IIT Bombay

IE 616 Project

Analysis and Simulation of the Evolutionary Behaviour of Spatial Prisoner's Dilemma

---

Student Name	Student ID
1. Nahush Rajesh Kolhe	210260034
2. Shravya Suresh	210260046
3. Aman Sharma	210100011

Lecturer in charge:

Prof. Urban Larsson

Date : 25/04/23

Contents

1 Introduction **2**
1.1 General Matrix Games 2
1.2 The Prisoner's Dilemma 2
1.3 Evolutionary Game Theory 3
1.4 The Spatial Prisoner's Dilemma 3

2 Methodology **3**
2.1 Setting Up 3
2.2 Main loop 4
2.3 Varying some parameters 4

3 Data Visualization **5**
3.1 Variation 1 5
3.2 Variation 2 6
3.3 Variation 3 7

4 Results and Conclusion **8**

5 Appendix: Code **9**
5.1 Variation 1 9
5.2 Variation 2 12
5.3 Variation 3 14

Abstract

Matrix games are an informative method of representing a decision-making scenario between two people. Of these, the Prisoner's Dilemma is the most popular one, representing a trade-off between optimisation and equivalence in the outcome of each individual involved in the scenario. General matrix games can be extended into real-life applications to better understand the world around us. A very common utility of such games is to understand the evolutionary nature of a system. In this project, we analyse and simulate a variant of the Prisoner's Dilemma, known as Spatial Prisoner's Dilemma.

1 Introduction

1.1 General Matrix Games

When a game played between two people uses a fixed matrix that determines the 'winner' between the two players, it is called a 'matrix game'.

1.2 The Prisoner's Dilemma

Consider this situation: Persons A and B have been convicted on the charges of robbery. Both of them are taken in for interrogation, and are made to sit in two different rooms. The interrogating officer tells each of them that if they produce evidence against the other person (i.e. *defect* against the other person), they will be allowed to walk free, without serving any time in jail. However, if they do not confess the truth (i.e. remain in *confidence* with the other person), then they will have to serve some amount of time in jail.
Given below is the payoff matrix for the above scenario.

		B	
		Confidence	Defect
A	Confidence	(1, 1)	(10, 0)
	Defect	(0, 10)	(5, 5)

The entries in the table, (A, B), indicate the number of years that A and B respectively have to serve in jail. For example, if A remains in confidence and B defect, the outcome is (10, 0). This means that A will be given a sentence of 10 years, while B will walk free. These respective entries in the matrix are referred to as the 'payoff' of A and B respectively. Overall, in agreement with these payoff values, this matrix represents the Prisoner's Dilemma.

A prisoner's dilemma is a situation where individual decision-makers always have an incentive to choose in a way that creates a less than optimal outcome for the individuals as a group. In a rational play of prisoner's dilemma, the individuals receive the greatest payoffs if they betray the other and defect, rather than co-operate with them and remain in confidence. However, in this case, the better option for both individuals would be to cooperate with each other and remain in confidence so they get to serve a shorter time in jail. As deduced from the payoff matrix, (1, 1) is the better option than (5, 5). Thus, both parties receive the highest reward only if they cooperate with each other, unlike the expected rational play.

To understand this better, we refer to the concept of a Nash Equilibrium. The Nash equilibrium is a decision-making concept that states that an individual can achieve their most desired outcome by playing their dominant strategy. If we utilise this idea on the payoff matrix shown earlier, then we obtain the Nash equilibrium at (5, 5). This is in agreement with the idea that was discussed earlier; by rational play, each individual will want to choose a strategy that helps them maximise their own payoff.

However, upon closer, intuitive analysis, we observe that there is a better alternative for both the individuals in this situation. Rather than playing their dominant strategies, if both the individuals cooperate with each other and remain in confidence, then they serve a much shorter time in jail as compared to when they defect against each other. This is contrary to the idea of adopting the Nash equilibrium pathway, but it benefits both the individuals in this scenario.

1.3 Evolutionary Game Theory

Evolutionary game theory originated as an application of the mathematical theory of games to biological contexts. But now, it has become of increased interest to people from all walks of life - economists, sociologists, anthropologists, as well as philosophers. Although it has provided numerous insights to particular evolutionary questions, it has the potential to address a number of deficiencies in the traditional theory of games. For starters, it effectively addresses the equilibrium selection, generally achieved through the concept of a Nash Equilibrium.
In the context of our project, this evolutionary concept is displayed by a variation of the Prisoner's dilemma, known as the Spatial Prisoner's dilemma.

1.4 The Spatial Prisoner's Dilemma

Over the last decades, the prisoner's dilemma game has been adopted in a variety of studies which seek to explore and resolve the dilemma of cooperation. In this project, we expand upon this idea to analyse its evolutionary behaviour over a spatial format. The main idea behind Spatial Prisoner's Dilemma is this: each individual element in the given space will adopt the most optimal strategy amongst its neighbours. Each individual will play the Prisoner's dilemma with all its neighbours, and out of all the games played, it will choose the one that yields the highest payoff and adopt that strategy. Thus, depending upon the payoff matrix defined for each game, simulating this situation results in some very interesting patterns.

2 Methodology

2.1 Setting Up

We have implemented this simulation using python. First we define a space which has a grid of 51x51 squares. These squares represent players playing the game and their strategies are represented by the colour of that square. Red and Blue corresponds to *Defect* and *Confidence* Strategies respectively.

Now, we define a strategy matrix called "*strat*" whose value corresponds to the strategies of a particular player. Numbers 1 and 0 corresponds to *defect* and *confidence* respectively. This matrix is of size 51x51. Similarly we define another matrix called "*newstrat*" which will contain the updated strategies of both the players. These strategies are initialized so that all the values are 0 except at the centre square of the "*strat*" matrix. This represents that all of them have *confidence* strategies except for the one who is in the middle.

Now we define the payoff matrix as

		B	
		Confidence	Defect
A	Confidence	(0, 0)	(1.37, 0)
	Defect	(0, 1.37)	(1, 1)

2.2 Main loop

Now we start the main loop of the game. Here we first define a 51x51 matrix called "*currscore*" which will contain the score of each player in a given iterate. We initialize it to 0 at the beginning of every loop.

The game is now being played among the neighbours. Here, each square plays the game with its neighbours. The neighbours of a square are the square to the immediate left and right, and immediately above and below it. This way, each player plays 4 games and the payoffs of each games are summed and stored into *currscore*.

Having done this, we then have to update the strategy of each player. The condition to do so is this: every player will look at the score of its neighbours. It will determine the highest score, and copy the strategy of that neighbour. This strategy will be stored into *newstrat*. After iterating over all the players, we copy the *newstrat* matrix into the *strat* matrix so that we can update all the players simultaneously rather than in real time by giving some bias.

Finally, we update our canvas representing the game using appropriate colors for the strategies, and end our main loop. Upon running this code, we observe various beautiful patterns evolving in the canvas.

2.3 Varying some parameters

Let **Variation 1** represent the procedure mentioned above. Then:

Variation 2 : In this variation we change the payoff matrix to a new one as:

		B	
		Confidence	Defect
A	Confidence	(1, 1)	(5, 0)
	Defect	(0, 5)	(3, 3)

Variation 3 : In this variation, we first change the initial strategy. Now the inital strategy of each player will be randomly alloted as 0 or 1.

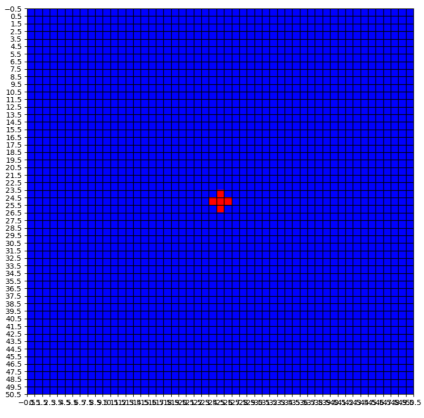
Then we introduce a factor of history in our game, which means that we will now keep track of the history of the scores of previous games. For this, we increase a dimension of the *currscore* array. Now it's a 51x51x10 dimensional array where the third dimension stores the scores of the 10 most recent games. So in our updation step, instead of comparing the *currscore* of players, we compare the sum of *currscore* of the past 10 games. In variation 1 we used to initialize *currscore* to 0 at each iterate, but now we remove that condition.

In this way, we have now implemented history into the game, and now the players will make choice based on all the 10 previous games instead of just the previous game.

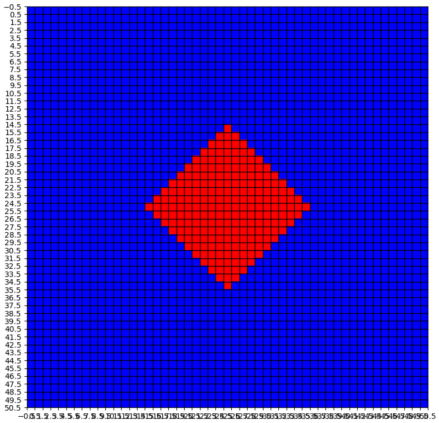
3 Data Visualization

Following are the simulated results obtained for each variation:

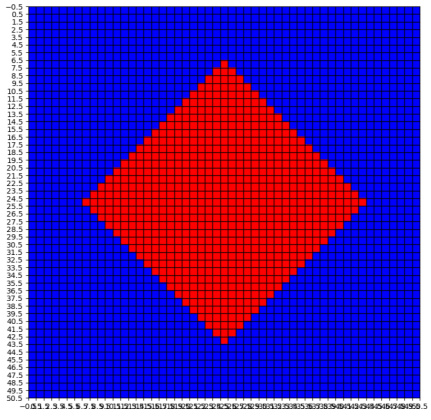
3.1 Variation 1



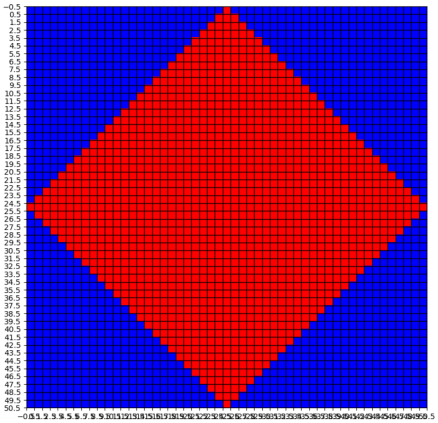
(a)



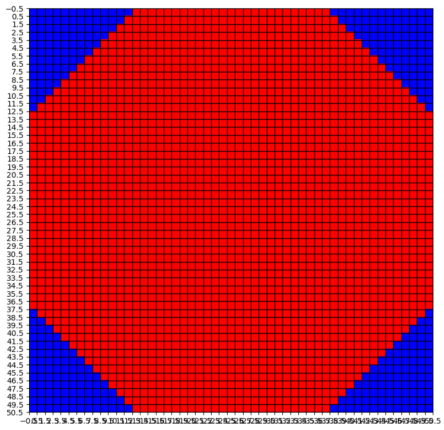
(b)



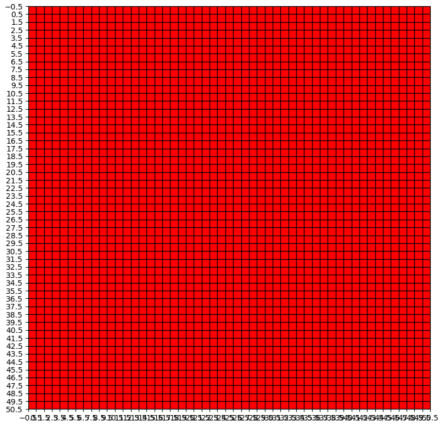
(a)



(b)

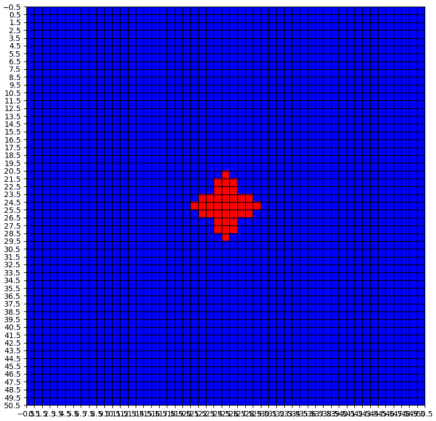


(a)

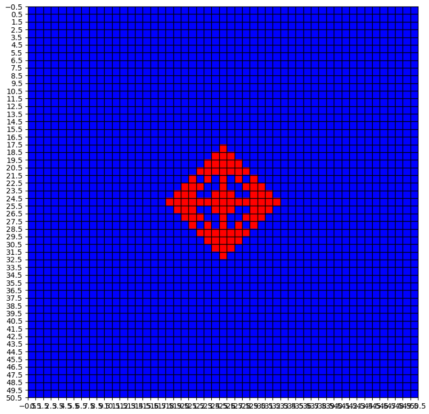


(b)

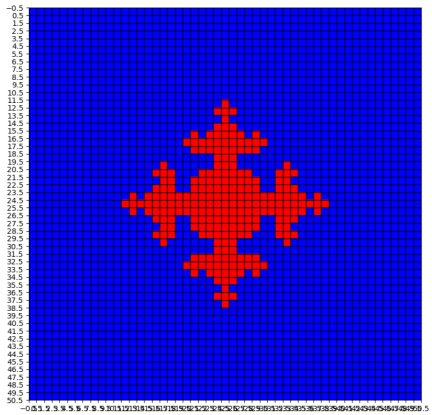
3.2 Variation 2



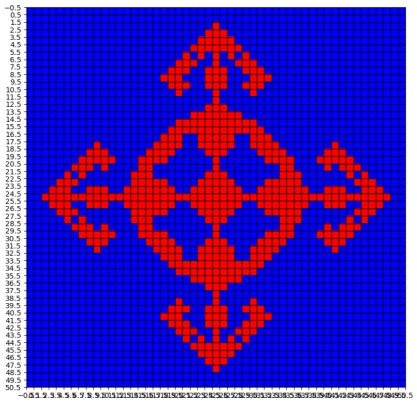
(a)



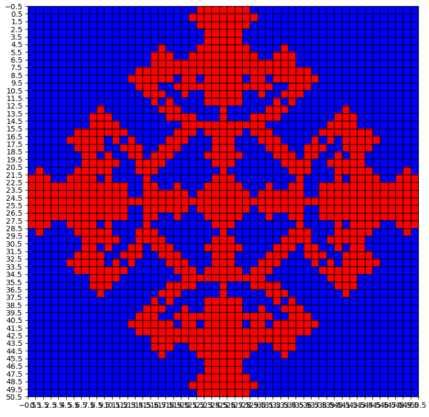
(b)



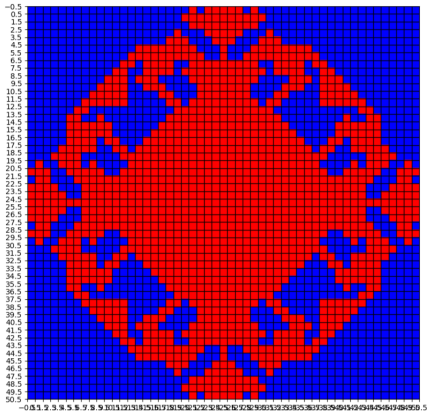
(a)



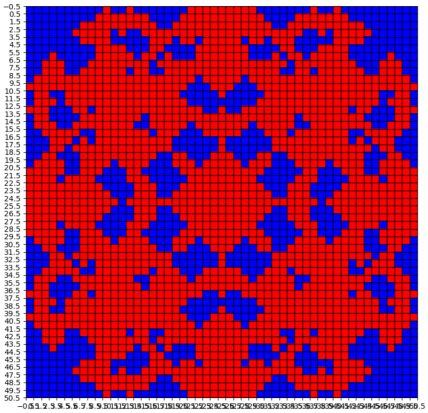
(b)



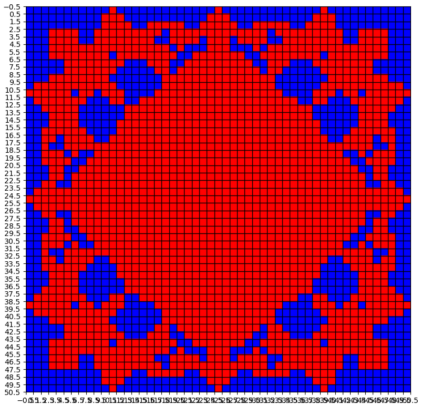
(a)



(b)

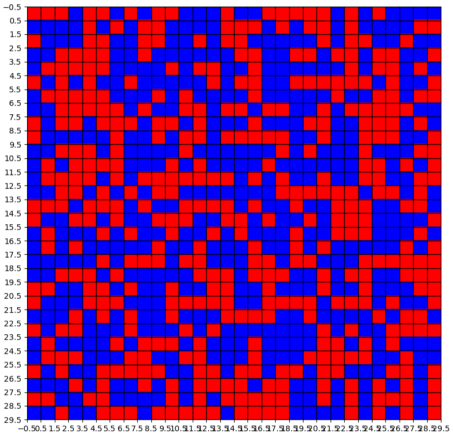


(a)

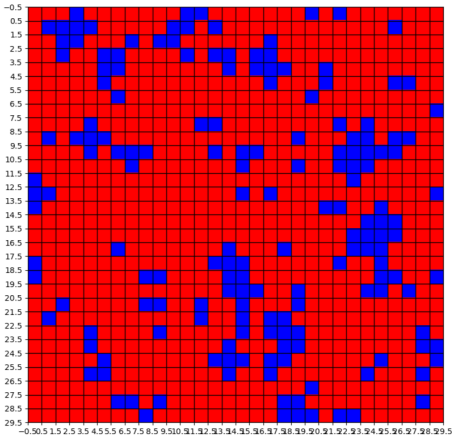


(b)

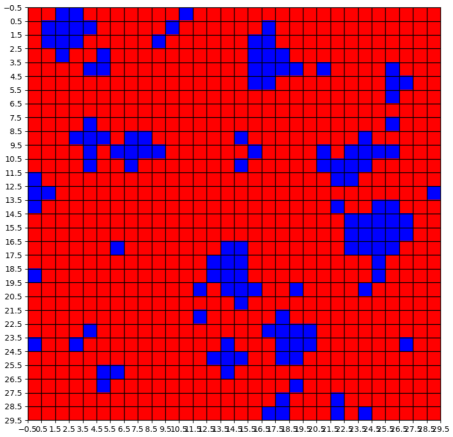
3.3 Variation 3



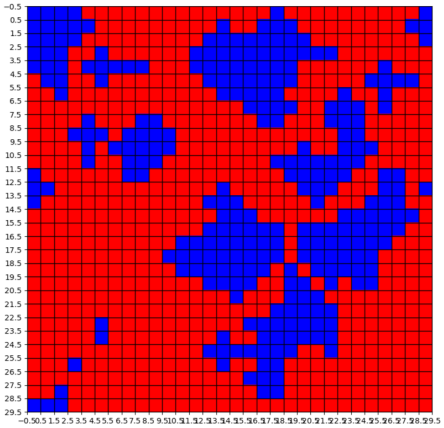
(a)



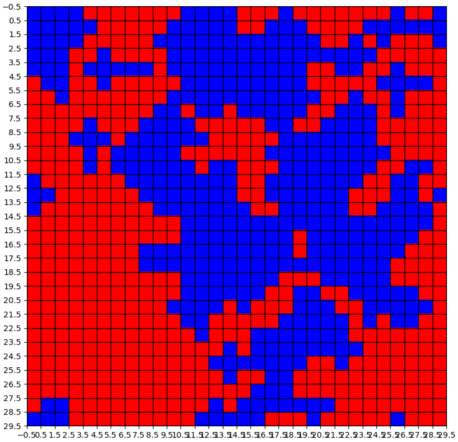
(b)



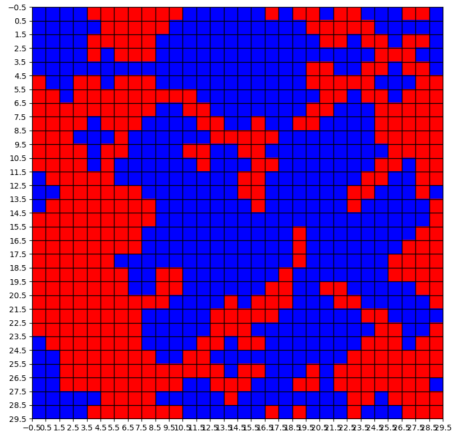
(a)



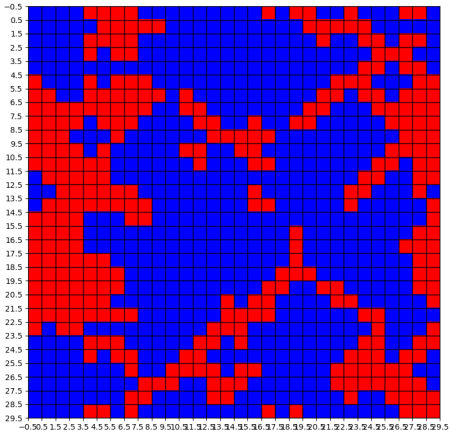
(b)



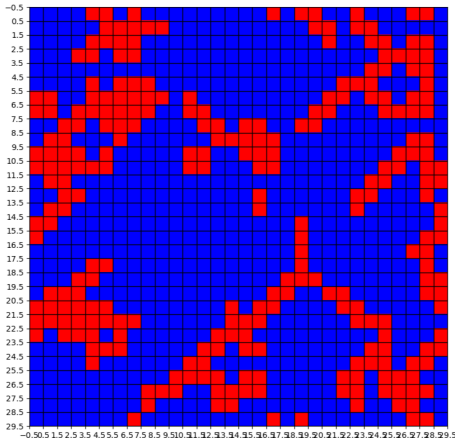
(a)



(b)



(a)



(b)

4 Results and Conclusion

Variation 1 : We observe that the red strategy spreads in all the directions evenly. This variation doesn't create patterns, here red just dominates all the blues and eventually all of them become red. This represents the Nash Equilibrium of the game where both the players follow the *defect* strategy. Note that this will be true for any initial strategy.

Variation 2 : We observe that the red strategy spreads in all the directions creating a very peculiar pattern. This pattern is of fractals. Hence, the red invades the blue and captures most of it but it doesn't fully convert all of them to blue. This is because the payoff matrix that we have chosen is such that we can see this pattern.

Variation 3 : We observe that initially, the strategies are randomly distributed as expected. At the very start, red quickly starts to spread but this doesn't last long. Blues then starts to dominate the red in a chaotic pattern where groups of blue stick together and expand their territory. After long time we observe that most of the players are blue and this continues. This represents the optimal strategy of the game where both the players cooperate and remain in *confidence* to get a higher payoff than when both of them *defect*. This can be explained as follows: when we introduce history, players make more informed choices as they account for the past experiences of their neighbours. This way, they learn that if both of them are blue, they will get a higher payoff than when both are red. At the very start of the game, they do not have any history of the past games. Hence red was dominating. But as soon as some amount of history was accumulated, the blues eventually dominated. This is thus a pretty accurate representation of evolutionary nature, where the most optimal features survive in the long run.

5 Appendix: Code

5.1 Variation 1

```
1
2  import matplotlib.pyplot as plt
3  import numpy as np
4  from matplotlib import colors
5
6  gameSize = 31
7
8  strat = np.random.randint(0,2,size=(gameSize,gameSize))
9  newstrat = np.zeros((gameSize, gameSize))
10 #strat[gameSize // 2][gameSize // 2] = 1
11 #strat[gameSize // 2][gameSize // 2 +1] = 1
12 #strat[gameSize // 2 +1][gameSize // 2] = 1
13 #strat[gameSize // 2 +1][gameSize // 2 +1] = 1
14
15 cmap = colors.ListedColormap(['blue', 'red'])
16 bounds = [0,0.5,1]
17 norm = colors.BoundaryNorm(bounds, cmap.N)
18
19
20 fig, ax = plt.subplots()
21 image = ax.imshow(strat, cmap = cmap, norm=norm)
22
23 ax.grid(which = 'major', axis = 'both', linestyle='-',
       color='k', linewidth=1)
```

```
24 ax.set_xticks(np.arange(-0.5, gameSize, 1))
25 ax.set_yticks(np.arange(-0.5, gameSize, 1))
26
27 k=1
28 #payoff_matrix_r = np.array([[1*k, 0*k],[2*k, 0*k]])
29 payoff_matrix_r = np.array([[1, 0],[1.37, 0]])
30 #payoff_matrix_r = np.array([[-1, -10],[0, -5]])
31
32 #currscore = np.zeros((20, 20))
33 newstrat = np.zeros((gameSize, gameSize))
34 while True:
35     print(strat)
36     currscore = np.zeros((gameSize, gameSize))
37     newstrat = np.zeros((gameSize, gameSize))
38     #neighbours =[(0,1),(0,-1),(1,0),(-1,0)]
39     h=10
40     neighbours = []
41     for j in range(-h, h-1):
42             for k in range(-h, h-1):
43                 neighbours.append((j,k))
44
45     for i in range(gameSize * gameSize):
46         row = i // gameSize
47         col = i % gameSize
48         length = 20
49
50         for j in range(length):
51             opponent = neighbours[np.random.randint(0,len(
                neighbours))]
52             currscore[row][col] += payoff_matrix_r[int(
                strat[row][col]), int(strat[(row + opponent
                [0]) % gameSize][(col + opponent[1]) %
                gameSize])]
53
54     for i in range(gameSize * gameSize):
55         row = i // gameSize
56         col = i % gameSize
57
58         max_score = -1000
59         max_x = 0
60         max_y = 0
61
62         for j in range(-1, 2):
63             for k in range(-1, 2):
64                 if (j != k and j != -k) or (j == 0 and k
                    == 0):
65                     if  (max_score < currscore[(j + row) %
                          gameSize][(k + col) % gameSize])
                        or (max_score == currscore[(j + row
                        ) % gameSize][(k + col) % gameSize]
```

```
                                and strat[(j + row) % gameSize][(k
                                + col) % gameSize] == 1):
66
67                                  max_score = np.copy(currscore[(j +
                                        row) % gameSize][(k + col) %
                                        gameSize])
68                                  max_x = (j + row) % gameSize
69                                  max_y = (k + col) % gameSize
70
71              newstrat[row][col] = np.copy(strat[max_x][max_y])
72
73          strat = np.copy(newstrat)
74          image.set_data(strat)
75          fig.canvas.draw_idle()
76          plt.pause(0.5)
```

5.2 Variation 2

```
1      import matplotlib.pyplot as plt
2  import numpy as np
3  from matplotlib import colors
4
5  gameSize = 51
6
7  strat = np.zeros((gameSize, gameSize))
8  newstrat = np.zeros((gameSize, gameSize))
9  strat[gameSize // 2][gameSize // 2] = 1
10 #strat[gameSize // 2][gameSize // 2 +1] = 1              for
       a square initial starting position
11 #strat[gameSize // 2 +1][gameSize // 2] = 1
12 #strat[gameSize // 2 +1][gameSize // 2 +1] = 1
13
14 cmap = colors.ListedColormap(['blue', 'red'])
15 bounds = [0,0.5,1]
16 norm = colors.BoundaryNorm(bounds, cmap.N)
17
18
19 fig, ax  = plt.subplots()
20 image = ax.imshow(strat, cmap = cmap, norm=norm)
21
22 ax.grid(which = 'major', axis = 'both', linestyle='-',
       color='k', linewidth=1)
23 ax.set_xticks(np.arange(-0.5, gameSize, 1))
24 ax.set_yticks(np.arange(-0.5, gameSize, 1))
25
26 k=1
27 payoff_matrix_r = np.array([[1*k, 0*k],[1.3*k, 0*k]])
28 #payoff_matrix_r = np.array([[1, 0],[1.45, 0]])
29 #payoff_matrix_r = np.array([[-1, -10],[0, -5]])
30
31 #currscore = np.zeros((20, 20))
32 newstrat = np.zeros((gameSize, gameSize))
33 while True:
34      print(strat)
35      currscore = np.zeros((gameSize, gameSize))
36      newstrat = np.zeros((gameSize, gameSize))
37
38      for i in range(gameSize * gameSize):
39          row = i // gameSize
40          col = i % gameSize
41
42          for j in range(-1, 2):
43              for k in range(-1, 2):
44                  if k != j and k != -j:
45                      currscore[row][col] += payoff_matrix_r
                           [int(strat[row][col]), int(strat[(
```

```
                        row + j) % gameSize][(col + k) %
                        gameSize])]
46                    # if row == 10 and col == 12 :
47                    #     print(j, k, currscore[row][col])
48
49      for i in range(gameSize * gameSize):
50          row = i // gameSize
51          col = i % gameSize
52
53          max_score = -1000
54          max_x = 0
55          max_y = 0
56
57          for j in range(-1, 2):
58              for k in range(-1, 2):
59                  if (j != k and j != -k) or (j == 0 and k
                        == 0):
60                      if  (max_score < currscore[(j + row) %
                            gameSize][(k + col) % gameSize])
                            or (max_score == currscore[(j + row
                            ) % gameSize][(k + col) % gameSize]
                            and strat[(j + row) % gameSize][(k
                            + col) % gameSize] == 1):
61
62                          max_score = np.copy(currscore[(j +
                                row) % gameSize][(k + col) %
                                gameSize])
63                          max_x = (j + row) % gameSize
64                          max_y = (k + col) % gameSize
65
66          newstrat[row][col] = np.copy(strat[max_x][max_y])
67
68      strat = np.copy(newstrat)
69      image.set_data(strat)
70      fig.canvas.draw_idle()
71      plt.pause(0.01)
```

5.3 Variation 3

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import colors

gameSize = 30

strat = np.random.randint(0,2,size=(gameSize,gameSize))
newstrat = np.zeros((gameSize, gameSize))
strat[gameSize // 2][gameSize // 2] = 1
#strat[gameSize // 2][gameSize // 2 +1] = 1
#strat[gameSize // 2 +1][gameSize // 2] = 1
#strat[gameSize // 2 +1][gameSize // 2 +1] = 1

cmap = colors.ListedColormap(['blue', 'red'])
bounds = [0,0.5,1]
norm = colors.BoundaryNorm(bounds, cmap.N)

fig, ax = plt.subplots()
image = ax.imshow(strat, cmap = cmap, norm=norm)

ax.grid(which = 'major', axis = 'both', linestyle='-',
    color='k', linewidth=1)
ax.set_xticks(np.arange(-0.5, gameSize, 1))
ax.set_yticks(np.arange(-0.5, gameSize, 1))

k=1
#payoff_matrix_r = np.array([[3*k, 0*k],[5*k, 1*k]])
payoff_matrix_r = np.array([[1, 0],[1.37, 0]])
#payoff_matrix_r = np.array([[-1, -10],[0, -5]])

histlen = 10

currscore = np.zeros((gameSize, gameSize, histlen))
#currscore = np.zeros((20, 20))
newstrat = np.zeros((gameSize, gameSize))
i=0
while True:
    print(strat)
    #currscore = np.zeros((gameSize, gameSize))
    newstrat = np.zeros((gameSize, gameSize))
    h = 100

    for i in range(gameSize * gameSize):
        row = i // gameSize
        col = i % gameSize

        for j in range(-1, 2):
            for k in range(-1, 2):
```

```
48                  if k != j and k != -j:
49                      currscore[row][col][0] +=
                            payoff_matrix_r[int(strat[row][col
                            ]), int(strat[(row + j) % gameSize
                            ][(col + k) % gameSize])]
50                      # if row == 10 and col == 12 :
51                      #     print(j, k, currscore[row][col])
52
53          print(currscore[row][col])
54
55      for i in range(gameSize * gameSize):
56          row = i // gameSize
57          col = i % gameSize
58
59          max_score = -1000
60          max_x = 0
61          max_y = 0
62
63
64          for j in range(-1, 2):
65              for k in range(-1, 2):
66                  if (j != k and j != -k) or (j == 0 and k
                        == 0):
67                      if  (max_score < np.sum(currscore[(j +
                            row) % gameSize][(k + col) %
                            gameSize])) or (max_score == np.sum
                            (currscore[(j + row) % gameSize][(k
                            + col) % gameSize]) and strat[(j +
                            row) % gameSize][(k + col) %
                            gameSize] == 1):
68
69                          max_score = np.copy(np.sum(
                                currscore[(j + row) % gameSize
                                ][(k + col) % gameSize]))
70                          max_x = (j + row) % gameSize
71                          max_y = (k + col) % gameSize
72
73          newstrat[row][col] = np.copy(strat[max_x][max_y])
74
75      for i in range(gameSize):
76          for j in range(gameSize):
77              k = histlen - 1
78              while k > 0:
79                  currscore[i, j, k] = currscore[i, j, k -
                        1]
80                  k -= 1
81
82              currscore[i, j, 0] = 0
83
84
```

```
85        strat = np.copy(newstrat)
86        image.set_data(strat)
87        fig.canvas.draw_idle()
88        plt.pause(0.01)
```

References

[1] Investopedia

[2] Nature Magazine

[3] Stanford Education