HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

SCIENTIFIC COMPUTING PROJECT REPORT

SCIENTIFIC COMPUTING
(IT4110E)

Diffusion-Limited Aggregation model

Team members	*Email*
Vu Duc An 20215174	an.vd215174@sis.hust.edu.vn
Le Ha Ngan 20215230	ngan.lh215230@sis.hust.edu.vn
Dang Viet Hoang 20215206	hoang.dv215206@sis.hust.edu.vn
Trinh Diem Quynh 20210737	quynh.td210737@sis.hust.edu.vn
Trinh Thi Dieu Thuy 20215247	thuy.ttd215247@sis.hust.edu.vn

Class
727616 - IT4110E
Lecturer
Vu Van Thieu
thieu.vuvan@hust.edu.vn

May 30, 2024

Time period: 23ndJune - 7thJuly

Scientific Computing Report
Diffusion-Limited Aggregation model
Class: 727616
Lecturer: Vu Van Thieu

Contents

I Introduction

I.1 Diffusion-Limited Aggregation

The Diffusion-Limited Aggregation (DLA) problem is a mathematical problem that involves simulating the growth of a random cluster of particles that are randomly moving in space and sticking together when they come into contact with each other.

In this context, DLA is often used to model the growth and aggregation of various entities, such as particles, cells, or organisms.

I.2 Real-world applications:

The DLA problem has practical applications in various fields such as physics, chemistry, biology and computer science. It is commonly used to simulate the growth and development of living organisms, including viruses, algae, and cells. Additionally, DLA can be applied to understand and describe physical processes such as dissolution and permeation. By studying DLA, researchers can gain insights into the dynamics and patterns of growth in complex systems.

I.3 Problem statement:

In the scope of this report, we focus on simulating the growth of virus entities within a limited environment that contains a food source. The objective is to model how the viruses interact with their surroundings, consume food, and undergo continuous growth. By analyzing the simulated growth patterns, we aim to gain a better understanding of the dynamics and behavior of virus populations in confined environments. This information can have implications in various fields, including biology, epidemiology, and materials science.

II Theory

II.1 Theorem statement:

During the description process, bacteria will grow from an initial cell. The growth of bacteria in the food environment is probabilistic and depends on the food concentration in the surrounding cells.

The food concentration in the environment is continuously changing, and cells containing bacteria will deplete the available food. The difference in food availability will be influenced by the diffusion equation.

The diffusion equation simulates the changes in food concentration, which occur much faster compared to the growth rate of the virus. Therefore, the diffusion process can be considered instantaneous, and a time-independent diffusion equation will be used to determine the food concentration.

$$\frac{\partial c}{\partial t} = D \bigtriangledown^2 c \tag{1}$$

D: Diffusion coefficient.
C: Concentration.

Since the equation is time-independent, the derivative of the concentration with respect to time can be considered as zero.

The above equation can be solved directly or using iterative method. The Successive Over Relaxation (SOR) method will be presented in the following section of the report.

II.2 SOR method:

The Successive Over Relaxation (SOR) method uses the following formula to calculate Ci,j:

$$c_{i,j}^{k+1} = \frac{\omega}{4}\left(c_{i+1,j}^{(k)} + c_{i-1,j}^{(k+1)} + c_{i,j+1}^{(k)} + c_{i,j-1}^{(k+1)}\right) + (1-\omega)c_{i,j}^{k} \tag{2}$$

Where ω is the relaxation parameter. The parameter ω indicates how much the concentration at a position in the previous iteration influences the next iteration. In the SOR method, we use a value of ω within the range of $(1, 2)$.

Specifically, in this report, we will investigate the SOR method with a value of $\omega = 1.89$.

II.3 Application to the DLA problem:

The basic algorithm for simulating the DLA process is as follows:

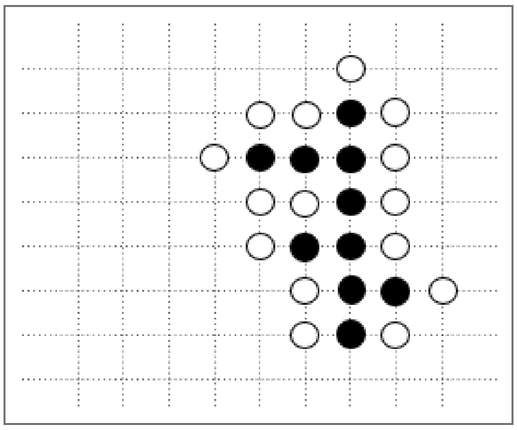
1. Solve the diffusion equation to obtain the food concentration matrix.
2. Allow viruses to reproduce (according to probabilities).

3. Repeat steps 1 and 2.

Specifically:

Step 1: The concentration matrix is generated using the SOR method.

Step 2: Each neighboring cell of a virus has a probability of being infected, which is proportional to the food concentration in that cell.



The image above illustrates a stage in the virus development process. The black cells represent infected cells, while the white cells represent potential sites for further virus growth. The probability of virus growth in each white cell is calculated using the equation:

$$P_g((i,j) \in \circ \to (i,j) \in \bullet) = \frac{(c_{i,j})^\eta}{\sum\limits_{(i,j)\in\circ} (c_{i,j})^\eta} \tag{3}$$

Where the exponent η determines the shape of the virus cluster and is typically chosen between 0.5 and 2. In this report, different values of η will be investigated.

III Implementation:

```python
#### import numpy as np

# Size of the grid: size
# Exponent p (eta) - in the probability formula
# Place the first virus at position x_start, y_start
def simulate(size, p, x_start, y_start):

    n = 4000 # Number of viruses
    nVirus = 1

    # Food concentration array
    C = np.zeros((size, size))
    temp = np.zeros((size, size))

    # Setting omega parameter
    w = 1.5

    # Grow array marks  locations where the viruses grow
    grow = np.zeros((size, size))
    grow[x_start, y_start] = 1

    # Setting default food
    for i in range(0, size ):
        for j in range(0, size ):
            if i == size-1:
                C[i, j] = 1
            else:
                C[i,j] = 0.001

    C[x_start, y_start] = 0

    while True:
        candidate = np.zeros((size, size))
        sumOfChance = 0

        # SOR method to calculate at step k+1
        for i in range(0, size):
            for j in range(0, size):
                if (i == size - 1):
                    C[i, j] = 1
                elif i == 0:
                    C[i,j] = 0;
                elif j == 0:
                    C[i, j] = (w/4) * (C[i+1, j] + C[i-1, j]
                    + C[i, j+1] + C[i, size-1]) + (1-w) * C[i, j]
                elif j == size-1:
                    C[i, j] = (w/4) * (C[i+1, j] + C[i-1, j]
                    + C[i, 0] + C[i, j-1]) + (1-w) * C[i, j]
                else:
                    C[i, j] = (w/4) * (C[i+1, j] + C[i-1, j]
                    + C[i, j+1] + C[i, j-1]) + (1-w) * C[i, j]

        # Find candidates
        for i in range(0, size-1):
            for j in range(0, size-1):
                if grow[i, j] == 1:
                    C[i, j] = 0
                    if i == 0:
                        if grow[0, j] == 0 and candidate[0, j] == 0:
                            candidate[0, j] = 1
                        if grow[i+1, j] == 0 and candidate[i+1, j] == 0:
                            candidate[i+1, j] = 1
                        if grow[i, j-1] == 0 and candidate[i, j-1] == 0:
                            candidate[i, j-1] = 1
                        if grow[i, j+1] == 0 and candidate[i, j+1] == 0:
                            candidate[i, j+1] = 1
                    elif i == size-1:
                        if grow[i-1, j] == 0 and candidate[i-1, j] == 0:
                            candidate[i-1, j] = 1
                        if grow[size-1, j] == 0 and candidate[size-1, j] == 0:
                            candidate[0, j] = 1
                        if grow[i, j-1] == 0 and candidate[i, j-1] == 0:
                            candidate[i, j-1] = 1
                        if grow[i, j+1] == 0 and candidate[i, j+1] == 0:
                            candidate[i, j+1] = 1
                    elif j == 0:
                        if grow[i-1, j] == 0 and candidate[i-1, j] == 0:
                            candidate[i-1, j] = 1
                        if grow[i+1, j] == 0 and candidate[i+1, j] == 0:
                            candidate[i+1, j] = 1
                        if grow[i, size-1] == 0 and candidate[i, size-1] == 0:
                            candidate[i, size-1] = 1
                        if grow[i, j+1] == 0 and candidate[i, j+1] == 0:
                            candidate[i, j+1] = 1
                    elif j == size-1:
                        if grow[i-1, j] == 0 and candidate[i-1, j] == 0:
                            candidate[i-1, j] = 1
                        if grow[i+1, j] == 0 and candidate[i+1, j] == 0:
                            candidate[i+1, j] = 1
                        if grow[i, j-1] == 0 and candidate[i, j-1] == 0:
                            candidate[i, j-1] = 1
                        if grow[i, 0] == 0 and candidate[i, 0] == 0:
                            candidate[i, 0] = 1
                    else:
                        if grow[i-1, j] == 0 and candidate[i-1, j] == 0:
                            candidate[i-1, j] = 1
                        if grow[i+1, j] == 0 and candidate[i+1, j] == 0:
                            candidate[i+1, j] = 1
                        if grow[i, j-1] == 0 and candidate[i, j-1] == 0:
                            candidate[i, j-1] = 1
                        if grow[i, j+1] == 0 and candidate[i, j+1] == 0:
                            candidate[i, j+1] = 1

        # Calculate the denominator of P
        for i in range(0, size ):
            for j in range(0, size ):
                if candidate[i, j] == 1:
                    sumOfChance += C[i, j]**p

        # Random grow
        for i in range(0, size ):
            for j in range(0, size ):
                if candidate[i, j] == 1:
                    randPos = np.random.rand() / 10
                    curChance = (C[i, j]**p) / sumOfChance
                    if randPos < curChance:
                        grow[i, j] = 1
                        if nVirus < n:
                            nVirus += 1

        print(nVirus, end="\r")

        if nVirus == n:
            x = np.arange(size)
            y = np.arange(size)
            X, Y = np.meshgrid(x, y)

            # Make a fig
            fig, axes = plt.subplots(2, 2, figsize=(10, 10)
                                , subplot_kw={'projection': '3d'})

            axes[0, 0].plot_surface(X, Y, C, cmap='jet')
            axes[0, 0].view_init(azim=-120, elev=30)

            axes[0, 1].contourf(X, Y, C, cmap='jet')
            axes[0, 1].view_init(azim=-90, elev=90)
            axes[0, 1].set_zticks([])

            axes[1, 0].contourf(X, Y, grow, cmap='jet')
            axes[1, 0].view_init(azim=-90, elev=90)
            axes[1, 0].set_zticks([])

            axes[1, 1].axis('off')

            filename = '{0}x{0}_X{1}Y{2}p{3}plot.png'.format(size, x_start, y_start
            plt.savefig(filename, transparent=True)
            plt.show(block=False)

            break

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def run_sim(size, p, x_start, y_start):
    simulate(size, p, x_start, y_start)

# Make directory
import os
path = "data"
# Check whether the specified path exists or not
isExist = os.path.exists(path)
if not isExist:
   os.makedirs(path)
```

Program used Python, Jupyter Notebook/Jupyter Lab to run the code and libary matplotlib, mpltoolkits to get result images. To run the program, use command `run_sim` with arguments:

`p` : Value of exponent η

`size` : The size of simulation

`x_start, y_start` : Starting position of the virus

`name` : Additional string to append to the output image file

For example, run one of the following:

```
run_sim(size = 200, p = 0, x_start = 2, y_start = 100)
run_sim(size = 200, p = 1, x_start = 2, y_start = 100)
run_sim(size = 500, p = 2, x_start = 250, y_start = 250)
```

III.1 Data structure:

2D `Numpy` array `candidate[][]` contains a list of candidates that can be infected with the virus in the current step

`C[][]` array represents the remaining food in the area, initialized to 1, except for a single cell containing the first virus

The `grow[][]` array is used to mark the position of the viruses.

III.2 Program structure:

1. Initialize `C[][]` equal 1 (because have no virus eat food), `grow[][]`, `candidate[][]` equal 0
2. Perform the SOR iteration method to calculate the entire `C[][]` matrix.
3. Find candidates place where having growth of viruss
4. Randomly generate viruses
5. Loop untill find enough viruss

III.3 Graph plotting:

The result matrix is stored in the .png file. This will create a visualization of the matrix using colors to represent different values.

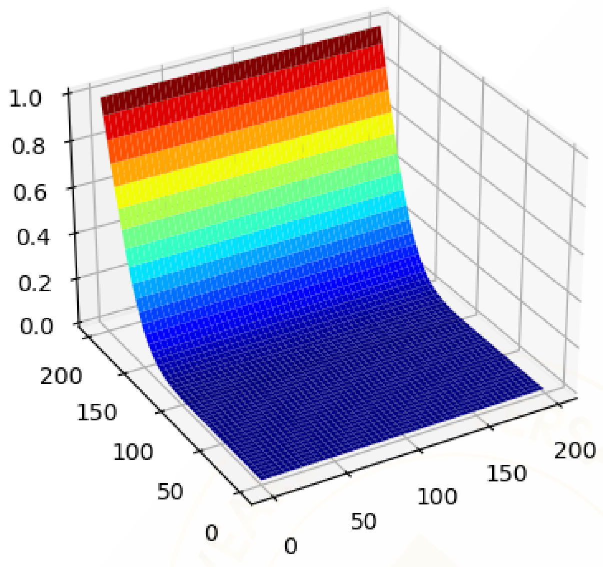
IV Experimental results:

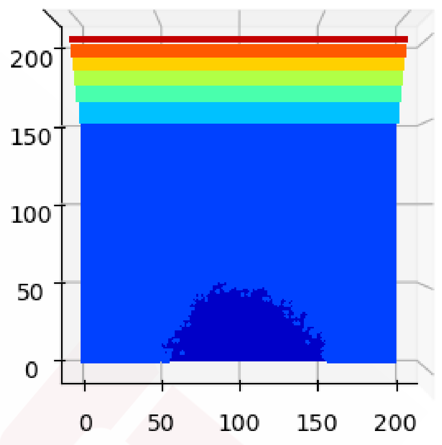
We set the boundary condition as follow: top always has food, bottom no food, left and right boundary always have the same food.

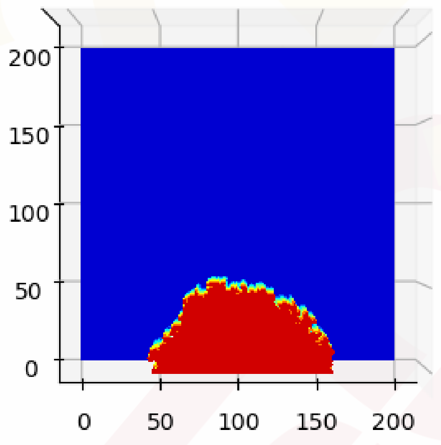
We release the first virus at the bottom of the screen and run simulations with different sizes: The redder the position, the higher the food concentration. Light red or yellow positions indicate areas with less food, while the darkest blue indicates the presence of viruses.

Running with a size of **200x200**, producing **4,000** viruses.

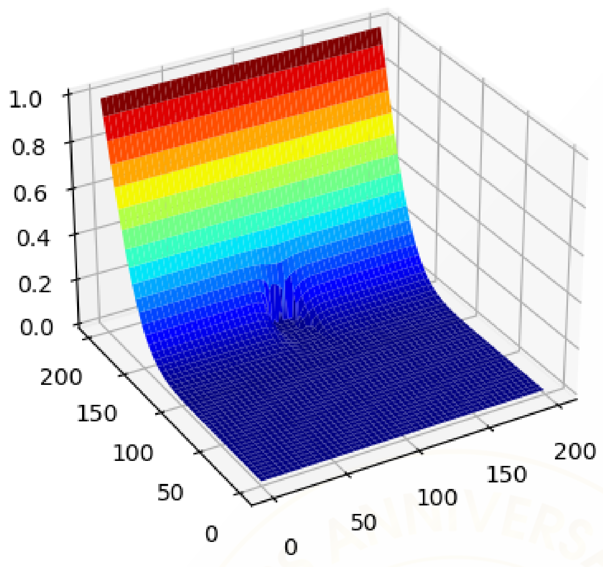
- With $\eta = 0$:



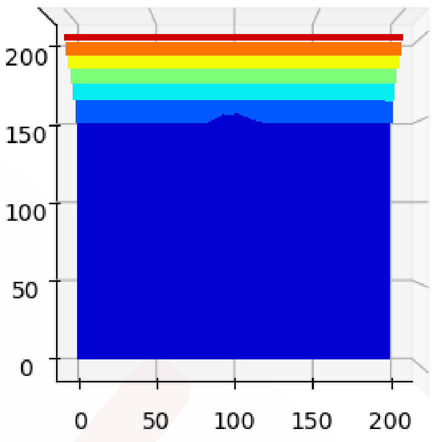




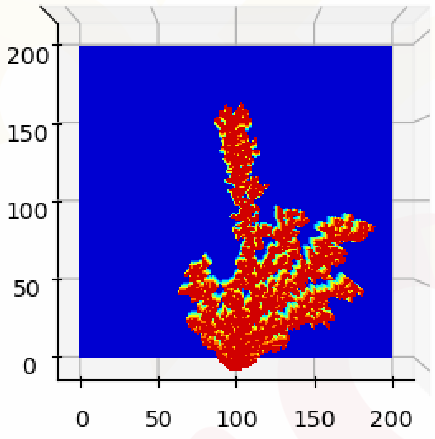
- With $\eta = 1$:



1.0
0.8
0.6
0.4
0.2
0.0
200
150
100
50
0
0
50
100
150
200

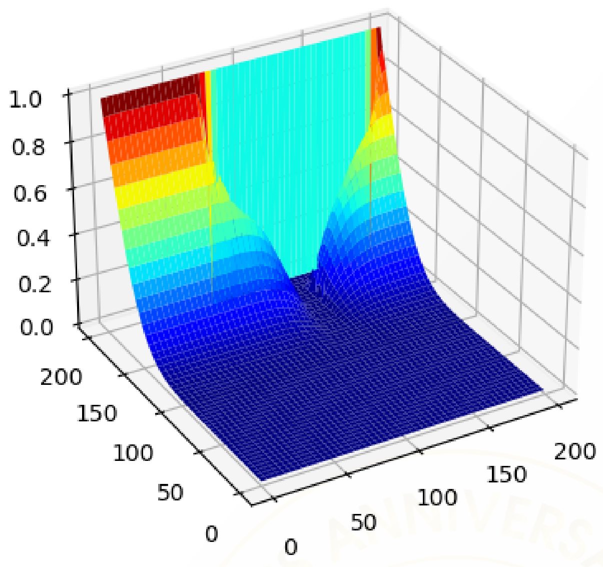


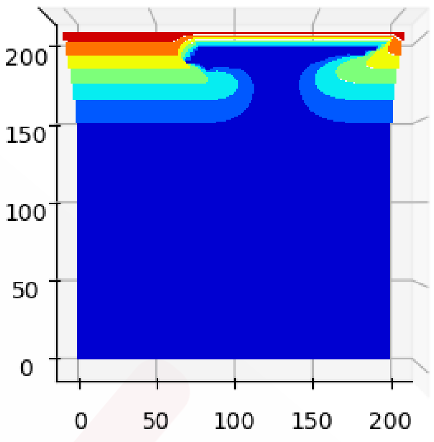
200
150
100
50
0
0
50
100
150
200

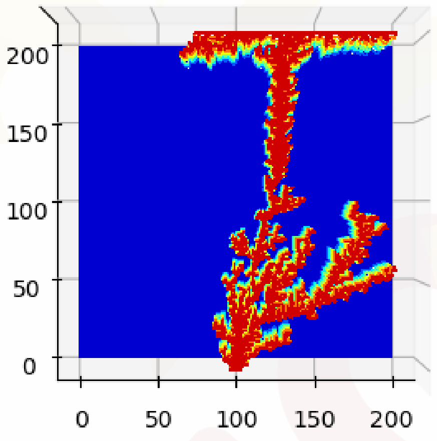


200
150
100
50
0
0
50
100
150
200

- With $\eta = 2$:

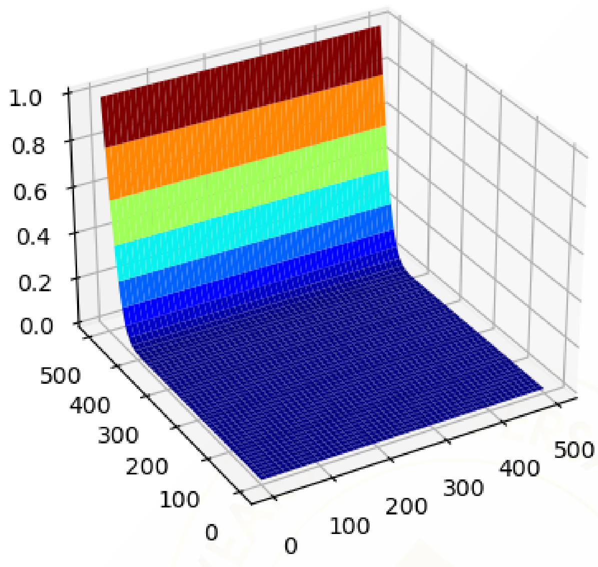

1.0
0.8
0.6
0.4
0.2
0.0
200
150
100
50
0
0
50
100
150
200

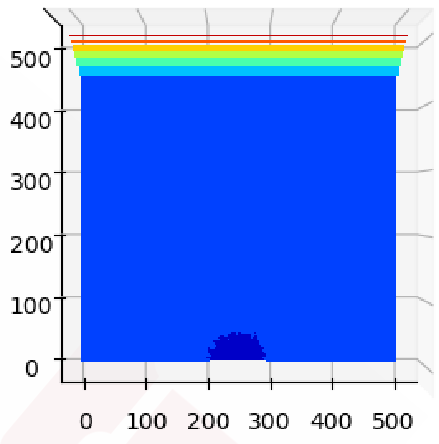

200
150
100
50
0
0
50
100
150
200

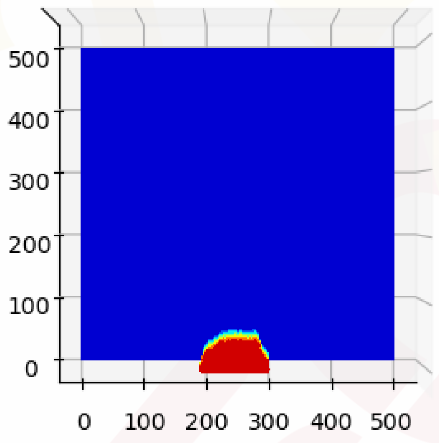

200
150
100
50
0
0
50
100
150
200

Running with a size of **500x500**:

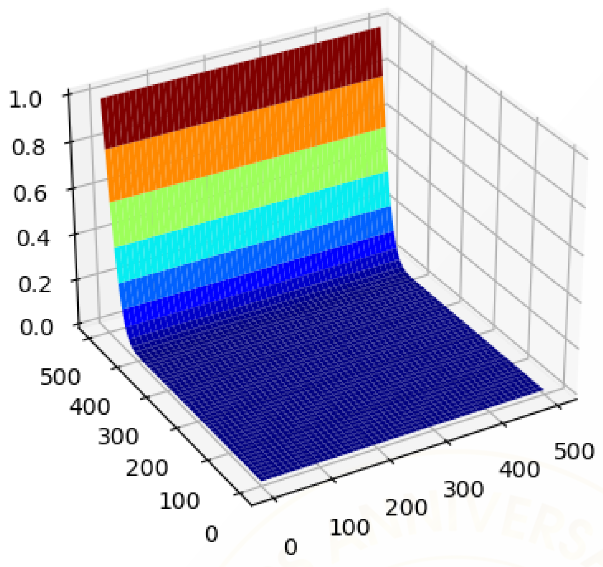
- With $\eta = 0$:



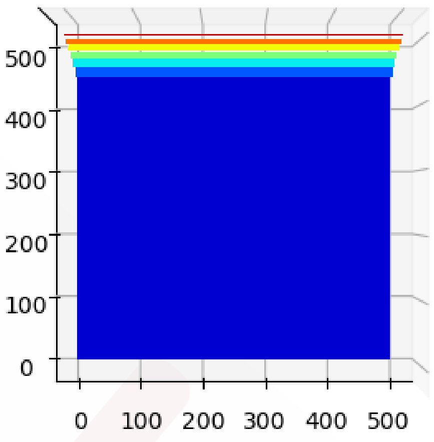




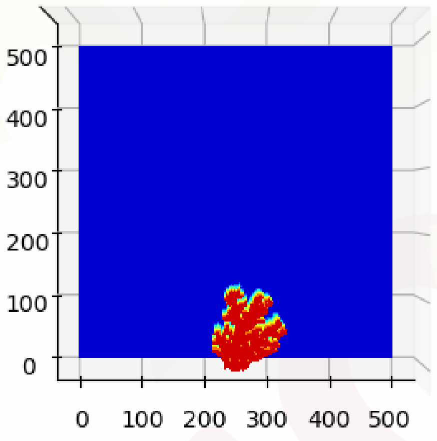
- With $\eta = 1$:



1.0
0.8
0.6
0.4
0.2
0.0
500
400
300
200
100
0
0
100
200
300
400
500

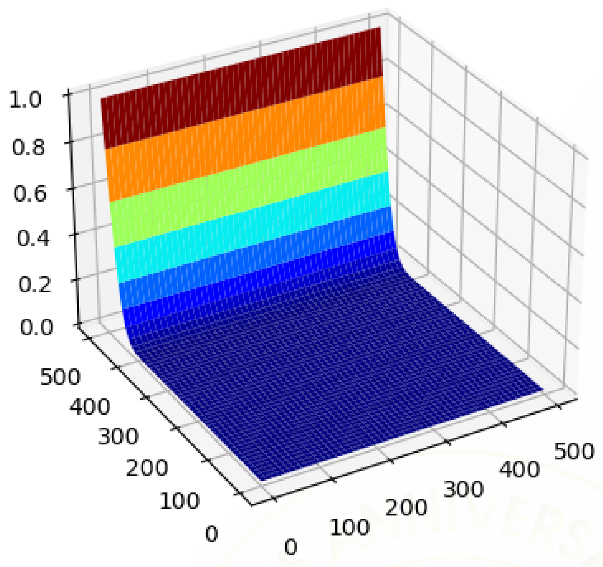


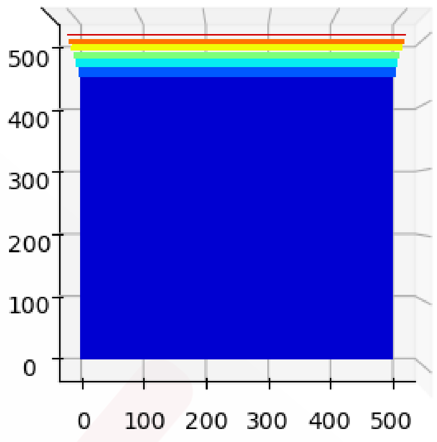
500
400
300
200
100
0
0
100
200
300
400
500

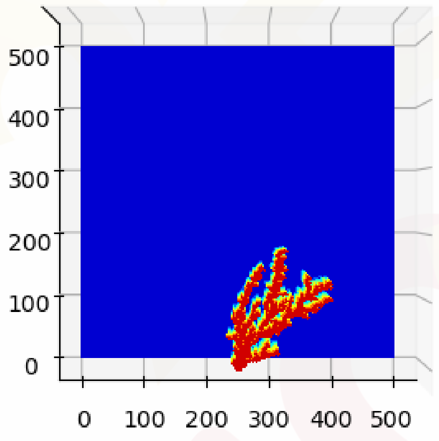


500
400
300
200
100
0
0
100
200
300
400
500

- With $\eta = 2$:

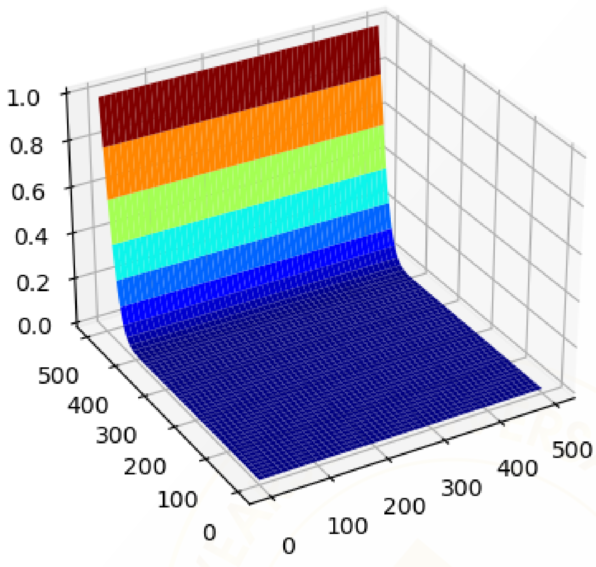

1.0
0.8
0.6
0.4
0.2
0.0
500
400
300
200
100
0
0
100
200
300
400
500

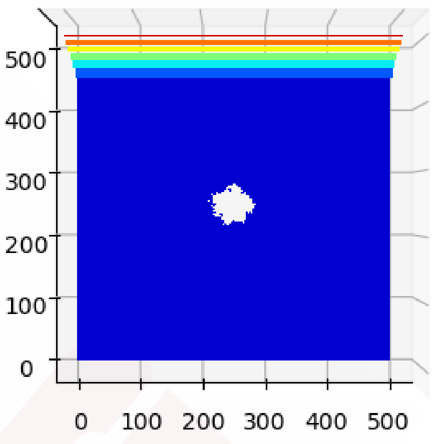

500
400
300
200
100
0
0
100
200
300
400
500

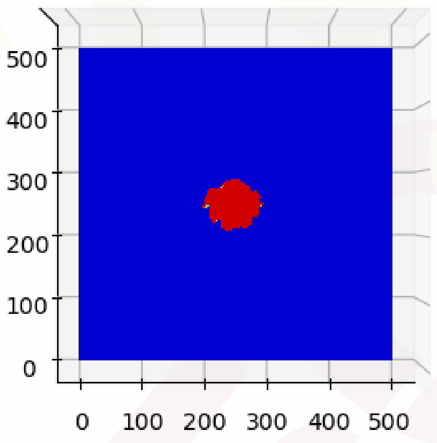

500
400
300
200
100
0
0
100
200
300
400
500

Similarly, but with viruses growing from the center of the screen:

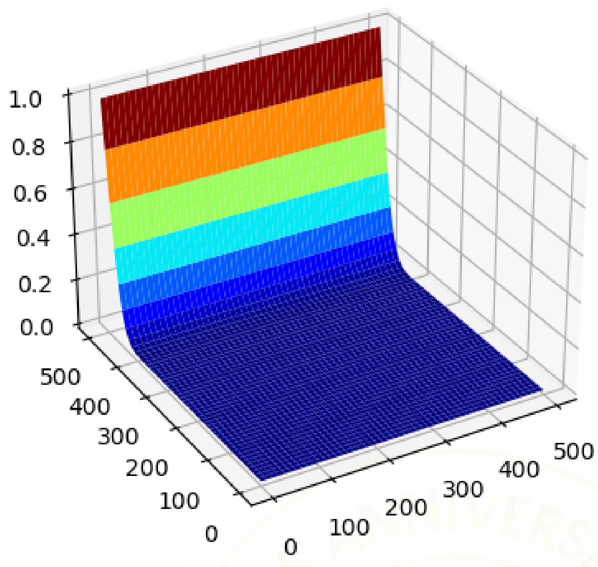
- With $\eta = 0$:



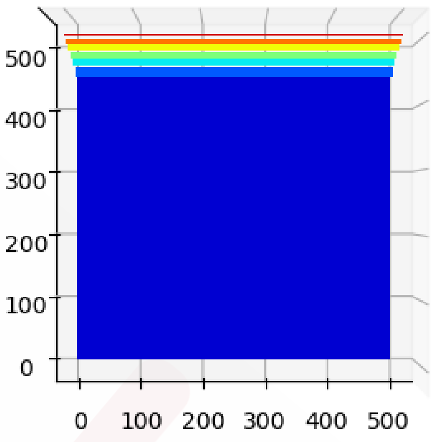




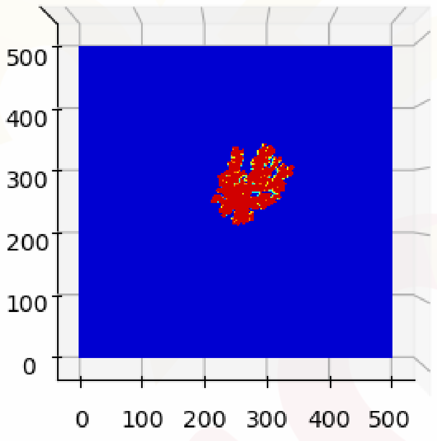
- With $\eta = 1$:



1.0
0.8
0.6
0.4
0.2
0.0
500
400
300
200
100
0
0
100
200
300
400
500

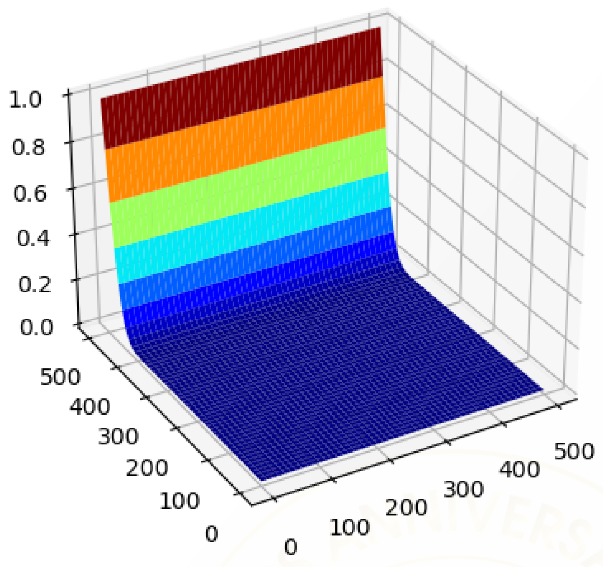


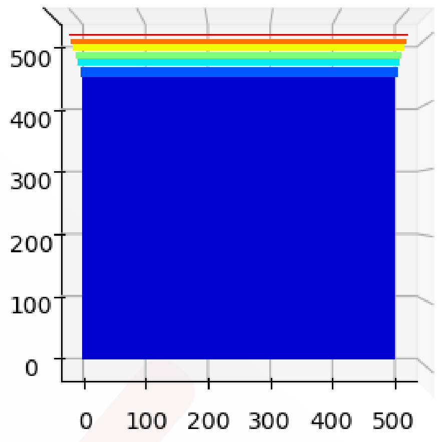
500
400
300
200
100
0
0
100
200
300
400
500

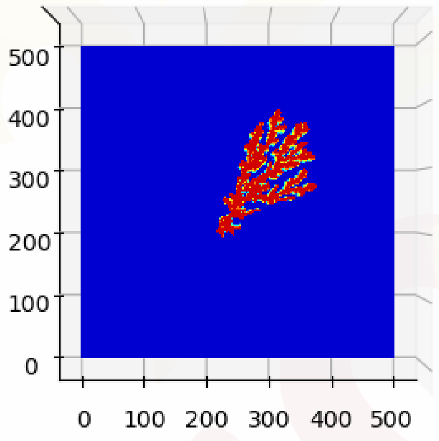


500
400
300
200
100
0
0
100
200
300
400
500

- With $\eta = 2$:


1.0
0.8
0.6
0.4
0.2
0.0
500
400
300
200
100
0
0
100
200
300
400
500


500
400
300
200
100
0
0
100
200
300
400
500


500
400
300
200
100
0
0
100
200
300
400
500

Observations:

The exponent η affects the virus growth pattern. With $\eta = 0$, the probability of new virus appearance is the same everywhere, resulting in a symmetrical pattern. With $\eta = 1$, the probability of new virus appearance is proportional to the available food concentration, which is reasonable in nature. The pattern typically exhibits a few branching structures.

With $\eta = 2$, the probability is proportional to the square of the food concentration. Regions with higher food concentration have a significantly higher probability of new virus appearance. Consequently, the virus growth pattern exhibits fewer branching structures.

V Conclusion:

The report provides an overview of the principles and algorithms used in the DLA simulation method. It discusses the concepts of diffusion-limited aggregation and its applications in various fields. The theoretical aspects, including the theorem statement and the SOR method, are explained in detail. The implementation section covers the data structure, program structure, and graph plotting techniques employed in the simulation. The report concludes with the experimental results obtained from running the simulation and provides evaluations and conclusions drawn by the team based on these results.

VI Reference

References

[1] Nguyen Khanh Duy. *DLA Virus Development*. Dec. 2022. URL: https://github.com/khanhnguyenduy147/DLA-Virus-Development/tree/main.

[2] LewisElshan Shaloy. *Solving the steady and unsteady 2D heat conduction equations.: Skill-lync*. Dec. 2022. URL: https://skill-lync.com/student-projects/week-5-mid-term-project-solving-the-steady-and-unsteady-2d-heat-conduction-problem-26.