Uppsala University

Modelling Complex Systems

1MA256

Lab2 report

Author:
Kristensen.Samuel
Place of publication: Uppsala

April 21, 2024

1 Lorenz system

1.1 Task 1: Lorenz attractor

The lorenz system is a system of ordinary differiental equations on the form

$$\frac{dx}{dt} = \sigma(y - x) \tag{1}$$
$$\frac{dy}{dt} = x(\rho - z) - y \tag{2}$$
$$\frac{dz}{dt} = xy - \beta z \tag{3}$$

To solve and plot the ODE we implement the fourth order runge-kutta method

$$y_{n+1} = \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$
$$t_{n+1} = t_n + h$$

For $n = 0, 1, 2, 3, ...$ using

$$k_1 = f(t_n, y_n),$$
$$k_2 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2},\right)$$
$$k_3 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right),$$
$$k_4 = f(t_n + h, y_n + hk_3).$$

We do this for $y_n = x_n, y_n, z_n$ and where $f(t_n, y_n)$ are the right hand sides corresponding to the equations in $(1), (2), (3)$.

The classical parameters for the lorenz equation are $\sigma = 10, \beta = \frac{8}{3}, \rho = 28$. The initial data are set to $x(0) = 0, y(0) = 1, z(0) = 1.05$ and the code to solve and plot the lorenz equation is in section 3.1.

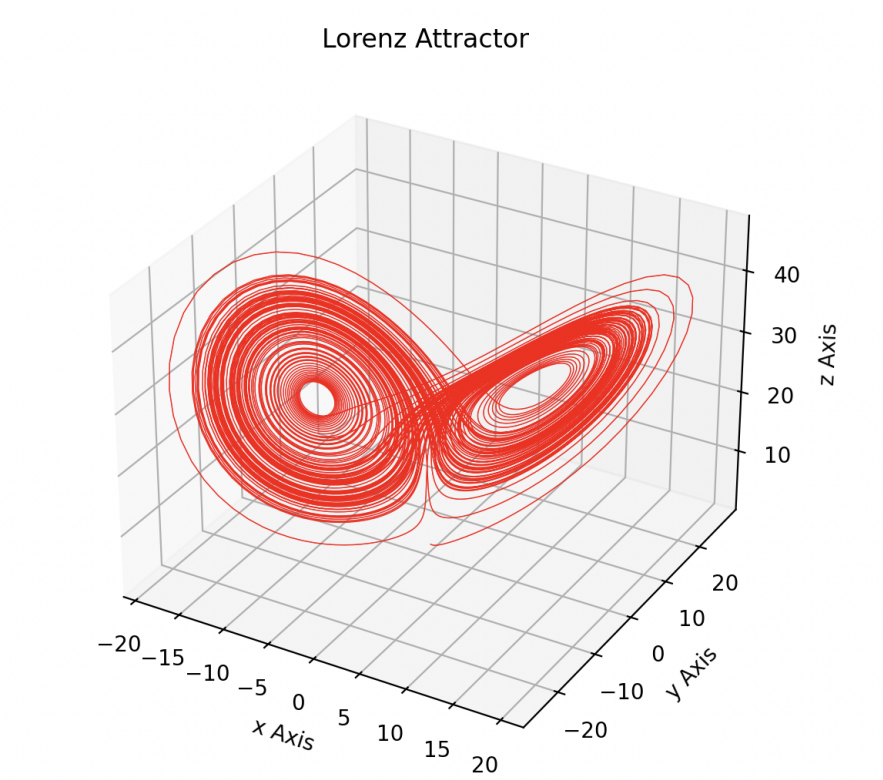


Figure 1: Lorenz attractor for the classical parameters

Figure 1 shows the 3D plot of the lorenz attractor.

1.2 Task 2: Lyapunov exponent

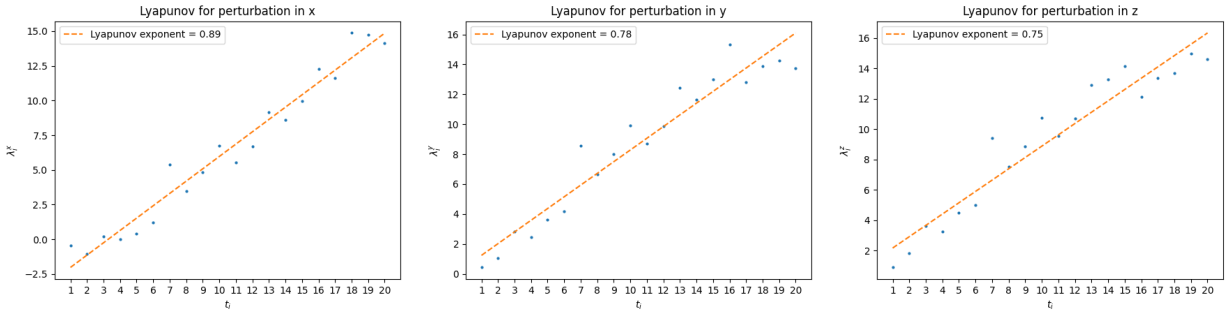
To calculate the lyapunov exponent, we start with choosing initial conditions (x_0, y_0, z_0) so that the orbit is in the basin of attraction of the Lorenz attractor. We choose $x_0 = y_0 = z_0 = 10$ and purturbate this initial condition wit a small number epsilon ($\epsilon = 10^{-5}$). We then solve the ODE for each timestep and compare the original solution to the perturbated one. We do this for all three dimension, i.e. $x_0^p = x_0 + \epsilon$ and so on.

The λ_i^x can then be measured for every timestep by using

$$\lambda_i^x = ln \frac{|\delta \bar{x}(t_i)|}{|\delta x(0)|} \tag{4}$$

Where $\delta \bar{x}(t_i)$ is the distance between the perturbated value of the Lorenz system with the original at time t_i and $\delta x(0)$ is the distance between the perturbated value of the Lorenz system with the original at the start, i.e. ϵ

We can then get the Lyapunov exponent for the three dimensions by taking the slope in the graphs. The code for this experiment is in section 3.2.



(a) λ^x (b) λ^y (c) λ^z

Figure 2: Lyapunov for the three dimensions

Figure 2 shows us the logarithmic difference difference in position for when a small perturbation is applied to the three dimensions x,y and z. We can see from the figures that the biggest Lyapunov exponent is when x is perturbated ant thus $\lambda_{max} = max(\lambda^x, \lambda^y, \lambda^z) = \lambda^x = 0.89$.

The Lyapunov value is positive which suggests that there is chaos.

2 Greenberg-Hastings CA (GHCA)

2.1 Task 1: python model

To replicate the Greenberg-Hastings cellular automation we consider a grid of $x \times n$ with two integers $N \leq 2$ and $e \in [1, N-2]$. Each point on the grid is in a state $0, 1, ..., N-1$ and the states $1, 2, ..., e$ are called excited states. cited states. The four neighbors of a cell are the cells situated north, south, west, and east (von Neumann's neighborhood) and the rules are as follows:

- if a cell is in state k with $1 \leq k \leq N-2$, then its next state is k + 1;
- if a cell is in state N 1, then its next state is 0;
- if a cell is in state 0,
  - if one of its neighbors is in an excited state, then its next state is 1,
  - if none of its neighbors is in an excited state, then its next state is 0.

We implement these rules in python and randomly generate some initial data with each of the points being a random number of the states. The code for this task is in section 3.3 We also draw a figure of the last state after advancing this configuration 50 steps.

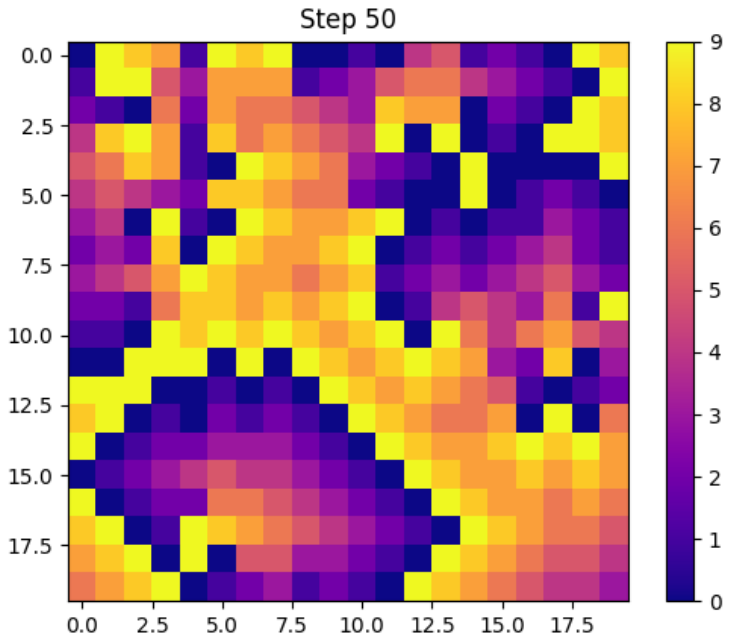


Figure 3: GHCA after 50 steps

Figure 3 shows us the states of each cell after 50 steps using $N = 10$ and $e = 2$ on a 20×20 grid.

To find the cardinality of the set or the number of all possible configurations we use the fact that each cell can have N different states and we have $n \times m$ cells then the cardinality is

$$\text{cardinality} = N^{n \times m} \tag{5}$$

So for the example that was run previously we had $N = 10$ and $n = 20, m = 20$ so the cardinality is $10^{20 \times 20} = 10^{400}$

2.2 Task 2: Periodicity

We are now going to find a periodic comfiguration, i.e. $m \geq 2$ for

$$F^{m+k}(C) = F^k(C) \tag{6}$$

We can do this by saving the states for each step and then compare the states with each other, once we find the states that match we can append the step to a list and then finally plot the first occurrence of periodicity. The code to do this is in section 3.4.

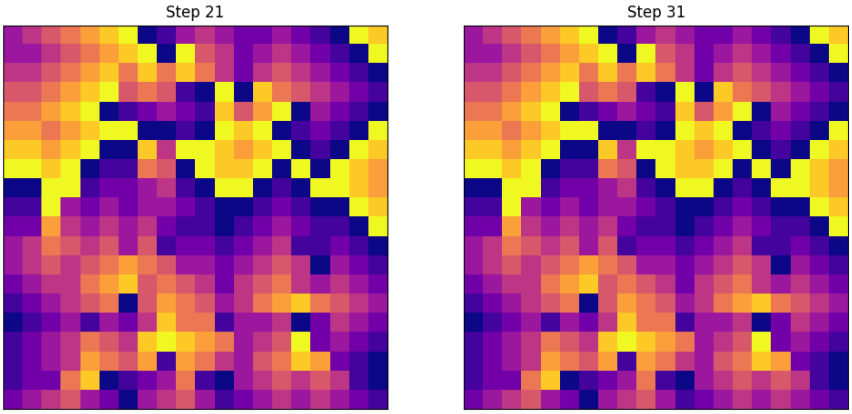


Figure 4: First two equal states

Figure 4 shows the first two states that are equal to each other, we see that the steps between them is 10 and by printing out the other pairs of equal states we can confirm that the periodicity is $m = 10$. We used again $N = 10$ and $n \times m = 20 \times 20$.

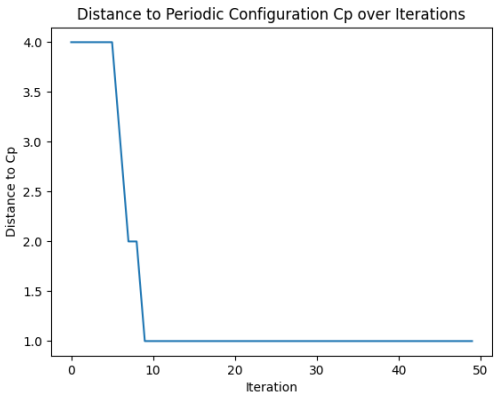
This is expected since the notes on cellular automata states that if f $N \geq 5$ and if $e \geq 2$, any structure that is living has period N.

2.3 Task 3: Attracting or repelling

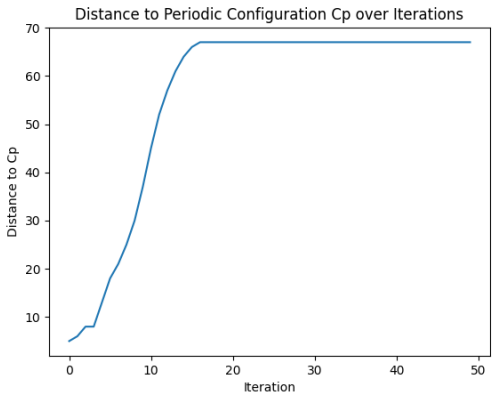
We are now going to study if for some periodic configuration C_p it's attracting or repelling. To do this we choose a periodic configuration, for example we can choose C_p as the fist instance where we found the periodic states, i.e. the left figure in figure 4 from section 2.2. After doing this we change a number of cells in it to some random states and iterate it and check if its orbit eventually begins to coincide with that of C_p or not.

Let C be the matrix that only differs from C_p at only several cells, we then take the orbit from C_p, i.e. the matrices from step 21 to step 31 in figure 4. We then iterate C and compare for each step with the orbit for C_p. The code for this is in section 3.5.

We test this by changing 5 cells from C_p



(a) One run



(b) Another run

Figure 5: Two runs for the distance from the orbit between C and C_p

We see from figure 5 that C does not coincide with the orbit of C_p. The figure shows two different runs where the initial configuration is randomly initiated. The distance between C and C_p can quite dramatically change, we see in the left figure that it can come close to each others orbit but one cell is still different all the time and for the right one the orbits keeps getting more different over time to a limit.

2.4 Task 4: Additive

We shall now check if the GHCA is additive, to do this we create two matrices C_1 and C_2 and also $C_{12} = C_1 + C_2$, where the sum of two configurations may be defined cell-wise, for example, as $c_1^{i,j} + c_2^{i,j}$ mod N, where $c_k^{i,j} \in \{0, 1, ..., N-1\}$ are values of (i, j)-th cells in configuration $C_k, k = 1, 2$

We are iterating C_1, C_2 and C_{12} with the rules of GHCA and then checking if $F(C_1) + F(C_2) = F(C_{12})$. We do this 10 times for different random matrices and comparing the final matrices after 100 steps.

The code for this can be found in section 3.6 and the result was that for all the different initial configurations the matrices were not the same, thus $F(C_1) + F(C_2) \neq F(C_{12})$

3 Codes

3.1 Code for Task 1.1

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import Axes3D

# Lorenz equations
def xt(x, y, z, t):
    return sigma * (y - x)

def yt(x, y, z, t):
    return x * (rho - z) - y

def zt(x, y, z, t):
    return x * y - beta * z

# Runge-Kutta 4th order method
def rk4(t, dt, x, y, z):
    # k for x, l for y, m for z.
    k1 = xt(x, y, z, t)
    l1 = yt(x, y, z, t)
    m1 = zt(x, y, z, t)
    k2 = xt(x + 0.5 * k1 * dt, y + 0.5 * l1 * dt, z + 0.5 * m1 * dt, t + dt / 2)
    l2 = yt(x + 0.5 * k1 * dt, y + 0.5 * l1 * dt, z + 0.5 * m1 * dt, t + dt / 2)
    m2 = zt(x + 0.5 * k1 * dt, y + 0.5 * l1 * dt, z + 0.5 * m1 * dt, t + dt / 2)
    k3 = xt(x + 0.5 * k2 * dt, y + 0.5 * l2 * dt, z + 0.5 * m2 * dt, t + dt / 2)
    l3 = yt(x + 0.5 * k2 * dt, y + 0.5 * l2 * dt, z + 0.5 * m2 * dt, t + dt / 2)
    m3 = zt(x + 0.5 * k2 * dt, y + 0.5 * l2 * dt, z + 0.5 * m2 * dt, t + dt / 2)
    k4 = xt(x + k3 * dt, y + l3 * dt, z + m3 * dt, t + dt)
    l4 = yt(x + k3 * dt, y + l3 * dt, z + m3 * dt, t + dt)
    m4 = zt(x + k3 * dt, y + l3 * dt, z + m3 * dt, t + dt)

    x_next = x + (dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6)
    y_next = y + (dt * (l1 + 2 * l2 + 2 * l3 + l4) / 6)
    z_next = z + (dt * (m1 + 2 * m2 + 2 * m3 + m4) / 6)

    return x_next, y_next, z_next

# Parameters
sigma = 10.0
beta = 8.0 / 3.0
rho = 28.0
n = 5000
T = 50
dt = T / n

# Initial conditions
x0 = 10
```

```
y0 = 10
z0 = 10

# Initialize arrays
t_values = np.zeros(n + 1)
x_values = np.zeros(n + 1)
y_values = np.zeros(n + 1)
z_values = np.zeros(n + 1)

# Set initial values
t_values[0] = 0
x_values[0] = x0
y_values[0] = y0
z_values[0] = z0

# Create figure and 3D axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot initial point
line, = ax.plot([], [], [], 'r-', linewidth=0.7)

# Set axis limits
ax.set_xlim(-20, 20)
ax.set_ylim(-30, 30)
ax.set_zlim(0, 50)

# Set axis labels
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')

# Text annotation for time
time_text = ax.text2D(0.05, 0.95, '', transform=ax.transAxes)

# Initialization function: plot the background of each frame
def init():
    line.set_data([], [])
    line.set_3d_properties([])
    time_text.set_text('')
    return line, time_text

# Animation function. This is called sequentially
def animate(i):
    global x_values, y_values, z_values
    x_next, y_next, z_next = rk4(t_values[i], dt, x_values[i], y_values[i],
        z_values[i])
    t_values[i + 1] = t_values[i] + dt
    x_values[i + 1] = x_next
    y_values[i + 1] = y_next
    z_values[i + 1] = z_next
```

```
    line.set_data(x_values[:i], y_values[:i])
    line.set_3d_properties(z_values[:i])
    time_text.set_text('Time = {:.1f}'.format(t_values[i]))
    return line, time_text

# Call the animator. blit=True means only re-draw the parts that have changed.
# Call the animator. Set repeat=False to stop after one run.
ani = FuncAnimation(fig, animate, init_func=init, frames=n, interval=1,
    blit=True, repeat=False)


plt.show()
```

3.2 Code for Task 1.2

```
import numpy as np
import matplotlib.pyplot as plt
from Lorenz import rk4, xt, yt, zt

# Parameters
sigma = 10.0
beta = 8.0 / 3.0
rho = 28.0
n = 2000
T = 20
dt = T / n

epsilon = 1e-5

# Initial conditions
x0 = 10
y0 = 10
z0 = 10

# Initialize arrays
t_values = np.zeros(n + 1)
x_values = np.zeros(n + 1)
y_values = np.zeros(n + 1)
z_values = np.zeros(n + 1)

# Set initial values
t_values[0] = 0
x_values[0] = x0
y_values[0] = y0
z_values[0] = z0

# Initial conditions perturbated
x0p = x0
y0p = y0
```

```
z0p = z0 + epsilon

# Initialize arrays perturbated
x_valuesp = np.zeros(n + 1)
y_valuesp = np.zeros(n + 1)
z_valuesp = np.zeros(n + 1)

# Set initial values perturbated
x_valuesp[0] = x0p
y_valuesp[0] = y0p
z_valuesp[0] = z0p


for i in range(n):
    x_next, y_next, z_next = rk4(t_values[i], dt, x_values[i], y_values[i],
        z_values[i])
    t_values[i + 1] = t_values[i] + dt
    x_values[i + 1] = x_next
    y_values[i + 1] = y_next
    z_values[i + 1] = z_next

    x_nextp, y_nextp, z_nextp = rk4(t_values[i], dt, x_valuesp[i],
        y_valuesp[i], z_valuesp[i])
    x_valuesp[i + 1] = x_nextp
    y_valuesp[i + 1] = y_nextp
    z_valuesp[i + 1] = z_nextp

def get_length(x, y, z, xp, yp, zp):
    # Calculate the difference in coordinates
    dx = xp - x
    dy = yp - y
    dz = zp - z

    # Calculate the Euclidean distance
    length = np.sqrt(dx**2 + dy**2 + dz**2)

    return length

delta_xt0 = epsilon
lambda_values = []
t_plot = []
for i in range(0, n+1, 100):
    t_plot.append(t_values[i])
    delta_xti = get_length(x_values[i],y_values[i],z_values[i],
        x_valuesp[i],y_valuesp[i],z_valuesp[i])
    lambda_values.append(np.log(np.abs(delta_xti)/delta_xt0))

slope, intercept = np.polyfit(t_plot[1:], lambda_values[1:], 1) # fit line
d_array = [val*slope+intercept for val in t_plot[1:]] # calculate line

plt.plot(t_plot[1:],lambda_values[1:], 'o', markersize=2)
```

```
plt.plot(t_plot[1:], d_array, '--', label=f'Lyapunov exponent = {slope:.2f}')
plt.title('Lyapunov for perturbation in z')
plt.xlabel('$t_{i}$')
plt.ylabel('$\lambda_{i}^{z}$')
# Set custom x-ticks
plt.xticks(np.arange(1, 21, 1))
plt.legend()

plt.show()
```

3.3 Code for Task 2.1

```
import numpy as np
import random
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# initialize hastings with 20x20 matrix size
n = 20
m = 20

C = np.empty([n,m], dtype=int)

N = 10     # Number of states
states = [0,N-1]

# excited states
e = [1,2]
threshold = 1

# randomly generate an initial configuration for GHCA.
for i in range(n):
  for j in range(m):
    C[i][j] = random.randint(states[0],states[1])

# returns the von-moore neighbours of a cell at i,j for matrix C
def neighbours(i,j,C):
    neighbours = []
    if max(i - 1, 0) != i:
        neighbours.append(C[max(i - 1, 0), j])

    if max(j - 1, 0) != j:
        neighbours.append(C[i, max(j - 1, 0)])

    if min(i + 1, len(C) - 1) != i:
        neighbours.append(C[min(i + 1, len(C) - 1)][j])

    if min(j + 1, len(C[i]) - 1) != j:
        neighbours.append(C[i][min(j + 1, len(C[i]) - 1)])
```

```
    return neighbours

# function counting the number of excited neighbours.
def count(neighbours, excited_cells):
    counts = 0
    for i in neighbours:
        if i in excited_cells:
            counts += 1
    return counts

# GHCA applied 100 times giving 100 configurations
steps =50
# Create initial plot with color bar

fig, ax = plt.subplots()
cax = ax.imshow(C, cmap='plasma', interpolation='nearest')
cbar = fig.colorbar(cax)

# Function to update the plot for each step
def update(frame):
    global C
    # Update C
    C_temp = np.empty_like(C) # Create empty matrix to store updated values

    for i in range(len(C)):
        for j in range(len(C[i])):
            if 1 <= C[i][j] <= N - 2:
                C_temp[i][j] = C[i][j] + 1
            elif C[i][j] == N - 1:
                C_temp[i][j] = 0
            else:
                counts = count(neighbours(i, j, C), e)
                if counts >= threshold:
                    C_temp[i][j] = 1
                else:
                    C_temp[i][j] = 0

    C = C_temp

    # Update plot
    cax.set_data(C)
    plt.title(f"Step {frame + 1}")
    plt.pause(0.1)

# Create animation
animation = FuncAnimation(fig, update, frames=steps, repeat=False)

plt.show()

# save the final matrix in final_matrice.txt
fin_matr = open('final_matrice.txt','a')
```

```
fin_matr.writelines([str(C[i,j])+' ' if j<m-1 else str(C[i,j])+'\n' for i in
    range(len(C)) for j in range(len(C[i]))])

fin_matr.close()
```

3.4 Code for Task 2.2

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

# initialize hastings with 20x20 matrix size
n = 20
m = 20

C = np.empty([n,m], dtype=int)

N = 10
states = [0,N-1]

# excited states
e = [1,2]
threshold = 1

# randomly generate an initial configuration for GHCA.
for i in range(n):
  for j in range(m):
    C[i][j] = random.randint(states[0],states[1])

# returns the von-moore neighbours of a cell at i,j for matrix C
def neighbours(i,j,C):
    neighbours = []

    if max(i - 1, 0) != i:
        neighbours.append(C[max(i - 1, 0), j])

    if max(j - 1, 0) != j:
        neighbours.append(C[i, max(j - 1, 0)])

    if min(i + 1, len(C) - 1) != i:
        neighbours.append(C[min(i + 1, len(C) - 1)][j])

    if min(j + 1, len(C[i]) - 1) != j:
        neighbours.append(C[i][min(j + 1, len(C[i]) - 1)])

    return neighbours

# function counting the number of excited neighbours.
def count(neighbours, excited_cells):
    counts = 0
```

```
    for i in neighbours:
        if i in excited_cells:
            counts += 1
    return counts


# these 2 lists below store the configuration and respective
# time step for every time steps where (step+1) % N==0
period_mats = []
period_mats.append(C)
period_time = []

# GHCA applied 50 times giving 100 configurations
steps = 50
for step in range(steps):
    C_temp = np.empty([n, m], dtype=int)

    for i in range(n):
        for j in range(m):
            if 1<= C[i][j] <=N-2:
                C_temp[i][j] = C[i][j] + 1

            elif C[i][j]== N-1:
                C_temp[i][j] = 0

            else:
                counts = count(neighbours(i, j, C), e)
                if counts >= threshold:
                    C_temp[i][j] = 1
                else:
                    C_temp[i][j] = 0

    C = C_temp

    period_mats.append(C)

# the for loop below plots the non-transient orbits
for i in range(0,len(period_mats)):
    for j in range(i+1,len(period_mats)):
        if np.array_equal(period_mats[i],period_mats[j]):
            period_time.append((i,j))
            break

# Plot the first pair of matrices in period_time
if period_time:
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))

    for idx, index in enumerate(period_time[0]):
        ax = axes[idx]
        ax.imshow(period_mats[index], cmap='plasma', interpolation='nearest')
        ax.set_title(f"Step {index}")
```

```
        ax.set_xticks([])
        ax.set_yticks([])
else:
    print("No periodic configuration found.")
```

3.5 Code for Task 2.3

```
# The previous code from Task 2.2

def compute_distance(config1, config2):
    return np.sum(config1 != config2)

orbit = []
for i in range(period_time[0][0], period_time[0][1]):
    orbit.append(period_mats[i])

#print(period_time)
C_p = np.empty([n,m], dtype=int)
C_p = period_mats[period_time[0][0]]

C = C_p.copy()

# Randomly modify a few cells
num_modifications = 5
for _ in range(num_modifications):
    i, j = random.randint(0, n-1), random.randint(0, m-1)
    C[i, j] = random.randint(0, N-1)

max_steps = 50
distances = []
for step in range(0,max_steps):
    new_C = np.empty_like(C)

    for i in range(n):
        for j in range(m):
            if 1<= C[i][j] <=N-2:
                new_C[i][j] = C[i][j] + 1

            elif C[i][j]== N-1:
                new_C[i][j] = 0

            else:
                counts = count(neighbours(i, j, C), e)
                if counts >= threshold:
                    new_C[i][j] = 1
                else:
                    new_C[i][j] = 0

    min_distaces = []
    for i in range(len(orbit)):
```

```
        distance = compute_distance(C,orbit[i])
        min_distaces.append(distance)

    min_dist = min(min_distaces)
    distances.append(min_dist)

    # Update C for the next iteration
    C = new_C.copy()

# Plot the distance over time
plt.figure(2)
plt.plot(range(len(distances)), distances)
plt.xlabel("Iteration")
plt.ylabel("Distance to Cp")
plt.title("Distance to Periodic Configuration Cp over Iterations")

plt.show()
```

3.6 Code for Task 2.4

```
# Same code as in previous Task

for i in range(10):
    # initialize hastings with 20x20 matrix size
    n = 20
    m = 20

    N = 10
    states = [0,N-1]

    # excited states
    e = [1,2]
    threshold = 1


    C1 = random_matrix()
    C2 = random_matrix()
    C12 = add_matrix(C1,C2,n,m)

    FC1 = GHCA(C1)
    FC2 = GHCA(C2)

    FC12 = GHCA(C12)

    FC1_FC2 = add_matrix(C1,C2,n,m)

    print(np.array_equal(FC1_FC2,FC12))
```