Aplicaciones de la nanotecnología computacional
Proyecto 9 - Difusión de una sustancia en 2D usando métodos de Montecarlo

Jair Othoniel Domínguez Godínez 1824524, Jesús Eduardo Flores Rodríguez 1837492

15 de noviembre de 2021

Resumen

En este proyecto se simula la difusión de una sustancia en dos dimensiones. Se utiliza una perspectiva de Montecarlo para seleccionar la casilla a la que se va mover, a partir de un criterio que se define como distancia de corte. La aproximación consiste en considerar al sistema como un mapa de píxeles donde la mancha o la sustancia que se difunde es un píxel con valor 1 y donde no se ha difundido tiene valor 0.

Palabras Clave: Método de Montecarlo, difusión, píxel, arreglo

---

1. Objetivo general

Simular la difusión de una sustancia en $2D$ y obtener una imagen representativa del proceso.

2. Objetivo específico

I. Obtener una imagen del estado inicial y final del sistema.

II. Obtener una gráfica del número de iteraciones contra el número de píxeles manchados.

3. Introducción

El problema de la difusión de una sustancia se puede resolver tomando como punto de partida una ecuación diferencial parcial (EDP), esta puede ser en 1 dimensión, 2 o 3 dimensiones y además tener como parámetro el tiempo. Este tipo de ecuaciones pueden llegar a ser muy complejas en el sentido analítico de obtener una solución analítica del sistema. Existe una perspectiva diferente de estudiar el fenómeno y es a partir de las ciencias de la complejidad y la computación. Esta aproximación parte del supuesto de considerar al sistema como una matriz cuyos elementos son unos y ceros. Aunque la aproximación pueda parecer poco aproximada o burda, la realidad es que resulta ser un buen modelo de juguete. También resulta ser la base del modelo de agregación limitada por difusión **DLA** por sus siglas en inglés. Estos modelos tratan el movimiento Browniano por medio de paseos aleatorios. ¿Qué podría ser un paseo aleatorio? La electrodeposición química, como se forman los agregados moleculares como el ADN o las proteínas. La imagen 1 resulta ser un ejemplo de un fractal y también un proceso DLA. El modelo que se trata en este proyecto es una base mas simple de este proceso.

4. Marco Teórico

4.1. Modelo de difusión 2D

Consideremos una malla de dimensión N como se muestra en la figura 2a. La mancha roja se puede mover hacia uno de los vecinos verdes, esta elección es de forma aleatoria y es aquí donde se introduce el elemento

indeterminista. Así el elemento rojo con coordenadas (i, j) (subíndices de los elementos en la matriz 2) se puede mover hacia la las posiciones

$$\begin{gathered}(i+1, j) \\ (i+1, j+1) \\ (i, j+1) \\ (i+1, j-1) \\ (i, j-1) \\ (i-1, j-1) \\ (i-1, j) \\ (i-1, j+1)\end{gathered} \tag{1}$$

(las coordenadas correspondientemente de los cuadros verdes) vemos que el movimiento de la mancha está limitado a pasos unitarios, los índices de la posición de los ceros y los unos toman valores positivos y enteros $\{0, 1, 2, \cdots\}$ en las direcciones i y j. Una aproximación conveniente es considerar que la malla puede ser representada por una matriz de la forma

$$A = (a_{i,j})_{N\times N} = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \vdots & 0 & 1 & 0 & \vdots \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix}_{N\times N} \tag{2}$$

donde el cero corresponde a un cuadro no pintado, el 1 a un cuadro pintado, si se piensa que lo que está dentro es una gota de pintura o alguna sustancia que se difunde. Las coordenadas (i, j) corresponden a la posición del cuadro rojo en la matriz. Otras consideraciones interesantes es que la matriz computacionalmente representa un arreglo en Python y se puede interpolar el valor de cero a un píxel negro y 1 a un píxel blanco o cualquier otro par de colores. Las coordenadas de los elementos forman una lista de puntos con los cuales vamos a calcular distancias euclidianas con la ecuación

$$r_{P_0,P_1} = \sqrt{(i_0 - i_1)^2 + (j_0 - j_1)^2} \tag{3}$$

donde r_{P_0,P_1} denota la distancia entre lo dos puntos inicial P_0 y posterior P_1.

4.2. Algoritmo

A continuación se detalla de forma general el algoritmo del modelo de difusión.

I. Se inicializa el sistema dando como parámetros fijos el valor de la longitud del arreglo N (en términos matemáticos el número de filas y columnas de la matriz 2) y la distancia de corte D_T. Crear el arreglo lleno de ceros y tomar el elemento central que tendrá coordenadas

$$P_0 = (\frac{N-1}{2}, \frac{N-1}{2}) \tag{4}$$

este es nuestro valor semilla del sistema, del cual va a partir la caminata del elemento rojo.

II. Elegir una celda central de mallado, valor semilla.

III. Se debe ubicar a los ocho vecinos que rodean al cuadro rojo y que tienen las coordenadas que se indican en las expresiones 1.

IV. Se elige alguno de forma aleatoria y se mide la distancia r_{P_0,P_1}.

V. Se compara esta distancia con el valor D_T y si se cumple $r_{P_0,P_1} < D_T$ se cambia el cero del arreglo por un 1, si no se cumple se deja el cero y se vuelve a inspeccionar de nuevo uno de los ocho vecinos.

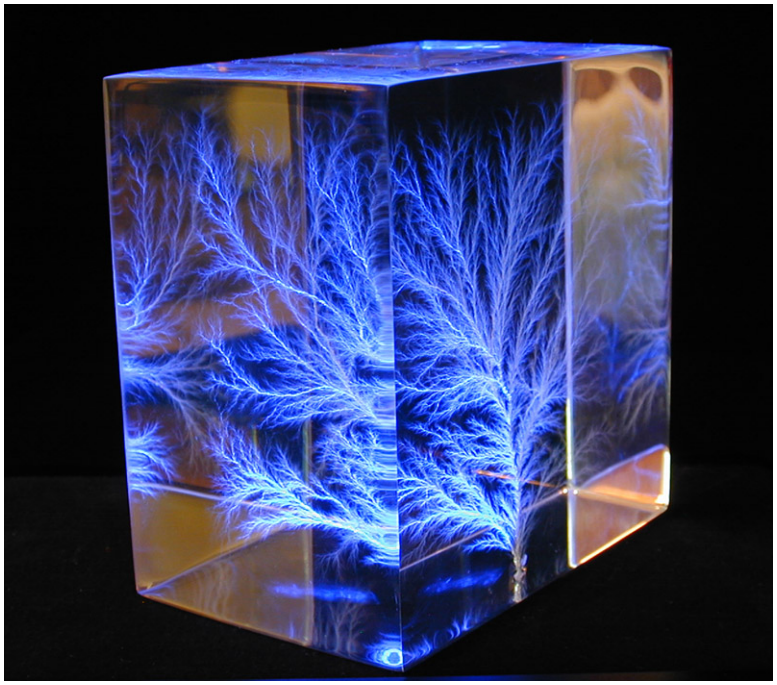
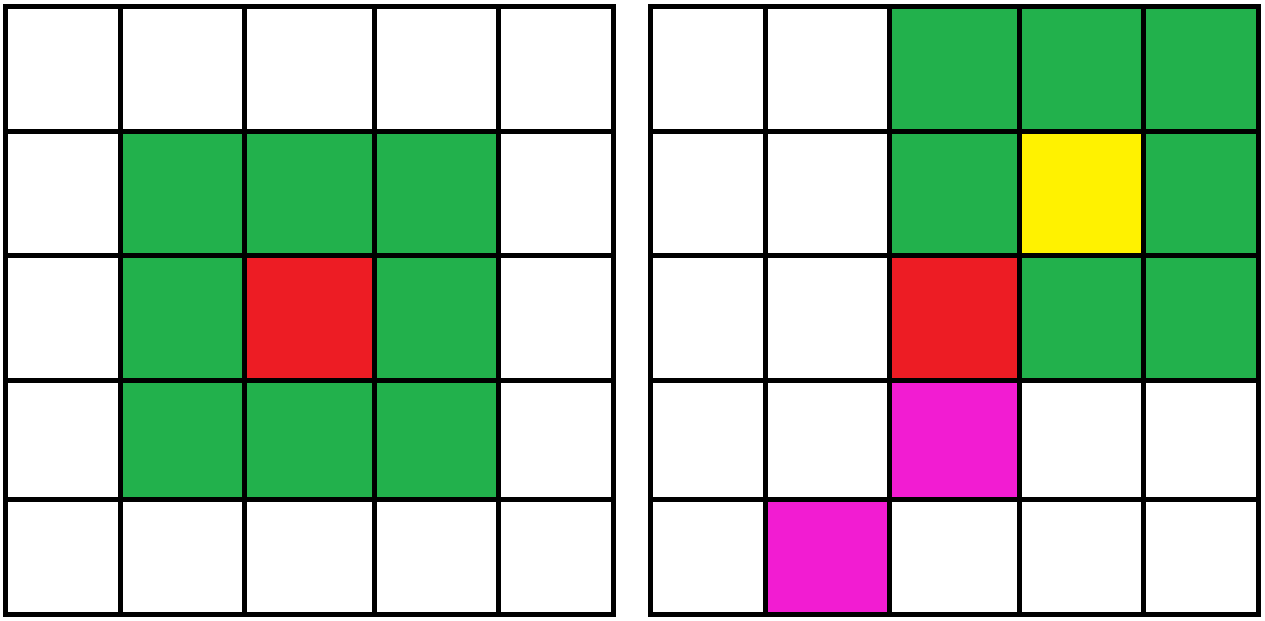


Figura 1: Figura de Litchenberg



(a) **Estado inicial del sistema** (b) **Evolución del sistema**

Figura 2: El elemento rojo representa la gota semilla del sistema que se va a difundir hacia uno de los posibles elementos verdes. Los magentas representan los que han cumplido el criterio en el paso V. El amarillo representa el último que ha cumplido el criterio y le rodean ocho de sus posibles elementos hacia los cuales podría hacer su caminata.

4.3. Código

El código se hizo en Python, con el intérprete de Spyder. A continuación se describen de forma general general las partes el código. Para hacer el GIF de la simulación son necesarias las librerías para animar los cuadros que se producen:

- **from copy import copy**
- **from matplotlib.animation import PillowWriter**.

Para hacer la regresión lineal de los datos y calcular la pendiente de la recta usamos la libreria **from sklearn.linear_model import LinearRegression**. Al final en lugar de usar el comando para graficar bits usamos el comando graficar un mapa de calor donde los valores solo son cero y uno. Comenzamos por exportar las librerías necesarias, damos formato a las gráficas e inicializamos el entorno de la regresión lineal. Todos estos comandos se muestran el la figura 3. Inicializamos el sistema, es decir, declaramos la matriz

```
import matplotlib.pyplot as plt
import numpy as np
import random
from matplotlib import animation
from copy import copy
from matplotlib.animation import PillowWriter
from sklearn.linear_model import LinearRegression

regresion_lineal = LinearRegression() # creamos una instancia de la funcion



#Damos formato de estilo a las gráficas
plt.style.use(['seaborn'])
font = {'family': 'serif',
        'color':  'black',
        'weight': 'normal',
        'size': 16,
        }
```

Figura 3: Primera parte del código

de dimensión $N \times N$ dando el valor N, declaramos la semilla, inicializamos en cero el número de pasos. Declaramos listas donde vamos a guardar las matrices de evolución del sistema, y las coordenadas de los valores que son uno y también los valores del radio. En la siguiente figura 4 se muestra la segunda parte del código. Comenzamos la inspección de los vecinos, si se cumple el criterio se guarda el valor del radio, la matriz

```
#Inicialización del sistema
N =200 #Numero de elementos, debe ser impar
pasos = 1000 #Numero de iteraciones
sistema = np.zeros((N,N),dtype=int) #Matriz inicial
num_ite = []
num_cuadros = []
difusion = []#Almacena las posiciones
config = []
semilla = int((N-1)/2) #Numero que nos asegura posicionarnos en el centro del arreglo
sistema[semilla,semilla] = 1 #Colocamos la semilla
difusion.append((semilla,semilla))

#Indices sobre los que corre la posicion de los pixeles
nx=0
ny=0
k=0
r_list = [] #Distancias que si cumplen el valor del criterio
i=0 #Detener la iteracion
step=0

#Graficamos el estado inicial
plt.figure(figsize = (6,6))
plt.contourf(sistema, cmap = 'plasma')
```

Figura 4: Inicialización del sistema

y las coordenadas. En la figura 5 se muestra de manera explicita el ciclo while que usamos. Recordemos que se elige uno de los ocho vecinos al azar. En la figura 6 se muestra que hay que extraer la información de los arreglos para graficar el estado final. También en esta parte se calcula la regresión lineal de los datos, en este caso la información que corresponde al valor x son el número de cuadros por cada paso temporal que en este caso es el dato y. De esta forma la regresión lineal aprende y optimiza la mejor recta de la forma $y = mx + b$. Finalmente creamos el GIF con los siguientes comandos que se muestran en la figura 7.

```
while step <= pasos:
    i, j = random.choice(difusion) #Elegimos un pixel con valor 1
    nx = i + random.randint(-1,1) #Nos podemos mover a la izquierda o a la derecha de forma aleato
    ny = j + random.randint(-1,1) #Nos podemos mover arriba o abajo de forma aleatoria
    while nx < 0 or nx> N-1 or ny<0 or ny>N-1: #Nos aseguramos de que siempre tengamos 8 vecinos
        nx = i + random.randint(-1,1)
        ny = j + random.randint(-1,1)

    pi,pj = difusion[-1] #Extraemos el ultimo elemento de la lista
    r = np.sqrt((pi-nx)**2 + (pj-ny)**2) #Calculamos la distancia con los indices de la posicion
    criterio = 2.5 #Distancia de corte
    if sistema[nx,ny] == 0 and r<criterio: #Si tenemos un cero y además se cumple la condición
        sistema[nx,ny]=1                   #cambiamos el 0 por un 1
        r_list.append(r)                   #Guardamos el valor del radio
        config.append(copy(sistema))
        difusion.append((nx,ny)) #Guardamos las coordenadas
        step +=1 #Contador
    num_cuadros.append(len(difusion)) #Guardamos el número de cuadros por cada evolución temporal
    k+=1 #Contador temporal
    num_ite.append(k) #Guardamos en una lista el contador
```

Figura 5: Inspección de los vecinos y evolución del sistema

```
#Convertimos en arreglos numericos para poder graficar
x1=np.array(num_ite)
y1=np.array(num_cuadros)

#Hacemos una regresión lineal para calcular la pendiente de la recta
regresion_lineal.fit(x1.reshape(-1,1), y1) # instruimos a la regresión lineal que aprenda
# vemos los parámetros que ha estimado la regresión lineal
print('w = ' + str(regresion_lineal.coef_) + ', b = ' + str(regresion_lineal.intercept_))

#Graficando la pendiente
def f(m):  # función f(x) = w*x + b + Ruido_Gaussiano
    z = regresion_lineal.coef_*x2 +  regresion_lineal.intercept_
    return z
x2 = np.arange(0, 2500, 1)
y2 = f(x2)
#grafica estado final
plt.figure()
plt.plot(x2,y2,label='data', color='green')
plt.plot(num_ite,num_cuadros,"om",label="Puntos del modelo")
plt.ylabel("Área")
plt.xlabel("Tiempo")
plt.title("Evolución del sistema")
plt.legend()
plt.grid()
plt.show()
plt.figure(figsize = (6,6))
plt.title('Difusión, {} iteraciones, r_prom = {:3.2f}'
          . format(pasos,  np.mean(r_list)),
          fontdict = font)

plt.contourf(sistema, cmap = 'plasma')
plt.show()
```

Figura 6: Cálculo de la pendiente y gráficas

```
fig, ax = plt.subplots(figsize = (6,6))
#GIF
def init():
    ax.contourf(config[0], cmap = 'plasma')
    ax.set_xlabel('$X$', fontdict = font)
    ax.set_ylabel('$Y$', fontdict = font)
    ax.set_title('Difusión, {} iteraciones, r_prom = {:3.2f}'. format(pasos, np.mean(r_list)), fontd
    return fig,

def animate(i):
       im = ax.contourf(config[i], cmap = 'plasma')
       return im,

video = animation.FuncAnimation(fig,
                animate,
                init_func = init,
                frames = pasos,
                interval = 1)

titulo_video = 'Difusion_mod.gif'
video.save(titulo_video, writer = 'pillow', fps = 10)
```

Figura 7: Creación del GIF

5. Resultados

Debido a lo pesado del código decidimos correr una simulación para obtener resultados con valores de iteración relativamente grandes pero sin calcular la regresión lineal porque la función actuaba sobre muchos datos y la laptop tronaba cuando intentábamos hacer estas dos cosas a la vez con el GIF, con valores de $pasos = 1500$. Comentamos la parte del gif porque consumía mucha capacidad de nuestra laptop (solo te-

nemos dos núcleos). La primer simulación corrió con los valores de $N = 50$, $pasos = 200$ y $D_T = 2{,}5$. Los resultados son los siguientes:
Para el estado inicial del sistema se observa en la figura 8 el punto amarillo representa la sustancia semilla que se va a difundir. Las partes purpuras corresponden a las partes del sistema donde no se ha difundido la sustancia. Un cálculo propuesto por el profesor fue la pendiente de la recta que se ajusta a los datos,

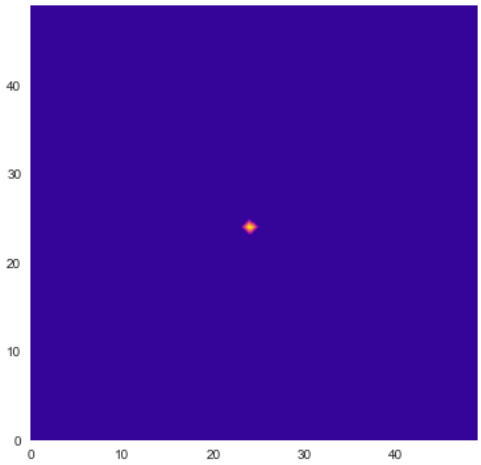


Figura 8: **Estado inicial del sistema.**

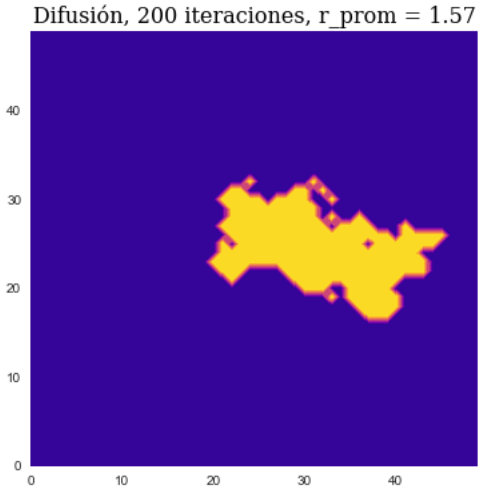


Figura 9: **Estado final del sistema.**

para eso hicimos una regresión lineal en Python haciendo uso de la documentación en [3]. Lo que obtenemos de esta función es el coeficiente $m = 0{,}01930976$ y la intersección con el eje b dados por la ecuación $y = (0{,}01930976)x + 45{,}2655548$ En este caso una aproximación del coeficiente de difusión está dada por $C_{difusion} = m$. Se muestra en la gráfica 10 la evolución del área en función del tiempo. Se consideró que el área podría ser proporcional al número de píxeles por cada paso temporal (steps). Lo que observamos es que el área aumenta mientras que converge hacia algún valor a medida que crece el tiempo, esto se observa para iteraciones mas grandes como en la figura 11. En el siguiente link podrá observar el GIF que se obtiene de la evolución del sistema.
`https://uanledu-my.sharepoint.com/:p:/g/personal/othoniel_dominguezgdnz_uanl_edu_mx/EU3Uxs4J3zBBriuHM`
`e=xpa0cJ`

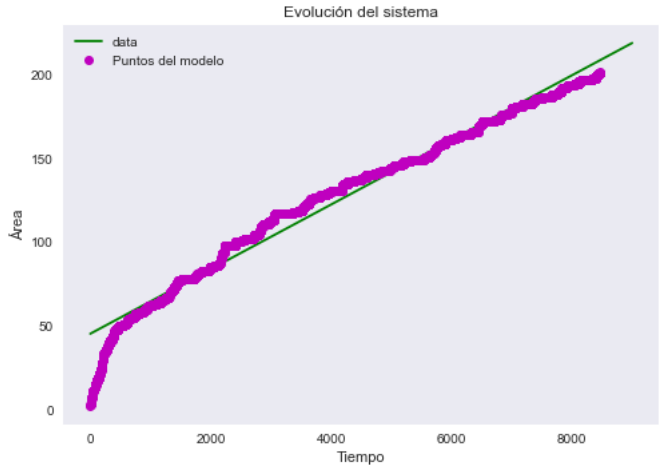


Figura 10: **Número de píxeles contra pasos temporales.** Con un número de iteraciones $pasos = 200$, $D_T = 2{,}5$ y dimensión $N = 50$

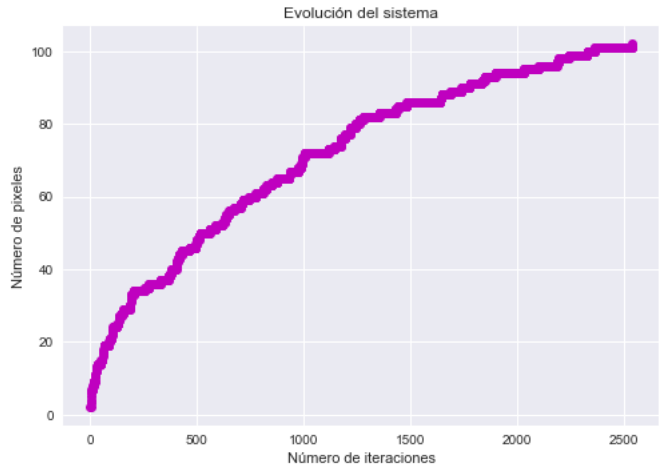


Figura 11: **Número de píxeles contra pasos temporales.** Con un número de iteraciones $pasos = 500$ y dimensión $N = 50$

6. Conclusión

Con un mejor equipo de computo pudimos haber obtenido curvas mas suaves y también pudimos haber aumentado el número de mallas para obtener resultados mas finos. Lo mas interesante son las aplicaciones de este modelo. Por las referencias nos dimos cuenta que este algoritmo podría ser la base para crear otro algoritmo que nos grafique el proceso DLA. Así obtendríamos fractales.

Referencias

[1] What are Lichtenberg figures, and how do we make them? `http://capturedlightning.com/frames/lichtenbergs.html`

[2] A Student's Guide to Python for Physical Modeling. Jesse M. Kinder and Philip Nelson

[3] Documentación de Python sobre la regresión lineal `https://scikit-learn.org/stable/modules/linear_model.html`