

Integrantes

- Oscar Juian Rogriguez Cardenas
- Tania Julieth Araque Dueñas
- Angela Sofia Rubiano Quintero

✓ Ejercicio 1

Considere el modelo Ising: en el lattice $k \times k$ ($10 \leq k \leq 20$)

$\pi_\beta(\eta) = \frac{1}{Z_\beta} e^{-\beta H(\eta)}$. Donde $H(\eta) = - \sum_{x \sim y} \eta_x \eta_y$ y $\beta > 0$ es el inverso de la temperatura.

Solución:

-Veremos algunas de las simulaciones para estudiar su comportamiento.

A) La primera haciendo uso de el algoritmo MCMC Gibbs sampler en este caso para generar 100 muestras **aproximadas** del modelo Ising con inverso de temperatura , en $\beta = \{0, 0.1, 0.2, \dots, 0.9, 1\}$

B) La segunda haciendo uso de el algoritmo de Propp-Wilson Sandwiching para obtener 100 muestras **exactas** del modelo Ising con las mismas temperaturas que en item a) $\beta = \{0, 0.1, 0.2, \dots, 0.9, 1\}$

C) Estimación y reportes

✓ a) 100 muestras aproximadas

```
import numpy as np
import matplotlib.pyplot as plt
import random
from ipywidgets import interact, IntSlider, FloatSlider

# Función auxiliar para calcular la probabilidad de actualización
```

```

def calcular_probabilidad(beta, delta_energia):
    return 1 / (1 + np.exp(-2 * beta * delta_energia))

# Función para encontrar vecinos del vértice en la cuadrícula
def calcular_vecinos(lattice, i, j):
    vecinos = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    positivos = sum(lattice[(i + dx) % lattice.shape[0], (j + dy) % lattice.shape[1]]
                    for dx, dy in vecinos)
    negativos = 4 - positivos # 4 vecinos en total, resto son negativos
    return positivos, negativos

# Función para el muestreador de Gibbs
def gibbs_sampler(lattice, beta):
    # Selección de una posición aleatoria
    i, j = random.choices(range(lattice.shape[0]), k=2) # Selección de una posición
    vecinos_positivos, vecinos_negativos = calcular_vecinos(lattice, i, j)
    delta_energia = vecinos_positivos - vecinos_negativos

    # Calcular probabilidad de cambio usando la función auxiliar
    probabilidad_cambio = calcular_probabilidad(beta, delta_energia)
    lattice[i, j] = 1 if random.random() < probabilidad_cambio else -1

    return lattice

# Función interactiva para ejecutar el muestreador de Gibbs
def ejecutar_muestreador(beta, tamaño):
    iteraciones = [10**3, 10**4, 10**5] # Iteraciones donde se tomaran muestras

    lattice = np.full((tamaño, tamaño), -1) # Inicializar cuadrilla con -1
    print(f"\nMuestras para beta = {beta}:")

    # Iteración para el muestreo
    for paso in range(1, max(iteraciones) + 1):
        lattice = gibbs_sampler(lattice, beta)

    # Muestreo y visualización en iteraciones especificadas
    if paso in iteraciones:
        print(f"Iteración {paso}:")
        plt.imshow(lattice, cmap='Purples', interpolation='nearest')
        plt.title(f"Beta = {beta}, Iteración = {paso}")
        plt.show()

# Crear sliders interactivos para beta y tamaño
interact(ejecutar_muestreador,
        beta=FloatSlider(min=0, max=1, step=0.1, value=0.5, description='Beta'),
        tamaño=IntSlider(min=10, max=20, step=1, value=10, description='Tamaño')

```

```
 tamaño=IntSlider(min=10, max=20, step=1, value=10, description= 'k' )
```



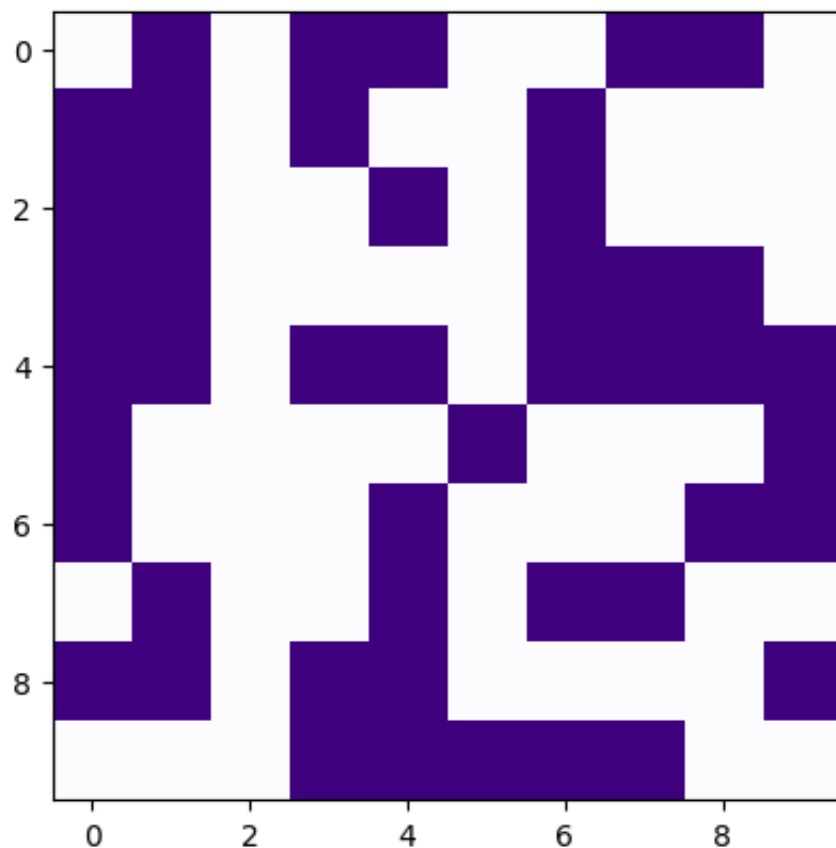
Beta

k

Muestras para beta = 0.0:

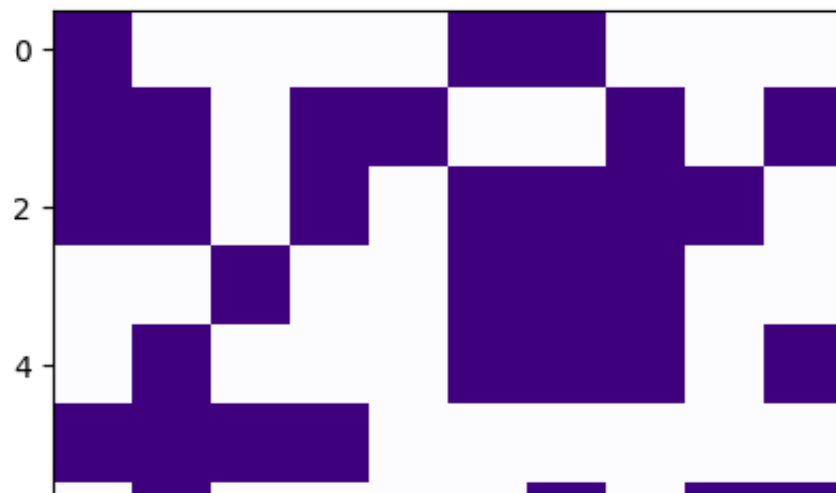
Iteración 1000:

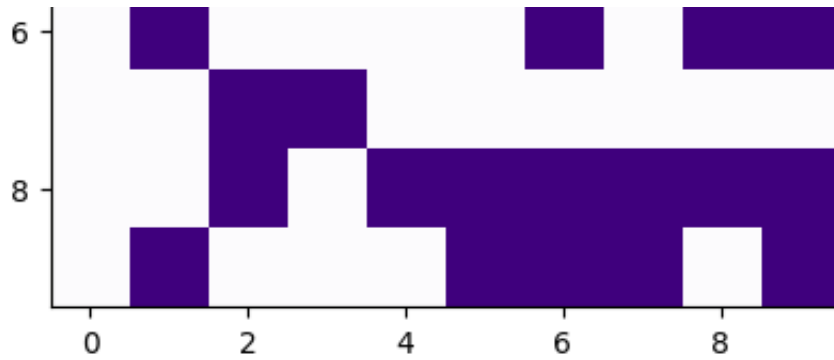
Beta = 0.0, Iteración = 1000



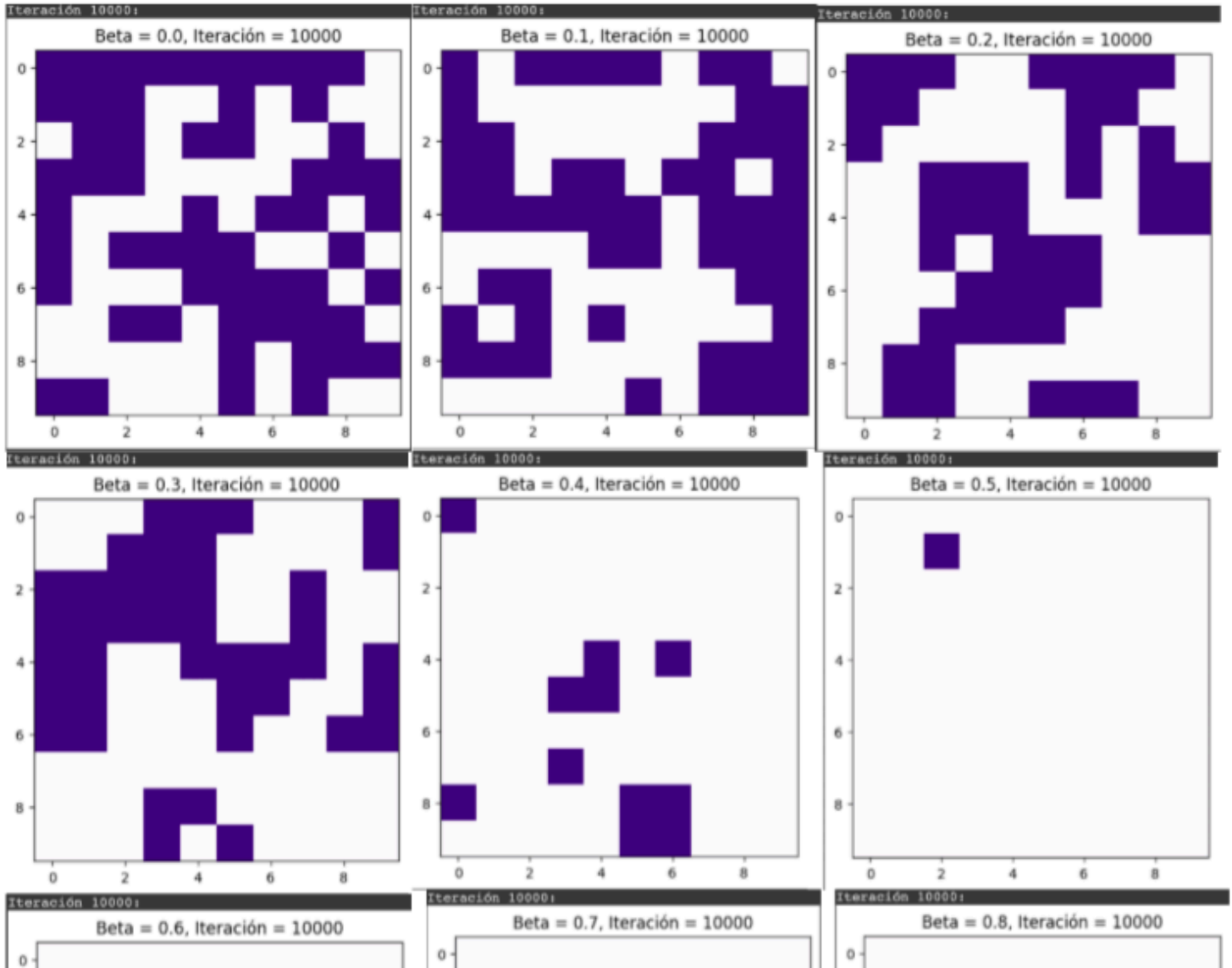
Iteración 10000:

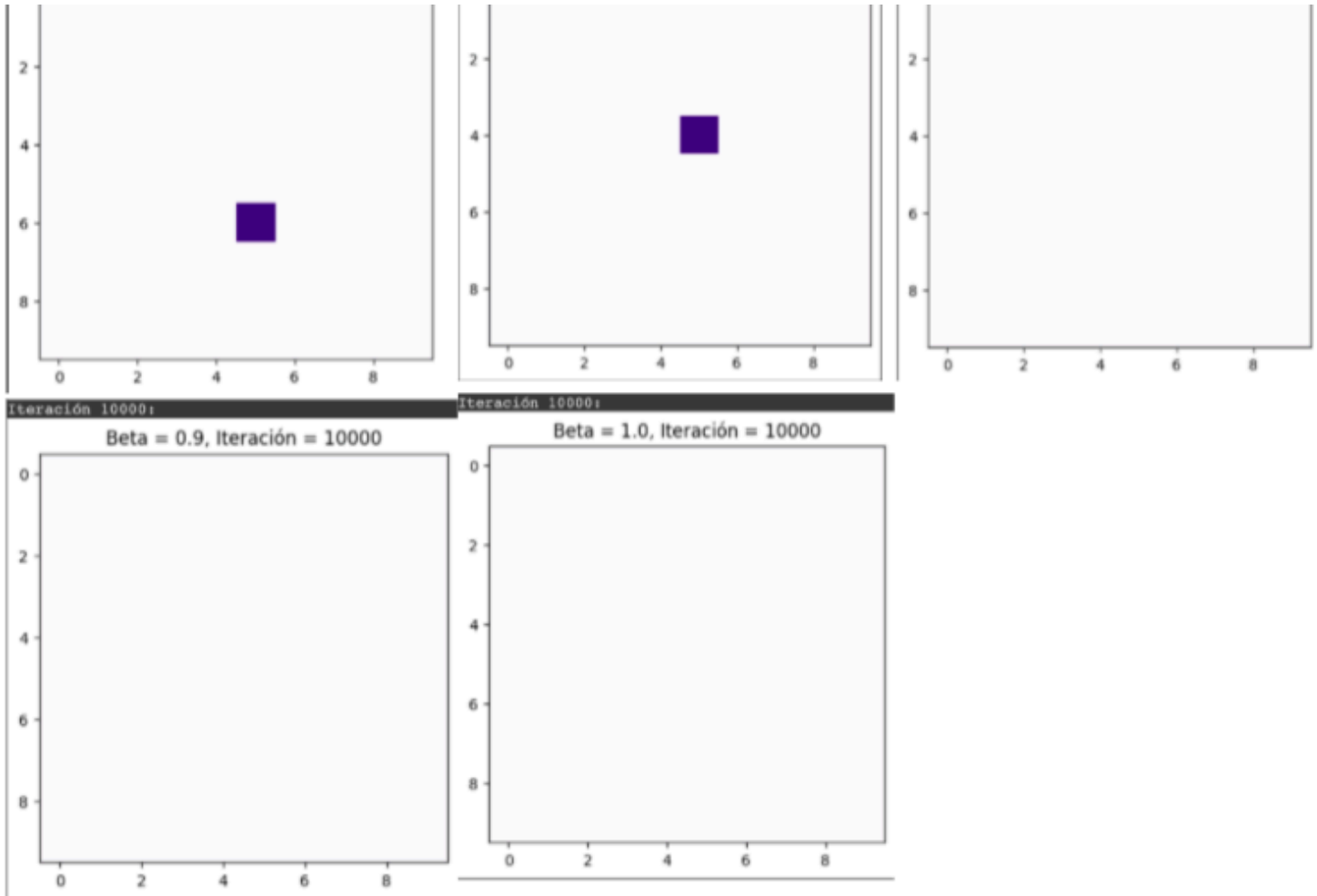
Beta = 0.0, Iteración = 10000





Tomamos por ejemplo en este caso la interacion 10000, por motivos de desarrollo de los siguientes puntos , para los diferentes valores de β en el lattice $k=10$, obteniendo asi muestras que siguen este comportamiento (en el codigo anterior se puede visulizar el dinamismo de las muestras haciendo uso de las barras interactivas)





Generemos 100 muestras por ejemplo

```
import numpy as np
import matplotlib.pyplot as plt
import random
from ipywidgets import interact, IntSlider, FloatSlider

# Función auxiliar para calcular la probabilidad de actualización
def calcular_probabilidad(beta, delta_energia):
    return 1 / (1 + np.exp(-2 * beta * delta_energia))

# Función para encontrar vecinos del vértice en la cuadrícula
def calcular_vecinos(lattice, i, j):
    vecinos = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    positivos = sum(lattice[(i + dx) % lattice.shape[0], (j + dy) % lattice.shape[1]] for dx, dy in vecinos)
    negativos = 4 - positivos # 4 vecinos en total, resto son negativos
```

```
return positivos, negativos
```

```
# Función para el muestreador de Gibbs
```

```
def gibbs_sampler(lattice, beta):
    # Selección de una posición aleatoria
    i, j = random.choices(range(lattice.shape[0]), k=2) # Selección de una posición
    vecinos_positivos, vecinos_negativos = calcular_vecinos(lattice, i, j)
    delta_energia = vecinos_positivos - vecinos_negativos

    # Calcular probabilidad de cambio usando la función auxiliar
    probabilidad_cambio = calcular_probabilidad(beta, delta_energia)
    lattice[i, j] = 1 if random.random() < probabilidad_cambio else -1

    return lattice
```

```
# Función para ejecutar el muestreador de Gibbs y generar 100 muestras para diferentes valores de beta
```

```
def generar_muestras(tamaño, temp_inversas):
    num_total_muestras = 100 # Número total de muestras deseadas
    muestras_por_beta = num_total_muestras // len(temp_inversas) # Número de muestras por beta
    muestras = [] # Lista para almacenar todas las muestras

    for beta in temp_inversas:
        lattice = np.full((tamaño, tamaño), -1) # Inicializar cuadrícula con -1

        # Iteración para el muestreo
        for _ in range(muestras_por_beta):
            # Ejecutar suficientes pasos de Gibbs para obtener una muestra
            for _ in range(tamaño**2): # Actualizar todos los espines una vez
                lattice = gibbs_sampler(lattice, beta)
            muestras.append(np.copy(lattice)) # Almacenar la muestra

    print("100 muestras generadas para diferentes valores de beta.")
    return muestras
```

```
# Función para visualizar algunas muestras generadas
```

```
def graficar_muestras(muestras, temp_inversas):
    plt.figure(figsize=(15, 10))
    num_muestras = len(muestras)
    for i in range(0, num_muestras, num_muestras // len(temp_inversas)): # Muestra cada beta
        plt.subplot(3, 4, i // (num_muestras // len(temp_inversas)) + 1)
        plt.imshow(muestras[i], cmap='Purples', interpolation='nearest')
        plt.title(f"Beta = {temp_inversas[i // (num_muestras // len(temp_inversas))]}")
        plt.axis('off')
    plt.tight_layout()
```

```
# Parámetros
tamaño_red = 10 # Dimensión de la red
temp_inversas = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0] # Lista de

# Generar y graficar las muestras
muestras = generar_muestras(tamaño_red, temp_inversas)
print(muestras)
graficar_muestras(muestras, temp_inversas)
```

```

100 muestras generadas para diferentes valores de beta.
[[-1, -1, -1, 1, -1, -1, -1, -1, -1, -1],
 [1, 1, 1, 1, 1, -1, -1, 1, -1, -1],
 [-1, -1, 1, -1, 1, 1, -1, 1, 1, -1],
 [-1, -1, -1, 1, 1, -1, -1, -1, -1, -1],
 [1, 1, -1, -1, -1, -1, -1, -1, -1, -1],
 [1, 1, 1, -1, -1, -1, -1, -1, -1, 1],
 [-1, 1, 1, -1, -1, 1, 1, -1, -1, 1],
 [1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
 [1, 1, -1, -1, 1, -1, 1, -1, -1, 1],
 [1, -1, -1, -1, -1, -1, 1, -1, -1, -1]], array([[ -1, -1, 1, 1, -1,
 [1, -1, 1, 1, 1, -1, 1, 1, -1, -1],
 [-1, 1, 1, -1, 1, 1, 1, -1, 1, -1],
 [-1, 1, 1, 1, 1, -1, -1, -1, 1, 1],
 [-1, -1, 1, -1, -1, -1, -1, -1, -1, 1],
 [-1, 1, 1, -1, -1, -1, 1, 1, -1, -1],
 [1, 1, 1, -1, -1, -1, 1, -1, -1, -1],
 [1, 1, -1, -1, 1, -1, -1, -1, 1, -1],
 [1, 1, 1, -1, -1, 1, 1, -1, -1, -1],
 [1, -1, -1, -1, 1, 1, 1, -1, 1, -1]], array([[ 1, -1, 1, -1, -1,
 [-1, 1, 1, -1, 1, 1, -1, -1, -1, 1],
 [-1, -1, 1, 1, 1, 1, 1, -1, 1, -1],
 [1, 1, -1, 1, -1, -1, 1, -1, 1, 1],
 [1, -1, 1, 1, -1, -1, 1, -1, -1, -1],
 [1, -1, 1, -1, 1, -1, 1, 1, -1, -1],
 [1, 1, -1, -1, -1, -1, -1, 1, 1, -1],
 [1, 1, -1, 1, -1, -1, 1, -1, -1, -1],
 [1, 1, 1, 1, 1, -1, -1, -1, 1, -1],
 [-1, -1, -1, -1, 1, -1, 1, -1, 1, 1]], array([[ 1, -1, 1, 1, -1,
 [-1, 1, 1, -1, 1, 1, -1, 1, -1, 1],
 [-1, -1, 1, -1, -1, -1, 1, 1, -1, -1],
 [1, -1, -1, -1, -1, -1, 1, 1, 1, 1],
 [1, -1, -1, 1, -1, -1, 1, 1, -1, -1],
 [1, -1, 1, -1, 1, 1, 1, 1, -1, -1],
 [1, -1, -1, -1, -1, 1, -1, 1, 1, 1],
 [1, 1, -1, 1, 1, -1, 1, -1, 1, 1],
 [1, 1, -1, 1, 1, 1, -1, -1, -1, -1],
 [1, -1, 1, -1, 1, -1, 1, -1, -1, 1]], array([[ -1, 1, 1, -1, 1,

```

```

[ 1, -1, 1, 1, -1, 1, -1, -1, -1, 1],
[ 1, -1, 1, -1, -1, 1, 1, 1, -1, 1],
[-1, 1, 1, 1, -1, 1, 1, 1, -1, -1],
[-1, 1, -1, 1, -1, -1, 1, 1, -1, -1],
[ 1, -1, 1, -1, 1, 1, -1, 1, -1, 1],
[-1, 1, 1, 1, -1, 1, -1, -1, 1, 1],
[-1, 1, -1, 1, 1, -1, 1, -1, 1, 1],
[ 1, 1, -1, 1, 1, 1, 1, -1, -1, -1],
[-1, -1, -1, -1, 1, -1, 1, -1, -1, -1]], array([[ -1, 1, 1, -1, 1,
-1, -1, 1, 1, 1, 1, 1, 1, -1, -1, 1],
[-1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1],
[-1, -1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
[-1, 1, -1, 1, -1, -1, 1, 1, -1, 1],
[-1, -1, 1, -1, 1, 1, -1, -1, -1, -1],
[-1, 1, 1, 1, -1, 1, 1, 1, 1, 1],
[ 1, 1, -1, -1, 1, 1, -1, -1, 1, 1],
[ 1, 1, 1, 1, 1, -1, 1, -1, -1, -1],
[ 1, 1, 1, -1, -1, 1, 1, -1, 1, -1]]], array([[ -1, 1, 1, -1, 1,
-1, -1, -1, -1, 1, 1, 1, -1, -1, -1],
[-1, 1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
[ 1, -1, 1, -1, -1, 1, 1, 1, 1, 1]

```

✓ b) 100 muestras exactas

```

[-1, 1, 1, 1, 1, 1, -1, -1, -1, -1],

import numpy as np

# Función para contar los vecinos positivos y negativos
def conteo_vecinos(lattice, pos_x, pos_y):
    movimientos = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    positivos = 0
    for dx, dy in movimientos:
        nuevo_x = (pos_x + dx) % lattice.shape[0]
        nuevo_y = (pos_y + dy) % lattice.shape[1]
        if lattice[nuevo_x, nuevo_y] == 1:
            positivos += 1
    negativos = 4 - positivos
    return positivos, negativos

# Función para obtener la probabilidad de un espín positivo
def obtener_probabilidad_espin(lattice, pos_x, pos_y, beta):
    positivos, negativos = conteo_vecinos(lattice, pos_x, pos_y)
    diferencia_energia = positivos - negativos
    return 1 / (1 + np.exp(-2 * beta * diferencia_energia))

# Función para realizar una actualización en la red

```



```

def actualizar_red(lattice, beta):
    for x in range(lattice.shape[0]):
        for y in range(lattice.shape[1]):
            prob = obtener_probabilidad_espin(lattice, x, y, beta)
            lattice[x, y] = 1 if np.random.rand() < prob else -1
    return lattice

# Función principal del algoritmo Propp-Wilson
def propp_wilson(dimension, beta, max_iter=1000):
    red_inferior = -np.ones((dimension, dimension))
    red_superior = np.ones((dimension, dimension))

    for iteracion in range(max_iter):
        red_inferior = actualizar_red(red_inferior, beta)
        red_superior = actualizar_red(red_superior, beta)

        if np.array_equal(red_inferior, red_superior):
            return red_superior

    return red_superior

# Función para generar una cantidad específica de muestras agrupadas por beta
def obtener_muestras_por_beta(dimension, betas, cantidad_muestras=100):
    muestras_por_beta = {beta: [] for beta in betas}
    for beta in betas:
        print(f"Generando muestras para beta = {beta}...")
        for _ in range(cantidad_muestras // len(betas)):
            muestra = propp_wilson(dimension, beta)
            muestras_por_beta[beta].append(muestra)
    return muestras_por_beta

# Función para calcular la magnetización promedio para cada beta
def calcular_magnetizacion_por_beta(muestras_por_beta):
    magnetizaciones_por_beta = {}
    for beta, muestras in muestras_por_beta.items():
        magnetizaciones = [np.mean(muestra) for muestra in muestras]
        magnetizaciones_por_beta[beta] = np.mean(magnetizaciones)
    return magnetizaciones_por_beta

# Parámetros
dim = 10 # Dimensión de la cuadrícula
betas = np.linspace(0.0, 1.0, 10) # Valores de beta
cantidad_muestras = 100 # Total de muestras deseadas

# Generar las muestras agrupadas por beta

```

```
muestras_por_beta = obtener_muestras_por_beta(dim, betas, cantidad_muestras)

# Calcular la magnetización promedio para cada beta
magnetizaciones_por_beta = calcular_magnetizacion_por_beta(muestras_por_beta)
```

```
[-1, -1, 1, -1, -1, -1, -1, -1, -1, -1],
```

Generemos muestras

```
1 1' 1' 1' 1' 1' 1' 1' 1' 1'
```

```
for idx, muestra in enumerate(MuestrasPW):
    print(f"Muestra {idx + 1}:")
    print(muestra)
```



Muestra 1:

```
[[ 1. -1. 1. -1. 1. -1. -1. -1. -1. -1.]
 [ 1. -1. -1. -1. -1. 1. -1. 1. 1. -1.]
 [-1. 1. 1. -1. -1. 1. 1. 1. -1. 1.]
 [ 1. -1. 1. -1. 1. -1. 1. -1. -1. -1.]
 [-1. 1. -1. -1. -1. 1. -1. -1. -1. 1.]
 [ 1. -1. -1. 1. -1. 1. 1. -1. -1. 1.]
 [-1. -1. -1. 1. 1. 1. 1. 1. 1. -1.]
 [ 1. -1. 1. 1. -1. -1. -1. -1. 1. -1.]
 [ 1. 1. 1. -1. -1. -1. -1. -1. 1. -1.]
 [ 1. 1. -1. -1. -1. -1. 1. 1. -1. -1.]]
```

Muestra 2:

```
[[ 1. -1. -1. 1. 1. 1. 1. 1. 1. -1.]
 [ 1. 1. 1. 1. -1. 1. -1. -1. 1. 1.]
 [-1. 1. -1. -1. -1. -1. -1. -1. 1. -1.]
 [-1. 1. -1. 1. -1. -1. -1. 1. 1. 1.]
 [-1. -1. 1. 1. -1. 1. 1. 1. -1. -1.]
 [ 1. -1. 1. -1. 1. -1. -1. 1. 1. -1.]
 [-1. -1. 1. 1. 1. 1. -1. 1. -1. -1.]
 [ 1. -1. -1. 1. -1. 1. 1. -1. 1. 1.]
 [ 1. -1. 1. 1. 1. 1. -1. 1. -1. 1.]
 [ 1. 1. -1. 1. 1. 1. 1. -1. -1. -1.]]
```

Muestra 3:

```
[[ 1. 1. -1. 1. -1. 1. 1. 1. 1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. 1. -1. 1.]
 [ 1. 1. 1. -1. 1. 1. 1. 1. -1. 1.]
 [-1. -1. -1. -1. -1. 1. -1. -1. 1. 1.]
 [-1. -1. 1. 1. -1. 1. -1. 1. 1. -1.]
 [ 1. 1. 1. 1. -1. -1. -1. 1. -1. -1.]
 [ 1. -1. 1. -1. -1. -1. -1. 1. -1. 1.]]
```

```

[-1. -1. -1. -1. -1.  1. -1.  1. -1.  1.]
[ 1. -1. -1. -1.  1.  1.  1.  1. -1. -1.]
[ 1.  1. -1.  1. -1. -1.  1.  1. -1. -1.]]

```

Muestra 4:

```

[[ 1.  1.  1. -1.  1.  1.  1. -1. -1. -1.]
 [-1. -1. -1.  1. -1. -1.  1.  1.  1.  1.]
 [ 1. -1. -1.  1. -1.  1. -1.  1.  1.  1.]
 [ 1. -1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [-1. -1. -1. -1. -1. -1.  1.  1. -1.  1.]
 [-1.  1. -1. -1.  1. -1.  1.  1.  1.  1.]
 [ 1.  1. -1. -1. -1.  1.  1.  1. -1.  1.]
 [ 1. -1. -1. -1.  1. -1. -1. -1. -1.  1.]
 [-1.  1.  1.  1.  1.  1. -1.  1.  1. -1.]
 [ 1. -1. -1.  1.  1.  1.  1.  1. -1.  1.]]

```

Muestra 5:

```

[[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1.  1.  1.  1.  1.  1. -1.  1. -1.  1.]
 [ 1.  1.  1.  1.  1.  1. -1. -1. -1.  1.]
 [ 1.  1. -1.  1.  1.  1.  1. -1. -1. -1.]
 [ 1.  1.  1. -1. -1. -1. -1.  1.  1. -1.]
 [-1. -1.  1. -1.  1. -1.  1. -1. -1.  1.]
 [ 1. -1. -1.  1. -1.  1.  1. -1. -1.  1.]
 [-1.  1.  1. -1. -1. -1.  1. -1. -1. -1.]
 [-1. -1.  1. -1.  1.  1. -1. -1.  1.  1.]
 [-1. -1.  1. -1.  1.  1.  1. -1.  1.  1.]]

```

Muestra 6:

```

[[-1.  1.  1.  1. -1. -1.  1. -1. -1.  1.]
 [-1. -1. -1.  1. -1. -1.  1. -1. -1.  1.]
 [-1.  1. -1. -1.  1.  1.  1. -1.  1.  1.]
 [ 1. -1. -1. -1.  1.  1.  1. -1.  1. -1.]]

```

✓ c) Estimación y reportes

```

[ 1, -1,  1,  1, -1,  1,  1, -1, -1, -1],
[-1, -1,  1, -1, -1,  1,  1,  1, -1, -1],
[-1,  1,  1,  1, -1,  1,  1, -1,  1, -1],
[-1,  1, -1,  1, -1, -1,  1, -1,  1,  1]], array([[ -1, -1, -1,  1,  1,
[ 1, -1, -1, -1, -1, -1, -1,  1, -1, -1],
[-1, -1, -1, -1,  1,  1, -1, -1, -1, -1],
[ 1, -1, -1, -1,  1,  1,  1,  1,  1,  1],
[ 1,  1,  1,  1,  1,  1,  1, -1,  1,  1],
[ 1,  1,  1, -1,  1,  1,  1,  1,  1,  1],
[-1, -1, -1,  1,  1,  1,  1, -1, -1, -1],
[-1, -1,  1,  1,  1,  1,  1,  1, -1, -1],
[-1,  1,  1,  1, -1,  1,  1, -1, -1, -1],
[ 1,  1, -1,  1, -1,  1,  1,  1,  1,  1]]), array([[ -1, -1, -1,  1,  1,
[ 1, -1,  1,  1,  1,  1, -1,  1, -1,  1],
[-1, -1, -1,  1,  1,  1,  1, -1, -1, -1],
[ 1,  1, -1,  1,  1, -1,  1,  1, -1,  1],
[ 1,  1,  1,  1, -1,  1,  1, -1,  1,  1]]

```

```
def estimar_magnetismo(muestras):
    magnetismos = []
    for muestra in muestras:
        magnetismo = np.mean(muestra)
        magnetismos.append(magnetismo)
    return np.array(magnetismos)

# Estimar el magnetismo para las muestras generadas por MCMC
magnetismos_mcmc = estimar_magnetismo(muestrasMC1)

print("Magnetismos estimados para las muestras generadas por MCMC:")
print(magnetismos_mcmc)
```

```
⇒ Magnetismos estimados para las muestras generadas por MCMC:
[-0.1  0.1 -0.1 ...  1.   1.   1. ]
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
```

```
def estimar_magnetismo_propp_wilson(muestras):
    magnetismos_propp_wilson = []
    for muestra in muestras:
        magnetismo = np.mean(muestra)
        magnetismos_propp_wilson.append(magnetismo)
    return np.array(magnetismos_propp_wilson)

# Estimar el magnetismo para las muestras generadas por Propp-Wilson
magnetismos_propp_wilson = estimar_magnetismo_propp_wilson(muestrasPW)

print("Magnetismos estimados para las muestras generadas por Propp-Wilson:")
print(magnetismos_propp_wilson)
```

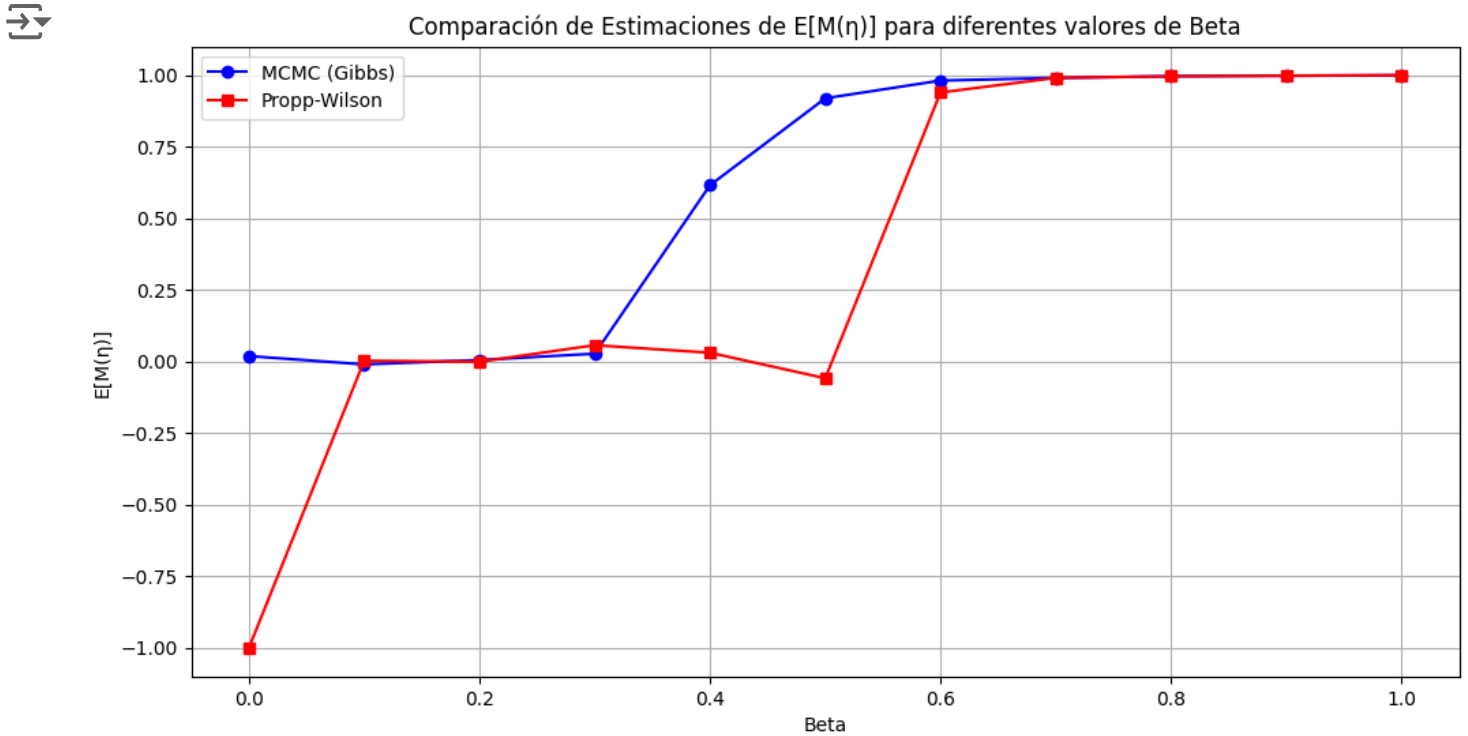
```
⇒ Magnetismos estimados para las muestras generadas por Propp-Wilson:
[-1. -1. -1. ...  1.   1.   1.]
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

Crear la gráfica con el diseño solicitado

```
def graficar_comparacion(muestras1, muestras2, temp_inversas):
    df = pd.DataFrame({'MCMC': muestras1, 'Propp-Wilson': muestras2, 'Beta': temp_inversas})
    plt.figure(figsize=(12, 6))
    plt.plot(df['Beta'], df['MCMC'], 'o-', label='MCMC (Gibbs)', color='blue')
    plt.plot(df['Beta'], df['Propp-Wilson'], 's-', label='Propp-Wilson', color='red')
    plt.xlabel('Beta')
    plt.ylabel('E[M(η)]')
    plt.title('Comparación de Estimaciones de E[M(η)] para diferentes valores de β')
    plt.legend()
    plt.grid(True)
```

```
plt.show()
```

```
# Graficar la comparación
graficar_comparacion(muestrasMC1, muestrasPW, temp_inversas)
```



```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
```

Como mencionamos en el inciso a) tomamos un valor de iteraciones en 10000 ,con respecto al tiempo de coalecencia , se llega al limite , por ende su tiempo de coalecencia en este paso es 10000 , es decir , en mas pasos se llegaría a un valor exacto

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]], array([[ -1, -1, -1, -1, -1,
```

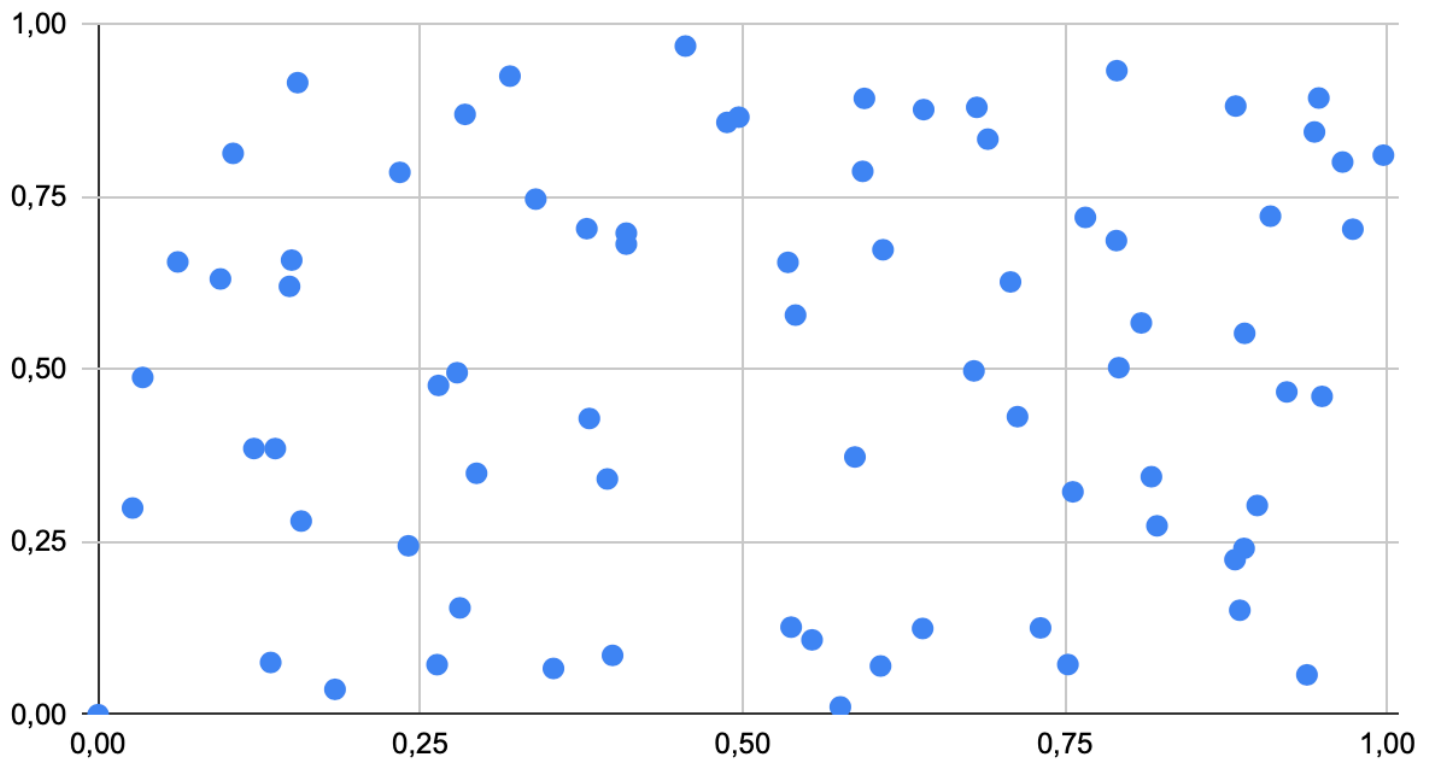
✓ Ejercicio 2

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
```

Una hormiga ha sido desalojada su colonia ubicada en el punto (0,0) de la parcela $[0, 1] \times [0, 1]$. Decide entonces la hormiga **VISITAR** todas la otras 75 colonias de su parcela **SIN REPETIR**

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1], array([[ -1, -1, -1, -1, -1,
```

Puntos por visitar partiendo del (0,0)



```
[-1, 1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, 1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, 1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, 1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, 1, -1, -1, -1, -1]], array([[ -1, -1, -1, -1, -1,
```

a) Use "simulated annealing" para ayudarle a la hormiga a encontrar el camino mas corto que recorra todas las parcelas.

Reporte -Esquema de enfriamiento usado -Distancia mínima obtenida -Mapa generado para la hormiga

b) Repita el item a) si se sabe que la hormiga retornará a su colonia original , despues de haber recorrido todas las otras colonias.

```
[ -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
```

✓ Solución

```
[ -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
```

Importar base de datos

```
[ -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
```


```
%config IPCompleter.greedy=True
import pandas as pd
import numpy as np
import xlrd
import seaborn as sb
import matplotlib.pyplot as plt
from matplotlib.ticker import PercentFormatter

[ -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]

from google.colab import files

[ -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]

from google.colab import drive
drive.mount('/content/drive')
```

 Mounted at /content/drive

```
[ -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import math

# Función para calcular la distancia euclidiana entre dos puntos
def distancia(p1, p2):
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
```

```
# Función para calcular la distancia total del recorrido
def calcular_distancia_total(ruta, colonias):
    distancia_total = 0
    for i in range(len(ruta) - 1):
        distancia_total += distancia(colonias[ruta[i]], colonias[ruta[i+1]])
    # Volver al inicio para cerrar el ciclo
    distancia_total += distancia(colonias[ruta[-1]], colonias[ruta[0]])
    return distancia_total

# Función de enfriamiento
def esquema_enfriamiento(temp, alpha):
    return temp * alpha

# Simulated Annealing
def simulated_annealing(colonias, temperatura_inicial, alpha, iteraciones_max):
    # Inicializar una ruta aleatoria
    n = len(colonias)
    ruta_actual = list(range(n))
    random.shuffle(ruta_actual)

    # Calcular la distancia de la ruta inicial
    mejor_ruta = list(ruta_actual)
    mejor_distancia = calcular_distancia_total(mejor_ruta, colonias)

    ruta = list(ruta_actual)
    distancia_actual = mejor_distancia

    temperatura = temperatura_inicial

    for i in range(iteraciones_max):
        # Generar una nueva ruta (perturbación) intercambiando dos colonias
        nuevo_ruta = list(ruta)
        idx1, idx2 = random.sample(range(n), 2)
        nuevo_ruta[idx1], nuevo_ruta[idx2] = nuevo_ruta[idx2], nuevo_ruta[idx1]

        # Calcular la nueva distancia
        nueva_distancia = calcular_distancia_total(nuevo_ruta, colonias)

        # Aceptar la nueva solución con una probabilidad dependiente de la temperatura
        if nueva_distancia < distancia_actual or random.random() < math.exp(-(nueva_distancia - distancia_actual) / temperatura):
            ruta = nuevo_ruta
            distancia_actual = nueva_distancia

        # Actualizar la mejor ruta encontrada
```



```

        if nueva_distancia < mejor_distancia:
            mejor_ruta = nuevo_ruta
            mejor_distancia = nueva_distancia

    # Enfriar el sistema
    temperatura = esquema_enfriamiento(temperatura, alpha)

    return mejor_ruta, mejor_distancia

# Datos del problema (las coordenadas de las colonias)
df = pd.read_excel('/content/drive/MyDrive/6 semestre/Cadenas de Markov/Problema 1')
df = df[:-6]
df = df[['Coordenada X', 'Coordenada Y']]
colonias = df[['Coordenada X', 'Coordenada Y']].values

# Parámetros del algoritmo
temperatura_inicial = 1000
alpha = 0.99
iteraciones_max = 10000

# Ejecutar Simulated Annealing
mejor_ruta, mejor_distancia = simulated_annealing(colonias, temperatura_inicial, iteraciones_max)

# Reportar los resultados
print(f"Esquema de enfriamiento: geométrico,  $T(k+1) = \{\alpha\} * T(k)$ ")
print(f"Distancia mínima obtenida: {mejor_distancia}")

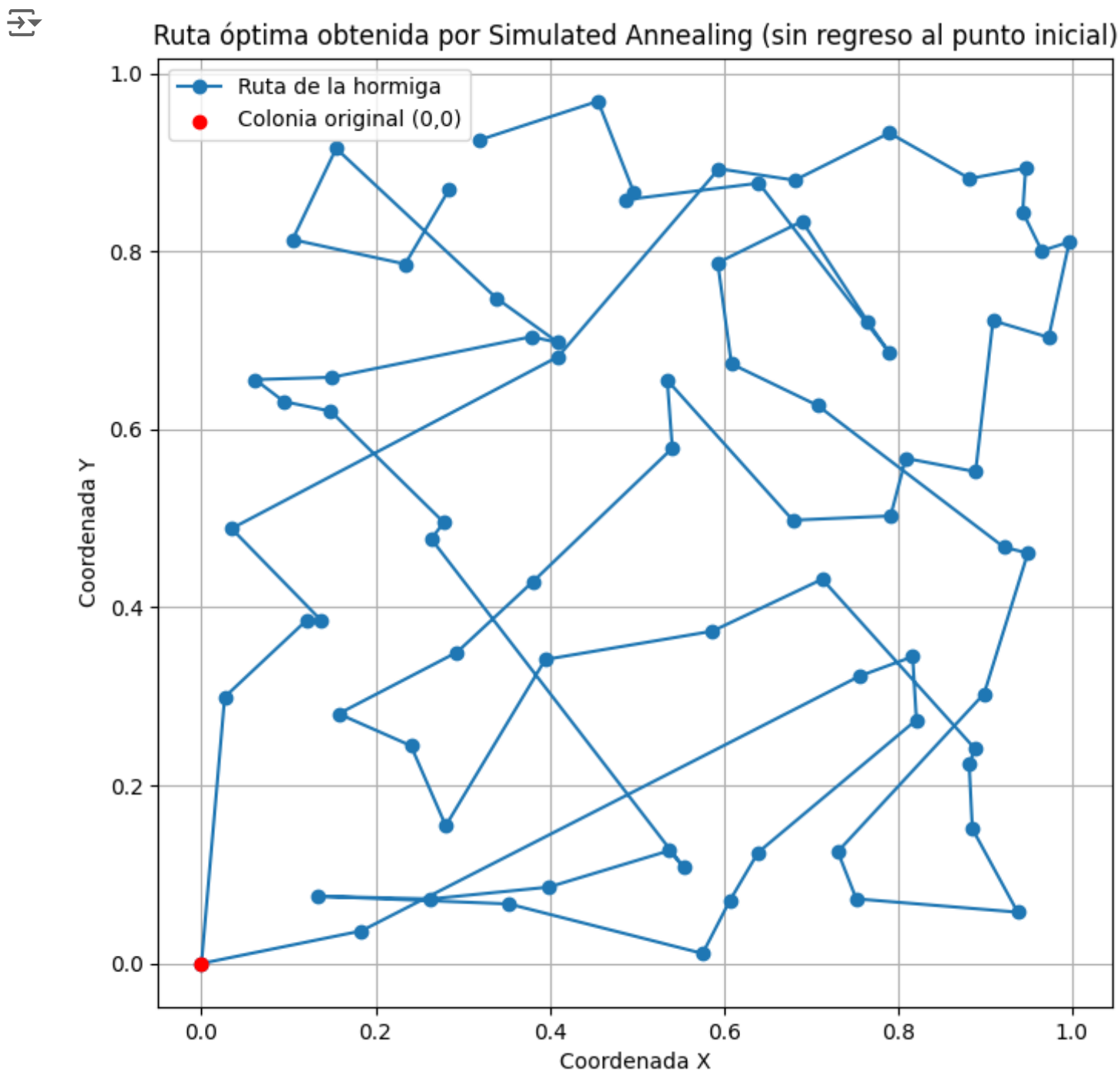
# Graficar la ruta obtenida
ruta_coords = [colonias[i] for i in mejor_ruta] + [colonias[mejor_ruta[0]]] # Para cerrar la ruta

# Esquema de enfriamiento: geométrico,  $T(k+1) = 0.99 * T(k)$ 
# Distancia mínima obtenida: 10.589167105576934
# Ruta obtenida sin volver al punto inicial
# Ruta sin cerrar
# Ruta sin cerrar

plt.figure(figsize=(8,8))
plt.plot(ruta_coords[:,0], ruta_coords[:,1], 'o-', label='Ruta de la hormiga')
plt.scatter(colonias[0,0], colonias[0,1], color='red', label='Colonia original (0,0)')
plt.title('Ruta óptima obtenida por Simulated Annealing (sin regreso al punto inicial)')
plt.xlabel('Coordenada X')
plt.ylabel('Coordenada Y')

```

```
plt.legend()
plt.grid(True)
plt.show()
```



```
[ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
[ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
[ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
```

Página 19 de 34

[illegible]

```
x_points = df['Coordenada X'].values # Example values
y_points = df['Coordenada Y'].values # Example values
```

```
final_neighbor = simulated_annealing_1(x_points, y_points)
print(final_neighbor)
```

```
final_distance = dist_path(final_neighbor["x"], final_neighbor["y"])
print("Final distance:", final_distance)
```

```
➦ {'x': array([0.          , 0.6401948 , 0.59425005, 0.49683601, 0.48770178,
0.45548882, 0.31936546, 0.28455705, 0.15470798, 0.10465497,
0.233981  , 0.33937816, 0.37896249, 0.40960775, 0.40960048,
0.53497468, 0.54084931, 0.6087055 , 0.59295341, 0.68997466,
0.6814314 , 0.79002582, 0.88221372, 0.94665169, 0.94329939,
0.96512103, 0.99677415, 0.97306404, 0.90914764, 0.76563671,
0.78967572, 0.70756197, 0.67921518, 0.80899694, 0.88914983,
0.94916593, 0.92180953, 0.79152632, 0.71299769, 0.58691362,
0.75594301, 0.81683198, 0.82119398, 0.89893308, 0.88871026,
0.881702  , 0.88541366, 0.93749301, 0.75209917, 0.730863  ,
0.63944283, 0.60678731, 0.53745222, 0.55366082, 0.57559441,
0.39895641, 0.35312074, 0.2628839 , 0.18370392, 0.13381271,
0.28056239, 0.24061812, 0.29345961, 0.39489794, 0.38092605,
0.26393333, 0.27838436, 0.14840974, 0.15007507, 0.09483427,
0.06174778, 0.03456659, 0.02671012, 0.12083757, 0.13735007,
0.15746602]), 'y': array([0.          , 0.87661905, 0.89276924, 0.8656721,
0.96868071, 0.92512318, 0.86969907, 0.91559441, 0.81320169,
0.78570425, 0.74692915, 0.70406391, 0.69756476, 0.6816464 ,
0.65525392, 0.57887761, 0.67359036, 0.78717681, 0.83375936,
0.87989073, 0.93297492, 0.88174926, 0.89338654, 0.8441392 ,
0.80058373, 0.81054482, 0.70335745, 0.72217  , 0.720404  ,
0.68678346, 0.62696419, 0.49801298, 0.56754334, 0.55241446,
0.46107779, 0.46759669, 0.50275123, 0.43170503, 0.37335144,
0.32287792, 0.34472775, 0.27363515, 0.30308251, 0.24091014,
0.22464644, 0.15131931, 0.05765701, 0.07265281, 0.12570163,
0.12486716, 0.07049558, 0.12683342, 0.10833723, 0.01126639,
0.08585402, 0.06692329, 0.07249196, 0.03665026, 0.07559025,
0.15472404, 0.24462974, 0.3496503 , 0.34148606, 0.42914589,
0.47695138, 0.49545332, 0.62044625, 0.65853008, 0.63122856,
0.65591373, 0.48858693, 0.29935022, 0.38562882, 0.38555454,
0.2805334 ])}
Final distance: 7.931671528041594
r 1 1 1 1 1 1 1 1 1 1 1 1
```

```
import matplotlib.pyplot as plt
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
import matplotlib.image as mpimg # Para cargar la imagen
```

```
# Cargar la imagen de la hormiguita
ant_image = mpimg.imread('/content/drive/MyDrive/6 semestre/Cadenas de Markov/png·

# Definir el tamaño de la figura
plt.figure(figsize=(10, 10))

# Graficar los puntos originales (x_points, y_points) usando scatter
plt.scatter(x_points, y_points, color='blue', label='Puntos Originales')

# Graficar la ruta optimizada (final_neighbor["x"], final_neighbor["y"])
plt.plot(final_neighbor["x"], final_neighbor["y"], color='red', label='Ruta Optim

# Elegir un punto donde quieras colocar la hormiga (ejemplo: el primer nodo)
x_ant = final_neighbor["x"][0] # Coordenada x del nodo donde pondrás la hormiga
y_ant = final_neighbor["y"][0] # Coordenada y del nodo donde pondrás la hormiga

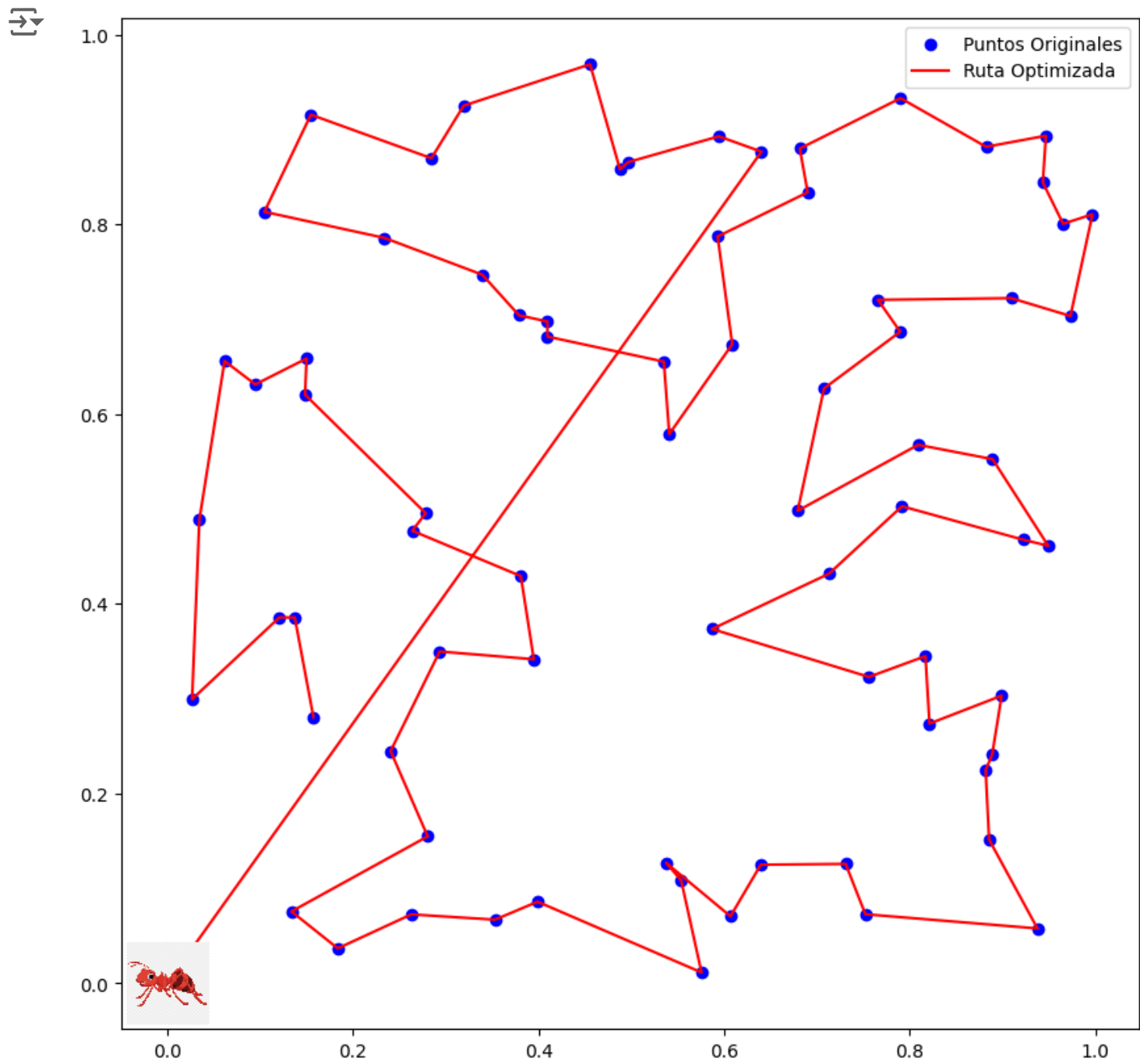
# Crear un OffsetImage para la hormiga
imagebox = OffsetImage(ant_image, zoom=0.05) # Ajusta el tamaño de la imagen con

# Crear el AnnotationBbox para colocar la imagen en el gráfico
ab = AnnotationBbox(imagebox, (x_ant, y_ant), frameon=False)

# Añadir la imagen al gráfico
plt.gca().add_artist(ab)

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```



```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
```

```
import matplotlib.pyplot as plt

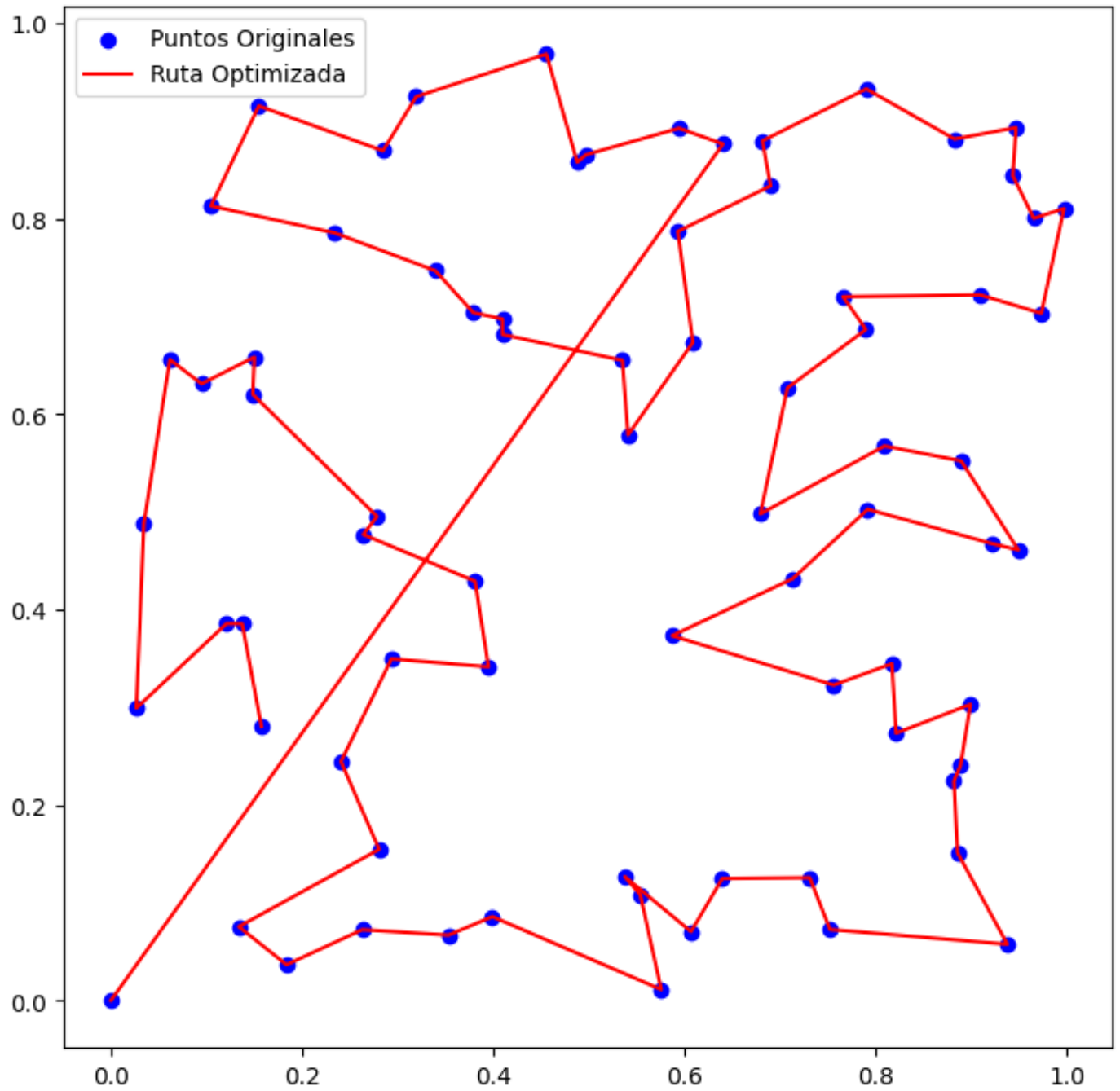
# Definir el tamaño de la figura
plt.figure(figsize=(8, 8))

# Graficar los puntos originales (x_points, y_points) usando scatter
plt.scatter(x_points, y_points, color='blue', label='Puntos Originales')

# Graficar la ruta optimizada (final_neighbor["x"], final_neighbor["y"])
plt.plot(final_neighbor["x"], final_neighbor["y"], color='red', label='Ruta Optim

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```

```
import math
import random
import numpy as np
import matplotlib.pyplot as plt
```

```
import math

def calcular_distancia_ruta(coordenadas_x, coordenadas_y):

    distancia_total = 0.0
    numero_puntos = len(coordenadas_x)

    for i in range(numero_puntos - 1):
        distancia_total += math.sqrt((coordenadas_y[i + 1] - coordenadas_y[i])**2)

    return distancia_total


import random
import numpy as np

def nueva_ruta(coordenadas_x, coordenadas_y):

    # Selecciona aleatoriamente dos índices i y j
    i = random.randint(2, 76)
    j = random.randint(i, 76)

    # Crea copias de las coordenadas originales
    nuevo_x = np.copy(coordenadas_x)
    nuevo_y = np.copy(coordenadas_y)

    # Invierte el orden de los puntos entre los índices i y j
    nuevo_x[i:j] = nuevo_x[i:j][::-1]
    nuevo_y[i:j] = nuevo_y[i:j][::-1]

    return nuevo_x, nuevo_y
```

```

def simulated_annealing(coordenadas_x, coordenadas_y):

    # Inicializa las coordenadas actuales
    puntos_actuales = {"x": np.copy(coordenadas_x), "y": np.copy(coordenadas_y)}

    # Inicialización de la temperatura y el ciclo de enfriamiento
    iteracion = 1
    while iteracion < 7:
        for k in range(1, 1000000):
            temperatura = 1 / (10 * iteracion) # Esquema de enfriamiento

            # Ruta a seguir
            nuevo_x, nuevo_y = nueva_ruta(puntos_actuales["x"], puntos_actuales["y"])

            costo_ruta_actual = calcular_distancia_ruta(puntos_actuales["x"], puntos_actuales["y"])
            costo_ruta_vecina = calcular_distancia_ruta(nuevo_x, nuevo_y)

            aleatorio = random.random()

            if aleatorio < min(1, math.exp((costo_ruta_actual - costo_ruta_vecina) / temperatura)):
                puntos_actuales = {"x": np.copy(nuevo_x), "y": np.copy(nuevo_y)}

        iteracion += 1

    return puntos_actuales

#Cargamos las coordenadas

df = pd.read_excel('/content/drive/MyDrive/6 semestre/Cadenas de Markov/Problema 1.xlsx')
df = df[:-6]
df = df[['Coordenada X', 'Coordenada Y']]

coordenadas_x = df['Coordenada X'].values
coordenadas_y = df['Coordenada Y'].values

# Ejecución del algoritmo del simulated annealing
ruta_final = simulated_annealing(coordenadas_x, coordenadas_y)

```

```
print("Ruta Final Recorrida por la hormiga:", ruta_final)
```

```
⇒ Ruta Final Recorrida por la hormiga: {'x': array([0.          , 0.6401948 , 0.59048882, 0.31936546, 0.28455705, 0.15470798, 0.10465497,
0.233981 , 0.33937816, 0.37896249, 0.40960775, 0.40960048,
0.54084931, 0.53497468, 0.6087055 , 0.59295341, 0.68997466,
0.6814314 , 0.79002582, 0.88221372, 0.94665169, 0.94329939,
0.96512103, 0.99677415, 0.97306404, 0.90914764, 0.78967572,
0.76563671, 0.70756197, 0.80899694, 0.88914983, 0.94916593,
0.92180953, 0.79152632, 0.67921518, 0.71299769, 0.58691362,
0.75594301, 0.81683198, 0.82119398, 0.881702 , 0.88871026,
0.89893308, 0.88541366, 0.93749301, 0.75209917, 0.730863 ,
0.63944283, 0.60678731, 0.57559441, 0.55366082, 0.53745222,
0.39895641, 0.35312074, 0.2628839 , 0.18370392, 0.13381271,
0.28056239, 0.24061812, 0.29345961, 0.39489794, 0.38092605,
0.27838436, 0.26393333, 0.15007507, 0.14840974, 0.09483427,
0.06174778, 0.03456659, 0.12083757, 0.13735007, 0.15746602,
0.02671012]), 'y': array([0.          , 0.87661905, 0.89276924, 0.8580071,
0.96868071, 0.92512318, 0.86969907, 0.91559441, 0.81320169,
0.78570425, 0.74692915, 0.70406391, 0.69756476, 0.6816464 ,
0.57887761, 0.65525392, 0.67359036, 0.78717681, 0.83375936,
0.87989073, 0.93297492, 0.88174926, 0.89338654, 0.8441392 ,
0.80058373, 0.81054482, 0.70335745, 0.72217 , 0.68678346,
0.720404 , 0.62696419, 0.56754334, 0.55241446, 0.46107779,
0.46759669, 0.50275123, 0.49801298, 0.43170503, 0.37335144,
0.32287792, 0.34472775, 0.27363515, 0.22464644, 0.24091014,
0.30308251, 0.15131931, 0.05765701, 0.07265281, 0.12570163,
0.12486716, 0.07049558, 0.01126639, 0.10833723, 0.12683342,
0.08585402, 0.06692329, 0.07249196, 0.03665026, 0.07559025,
0.15472404, 0.24462974, 0.3496503 , 0.34148606, 0.42914589,
0.49545332, 0.47695138, 0.65853008, 0.62044625, 0.63122856,
0.65591373, 0.48858693, 0.38562882, 0.38555454, 0.2805334 ,
0.29935022])})
```

```
# Cálculo de la distancia final
```

```
distancia_final = calcular_distancia_ruta(ruta_final["x"], ruta_final["y"])
```

```
print("Distancia final:", distancia_final)
```

```
⇒ Distancia final: 7.821196291742906
```

```
plt.figure(figsize=(8, 8))
```

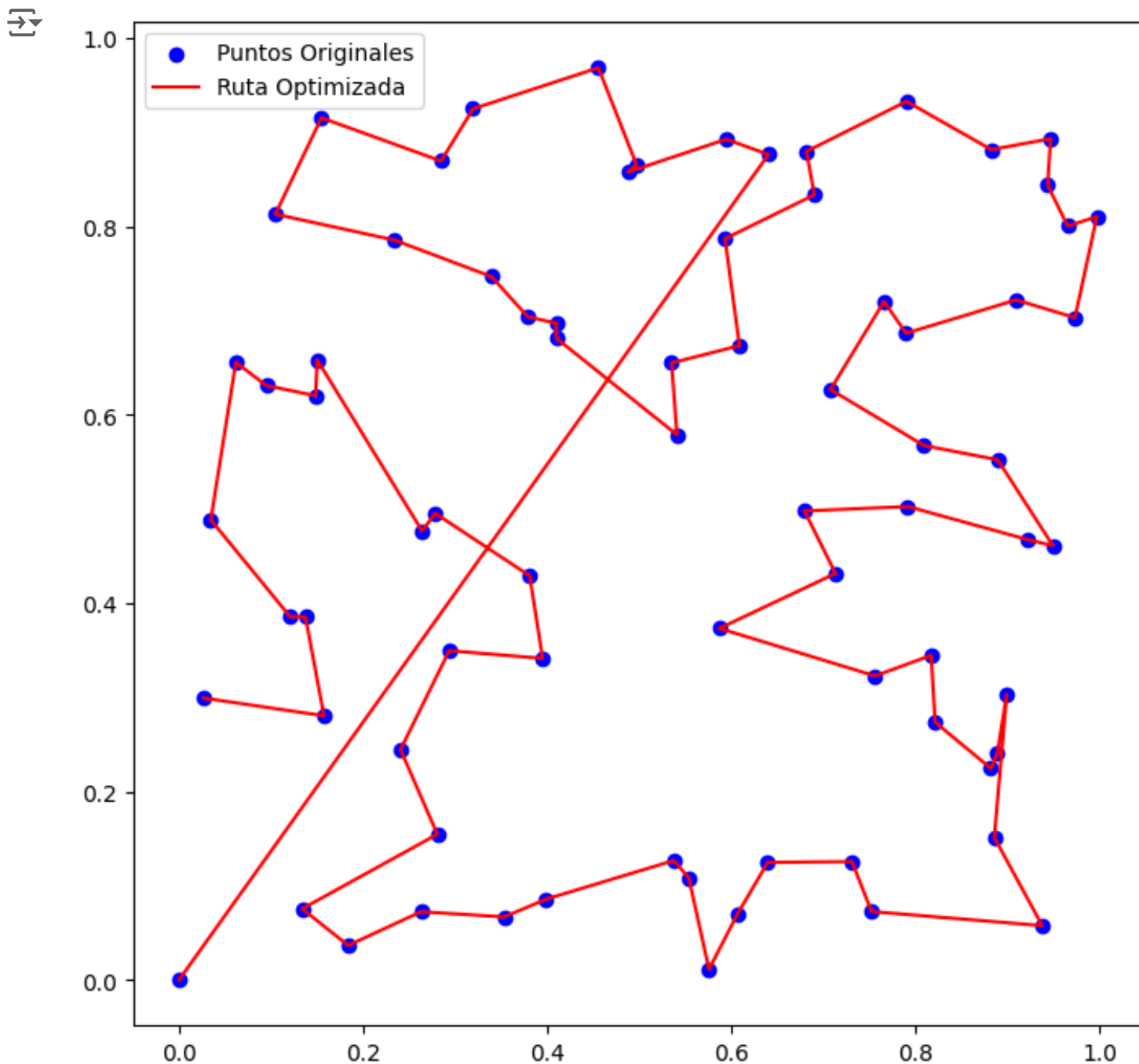
```
# Graficar los puntos originales (x_points, y_points) usando scatter
```

```
plt.scatter(coordenadas_x, coordenadas_y, color='blue', label='Puntos Originales')
```

```
# Graficar la ruta optimizada (final_neighbor["x"], final_neighbor["y"])
plt.plot(ruta_final["x"], ruta_final["y"], color='red', label='Ruta Optimizada')

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```



REPORTE

El esquema de enfriamiento que utilizamos corresponde a una disminución lineal de la temperatura inversamente proporcional al número de iteración n . Es decir:

$$T = \frac{1}{(10 * n)}$$

La temperatura inicial T_0 es $T = \frac{1}{10}$ en la primera iteración $n = 1$.

A medida que n aumenta, la temperatura disminuye de manera proporcional a $\frac{1}{n}$ lo cual implica que la temperatura se va "enfriando" a lo largo del tiempo.

La distancia mínima obtenida fue: 7.876269152202981

Punto B

Para regresar al origen bastará con agregar de nuevo el origen (0,0) a las colonias.

```
coordenadas_x = df['Coordenada X'].values
coordenadas_y = df['Coordenada Y'].values
```

```
coordenadas_x_C = np.copy(coordenadas_x)
coordenadas_y_C = np.copy(coordenadas_y)
```

```
# Se agrega el origen
```

```
coordenadas_x_C = np.append(coordenadas_x_C , 0)
coordenadas_y_C = np.append(coordenadas_y_C , 0)
```

```
ruta_final_con_origen = simulated_annealing(coordenadas_x_C , coordenadas_y_C )
```

```
# Resultado del vecino final optimizado
```

```
print("Distancia final con origen:", ruta_final_con_origen)
```

```
➡ Distancia final con origen: {'x': array([0.          , 0.6401948 , 0.59425005, (
    0.45548882, 0.31936546, 0.28455705, 0.233981 , 0.15470798,
    0.10465497, 0.15007507, 0.14840974, 0.09483427, 0.06174778,
    0.03456659, 0.13735007, 0.12083757, 0.02671012, 0.15746602,
    0.24061812, 0.29345961, 0.39489794, 0.38092605, 0.26393333,
    0.27838436, 0.33937816, 0.40960775, 0.37896249, 0.40960048,
    0.54084931, 0.53497468, 0.6087055 , 0.59295341, 0.68997466,
    0.6814314 , 0.79002582, 0.88221372, 0.94665169, 0.94329939,
    0.96512103, 0.99677415, 0.90914764, 0.97306404, 0.78967572,
    0.76563671, 0.70756197, 0.79152632, 0.92180953, 0.94916593,
    0.88914983, 0.80899694, 0.67921518, 0.71299769, 0.58691362,
    0.75594301, 0.81683198, 0.82119398, 0.89893308, 0.88871026,
    0.881702 , 0.88541366, 0.93749301, 0.75209917, 0.730863 ,
    0.63944283, 0.60678731, 0.57559441, 0.55366082, 0.53745222,
    0.39895641, 0.35312074, 0.28056239, 0.2628839 , 0.18370392,
    0.13381271, 0.          ]), 'y': array([0.          , 0.87661905, 0.8927691,
    0.96868071, 0.92512318, 0.86969907, 0.78570425, 0.91559441,
    0.81320169, 0.65853008, 0.62044625, 0.63122856, 0.65591373,
    0.48858693, 0.38555454, 0.38562882, 0.29935022, 0.2805334 ,
    0.24462974, 0.3496503 , 0.34148606, 0.42914589, 0.47695138,
    0.49545332, 0.74692915, 0.69756476, 0.70406391, 0.6816464 ,
    0.57887761, 0.65525392, 0.67359036, 0.78717681, 0.83375936,
    0.87989073, 0.93297492, 0.88174926, 0.89338654, 0.8441392 ,
    0.80058373, 0.81054482, 0.72217 , 0.70335745, 0.68678346,
    0.720404 , 0.62696419, 0.50275123, 0.46759669, 0.46107779,
    0.55241446, 0.56754334, 0.49801298, 0.43170503, 0.37335144,
    0.32287792, 0.34472775, 0.27363515, 0.30308251, 0.24091014,
    0.22464644, 0.15131931, 0.05765701, 0.07265281, 0.12570163,
    0.12486716, 0.07049558, 0.01126639, 0.10833723, 0.12683342,
    0.08585402, 0.06692329, 0.15472404, 0.07249196, 0.03665026,
    0.07559025, 0.          ]))}
```

```
# Cálculo de la distancia final
```

```
distancia_final = calcular_distancia_ruta(ruta_final_con_origen["x"], ruta_final_
```

```
print("Distancia final:", distancia_final)
```

```
➡ Distancia final: 8.128394122374488
```

```
plt.figure(figsize=(8, 8))
```

```
# Graficar los puntos originales (x_points, y_points) usando scatter
```

```
plt.scatter(coordenadas_x, coordenadas_y, color='blue', label='Puntos Originales')
```

```
# Graficar la ruta optimizada (final_neighbor["x"], final_neighbor["y"])
plt.plot(ruta_final_con_origen["x"], ruta_final_con_origen["y"], color='red', label='Ruta Optimizada')

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```