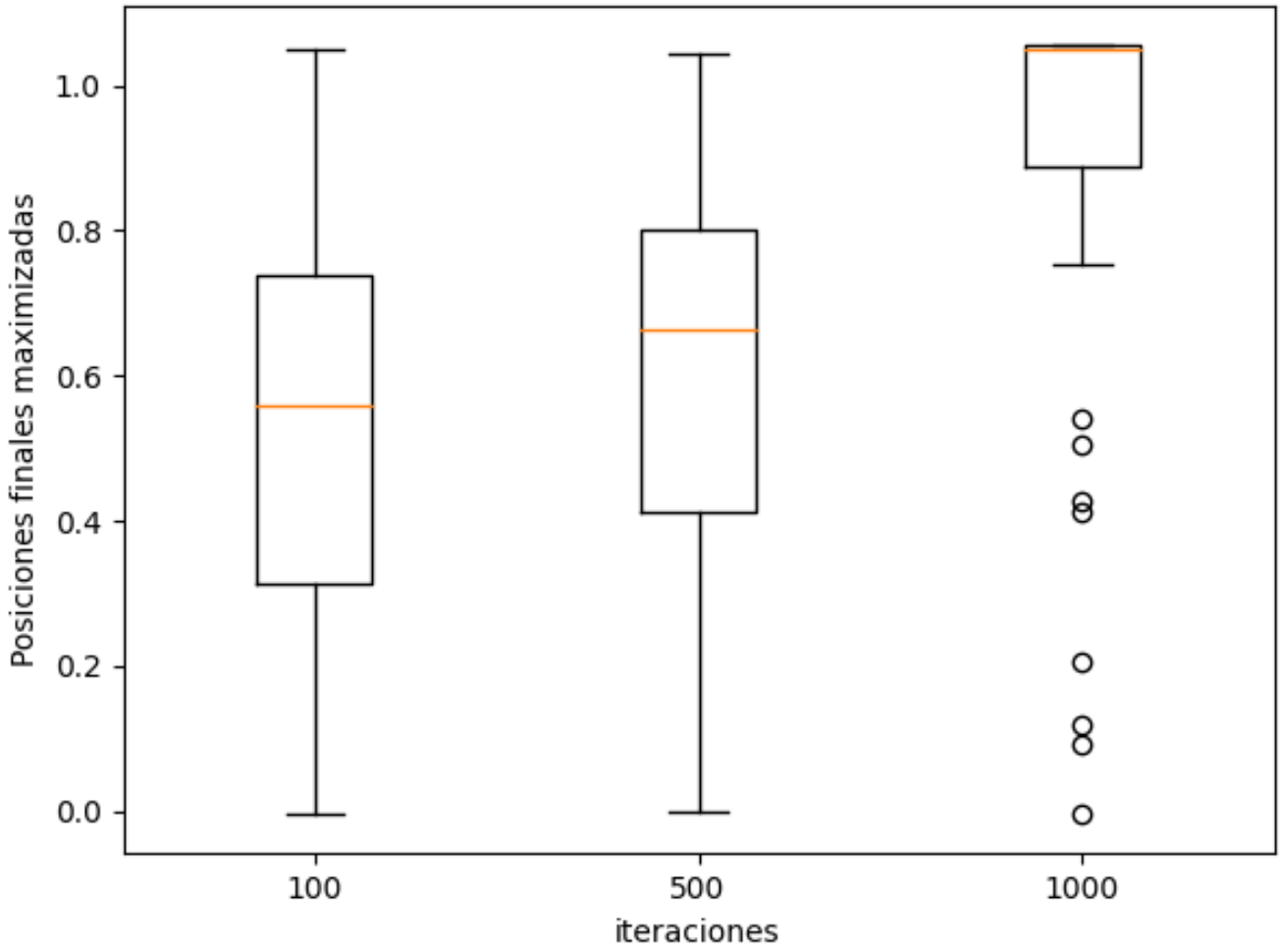


Figura 7: Diagrama de vecinos al azar

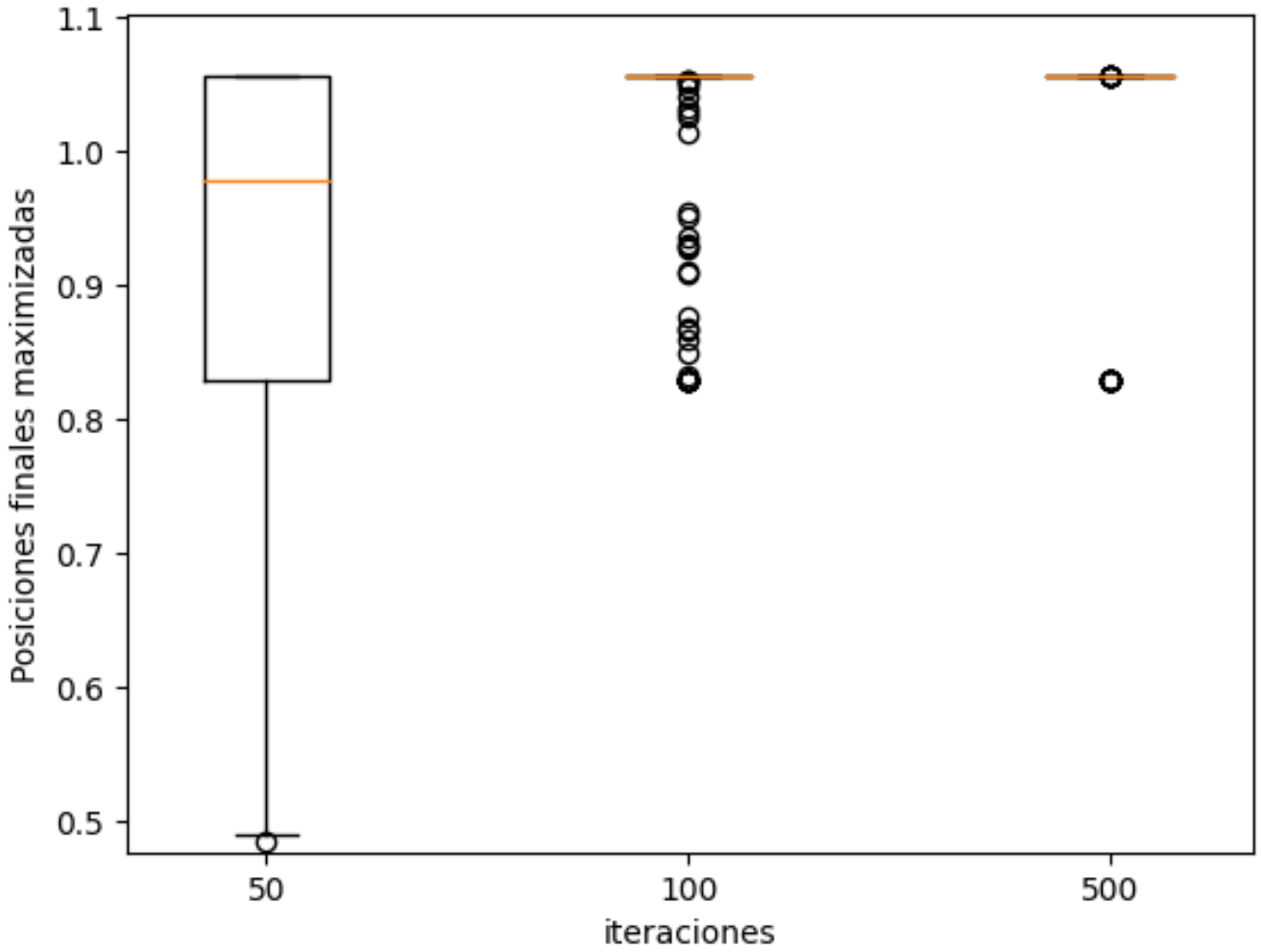


Figura 8: Diagrama de vecinos máximos

"P8" Modelo De Urnas

NESTOR

Abril 2022

1. Objetivo

El objetivo de la práctica consiste sobre fenómenos de coalescencia y fragmentación, donde las partículas se unen para formar cúmulos y estos cúmulos se pueden volver a descomponer en fragmentos menores. Supongamos que cúmulos con c o más partículas son suficientemente grandes para filtrar. Se estudiará 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]

2. Desarrollo

Basando el desarrollo en la codificación implementado por E. Schaeffer [1] y todas las instrucciones se encuentra en el repositorio repositorio de N. Rodríguez en GitHub.

Para comenzar se hace primero generar la función para graficar los cúmulos n/k que van a ser los cúmulos, implementados por E. Schaeffer, como se muestra a continuación en el código.

Código 1: Generamos la función

```
from random import random
from numpy.random import shuffle
import matplotlib.pyplot as plt
from math import exp, floor, log
import numpy as np
from random import randint
def rotura(x, c, d):
    return 1 / (1 + exp((c - x) / d))
def union(x, c):
    return exp(-x / c)
def romperse(tam, cuantos):
    if tam == 1: # no se puede romper
        return [tam] * cuantos
    res = []
    for cumulo in range(cuantos):
```

Posteriormente generaramos los ciclos. Para esta práctica se usará 6 variaciones de las 5 que se propusieron en clase como mínimo, esto para variar n/k que se puedan ver las réplicas, a continuación se muestra el código.

Código 2: Generamos los ciclos para variar n/k

```
k1=(500,8000,20000)
repeticiones=50
for k in k1:
    n1=((k*10),(k*500))
    contador=0
    for n in n1:
        print("############ k,n:",k,n,"##################")
        promedio=[]
```

Se realiza el valor c de los cúmulos, a continuación se muestra la siguiente instrucción:

Código 3: Generamos valor c

```
c = np.median(cumulos)
d = np.std(cumulos)
duracion = 50
```

```
digitos = floor(log(duracion, 10)) + 1
part_porc=[]
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)
```

Se realiza un ciclo for porcentaje para los cúmulos que se lograría filtrar por iteración para las partículas como se muestra la siguiente instrucción:

Código 4: Ciclo for para porcentaje de las partículas

```
grandes=[]
for s in cumulos:
    if s > c:
        grandes.append(s)
part_porc.append((len(grandes)/len(cumulos))*100)
```

Para esto se revisa todos los cumulos una vez que ya se filtraron y si el cumulo es mayor a mi valor crítico, entonces lo voy a guardar en una lista y en este caso son los que se quedaron en el filtro. Para este caso se guarda en un caja bigote como se muestra en el código 5

Código 5: Promedio

```
promedio.append((sum(part_porc)/len(part_porc)))
        CB.append(promedio)
        ticks.append((str(k),str(n)))
print(len(CB))
print(ticks)
plt.boxplot(CB, [1,2,3,4,5,6])
plt.xlabel('Tasa k/n')
plt.ylabel('Promedios retenidos en filtro (%)')
plt.xticks([1,2,3,4,5,6], ticks,rotation=10)
plt.show()
```

3. Resultados

En la caja bigote nos muestra las combinaciones de las réplicas que se obtiene una cantidad de promedios de porcentaje en el eje x se grafica de la tasa n/k y con en el eje y se grafica los promedios retenidos en el filtro para poder tener el porcentaje, se sacan varios porcentajes ya que se filtraron para poder tener el promedio.

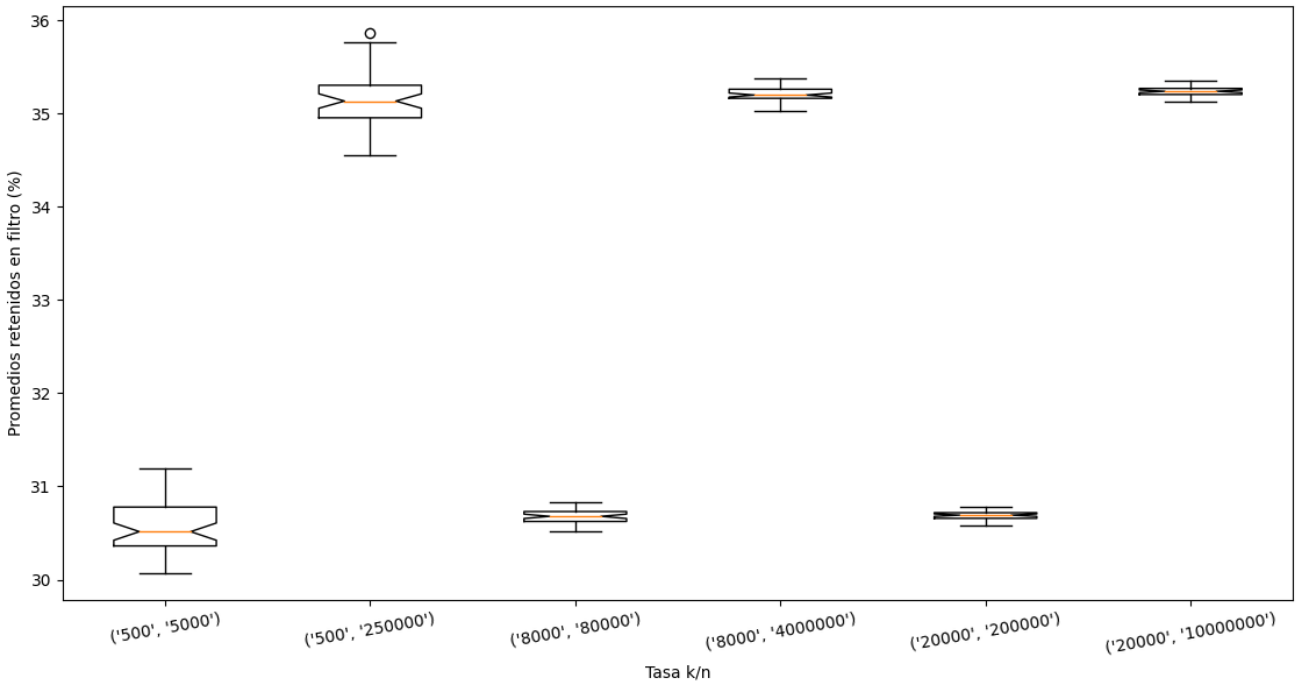


Figura 1: Diagrama de cúmulos n/k.

4. Conclusiones

Se concluye que los valores, si hacemos la prueba, esto nos da la variación mejor si aumentamos mas la tasa n/k esto para generar mejor el resultado, esto aumenta el promedio de retenidos en el filtro y nos da mejor el comportamiento pero para la tasa de n

5. Reto 1 Determinar el filtrado dependiendo del valor c

En este reto consiste en determinar cómo el momento idóneo de filtrado depende del valor de c, esto para saber si todo cambia y como si c ya no se asigna como la mediana inicial sino a un valor menor ó mayor A continuación se muestra en las siguientes instrucciones:

Código 6: Variación de c en tres diferentes métodos

```
k = 5000
n = 1000000
repeticiones=40
contador=0
metodos=3
for met in range(metodos):
    promedio=[]
    for rep in range(repeticiones):
```

Para este caso se filtra dependiendo de c,

Código 7: Valor filtrado dependiendo de c

```
if contador == 0:
    c = np.median(cumulos)
elif contador==1:
    c = min(cumulos)# factor arbitrario para suavizar la curva
elif contador==2:
    c = max(cumulos)
```

6. Reto 1 Resultados

Para la caja bigote vemos que en el eje x, se varía el valor crítico de los cúmulos aquí es con la mediana, el promedio anda de 35,0 a 35,2 porciento. Cuando toma el valor mínimo de la lista de cúmulos, el valor mínimo será la rejilla, entonces si un cúmulo es mayor al mínimo se quedará en el filtro, en este caso se tiene un mayor porcentaje que se quede retenidos en el filtro, ya que es el mejor comportamiento. Si se toma el valor máximo son pocos de rejillas aunque no es mejor que la mediana pero se retiene en el filtro.

En las réplicas se muestra que el mínimo, ya que tiene mayor porcentaje, en la réplica máximo si le compite al valor de la mediana pero ya cuando a mayor número de réplicas ya es comparable al de la mediana, en este caso si sea más réplicas tiene un mejor comportamiento y podría ganarle a la mediana a continuación se muestra los siguientes diagramas para el bigote y para las réplicas:

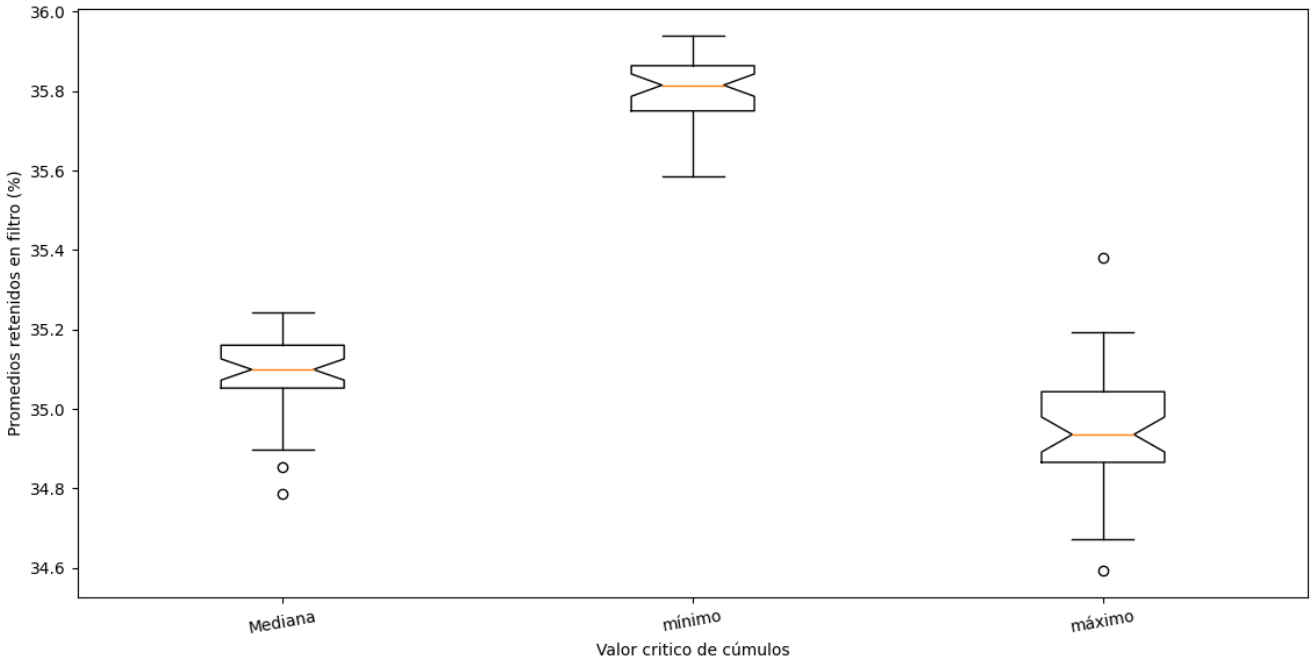


Figura 2: Diagrama bigote dependiendo del valor c.

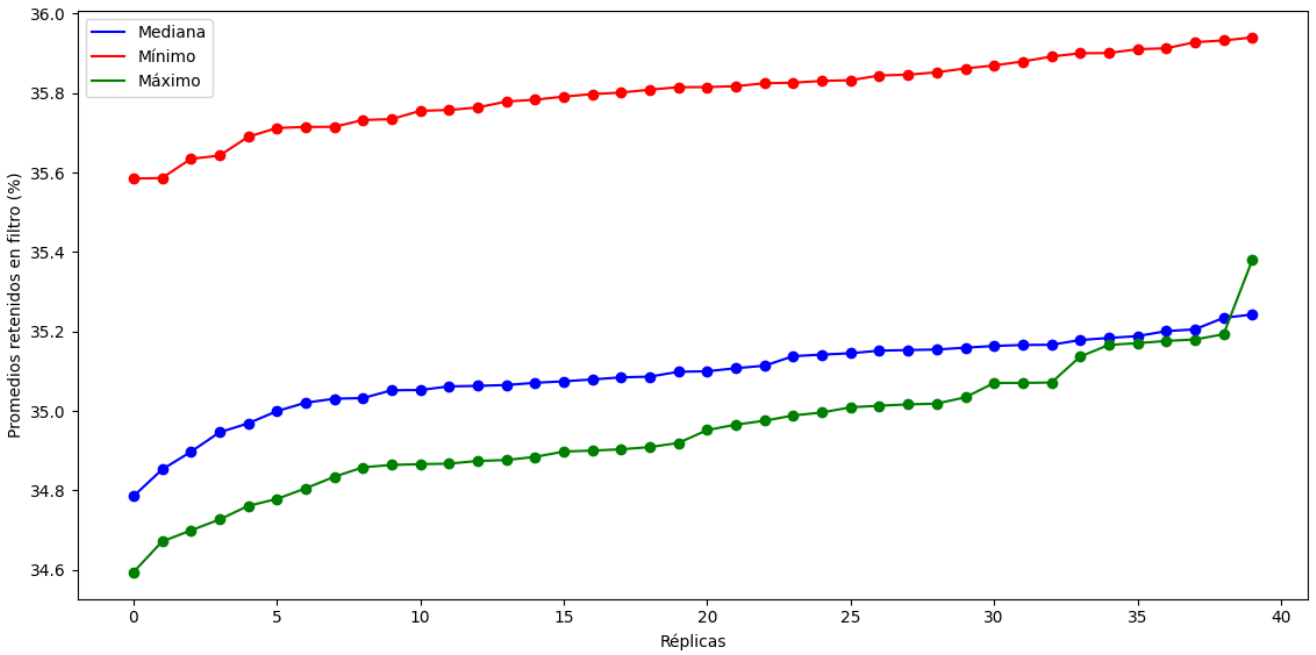


Figura 3: Diagrama de réplicas dependiendo del valor c.

7. Reto 2 Se estudia el efecto del parámetro suavizante d

En este reto consiste 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. A continuación se muestra en las siguientes instrucciones:

Código 8: Generación de repeticiones

```
k = 10000
n = 1000000
repeticiones=40
contador=0
metodos=4
for D in range(metodos):
    promedio=[]
    for rep in range(repeticiones):
```

Para este caso la mayor cantidad posible de partículas en el proceso suavizante en d.

Código 9: Promedio en el suavizante d

```
if contador == 0:
    d = np.std(cumulos) / 2
elif contador == 1:
    d = np.mean(cumulos)
elif contador==2:
    d = min(cumulos)# factor arbitrario para suavizar la curva
elif contador==3:
    d = max(cumulos)
for paso in range(duracion):
    assert sum(cumulos) == n
    assert all([c > 0 for c in cumulos])
    while len(j) > 1: # agregamos los pares formados
        cumulos.append(j.pop(0) + j.pop(0))
    if len(j) > 0: # impar
        cumulos.append(j.pop(0)) # el ultimo no alcanz  pareja
    assert len(j) == 0
    assert sum(cumulos) == n
    assert all([c != 0 for c in cumulos])
    grandes=[]
    for s in cumulos:
        if s > c:
            grandes.append(s)
    cuantos.append(len(grandes))
promedio.append((sum(cuantos)/len(cuantos)))
```

8. Reto 2 Resultados

Se muestra como se varía los promedios retenidos en el filtro con los parámetros del suavizante en d de los cúmulos, esto nos ayuda para tener la mayor cantidad de partículas que es el mejor comportamiento.

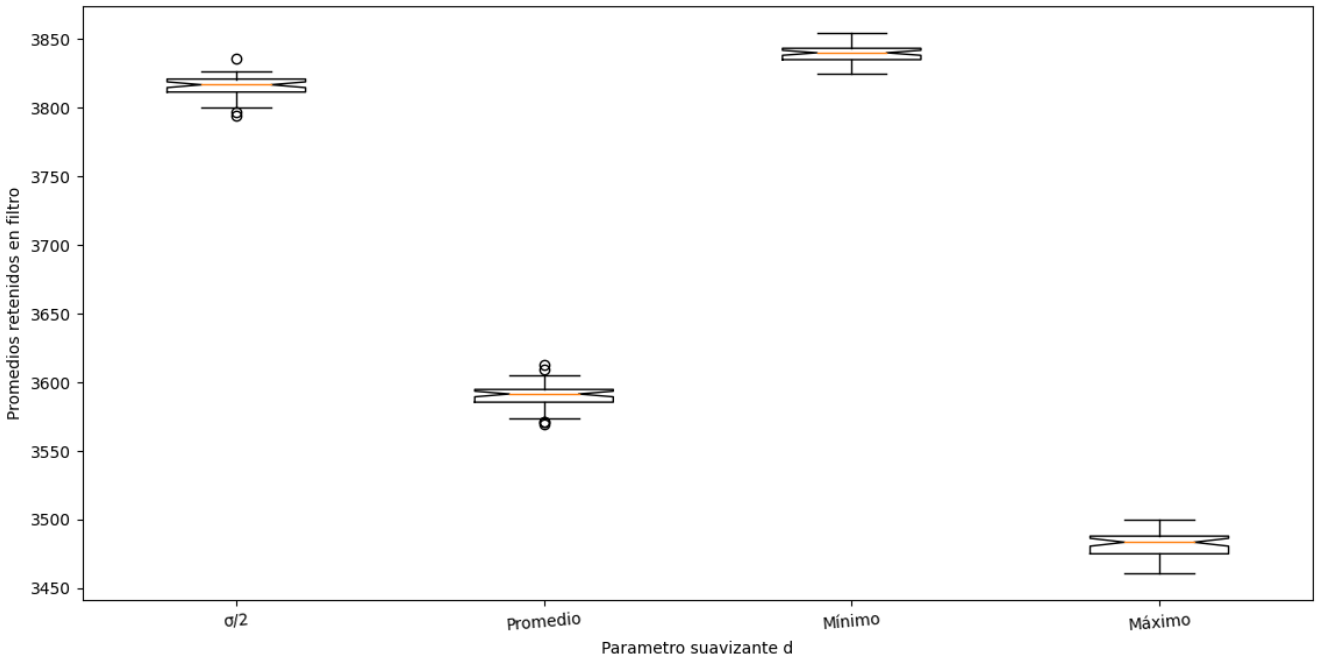


Figura 4: Diagrama de suavizante en d.

Referencias

[1] E. Schaeffer. Búsqueda local. *Repositorio, GitHub*, 2022. URL https://github.com/satuelisa/Simulation/blob/master/UrnModel/onlyAggr.py.

"P9" Interacciones entre partículas

NESTOR

Abril 2022

1. Objetivo

El objetivo de la práctica consiste en un modelo simplificado para los fenómenos de atracción y repulsión de física .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 euclideana entre las partículas [1].

2. Desarrollo

Basandome en el desarrollo en la codificación implementado por E. Schaeffer y todas las instrucciones se encuentran en el repositorio de N. Rodríguez en GitHub.

Para comenzar se hace primero generar la función para generar partículas de atracción y repulsión con esto para poder visualizar la masa

Código 1: Generamos las partículas

```
import numpy as np
import pandas as pd
from random import uniform
import matplotlib.pyplot as plt
paso = 256 // 10
niveles = [i/256 for i in range(0, 256, paso)]
eps = 0.001
```

Generamos las posiciones de las partículas en "x y y", se les asigna una carga y una masa con números random uniformemente con la finalidad de asignar masas variables.

Código 2: Partículas en "x y y"

```
if __name__ == "__main__":
    inicial=[]
    popen('rm -f p9p_t*.png') # borramos anteriores en el caso que lo hayamos corrido
    n = 25
    x = np.random.normal(size = n)
    y = np.random.normal(size = n)
    masa = [uniform(0,50)for i in range(n)]
    c = np.random.normal(size = n)
    #masa=[s * (-1) for s in masa]
```

Se realizó ya las masas variables, ahora se codifica que la masa cause un efecto ya que está asignada a una partícula, ya que con esto se hace varios pasos para que vayan avanzando las partículas, a continuación se muestra la codificación:

Código 3: masa

```
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.masa
    fx, fy = 0, 0
    for k in range(n):
        pk = p.iloc[k]
        ck = pk.c
        mk = pk.masa
        dire_c = (-1)**(1 + (ci * ck < 0))
        dire_m = (-1)**(1 + (mi * mk > 0))
        factor_c = dire_c * fabs(ci - ck) / (sqrt(dx**2 + dy**2) + eps)
        factor_m = dire_m * fabs(mi - mk) / (sqrt(dx**2 + dy**2) + eps)
        fx -= dx * factor_m * factor_c
        fy -= dy * factor_m * factor_c
    return (fx, fy)
```

Generamos la correlación de rango de Spearman, esto nos ayudará a correlacionar las tres variables en una medida de asociación lineal que se utilizará para los rangos.

Código 4: Spearman

```
### Correlacion de rango de Spearman
    from scipy.stats import spearmanr
    stat, p = spearmanr(x, z)
    print("correlacion carga con velocidad")
    print('stat=%.3f, p=%.3f' % (stat, p))
    if p > 0.05:
        print('Probablemente dependiente')
    else:
        print('Probablemente independiente')
    print("correlacion masa con velocidad")
    stat2, p2 = spearmanr(y, z)
    print('stat=%.3f, p=%.3f' % (stat2, p2))
    if p2 > 0.05:
        print('Probablemente dependiente')
    else:
        print('Probablemente independiente')
```

3. Resultados

Cuadro 1: Correlacion de rango de Spearman para las partículas

Mediciónes	Estadística
Relacion entre la velocidad con carga y velocidad con masa	0,37567824 porciento
Correlacion carga con velocidad	stat=0,380, p=0,061 (Probablemente dependiente)
Correlacion masa con velocidad	stat=−0,285, p=0,167 (Probablemente dependiente)

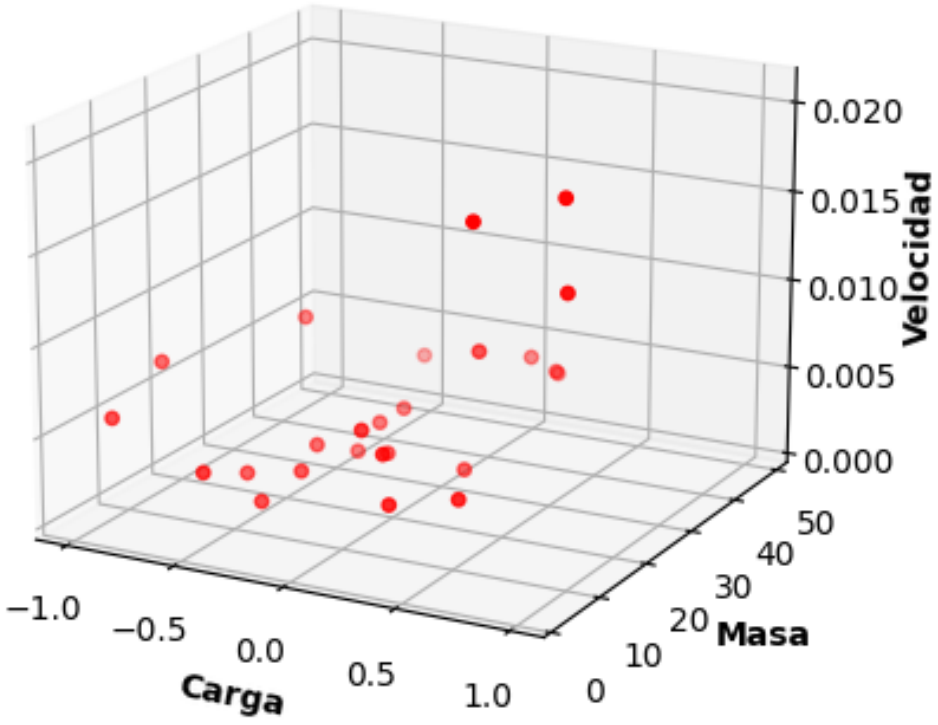


Figura 1: Partículas en la carga.

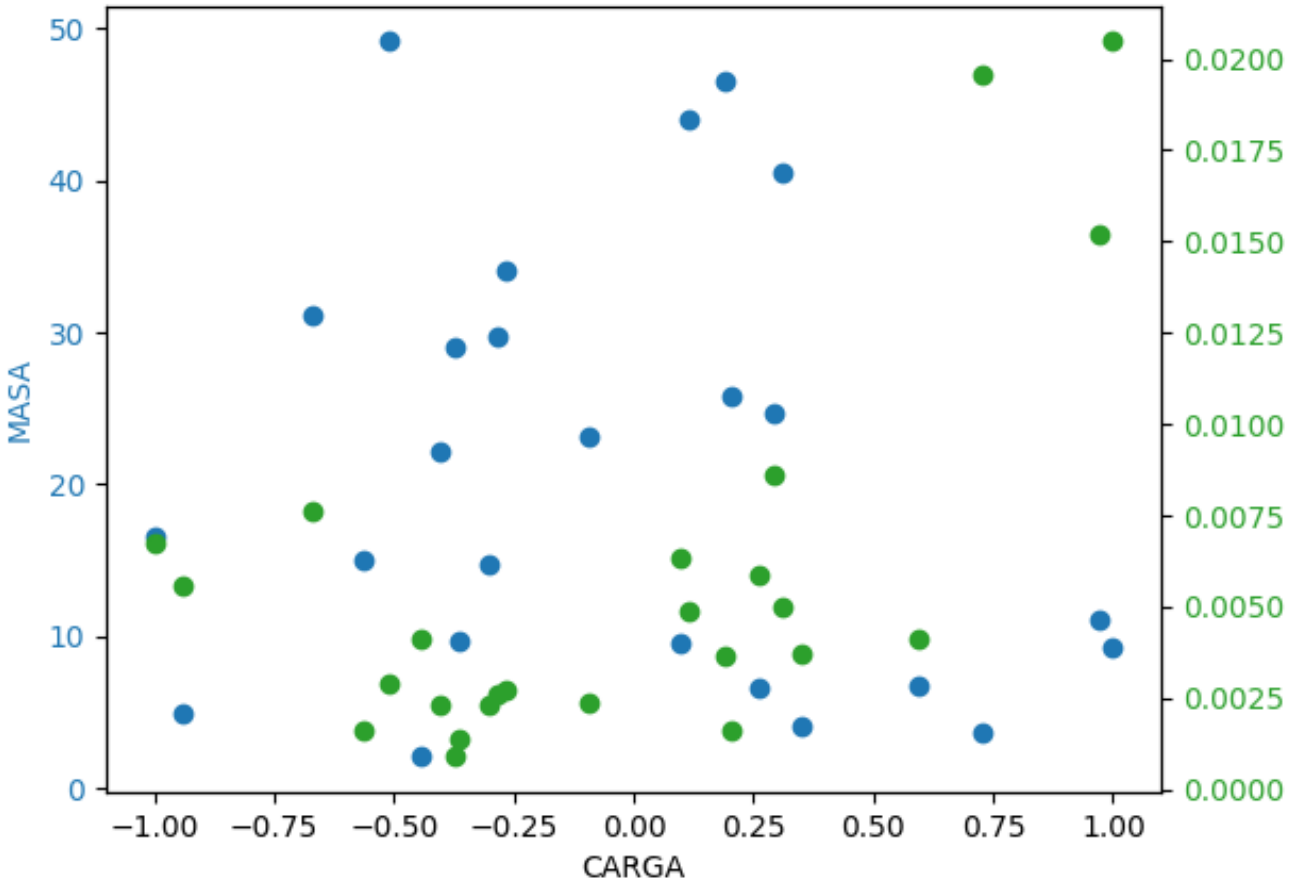


Figura 2: Comportamiento de carga.

4. Conclusiones

Se concluye que las partículas se correlacionan con las tres variables con respecto a la carga y la menor velocidad ya que se encuentra la diferencia en velocidades.

Referencias

[1] E. Schaeffer. Búsqueda local. *Repositorio, GitHub*, 2022. URL https://github.com/satuelisa/Simulation/blob/master/Particles/creation.py.

"P10" Algoritmo genético

NESTOR

Mayo 2022

1. Objetivo

El objetivo de la práctica consiste en 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 [2].

2. Desarrollo

Basandome en el desarrollo en la codificación implementado por E. Schaeffer y todas las instrucciones se encuentran en el repositorio de N. Rodríguez en GitHub.

Para comenzar se hace primero generar la función para generar partículas de atracción y repulsión con esto para poder visualizar la masa

Código 1: Algoritmo de la combinación óptima.

```
import math
from scipy.stats import expon
from time import time
def knapsack(peso_permitido, pesos, valores):
    assert len(pesos) == len(valores)
    peso_total = sum(pesos)
    valor_total = sum(valores)
    if peso_total < peso_permitido:
```

Generamos las posiciones de los pesos y valores de las instancias con esto de 1 a 3.

Código 2: Pesos y Valores

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

def valores(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
```

Para las instancias se crean $n = 40$ con los valores y pesos, a continuación se muestra la codificación:

Código 3: Parámetro de n

```
n = 40
VP=[]
Tr=[]
for regla in range(3):
    print("############### regla:",regla,"##################")
    if regla == 0:
        pesos = pesos1(n, 23, 100)
        valores = valores1(pesos, 5, 700)
    if regla == 1:
        valores = valores2(n, 5, 700)
        pesos = pesos2(valores, 23, 80)
```

Se hace la combinacion de valores y pesos para poder hacer las variaciones de mutaciones, reproducciones y la población. Con esto se genera las replicas e iteracciones.

Código 4: Generacion de funciones

```
for pm, init, rep in instancias:
        antesi=time()
        print("##############",pm, init, rep,"##############")
        replicas=3
        best=[]
        porc_dif=[]
        for K in range(replicas):
            p = poblacion_inicial(n, init)
            tam = p.shape[0]
            assert tam == init
            tmax = 100
            mejor = None
            mejores = []
            for t in range(tmax):
                for i in range(tam): # mutarse con probabilidad pm
                    if random() < pm:
                        p = np.vstack([p, mutacion(p[i], n)])
                for i in range(rep):  # reproducciones
                    padres = sample(range(tam), 2)
                    hijos = reproduccion(p[padres[0]], p[padres[1]], n)
                    p = np.vstack([p, hijos[0], hijos[1]])
                tam = p.shape[0]
                d = []
                for i in range(tam):
                    d.append({'idx': i, 'obj': objetivo(p[i], valores),
                              'fact': factible(p[i], pesos, capacidad)})
                d = pd.DataFrame(d).sort_values(by = ['fact', 'obj'], ascending = False)
                mantener = np.array(d.idx[:init])
                p = p[mantener, :]
                tam = p.shape[0]
                assert tam == init
                factibles = d.loc[d.fact == True,]
                mejor = max(factibles.obj)
                mejores.append(mejor)
            best.append(mejor)
            porc_dif.append(((optimo - mejor) / optimo)*100)
        CB.append(porc_dif)
        Ti.append(time()-antesi)
VP.append(CB)
Tr.append(Ti)
```

3. Resultados

Cuadro 1: Análisis de las instancias

Suma de los pesos de los objetos	Mediciones	Estadística
Instancia 1	0,09	2,997885194550087 porciento
	0,20	7,857709388825363 porciento
	1,0	5,923737554493455 porciento
Instancia 2	23	1,2345504465635582 porciento
	100	3,0208706337084634 porciento
	40	0,4934503866186409 porciento
Instancia 3	1,0	0,16747628028044556 porciento
	900	5,974668588789195 porciento
	80	3,0175308723987713 porciento

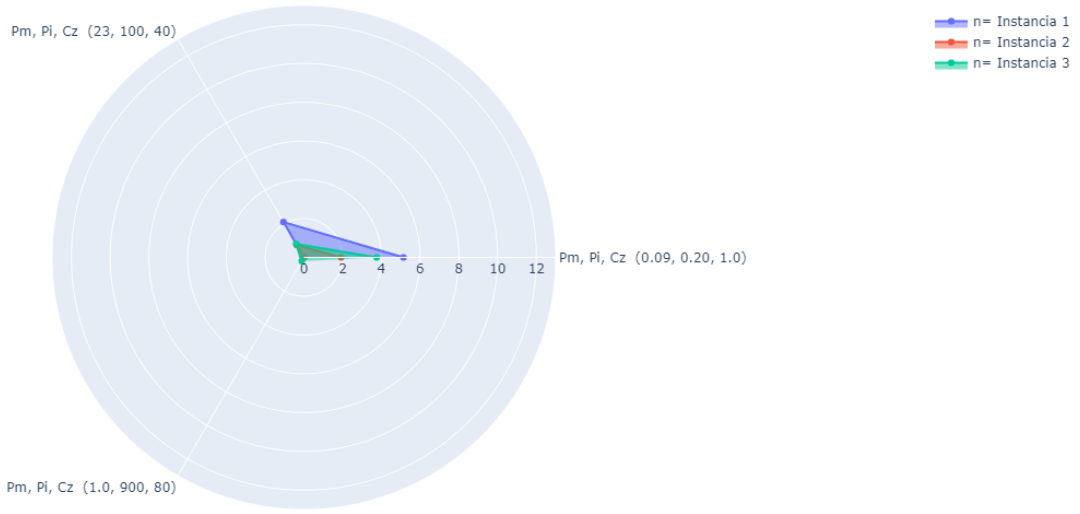


Figura 1: Analisis de los datos multivariados de las instancias.

4. Conclusiones

Como se puede apreciar en el diagrama de araña que es una herramienta muy útil para mostrar visualmente los gaps entre el estado actual y el estado ideal, se concluye que se puede implementar un algoritmo de genes para poder ejecutar problemas de complejidad en los análisis estadísticos como se observó en el cuadro de análisis de instancias ya que se varía los parámetros y con esto puede llegar a ser el valor óptimo por los valores que se pueden ejecutar con mayor fluidez.

Referencias

[1] N. Rodríguez. "p10.ªlgoritmo genético. *Repositorio, GitHub*, 2022. URL https://github.com/NestorZeus/SIMULACION-COMPUTACIONAL-DE-NANOMATERIALES/tree/main/P10.

[2] E. Schaeffer. Genetic algorithm. *Repositorio, GitHub*, 2022. URL https://github.com/satuelisa/Simulation/blob/master/GeneticAlgorithm/routines.py.

[1] [2]

"P11" Frentes de Pareto

NESTOR

Mayo 2022

1. Objetivo

El objetivo se trata de en optimización multicriterio, a un mismo conjunto de variables ocupa asignarse valores de tal forma que se optimizen 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. Desarrollo

De acuerdo con el desarrollo en el código de la codificación implementado por E. Schaeffer y todas las instrucciones se encuentran en el repositorio de N. Rodríguez en GitHub.

Para comenzar se implementa una solución aleatoria para polinomios que se generan al azar, con esto se crea la solución de polinomios y evaluación. A continuación se ejecuta el siguiente código:

```
import numpy as np
from scipy import stats
import pandas as pd
from itertools import compress
from random import randint, random
import matplotlib.pyplot as plt

def poli(maxdeg, varcount, termcount):
    f = []
    for t in range(termcount):
        var = randint(0, varcount - 1)
        deg = randint(1, maxdeg)
        f.append({'var': var, 'coef': random(), 'deg': deg})
    return pd.DataFrame(f)

def domin_by(target, challenger):
    if np.any(challenger < target):
        return False
    return np.any(challenger > target)

vc = 4
md = 3
tc = 5
```

```
#k = 2

replicas= 30
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) > 0.8
        n = 100
        sol = np.random.rand(n, vc)
        val = np.zeros((n, k))
        for i in range(n):
            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)
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize=(4, 10))
plt.ylabel('Porcentaje (%) solucion de pareto')
parts = ax.violinplot(CV, showmeans=False, showmedians=False, showextrema=False)
for p in parts['bodies']:
    p.set_facecolor('grey')
    p.set_edgecolor('green')
    p.set_alpha(1)
c='blue'
plt.xlabel('Funcion')
plt.close()
print(result)
```

3. Resultados

El rango de las soluciones de porcentaje no dominada se empieza a distribuir los porcentajes de las distribuciones de los porcentajes.

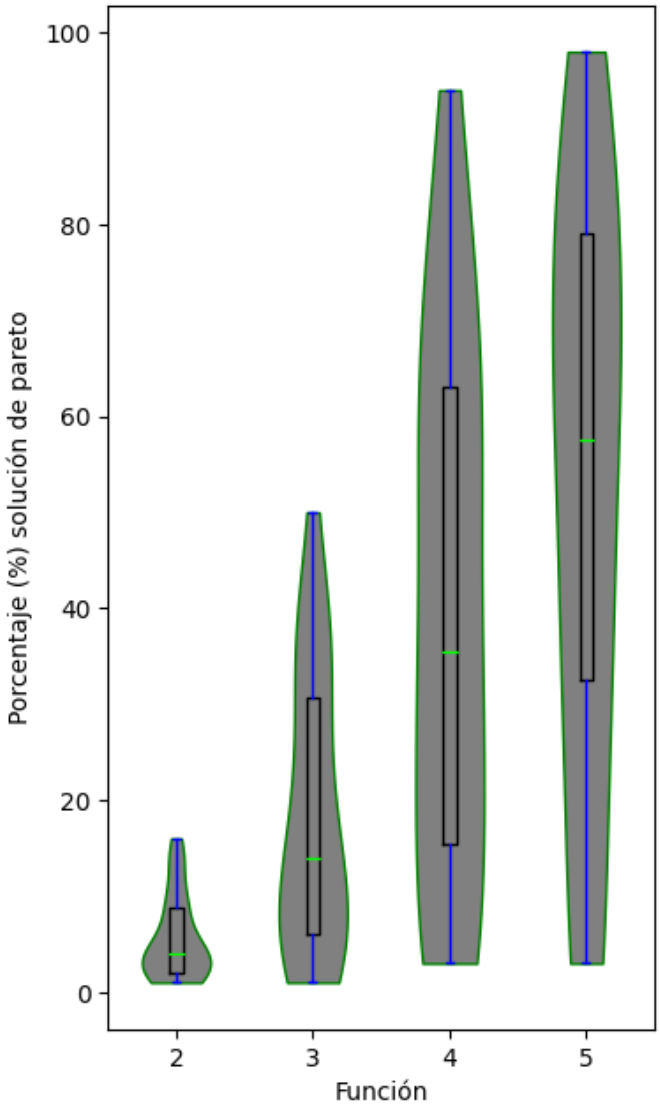


Figura 1: Diagrama violín distribucion de porcentaje

4. Conclusiones

Es complicado saber hayar una solución eficiente de una menor cantidad de funciones esto debido a que aumenta el rango de porcentajes de soluciones no encontradas.

Referencias

[1] E. Schaeffer. Genetic algorithm. *Repositorio, GitHub*, 2022. URL `https://github.com/satuelisa/Simulation/blob/master/ParetoFronts/poligen.py`.

"P12" Red neuronal

NESTOR

Mayo 2022

1. Objetivo

El objetivo de la práctica 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 (ngb), variando a las tres en un experimento factorial adecuado. [1]

2. Desarrollo

Basando el desarrollo en la codificación implementado por E. Schaeffer [1] y todas las instrucciones se encuentra en el repositorio de N. Rodríguez en GitHub.

Para comenzar se hace la función factorial para los modelos de las probabilidades negro, gris y blanco al número que se ingreso.

Código 1: Generamos la función

```
import itertools
from math import floor, log
import pandas as pd
factorial= itertools.product((1,0.5,0),(1,0.5,0),(1,0.5,0))
resultados=[]
for d1, d2, d3 in factorial:
    print('###############',d1,d2,d3,'###############')
    ciclos=10
    Rpl=[]
    for rpt in range(ciclos):
        modelos = pd.read_csv('digits.txt', sep=' ', header = None)
        modelos = modelos.replace({'n': d1, 'g': d2, 'b': d3})
```

Posteriormente se genera el F-score haciendo que se genere la matriz en confusión con la c

Código 2: Generamos F-score

```
c = pd.DataFrame(contadores)
c.columns = [str(i) for i in range(k)] + ['NA']
c.index = [str(i) for i in range(k)]
        arr=c.to_numpy()
        TP=sum(arr.diagonal())
        FP=(sum(sum(arr[:,:tope])))-TP
        FN= sum(sum(arr[:,-1:]))
        Precision= TP/(TP+FP)
        Recuperacion= TP/(TP+FN)
        puntajeF= 2*(Precision*Recuperacion)/(Precision+Recuperacion)
        Rpl.append(puntajeF)
resultados.append(Rpl)
```

3. Resultados

Se visualiza para la primer combinación (1, 1, 1) fue un F-score aproximado de 0,2, aquí lo que se busca es tener el F-score muy alto porqué se reconoció cai todos los números. ya que todas las combinaciones de las probabilidades se variaron y se hicieron réplicas y esas réplicas de (1, 0,5, 0,5) y (0, 0,5, 0,5) se obtuvo el cien porciento de número de detección correcta de verdaderos positivos un F-score alto.

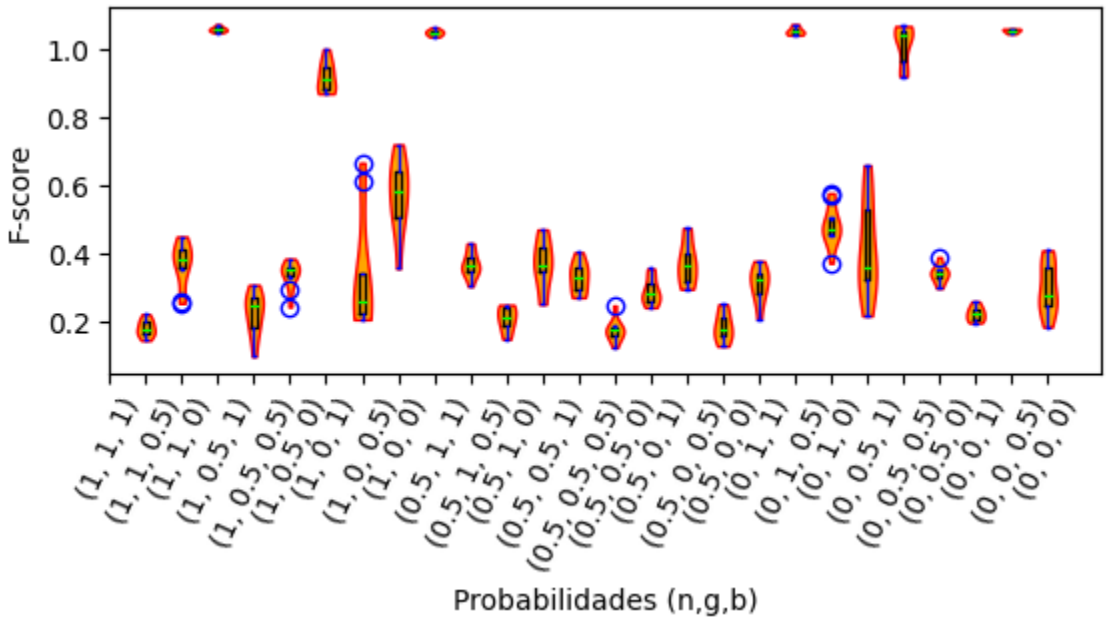


Figura 1: Diagrama de F-score

4. Conclusiones

Vimos un mejor comportamiento de F-score para las combinaciones (1, 1, 0), (1, 0,5, 0,5), (0, 0,5, 0,5), (0,5, 1, 1), (0, 1, 0,5) y (0, 0, 0,5) esto es un F-score muy alto.

5. Reto 1 Red neuronal con símbolos ASCII

En este reto consiste en la red neuronal para que reconozca además por lo menos doce símbolos ASCII adicionales, aumentando la resolución de las imágenes a 5 x 7 de lo original de 3 x 5 (modificando las plantillas de los dígitos acorde a este cambio).

Código 3: Se hace la modificación de modelos de 7 y 5

```
modelos = pd.read_csv('digits_12.txt', sep=' ', header = None)
modelos = modelos.replace({'n': d1, 'g': d2, 'b': d3})
r, c = 7, 5
dim = r * c
tasa = 0.15
tranqui = 0.99
tope = 21
```

6. Reto 1 Resultados

Se extendió la librería de reconocimiento de dígitos con 12 símbolos más y esta es la imagen de 5 x 7 mismo código.

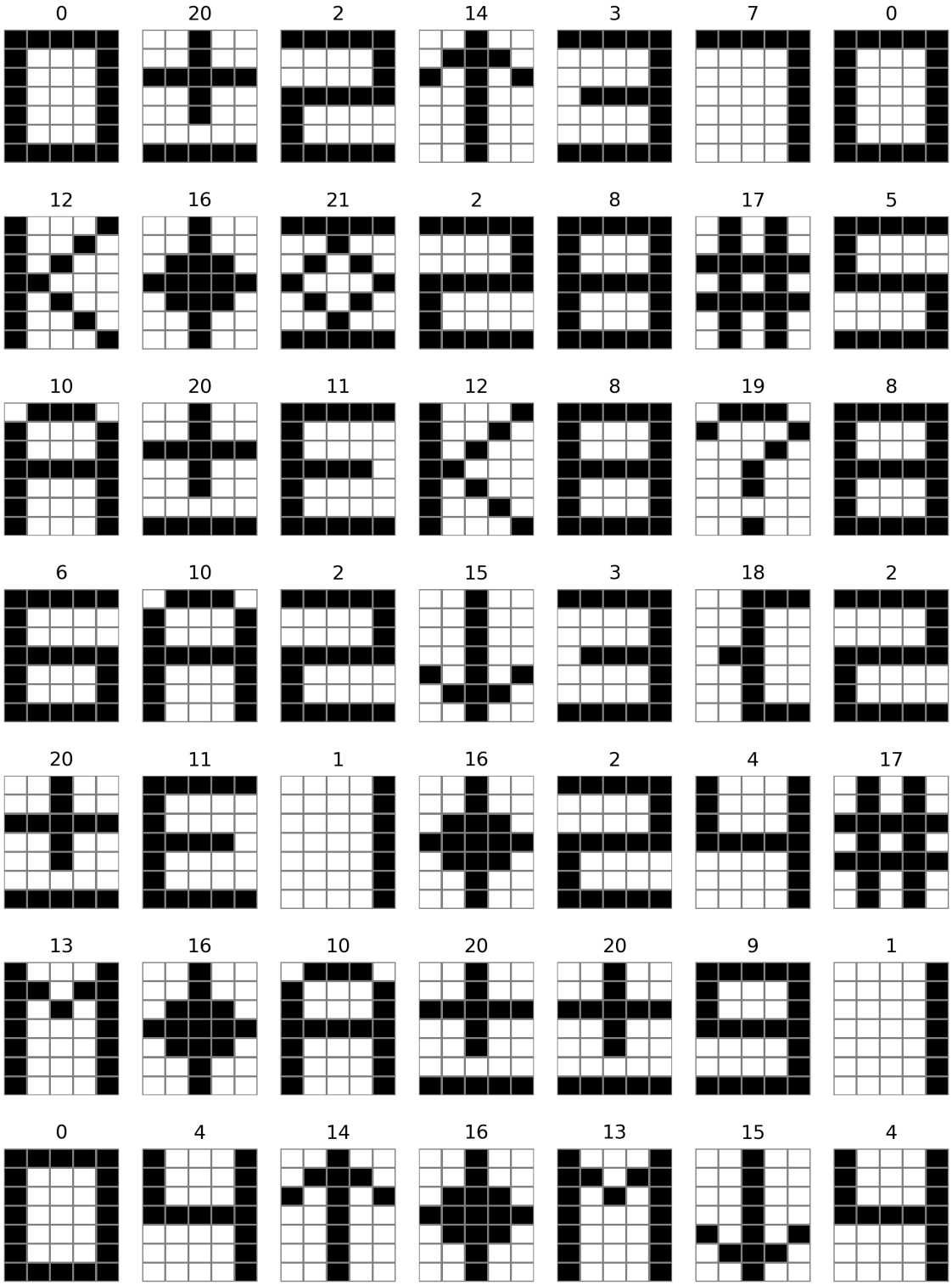


Figura 2: Diagrama de símbolos ASCII

7. Reto 2 Se estudia el ruido sal y pimienta

En este reto consiste 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).

Código 4: Se fijan los valores con el mismo ciclo

```
for pr in (0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1):
    print('###############',pr,'################')
    ciclos=10
    Rpl=[]
    for rpt in range(ciclos):
        modelos = pd.read_csv('digits.txt', sep=' ', header = None)
        modelos = modelos.replace({'n': 1, 'g': 0, 'b': 0})
        r, c = 5, 3
        dim = r * c
        tasa = 0.15
        tranqui = 0.99
        tope = 9
```

Para este de la fase de entrenamiento aquí está el cambio de sal y pimienta.

Código 5: Fijamos los valores

```
for t in range(5000): # entrenamiento
    d = randint(0, tope)
    pixeles = 1 * (np.random.rand(dim) < modelos.iloc[d])
    if (random.uniform(0, 1)) < pr:
        pixeles= random.randint(0, 1)* np.random.rand(dim)
for t in range(300): # prueba
    d = randint(0, tope)
    pixeles = 1 * (np.random.rand(dim) < modelos.iloc[d])
    if (random.uniform(0, 1)) < pr:
        pixeles= random.randint(0, 1)* np.random.rand(dim)
```

8. Reto 2 Resultados

Se muestra resultados de agresor ruido sal y pimienta son cuando la probabilidad es muy baja casi no se modifica la imagen y por lo tanto F-score sale alto ya que hubo muy buena detección de muchos verdaderos positivos en la matriz.

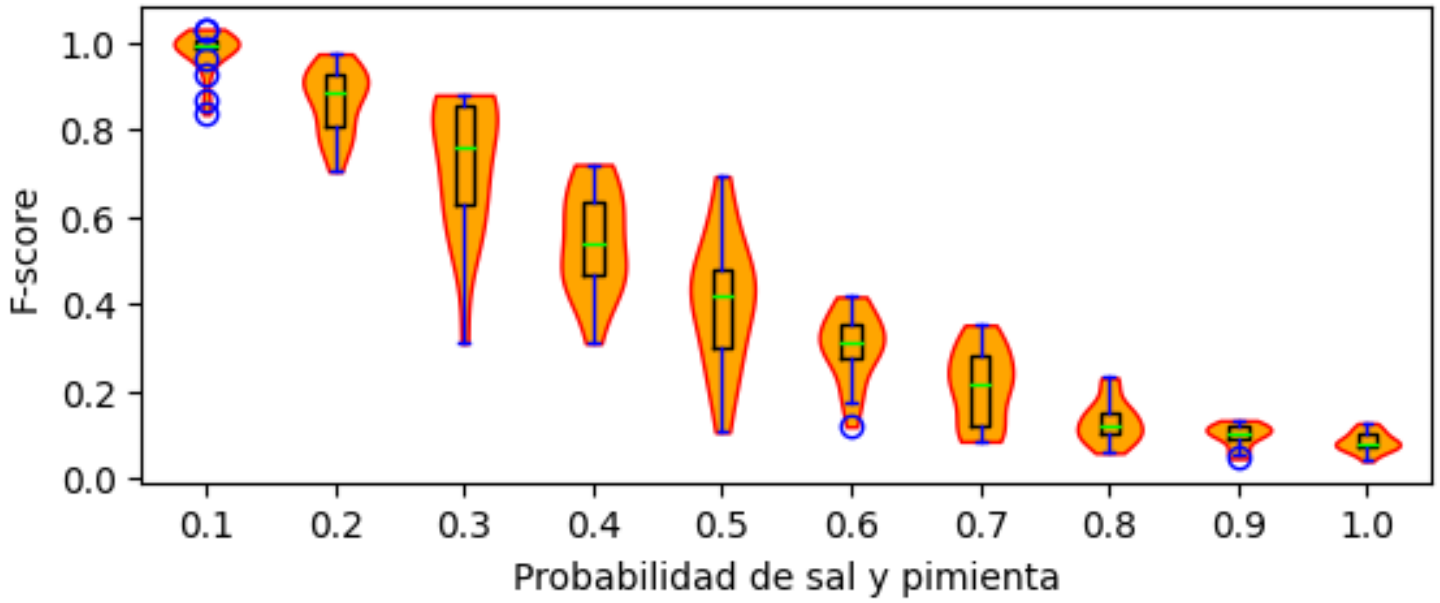


Figura 3: Diagrama de sal y pimienta.

Referencias

[1] E. Schaeffer. Neural network. *Repositorio, GitHub*, 2022. URL https://github.com/satuelisa/Simulation/blob/master/NeuralNetwork/perceptron.py.

Simulación del fenómeno de dispersión de electrones

Rodríguez Regalado N. D.
Correo electrónico: nestor.rodriguezrgl@uanl.edu.mx
: NestorZeus[1]

Maestría en Ciencias de la Ingeniería con Orientación en Nanotecnología

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

Resumen

La dispersión de electrones es utilizada para estudiar la materia haciendo que un haz de electrones incida sobre una muestra. Cuando se forma el haz, el choque de partículas (electrones) con las moléculas del aire, se dispersan, y esto se debe a la reducida masa de los electrones. Los resultados obtenidos son extraídos calculando el porcentaje de la cantidad de electrones que lograron dispersarse fuera del centro del haz ya que el estar en el centro implica que son contados como haz directo y si dispersan fuera del centro son importantes para la información cristalográfica del material.

Palabras clave: Electrón, dispersión, átomo.

1. Introducción

En el campo de la nanotecnología existe un fenómeno importante para el estudio y la caracterización cristalográfica de los nanomateriales para una comprensión de la estructura y sus propiedades [9], la interacción del electrón con los átomos de una muestra delgada se presenta en el microscopio electrónico de transmisión cuando se quiere analizar una muestra pequeña y estudiar la estructura cristalina para obtención de información cristalográfica. Cuando un haz de electrones es transmitido por la muestra existen interacciones como ionización, pérdida de energía, etc [1].

Una interacción importante característica del efecto en este microscopio es la fuerza eléctrica o interacción electrostática mejor conocido como fuerza de Coulomb que ocurre cuando un electrón que tiene carga negativa impacta contra un átomo (carga positiva) y su nube de electrones y ocasiona una atracción entre cada uno y a su vez una ligera repulsión, de esta manera dispersando el electrón en un ángulo dependiendo de la cercanía a la que pase del átomo. Estas dispersiones se utilizan como señales para obtener información cristalográfica de la muestra [10].

Comúnmente el análisis de `Monte Carlo` es utilizado para comprender este fenómeno en los microscopios y obtener una estimación de la dispersión dependiendo la muestra en el que analizan el efecto entre electrón-sólido [9].

En el presente trabajo de simulación se diseña un modelo de dispersión de electrones con valor de carga n que impacta a una muestra periódica de átomos correspondientes a muestras que se cambiaron para estudiar los distintos efectos de la dispersión debido a los tamaños de átomo y fuerzas de carga positiva de cada uno, obteniendo una estimación de electrones dispersados respecto a que tan cerca pasan del átomo. Como objetivos del trabajo se busca:

- Generar el efecto de la interacción del electrón con átomo a una distancia mínima y replicar pa-

ra una cantidad n de electrones,

- variar la muestra de átomos simulando distintos elementos en el que cambia la carga positiva que repele al electrón y su tamaño para revisar el efecto que tiene en la dispersión,
- realizar un análisis estadístico de la cantidad de electrones dispersados y que tanto se alejaron del centro respecto a la cantidad de electrones y el tipo de material utilizado como muestra.

Este trabajo esta dividido primeramente hablando de los antecedentes de este tipo de simulaciones o estudios realizados donde principalmente se utiliza el método `Monte Carlo` para las estimaciones de la trayectoria del electrón, continuando con la sección de trabajos relacionados donde se revisa las simulaciones en las que se basa este código para realizar la implementación que en el capítulo de implementación de simulación se explica aunado a las herramientas utilizadas. Como final cuenta con las secciones de experimentos donde se habla del diseño de la simulación, los resultados obtenidos y una discusión de dichos resultados, terminando con una conclusión general del fenómeno simulado.

2. Antecedentes

En el siguiente articulo reportado se propone simular las trayectorias que siguen los electrones de un haz que incide en una muestra de un determinado material. Tomando un promedio de las dispersiones con modelos probabilısticos que son abordados con el método de Monte Carlo.

Estos efectos simulados son observados en la microscopıa electrónica la cual es una técnica de formación de imágenes basada en la incidencia de un haz de electrones sobre una muestra, donde ocurren los efectos por la fuerza de Coulomb ocasionada entre el haz de electrones y los átomos del material muestra. Surgen efectos de dispersión elástica la cual puede desencadenar simultáneamente la desviación de la trayectoria del electrón, debido a la atracción eléctrica y la velocidad de impacto del electrón. Se describe que es complicado modelar para cada electrón los efectos que ocurren en el fenómeno, y se propone para solucionar esta tarea asignar probabilidades a ciertos eventos como que éste sea transmitido a través de la muestra a lo cual el método de Monte Carlo resulta eficiente [7].

En diversos trabajos se aborda el método Monte Carlo para resolver integrales o representar esta simulación de dispersión. Debido a que en los dominios de la física el método mencionado es popular para simular fenómenos descritos por la mecánica estadística como el transporte de neutrones reportado [3].

En otro trabajo similar utilizando el software `Python` producen haces de electrones paralelos dando condiciones de la velocidad de los electrones dado en kilovoltios y la desviación parabólica de cada uno dando un buen resultado a través de la simulación que se expone [11].

2.1. Trabajos relacionados

Esta simulación o implementación esta basada en dos trabajos, Interacción entre partículas y Sistema multiagente que pueden ser consultados en el repositorio de Schaeffer [4]. Estas prácticas son seleccionadas ya que simulan un comportamiento similar al del electrón cuando sufre esta dispersión respecto a la distancia que se acerca al átomo.

3. Modelo propuesto

El modelo propuesto en el presente trabajo se basa en el fenómeno descrito por Cárter y Williams [1] en donde se explica la dispersión del electrón cuando impacta un átomo aislado para estudiar su efecto tal como se puede observar en la figura 1 en como los electrones entre más cerca pasen del átomo sufrirán una mayor fuerza de atracción y por lo tanto el ángulo de dispersión sera mayor. Ampliando el concepto del átomo aislado, de igual manera se plantea lo que sucede si los electrones impactan a un conjunto de átomos periódicamente distribuidos y lograr simular el efecto que ocurre en un material cristalino ya que cada dispersión representa una posición atómica de la muestra.

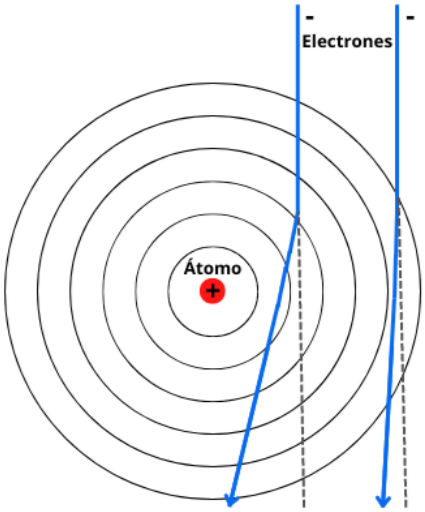


Figura 1: Interacción con átomo aislado.

4. Implementación de simulación

El desarrollo de este código fue realizado en el lenguaje de programación `Python` [5], así como librerías del mismo programa como `Numpy` [2] y `Matplotlib` [6] para la visualización de los gráficos y animaciones realizadas.

Partes importantes de la simulación del fenómeno se explican tal como puede ser visto código inicial 1 donde se declaran los electrones en sus posiciones al azar (x, y) al igual que los átomos solo que estos tendrán posiciones fijas.

```
electronX=[random.uniform(0,180) for i in
    range(n)]
electronY=[115 for i in range(n)]
planoX= [i for i in range(5,170,40)]
planoY= [80 for i in range(len(planoX))]
atomoP= [25 for i in range(len(planoX))]
```

Listing 1: Declaración de electrones y átomos.

Otra parte importante es donde cada electrón que se genera es comparada contra cada átomo obteniendo la distancia euclidiana y así de esta manera proceder en cada paso del electrón tal como se observa en el código 2, dichas distancias son utilizadas para cuando el electrón entre en rango de la fuerza de atracción que se genera debido a las cargas opuestas entre electrón y átomo y de esta manera se genere un desvío en la trayectoria del electrón respecto a la distancia, si es muy cercano al átomo esta fuerza se incrementará.

```
for xi, yi in electrones:
    A=[]
    electronY[cambioE]=electronY[cambioE]-
    paso
    for xf, yf in atomos:
        distancia=sqrt(((xf-xi)**2)+((yf-yi)*
    *2))
        if distancia <= Fuerza and yi>=yf:
            desvio=(pasomax/Fuerza)*(Fuerza-
    distancia)
```

Listing 2: Distancia entre electron y átomo.

Después de que el electrón entre en el rango de la fuerza de atracción del átomo se realiza una condición para que el electrón se desvié haciendo una probabilidad del cincuenta porciento, si se cumple esta condición entonces se realiza una comprobación de si la posición x del electrón (xi) esta en el lado derecho del átomo entonces sufrirá una desviación negativa y viceversa para simular el efecto de atracción tal como se desarrolla en el código 3.

```
Pd=0.5
if (random.uniform(0,1))<Pd:
    if xi >= xf:
        electronX[cambioE]=electronX[cambioE
    ]-desvio
        A.append('I')
    if xi < xf:
        electronX[cambioE]=electronX[cambioE
    ]+desvio
        A.append('D')
```

Listing 3: Probabilidad y direccion del desvió.

5. Experimentos

La simulación desarrollada como primera instancia se simuló una secuencia de interacción de un electrón con un solo átomo para validar la dispersión del átomo aislado que puede ser visualizado en el repositorio de Rodríguez [8], en dicha simulación se puede observar como el electrón comienza a desviarse hasta el centro del átomo para cuando lo atraviesa cambia su dirección de dispersión debido a la interacción de coulomb que sufrió (ver figura 2). Este primer experimento del fenómeno descrito por Williams [1] fue

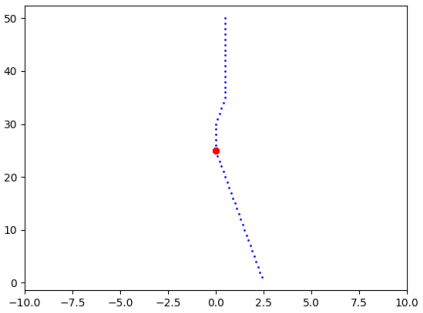


Figura 2: Electrón interacciona con átomo.

ampliado a un concepto simulado en el que se logró colocar un conjunto de átomos y se procede a impactar una cantidad n de electrones para estudiar la dispersión y entre más cerca del átomo se aproxime, mayor resulta la dispersión (ver figura 3) y de misma manera se puede observar en el repositorio [8].

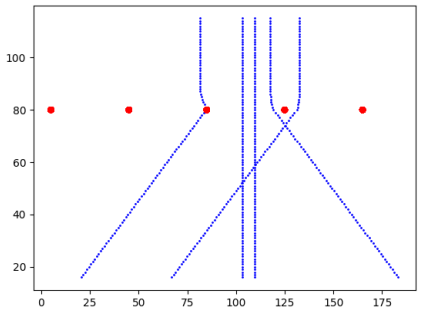


Figura 3: Interaccion de conjunto de electrones con átomos.

5.1. Resultados

Los resultados obtenidos son extraídos calculando el porcentaje de la cantidad de electrones que lograron dispersarse fuera del centro del haz ya que el estar en el centro implica que son contados como haz directo y si dispersan fuera del centro son importantes para la información cristalográfica del material.

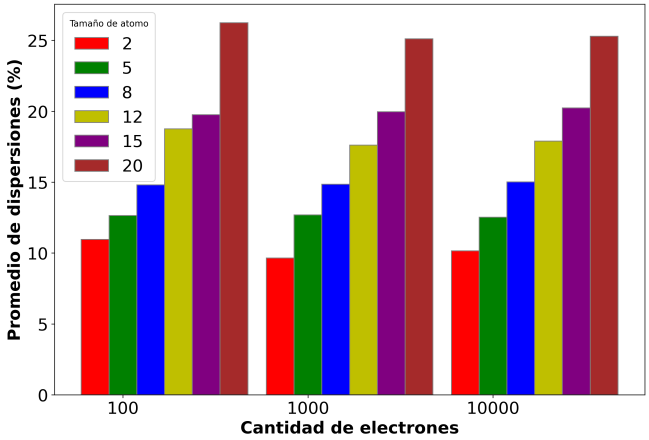


Figura 4: Gráfico de promedios de dispersión respecto al tamaño de átomo.

El porcentaje de dispersiones cambió respecto al tamaño del átomo así como en la cantidad de electrones y dichos resultados pueden ser observados gráficamente en la figura 4,

Los datos extraídos de los gráficos se puede visualizar en la tabla 1 donde se puede hacer una determinación de si a mayor número de electrones es significativo el mejoramiento en el porcentaje de electrones o si depende en su mayoría por el tamaño del átomo.

5.2. Discusión

Los resultados que fueron encontrados gráficamente se procesan mediante análisis estadísticos por el método de `Kruskal Wallis` el cual arroja si las variables utilizadas en el experimento son estadísticamente significativas.

De los resultados mostrados en la tabla 2 se pudo concluir que el valor `p` para el tamaño de átomos fue mayor a 0.05 mientras que para la cantidad de electrones no se cumplió.

Cuadro 1: Tabla de resultados en dispersiones por tamaño de átomos y cantidad de electrones.

Cantidad electrones	Tamaño de átomo	Electrones dispersos (%)
100	2	10.96
	5	12.64
	8	14.80
	12	18.76
	15	19.76
	20	26.24
1000	2	9.64
	5	12.68
	8	14.85
	12	17.61
	15	19.96
	20	25.10
10000	2	10.15
	5	12.52
	8	15.02
	12	17.89
	15	20.22
	20	25.28

Cuadro 2: Pruebas estadísticas `Kruskal Wallis`.

Datos	Prueba estadística	Valor P
Tamaño de átomo	0.035	0.982
Cantidad de electrones	16.578	0.005

6. Conclusiones

Se puede concluir con este trabajo que el efecto de dispersión de electrones depende en gran proporción a la cantidad de electrones que sean impactados para obtener mejor el porcentaje de dispersión así como el tamaño o fuerza del átomo, se observa que entre mayor sea esta fuerza la dispersión es mayor, por lo tanto menos electrones pasan sin sufrir efecto.

Y en base a la prueba estadística se logró concluir que el tamaño del átomo es significativo para la simulación del fenómeno mientras que la cantidad de electrones no resulta ser significativo en el experimento debido a que el valor de `p` resultó ser menor a 0,05.

7. Trabajo a futuro

Como trabajo a futuro pendiente, la experimentación se puede ampliar a una interacción mas completa donde cada átomo cuenta con nube de electrones que ocasionan fuerzas extra en el electrón incidente, aunado a esto agregar una velocidad al electrón ya que este en la literatura y en la practica se menciona como significativo para el fenómeno.

Referencias

[1] Williams D. & Carter C. *Transmission Electron Microscopy A Textbook for Materials Science*, volume 5. Springer Science & Business Media, 1999. ISBN 9780387765006.

[2] Charles R., Jarrod M., Gommers R., Virtanen P. & Cournapeau D. *Array programming with NumPy*, volume 585. Springer Science and Business Media LLC, 2020. doi: 10.1038/s41586-020-2649-2.

[3] Rubinstein R. & Kroese D. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016. ISBN 9781118632208.

[4] Schaeffer E. *Repositorio Github*. 2022. URL https://github.com/satuelisa/Simulation.

[5] Rossum G. *Python*. 2021. URL https://www.python.org/.

[6] Hunter J. *Matplotlib: A 2D graphics environment*, volume 9. IEEE computer society, 2007. doi: 10.1109/MCSE.2007.55.

[7] Saez W., Angulo D. & Murillo M. *Simulación de la trayectoria de un haz de electrones en un sólido usando el método de Monte Carlo*. URL https://www.academia.edu/32123432.

[8] Rodríguez N. *Repositorio Github*. 2022. URL https://github.com/NestorZeus/SIMULACION-COMPUTACIONAL-DE-NANOMATERIALES/tree/main/Proyecto%20Final.

[9] Hovington P., Dominique D. & Gauvin R. *CASINO: A new Monte Carlo code in C language for electron beam interaction—Part I: Description of the program*, volume 19. Wiley Online Library, 1997. doi: 10.1002/sca.4950190101.

[10] Janecek M. & Kral R. *Modern electron microscopy in physical and life sciences*, volume 2. Books on Demand, 2016. ISBN 9789535122524.

[11] Feng Y., Mao R., Li P., Kang X., Yin Y., Liu T., You Y., Chen Y., Zhao T. & Xu Z. *Beam distribution reconstruction simulation for electron beam probe*, volume 41. IOP Publishing, 2017. doi: 10.1088/1674-1137/41/7/077001.