COMPUTATIONAL ECONOMICS

Revised Edition

Cellular Automata in MATLAB

David A. Kendrick
Ruben P. Mercado
Hans M. Amman

1 Introduction

Cellular Automata originated in 1940s, from the pioneering work of Stanislaw Ulam and John von Neumann (von Neumann, 1966). From the 1980s on, research in this area received a big push from the work of Stephen Wolfram, whose 2002 book A New Kind of Science presents an extensive treatment of the topic, specially of one-dimensional Cellular Automata.

Cellular Automata are a modelling approach focused on the generation of aggregate complex behavior starting from very simple rules. They are a useful way to simulate and visually represent phenomena involving many interdependent units, where time evolves in a discrete way and where the notion of distance (be that of a spatial or relational nature) is relevant. As economies are systems of interdependent units located in space, Cellular Automata have found several economic applications. For example, they have been used to study the process of spatial racial segregation (Schelling, 1969), pricing in a spatial setting (Keenan and O'Brien, 1993), the diffusion of innovations (Bhargava, Kumar and Mukherjee, 1993; Fuks and Boccara, 1995) and interactions between consumers (Rouhaud, 2000). Also, Cellular Automata have been used to study evolutionary games (Nowak and May, 1993)[1].

In this chapter we present examples that build on the works of Wolfram and of Nowak and May. Following Wolfram (2002) and as a way of gaining familiarity with the concepts and workings of Cellular Automata, we present general examples of one-dimensional Cellular Automata. Later, following Nowak and May (1993), we present a spatial evolutionary game in a two-dimensional Cellular Automaton. In Chapter 13 (Genetic Algorithms and Evolutionary Games in MATLAB) we analyzed the Prisoner's Dilemma evolutionary game. Here we introduce a similar game but adding a spatial dimension.

[1] Cellular Automata share common traits yet have differences with Agent-Based Models such as the one we will present in Chapter 17 (Agent-Based Model in MATLAB). As we will see in this chapter, in Cellular Automata cells have fixed neighborhoods (the state of the cell can change, but its position is fixed) while in Agent-Based Models agents are free to move so that their nearest neighbors vary with time (state and position may change). Usually, Cellular Automata follow only a few and simple rules to update the state of each cell, depending on the state of their immediate neighbors, while in Agent-Based Models agents follow more complex rules of behavior. Thus, Cellular Automata are computationally simpler. As in the model we will see in Chapter 17, sometimes the environment of an Agent-Based Model is represented with a Cellular Automata. For a presentation and discussion of the analogies and differences of Cellular Automata and Agent-Based Models as modelling approaches, see Clarke (2014).

Figure 1: An 11-cell Cellular Automaton

2 A one-dimensional Cellular Automaton

A Cellular Automaton (CA) is a model of a dynamic system evolving in discrete steps. A CA can be represented with a grid, consisting of cells. Each cell of the CA can be in a finite number of states.

Figure 1 shows a one-dimensional CA represented by a 11-cell grid. In this example, each cell can be in one of two states (black or white). The central cell is in the black state, while all the other cells are in the white state.

Let's define a neighborhood of a cell as composed by its two most adjacent cells. For example, the neighborhood of the black central cell, that is cell number 6 counting from left to right, is made of cells number 5 and 7, both in the white state. The neighborhood of cell number 3, which is in the white state, is made of cells 2 and 4, both in the white state also.

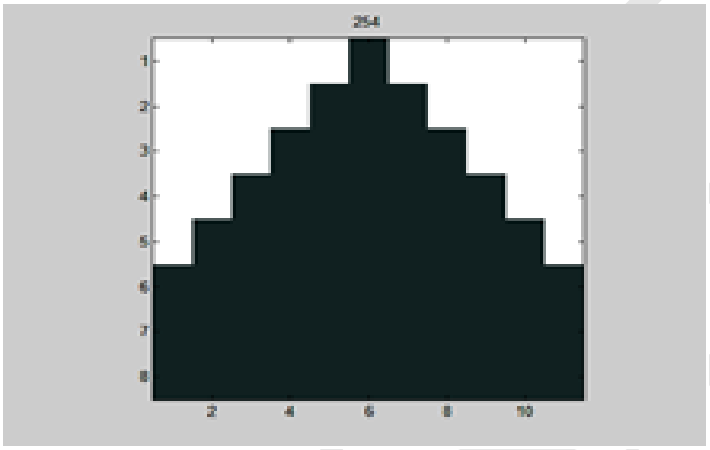
What about the neighborhoods of cells 1 and 11? If we define this one-dimensional CA as one with fixed boundary conditions, cells 1 and 11 will have only one neighbor (cells 2 and 9 respectively). However, if we define our CA as one with periodic boundary conditions, cell 1 will have cells 2 and 11 as its neighbors, while cell 11 will have cells 10 and 1 as its neighbors, so that the CA will have a circular representation.

Starting from an initial state of the CA, we can generate the next state or generation by applying a transition rule. This rule will tell us how to establish the next state of each cell as a function of its present state and, also, of the states of its neighboring cells. We repeat this process iteratively, to obtain the dynamic evolution of the CA.

Let's consider the following rule: each cell will be in the black state if itself or any of its neighbors was in the black state in the previous step of the CA. As we will see later, this rule corresponds to rule 254 in Wolfram's classification. Figure 2 shows its graphic representation.

Figure 2: A transition rule

Figure 3: Dynamic evolution of a CA



If we start, for example, from an initial condition such as the one depicted in Figure 1 (where all cells are white, except the central cell which is black), and we iterate the rule 8 times, we obtain the dynamic evolution of our one-dimensional CA as shown in Figure 3.

There are many possible rules like the one depicted in Figure 2, in which the transition of each cell depends on its previous state and on the state of its two immediate neighbors. In fact, there are exactly 256 possible rules. To see this, remember that the next state of a given cell depends of the previous states of three cells: itself and its two neighbors. Since each of these three cells can be in one of two states only (black or white), we obtain that there are 2 x 2 x 2 = 8 possible binary combinations for the three cells. And each of these 8 possible combinations is associated to a transition to a new cell taking one of two states (black or white), so that there are a total of $2^8 = 256$ possible rules.

Wolfram provides a method to associate a number to each rule. Assigning the value 1 to the black state, and 0 to the white state, from Figure 2 we obtain:

The lower row is the binary representation of the rule. The number 11111110

Figure 4: Binary representation of Figure 2

1	1	1	1	1	0	1	0	1	1	0	0	0	1	1	0	1	0	0	0	1	0	0	0
	1			1			1			1			1			1			1			0	

Figure 5: Wolfram's rule 90

1	1	1	1	1	0	1	0	1	1	0	0	0	1	1	0	1	0	0	0	1	0	0	0
	1			1			1			1			1			1			1			0	

is an eight-bit binary number. To transform this number into a decimal number, we multiply each bit by the corresponding power of 2, and we obtain as a result the decimal number 254:

$1x2^7 + 1x2^6 + 1x2^5 + 1x2^4 + 1x2^3 + 1x2^2 + 1x2^1 + 0x2^0 =$
$1x128 + 1x64 + 1x32 + 1x16 + 1x8 + 1x4 + 1x2 + 0x1 = 254$

Thus, in Wolfram's notation, the rule represented in Figures 2 and 4 is rule number 254. The binary representation of rule 0 will be 00000000, while the one corresponding to rule 255 will be 11111111. For rule 90, we would have:

3 One-dimensional Cellular Automata in MATLAB

In this section we describe the MATLAB code for a one-dimensional CA. The first step in the code is to create a matrix representing the grid of the CA. To do so, the number of rows and columns is established, and all the elements of the matrix, named grid, are set as equal to zero, that is, all the cells of the CA take the white color. The clear all command at the beginning of the code tells MATLAB to erase all the stored information from the previous run, so that the new run starts afresh.

```
clear all;
nrows=8;
ncolumns=11;
grid(nrows,ncolumns)=0;
```

Next, the initial condition of the CA is specified. For example, if we want to start with all cells in the white color, except the central cell, which will be the black color, we proceed as follows. First, we create a variable named initposition, divide the number of columns by 2, and then, using the MATLAB ceil function, assign to initposition the nearest integer greater than the result. Thus, if we have 11 columns, the result of this operations will be the number 6.

```
initposition = ceil(ncolumns/2)
```

The next command assigns the number one (the black color) to the central cell of the grid.

```
grid(1,initposition) = 1;
```

The second step in the code is to establish the decimal number corresponding to the transition rule we want to simulate, then transform that number into its binary equivalent. To do this, we make use of the dec2bin function, which converts a decimal number into a binary string. In the example below, dec2bin converts the number contained in the variable rulenumber into a binary string with at least 8 characters and assigns the result to the rulebin variable.


```
rulenumber=254;
rulebin=dec2bin(rulenumber,8);
```

Finally, we create an eight-element vector named r, each of its elements containing one bit of the binary representation of the rule. To do so, we make use of the str2num command, which converts a character array or string scalar to a numeric matrix or vector. The i ndex allows us to select each element of the string contained in rulebin, and assign its content to each element of the r variable.

```
for i=1:8
    r(i)=str2num(rulebin(i));
end
```

The third step in the code is to apply the rule to each element of the grid to obtain the next state or generation of the CA, then iterate the procedure as many times as the number of rows in the grid. We do so with the two loops shown below.

```
for i=1:nrows-1;
    for j=1:ncolumns;

   % implement periodic boundary conditions
        if ((j-1) < 1) topleft = grid(i,ncolumns);
            else topleft = grid(i,j-1); end

        top = grid(i,j);

        if ((j+1) > ncolumns) topright = grid(i,1);
            else topright = grid(i,j+1); end

    % apply the rule
    if (topleft==1 & top==1 & topright ==1) grid(i+1,j) = r(1); end
    if (topleft==1 & top==1 & topright ==0) grid(i+1,j) = r(2); end
    if (topleft==1 & top==0 & topright ==1) grid(i+1,j) = r(3); end
    if (topleft==1 & top==0 & topright ==0) grid(i+1,j) = r(4); end
    if (topleft==0 & top==1 & topright ==1) grid(i+1,j) = r(5); end
    if (topleft==0 & top==1 & topright ==0) grid(i+1,j) = r(6); end
    if (topleft==0 & top==0 & topright ==1) grid(i+1,j) = r(7); end
    if (topleft==0 & top==0 & topright ==0) grid(i+1,j) = r(8); end
    end
end
```

Remember that, as we mentioned earlier in the chapter, a CA can have different boundary conditions. For our example, we choose to implement a periodic boundary condition. This is done in the first part of the loop. We check if the column we are dealing with is the first one or the last one in the grid. If that is the case, we change the value of the column index j so that it identifies the corresponding neighbor.

Remember also that the next state of each cells depends on its present state and the states of its two immediate neighbors. Looking at figures 2, 4 or 2 earlier in the chapter, we can say that the next state depends on the present state of the top left, the top and the top right cells.

Then we apply the rule to obtain the corresponding transition values. To do so, we check the values of the top left, top and topright cell, and assign to the next row cell the corresponding value for that rule, which is contained in the corresponding element of the r vector.

The fourth and final step in the code is to display the dynamic evolution of the CA, which will be contained in the matrix named grid. A simple way to do this would be to use the spy(grid) function. For our example, this function would return a figure with the display of the matrix, with each cell in black if its value equals one, or in white if its value equals zero. However, the cell markers would not look good.

A better alternative is to use the imagesc function. First, we use the colormap function, which sets the colormap of the figure we want to create. For example, we choose two colors from the bone colormap to represent each cell state. In our case they will be white and black, thus we set:

```
colormap(bone(2));
```

Since imagesc will display cells with low values as black and cells with high values as white (while in our example black corresponds to 1 and white to 0), we make the ones in the grid equal to minus one:

```
imagesc(grid * (-1));
```

Finally, we assign the rulenumber variable as the title of the figure, then specify that the axes are of equal lengths.

```
title(rulenumber);
axis square;
```

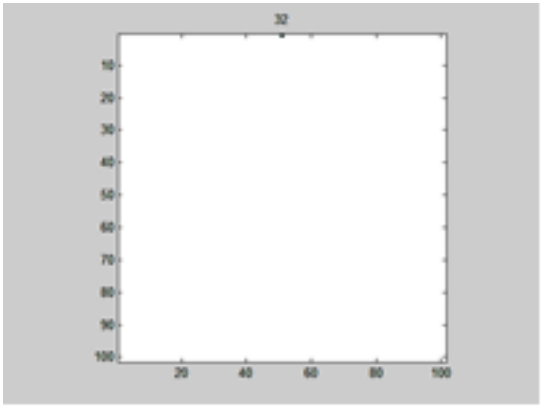
4 Some examples of one-dimensional Cellular Automata

Wolfran classified the 256 rules governing the behavior of one-dimensional Cellular Automata in four classes according with the following criteria:

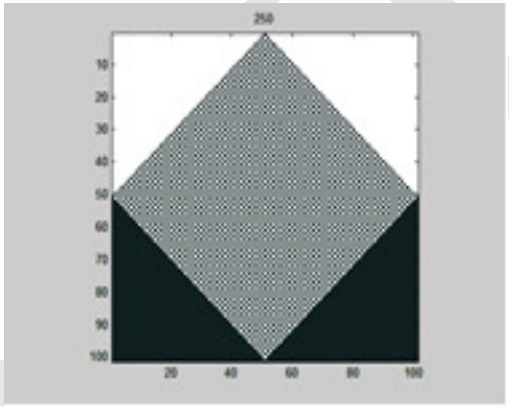
- Class 1: rapid convergence to a uniform state.
- Class 2: rapid convergence to a stable state.
- Class 3: appears to remain in a random state
- Class 4: displays areas of stable states, but also structures interacting in complex ways.

Figure 4 shows one example of each class, for a one-dimensional CA with 101 cells, and for 101 iterations.:

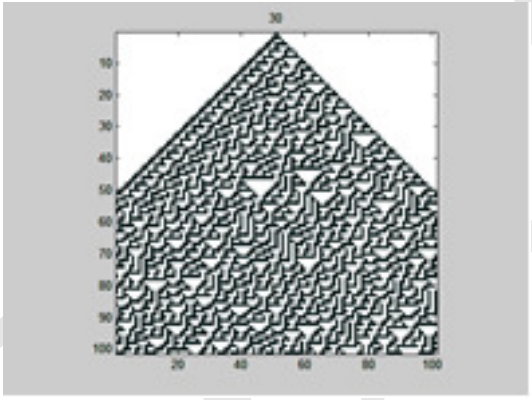
Figure 6: Examples of the dynamics one-dimensional Cellular Automata



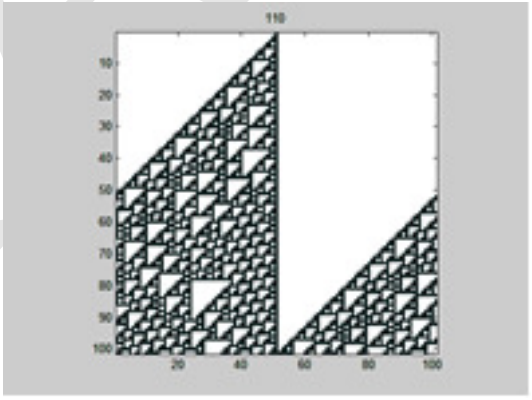
Class 1: rule 32



Class 2: rule 250

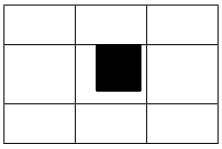


Class 3: rule 30



Class 4: rule 110

Figure 7: A player's neighborhood



5 A two-dimensional Cellular Automaton: Prisoner's Dilemma evolutionary game in spatial form

In Chapter **??** we analyzed the Prisoner's Dilemma evolutionary game. Here we introduce a similar game, but adding a spatial dimension, following the work of Nowak and May (1993).

Let's assume that we have a two-dimensional CA, made of a grid of nxn cells. Each cell is occupied by a player. At variance with Chapter xx, here we will assume that each player has a chromosome with only one gen, and that this gen can take only one of two values, representing an action in the Prisoner's Dilemma game: defect (D) or cooperate (C).

Each player plays the game against all its mmediate neighbors (one at a time) and against itself. Thus, it will play a total of nine times.

If it wins all the games, it will keep its strategy without modification for the next round of games. If not, the player will adopt, for the next round of games, the strategy of the neighbor that obtained the highest cumulative payoffs during the previous round of games. Notice that, at variance with the game in Chapter xx, here we are not assuming any combination or random mutation of chromosomes. In this game, each "generation" of players inherits its gens (its actions) from the previous generation, according to the procedure just described. And notice also that that procedure describes, in a way, the transition rule of the Cellular Automaton. Finally, the dynamic evolution of the CA will be given by the changes in the two-dimensional grid resulting from the successive rounds of games.

Depending on the structure of payoffs, this game generates a rich pattern of evolutionary dynamics of the CA. Figure 5 shows a Prisoner's Dilemma game with a structure of payoffs such as the one studied by Nowak and May.

Figure 8: Prisoner's Dilemma matrix game

		Player 2	
		D	C
Player 1	D	0, 0	b, 0
	C	0, b	1, 1

Depending on the value of the b parameter, different patterns of behavior emerge. We will study the dynamic evolution of a CA with a 49x49 grid, for 25 rounds of games. We will assume that the game begins with a generation of all cooperators, except the player located in the center of the grid, which is a defector. We will assume that the CA has periodic boundary conditions. Also, we will use the following coloring code for each player, depending on its actions in the previous and in the present generation:

- Blue: was a cooperator and is still a cooperator
- Yellow: was a cooperator and is now a defector
- Red: was a defector and is still a defector
- Green: was a defector and is now a cooperator

The amount of yellow and green color will indicate how many cells change from one generation to the next, while blue and red patterns will indicate no change.

6 The spatial game in MATLAB

The MATLAB representation of the spatial evolutionary game consists of a main program named spatialgame.m and several functions. Below is the code of the main program. The action code for each player will be 1 for cooperation and 0 for defection. Also, the MATLAB color code for each player will be as follows: 1 for blue, 2 for green, 3 for yellow and 4 for blue.

```
%Spatialgame

% Initialization of variables
```

```
clc;
clear all;
size = 49;
nruns = 16;
b = 1.85;

% Assign all cooperators to the grid
 for i = 1:size;
     for j = 1:size;
     a_str(i,j).actionpresent = 1;
     a_str(i,j).actionprevious = 1;
     a_str(i,j).color = 4;
     end
 end

% Place one defector on the center of the grid
i = ceil(size/2);
j = ceil(size/2);
a_str(i,j).actionpresent = 0;
a_str(i,j).actionprevious = 0;
a_str(i,j).color = 1;

% Begin main loop
for runs = 1:nruns;

% Display results
displayresults(a_str, size, nruns, runs);

% Play games: each player plays against its neighbors and itself
a_str = playgames(a_str, size, b);

% Adopt best action: each  adopts the action with highest accumulated
% payoff amongst its neighbors and itself
a_str = adoptbest(a_str, size);

% Adopt color: each agent compares its previous action against
% its present action and adopts the corresponding color
a_str = adoptcolor(a_str, size);

end
```

The program begins with the initialization of parameters (size of the grid, number or runs or, equivalently, generations, and value of the b parameter) and the setting of players characteristics and positions on the grid. The clc command clears the command window, while the clear all command clears the workspace, so that everything starts afresh.

To represent players, we will use a data type available in MATLAB called structure. A structure is an array with "data containers" named "fields". These fields can contain any kind of data. The following statements

```
a_str.color = 4;
a_str.accumpayoffs = 0;
```

create the simple 1x1 structure a_str containing two fields with the color code and accumulated payoffs for one player. If we use the statements

```
a_str.color(2) = 4;
a_str.accumpayoffs(2) = 0;
```

then a_str becomes a 1x2 array with two fields. Let's assume that we want to create a 49x49 grid populated by cooperators. And that we want to associate three fields to each player, containing its previous action, present action and its color. We can achieve this with the following statements

```
for i = 1 : size;
     for j = 1 : size;
     a_str(i,j).actionpresent = 1;
     a_str(i,j).actionprevious = 1;
     a_str(i,j).color = 4;
     end
end
```

To place a defector in the center of the grid, we use the following statements:

```
i = ceil(size/2);
j = ceil(size/2);
a_str(i,j).actionpresent = 0;
a_str(i,j).actionprevious = 0;
a_str(i,j).color = 1;
```

The ceil function assigns the highest nearest integer value to the result between parentheses. Then follows the main loop or the program, corresponding to the number of runs.

Inside this loop, there are four calls to the following functions: displayresults, which shows the evolution of the game; playgames, in which each player plays the game against its neighbors and against itself; adoptbest, in which each player adopts the action with highest accumulated payoffs; and, finally, adoptcolor, in which each player compares its previous action against its present action and adopts the corresponding color. From this overview of the main program we turn next to descriptions of the functions.

7 Functions

7.1 Displayresults

The displayresults function shows the evolution of the spatial game as a sequence of grids. Its code is shown below.

```
function displayresults(a_str, size, nruns, runs);
 for i = 1:size;
     for j = 1:size;
         a(i,j) = a_str(i,j).color;
     end
 end

 figure(1);
 subplot(ceil(sqrt(nruns)), ceil(sqrt(nruns)),runs),
 map = [1 0 1
        0 1 0
        1 1 0
        0 0 1];
 colormap(map);
 imagesc(a);

 axis square;
```

The function receives as inputs the data structure containing the characteristics of the players, the size and nruns initial parameters and the runs variable. It first transforms the color field of the data structure into a matrix named a, since

MATLAB does not allow to display the field directly.

The statement figure(1) tells MATLAB to display a figure with the successive grids showing the evolution of the game. Consider next the line of code:

```
subplot(ceil(sqrt(nruns)),ceil(sqrt(nruns)),runs)
```

The call to subplot divides the window into a number of panes. These statements thus allow us to display multiple images in a single figure such as the images of players' colors in successive runs of the program. The MATLAB function subplot(m,n,p) creates an axes in the pth pane of a figure divided into an m-by-n matrix of rectangular panes. For example, if we set the number of runs (nruns) parameter in the main program equal to 25, the subplot statement shown earlier - where ceil(sqrt(nruns) is the ceiling (i.e. the nearest higher integer) of the square root of the number of runs - will divide the figure into a matrix with 5 rows and 5 columns of panes to accommodate the images of the grids in successive runs.

The map matrix specifies the map of colors of the figure. To create a custom colormap in MATLAB, it is necessary to specify map as a three-column matrix of red-green-blue triplets, each row defining one color. The numbers specify the intensities of the red, green, and blue components of the color, and must be in the range [0,1]. For example, the [0 0 1] row vector defines the blue color, while the [1 0 1] vector defines the magenta color, as a combination of red and blue.

The colormap(map) statement sets the matrix map as the colormap to be used in the figure, while imagesc(a) plots the contents of matrix a as an image. Matrix a will contain numbers from 1 to 4, and each number will be associated with each row (from the first to the last) of the map matrix and thus with the corresponding color. Finally, the axis square statement specifies that the axes of each pane will be of equal length.

7.2 Playgames

This function receives as inputs the data structure and the size and b parameters and computes the accumulated payoffs for each player.

```
function a_str = playgames(a_str, size, b);
```

The function begins with a loop that creates a new data field named a_str(i,j).accumpayoffs, and sets it to zero.

```
% sets accumulated payoffs to zero

for i = 1:size;
    for j = 1:size;
        a_str(i,j).accumpayoffs = 0;
    end
end
```

Then a second loop, shown below, with indexes i and j goes over the games played by each player on the grid. For each player (i,j) an inner loop with indexes u and v going from -1 to 1 covers a 3x3 neighborhood of cells. Adding u to i and v to j two new indexes are created (iu and jv respectively) that go over the neighborhood of player (i,j), including itself.

Four conditional statements ("if") set periodic boundaries, checking if any of the iu and jv indexes goes outside the dimensions of the grid. Since we are dealing with a two-dimensional CA, this means that geometrically the CA grid forms a torus (an object whose surface looks like a donut). Then the present actions of the (i,j) player and the ones of its neighbors (including itself) are assigned to the auxiliary variables actionp1 and actionp2 respectively. Finally, the game is played according to its payoff matrix, and the resulting payoffs are accumulated.


```
% the game
for i = 1:size;
    for j = 1:size;

       % u and v indexes to move around neighborhood of player (i,j)
        for u = -1:1;
            for v = -1:1;
                iu = i + u;
                jv = j + v;

            % periodic boundaries (torus)
            if (iu > size) iu = 1; end
            if (iu < 1)    iu = size; end
            if (jv > size) jv = 1; end
            if (jv < 1)    jv = size; end

            % definition of actions for player 1 and player 2
            actionp1 = a_str(i,j).actionpresent;
```

```
            actionp2 = a_str(iu,jv).actionpresent;

            % the game is played
            if ((actionp1 == 0) && (actionp2 == 0))
              a_str(i,j).accumpayoffs = a_str(i,j).accumpayoffs + 0;
            end

            if ((actionp1 == 1) && (actionp2 == 0))
              a_str(i,j).accumpayoffs = a_str(i,j).accumpayoffs + 0;
            end

            if ((actionp1 == 0) && (actionp2 == 1))
              a_str(i,j).accumpayoffs = a_str(i,j).accumpayoffs + b;
            end
            if ((actionp1 == 1) && (actionp2 == 1))
              a_str(i,j).accumpayoffs = a_str(i,j).accumpayoffs + 1;
            end

            end
        end
    end
end
```

7.3 Adoptbest

This function receives as inputs the data structure and the size parameter and selects the best action to be adopted by each player as its new present action.

```
function a_str = adoptbest(a_str, size);
```

The main loop shown below, with indexes i and j, goes over the accumulated payoffs of each player. As in the playgames function described in the previous section, an inner loop goes over the player's neighborhood.

A 3x9 matrix named temp is created and filled with zeros. Each column of the matrix corresponds to a neighborhood player: in the first row are the player's accumulated payoffs, while in the second and third rows are the corresponding cell indexes for that player. A variable m, going from 1 to 9, will be used to transfer, for each neighborhood player, its accumulated payoffs and indexes to a column of the temp matrix. This is done after setting periodic boundaries for

the grid, as in the playgames function.

The MATLAB function max is used to place in the scalar top the largest element in the first row of the temp matrix and the corresponding index in the scalar topi. If there is more than one maximum, this function will choose only one. The corresponding indexes are stored in the variables bestiu and bestjv.

A new data field named a_str(i,j).actionbest is created, and the value of the best action found, contained in the data field a_str(bestiu, bestjv).actionpresent), is assigned to it.

```
for i = 1:size;
    for j = 1:size;

      temp = zeros(3,9); % creates a 3x9 temporary matrix with zeros
      m = 0;

         for u = -1 : 1;
            for v = -1 : 1;

                iu = i + u;
                jv = j + v;

            % periodic boundaries (torus)
            if (iu > size) iu = 1; end
            if (iu < 1)    iu = size; end
            if (jv > size) jv = 1; end
            if (jv < 1)    jv = size; end

            % transfers accumpayoffs and indexes to matrix
            m = m+1;
            temp(1,m) = a_str(iu,jv).accumpayoffs;
            temp(2,m) = iu;
            temp(3,m) = jv;
            end
         end


        % find maximum accumpayoff and corredponding indexes
        [top topm] = max(temp(1,:));
```

```
        bestiu = temp(2,topm);
        bestjv = temp(3,topm);

        a_str(i,j).actionbest = a_str(bestiu, bestjv).actionpresent;

    end
end
```

Finally, at the end of the adopbest function, a loop updates each player's actions.

```
% update actions
for i = 1:size;
    for j = 1:size;
        a_str(i,j).actionprevious = a_str(i,j).actionpresent;
        a_str(i,j).actionpresent = a_str(i,j).actionbest;
    end
end
```

7.4 Adoptcolor

The adoptcolor function is the last function called from the main program. It takes as inputs the data structure and the size parameter and assigns the corresponding color to each player comparing its present action to its previous action and applying the color code described earlier.

```
function a_str = adoptcolor(a_str, size);
for i = 1:size;
        for j = 1:size;

if (a_str(i,j).actionprevious == 0) && (a_str(i,j).actionpresent == 0)
         a_str(i,j).color = 1; %magenta red
         end
if (a_str(i,j).actionprevious == 0) && (a_str(i,j).actionpresent == 1)
         a_str(i,j).color = 2; %green
         end
if (a_str(i,j).actionprevious == 1) && (a_str(i,j).actionpresent == 0)
         a_str(i,j).color = 3; %yellow
         end
if (a_str(i,j).actionprevious == 1) && (a_str(i,j).actionpresent == 1)
```

```
          a_str(i,j).color = 4; %blue
          end
         end
end
```

8 Dynamic evolution of the spatial game

As we said earlier, the behavior of the CA for the game we are dealing with critically depends of the value of the b parameter. Figure 9 shows the evolution of the CA for b = 1.3. Remember that we assume that in the first generation all players are cooperators (blue color), except the one in the center of the grid which is a defector (red color). To show what's going on more clearly, each square in Figure 9 displays the central grid of only 9x9 cells instead of the complete grid of 49x49 cells, and only for 9 runs.

We can observe that in the initial generation there is a defector surrounded by cooperators. After the first round of games, all cooperators on the square around the defector become defectors (yellow color). This is because the cumulative payoff of the defector is equal to 8b: the defector wins all the games against its eight neighbors obtaining a payoff equal to b at each game, and 0 when playing against itself, thus getting a cumulative payoff of 8x1.3 = 10.4. While the cumulative payoff for each cooperator in that neighborhood is equal to 8: each cooperator gets 1 when playing against all the cooperators in its neighborhood and against itself, and 0 when playing against the defector which is a member of its neighborhood. Thus, the maximum cumulative payoff is 10.4, and all cooperators become defectors.

In the next round all defectors except the one in the center of the grid become cooperators (green color). At the corners of the central 3x3 square, cumulative payoffs are equal to 5b, while the maximum cumulative payoff for the surrounding cooperators is equal to 8. Since 5x1.3 = 6.5 is smaller than 8, corner defectors switch to cooperators. Along the edges of the square, defector's cumulative payoffs are equal to 3b, while the maximum cumulative payoffs for the surrounding cooperators is equal to 7. Since 3x1.5 = 4.5 is smaller than 7, those defectors also switch to cooperators. Thus, we are back to the initial situation. Thus the dynamic pattern of the game with b=1.3 will be one of cyclical changes from cooperators to defectors and back, for the players in the central 3x3 square surrounding the central defector.

Figure 9: Cyclical behavior (b = 1.3)

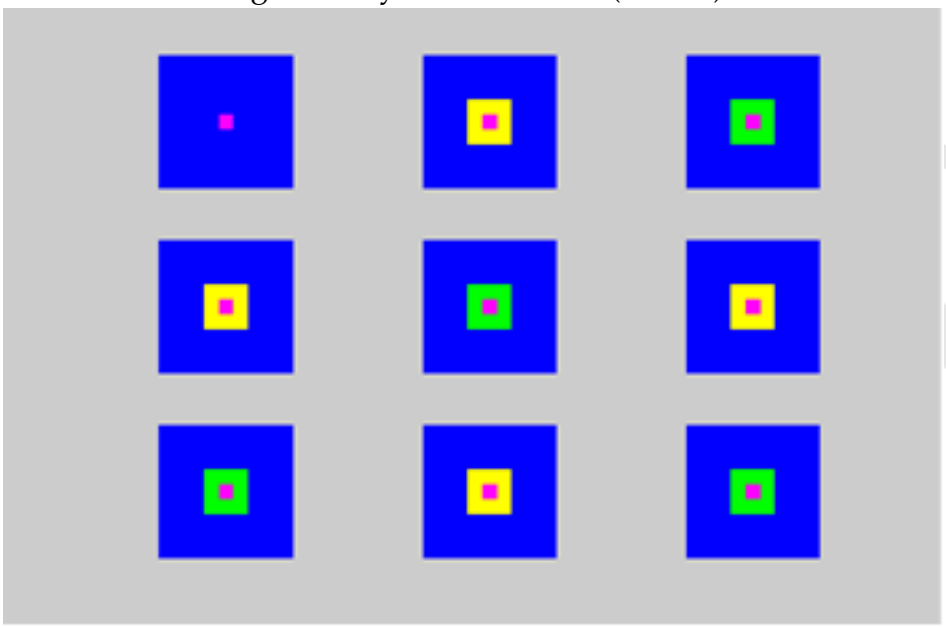


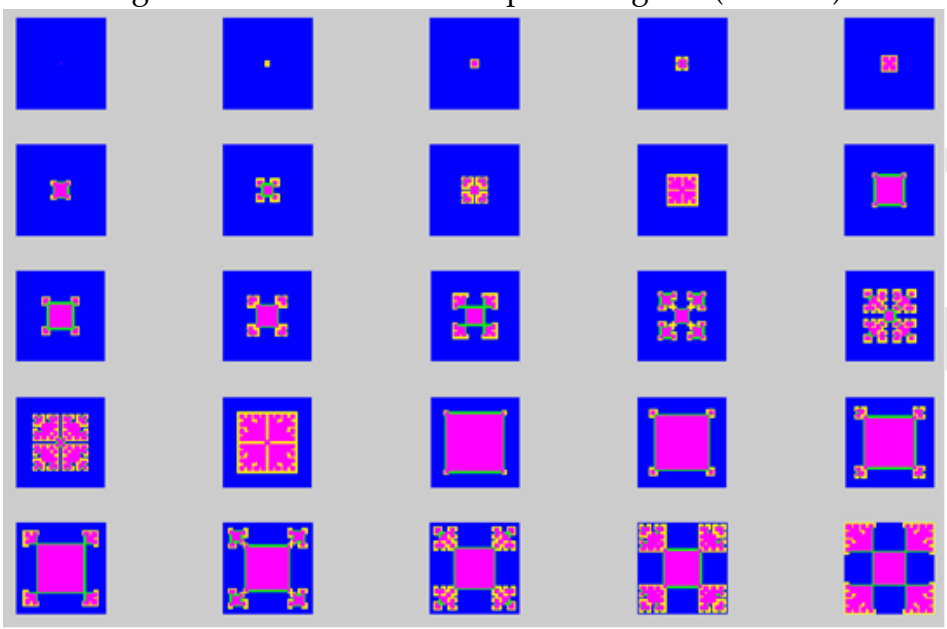
Figure 10 shows the behavior of the CA for b = 1.85. We can observe that the population of defectors grows up to a point, then defectors and cooperator grow in a beautiful fractal like way.

We have seen how the behavior of the CA, and the spatial evolutionary game it represents, depend critically of the value of the b parameter. For the payoff matrix we used, we can identify four parameter regions. When b is lower than 1, the only initial defector quickly disappears. For b greater than 1 and smaller than 1.8, the CA quickly reaches a stable or cyclical behavior. For b greater or equal to 1.8 and smaller than 2.3, defectors and cooperator growth in fractal like forms. While for b greater or equal to 2.3, defectors keep growing.

9 Experiments

You may want to explore the behavior of one-dimensional Cellular Automata for a variety of Wolfram's rules or increasing the size of the grid showing the successive simulations. You may also want to analyze the spatial game presented in this chapter with a larger grid and for a larger number of generations, and for other values of the b parameter. You may also experiment with fixed

Figure 10: Defectors and cooperators grow (b = 1.85)



boundary conditions. In such a case, players at the corners will have 3 neighbors only, while those along the edges will have 5 neighbors only. You may also impose infinite boundary conditions, setting a large size for the grid, then focus on the behavior or the CA for a number of generations that doesn't reach the borders of the grid. Finally, you may what to try different initial conditions: more than one defector, or an initial population with all defectors and one or more cooperators.

10 Further readings

For a systematic introduction to one-dimensional Cellular Automata, see Wolfram (20002). For a pioneering paper on spatial evolutionary games, see Nowak and May (1993). For a mathematical introduction to Cellular Automata models, see Boccara (2004).

References

Boccara, N.: 2004, *Modeling Complex Systems*, Springer, New York.

Nowak, M. A. and May, R. M.: 1993, Evolutionary games and spatial chaos, *International Journal of Bifurcation and Chaos* **3**, 3578.

Wolfram, S.: 20002, *A New Kind of Science*, Wolfram Media Inc.