**Introduction to Cellular Automata in Ecology**

Rob Morgante

University of Calgary

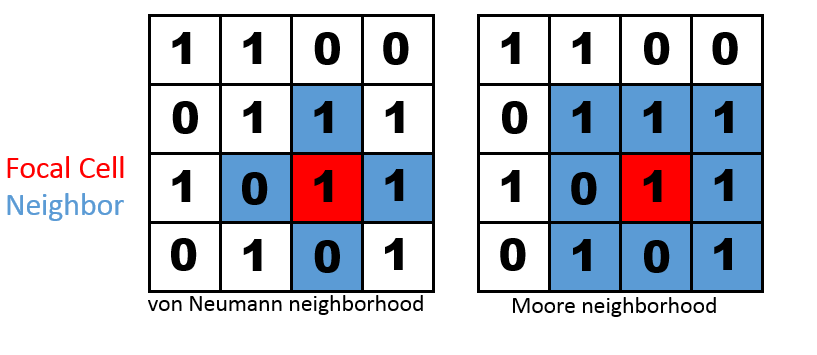
The goal of this tutorial is to explain how to create a simple cellular automata in R. I will give a quick overview of the background theory and the reasons for using a cellular automaton then walk through one example of a cellular automaton (CA).

**Introduction**

Large-scale patterns observed in natural populations are often the result of small-scale processes (Baltzer et al. 1998). Spatially explicit models allow for the modeling of population dynamics across an entire landscape while accounting for the small-scale biological mechanisms driving the system. Historically, partial differential equations have been one of the primary tools for studying spatial processes in ecology (Holmes et al. 1994). These partial differential equations are especially useful in ecological modeling, because they allow for the simultaneous study of both spatial and temporal dimensions of the system (Holmes et al. 1994). However, they are often very difficult or impossible to solve analytically. CA models are one way to rectify this by serving as a discrete approximations of complex partial differential equations (Omohundro 1984).

Cellular automata are especially useful due to their ability to examine spatiotemporal processes, relatively simple implementation, and flexibility to be modified to particular systems ( Breckling et al. 2011). They are especially useful in examining processes related to the introduction of a species into a novel community, such as invasion, range shift, and succession (van der Putten et al. 2009). CA have been used in the study of plant population dynamics, the spread of disease, interspecific competition, and fire spread, among many other ecological applications (Breckling et al. 2011, Baltzer et al. 1998). CA are also great for examining spatial pattern formation at large scales and their probabilistic nature helps mimic natural processes. If your goal is to model a spatially explicit process that is too complex to solve analytically, a cellular automaton may be appropriate.

A cellular automaton is defined as a lattice composed of cells that can exist in a finite number of predefined states where the state of any given cell is determined by the state of the cells at the previous time step and the given transition function. Typically, the number of cell states is small. For example, each cell may only show presence or absence of a species of interest. The cells in the grid are usually updated synchronously, i.e. the transitions are all calculated at once. However, some automata use asynchronous updating with one random focal cell chosen for each time step. A neighborhood is the amount of surrounding cells that are considered in the determination of the next cell state. The most commonly used types of neighborhoods are the von Neumann-neighborhood, which counts the 4 cells to the north, south, east, and west of the focal cell, and the Moore-neighborhood, which looks at all 8 of the surrounding cells in a 3 x 3 grid as shown below.



Another important consideration in the design of the model is the choice between a stochastic and deterministic CA. Stochastic CA are generally considered to be more applicable to ecological systems, but may become more complicated and lose heuristic value (Baltzer et al. 1998). Though its common to only have two possible cell states: on and off, cellular automata allow for much more complex models. You could model the population dynamics of a multispecies system where each cell represents an individual of one of several species.

**Example**

The remainder of the tutorial will focus on working through the code for a simple cellular automata in R. A full version of the code is included at the end of the tutorial. Before starting, we need to install and load the caTools package, which will be used later to create a GIF of our results.

install.packages("caTools")

library(caTools)

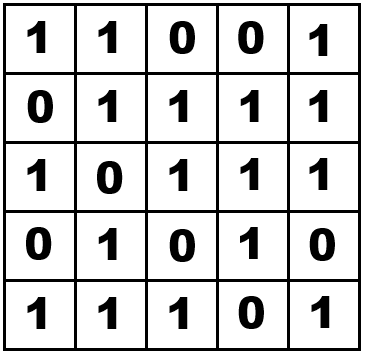
To begin creating the cellular automata, we first create the matrix that represents the physical space (forest, lake, plot, petri dish, etc. ) of the system. The matrix will be a square with side lengths equal to the variable **side**, which I have set equal to 5. This matrix will be called **X**.

side=5

X <- matrix ( nrow=5, ncol=5 )

Now we have **X** , a 5 x 5 matrix full of NAs. To fill X with values, I will use the **sample** function to randomly fill each cell with either a 0 or a 1 with a 75% chance to get a 0 and a 25% to get a 1. These initial conditions depend on the system you are trying to model. Within the sample function, **c(0,1)** creates the vector of values you want to sample from. My vector has the values of 0 and 1, so the **sample** function can only choose either a 0 or a 1. The next argument tells the **sample** function how many times you want it to sample a value from that vector. **Length(x)** instructs the function to sample an amount of values equal to the amount of cells in the matrix **X**. Setting **replace** equal to true allows each value to be selected multiple times, and the **prob** vector gives the probability of sampling 0 and 1 respectively. The first probability in the **prob** vector should correspond to the first value in the vector of values you are sampling from.

X[] <- sample( c (0,1) , length(X) , replace=T, prob=c ( .25 , .75 ) )



To be able to save the state of this matrix over time, an array is used with the 3rd dimension of the array representing time. **Steps** is the amount of iterations/generations you want the simulation to run for. Here, **steps** is set equal to 100. **Storage** is the name of the array that will store the results of the simulation at each time step. It is an array initially filled with zeroes, and has dimensions of **side** x **side** x **steps+1**.

steps = 100

storage <- array (0, c ( side, side, steps+1 ) )

We will fil the first slice of the array **storage** with the initial value of **X** from the random sample. The rest of the slices will contain the output of the simulation. Since we want the values to be stored in the 3rd dimension of the array, the two commas in the brackets preceding the 1 tell R to put the value in the 3rd spot.

storage[ , , 1]<-X

The rest of the code is the implementation of the simulation. Each part is explained in detail later, but I will start by giving an overview of the entire simulation. We begin by creating a for loop with an amount of iterations equal to **steps**. Lines 3-8 count the amount of neighbors each focal cell has by shifting the entire matrix in different directions. Lines 9-13 contain the rules for the simulation. In this example, the rules follow Conway’s Game of Life, which is a famous example of a simple cellular automaton. If an empty cell has 3 neighbors it becomes a 1, and if a cell with a 1 does not have 3 neighbors it becomes a 0. Finally, line 14 stores **X** at each time step in the array **storage**.

for (i in 1:steps)

{

allW = cbind ( rep ( 0 , side ) , X [ , -side ] )

allN = rbind (rep( 0 , side ) , X [ -side , ] )

allS = rbind (X [ -1 , ] , rep ( 0 , side ) )

allE = cbind (X [ , -1] , rep ( 0 , side ) )

neigh <- allW + allN + allE + allS

X3 <- X

X3[X==0 & neigh==3] <- 1

X3[X==1 & neigh<2] <- 0

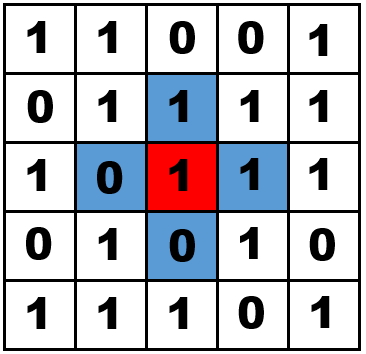
X3[X==1 & neigh>3] <- 0

X <- X3

storage[ , , i+1] <- X

}

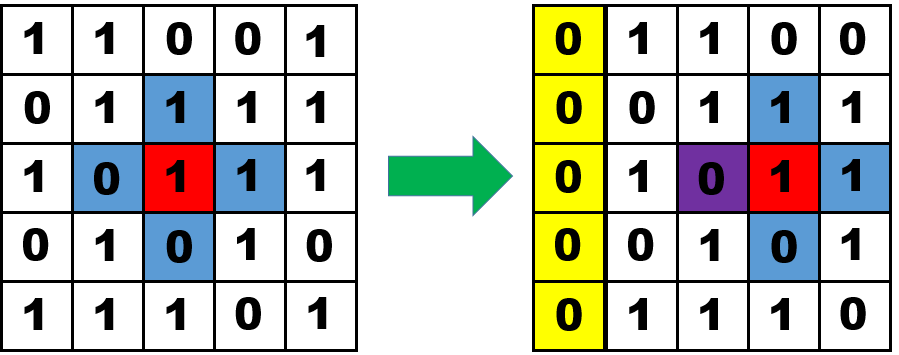
The simulation will count the amount of neighbors for every cell in the matrix at every iteration. To count the neighbors in a von Neuman-neighborhood for the cell in row 3 column 3 (shown in red), the simulation should look at the blue cell immediately above, below, to the left, and to the right of the focal cell. It should add the values from each of the 4 cells (1+1+0+0), giving a total of 2 neighbors.



To look at the cell to the to the left of the focal cell, we will shift all the cells in the matrix to the right, so that the “west” (**allW**) neighbor is moved to row 3 column 3. By column binding(**cbind)** a vector of 0’s and our **X** matrix,

allW=cbind(rep(0,side), X[,-side])

We get the shifted version of **X** called **allW** shown below. We added a new column of zeroes to the left side of the matrix and shifted all of the other columns to the right by 1 position. Now, the 0 that was initially to the left of the focal cell is located in the position of the focal cell (shown in purple).



Similarly, to count the cell to the right of the focal cell, we shift all of the cells to the left by adding the column of zeroes to the right side of **X**.

allE=cbind(rep(0,side), X[,-side])

Using **rbind**, we can add a row of zeroes to the top or bottom of **X** in order to count the cells above and below the focal cell. Shifting all the cells down (**allN**), the cell above the focal cell is counted. Shifting the cells up (**allS**) allows us to count the cell below.

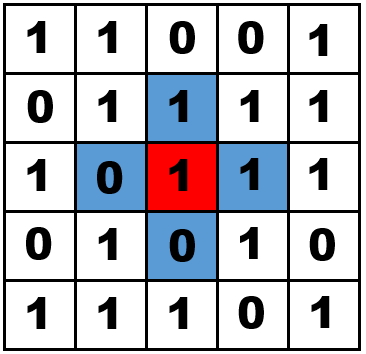
allN = rbind(rep(0,side),X[-side,])

allS = rbind(X[-1,],rep(0,side))

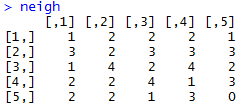
For each time step, there will be a value of **allN, allS, allE,** and **allW** for each cell, but we are just interested in the total number of neighbors each cell has. It doesn’t matter what direction the neighbor is in. Therefore, the 4 directional neighbor counts can simply be summed to get the total neighbor count for each cell, which will be stored in the matrix **neigh**.

neigh <- allW + allN + allE + allS

Going back to the original **X** matrix and focusing on the cell in row 3 column 3 again,



The value for **allW=0, allN=1, allE=1, and allS=0.** Adding these values, **neigh=2** at cell 3,3. Therefore, this cell has 2 neighbors in its von Neuman-neighborhood. In my example, I focused on 1 cell for simplicity. In the actual simulation, the amount of neighbors will be counted for every cell simultaneously. The matrix **neigh** for the values shown in **X** will look this this:



The value in each cell shows the amount of neighbors that cell has in the matrix **X**.

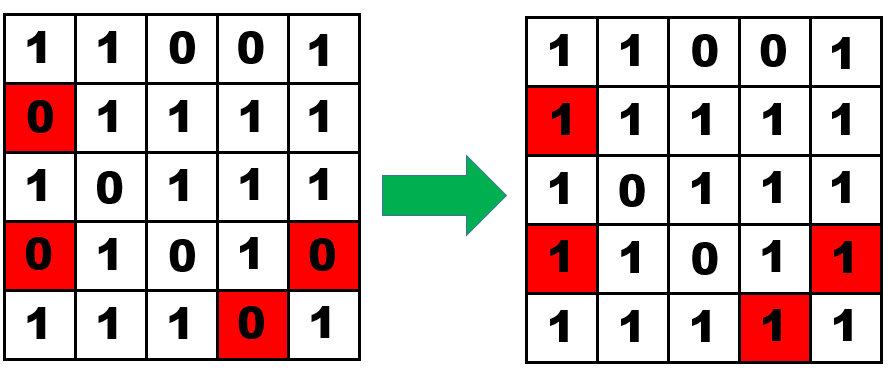
The next 5 lines of code implement the rules on the simulation. These rules define the mechanisms the simulation is based on. In this example, the rules from Conway’s Game of Life are used. This particular set of rules has limited biological relevance, but it generates interesting patterns and has been widely studied. The first line creates a new matrix, **X3**, which is initially identical to **X.**

X3 <- X

The next line implements the first rule of the Game of Life. Any cell that is currently empty and has exactly 3 neighbors will become a 1.

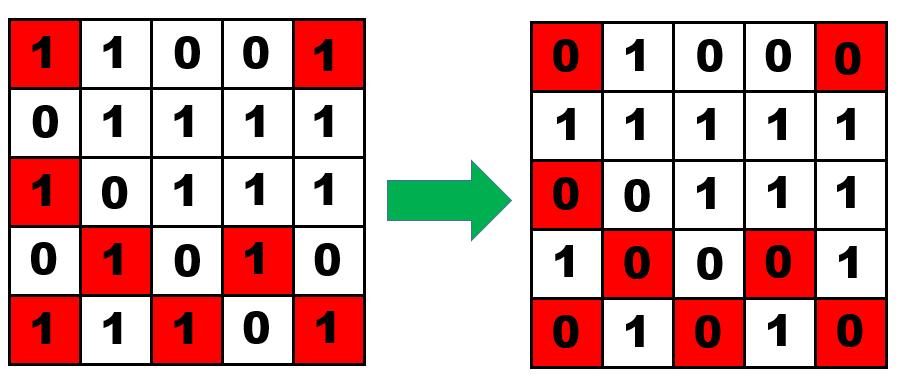
X3[X==0 & neigh==3] <- 1

This line of code will go through every cell in **X3** and check if they have a value of 0 in the corresponding position in the matrix **X** and a value of 3 in the corresponding position in the **neigh** matrix. The four cells that meet these two conditions are shown in red. All four of these cells are changed to a 1 according to the rules of the simulation.



The next rule changes cells that are currently a 1 and have less than 2 neighbors into a 0. It is important to note that it is using values from the original X matrix not the new X3 matrix made in the previous line.

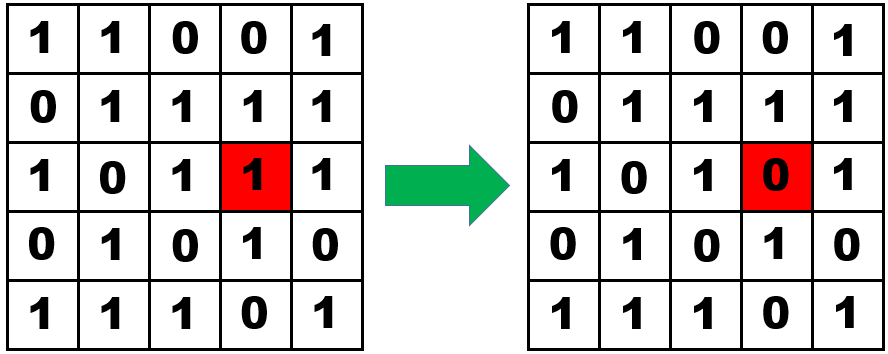
X3[X==1 & neigh<2] <- 0



There were a total of 8 cells that contained a 1 and had less than 2 neighbors. All of these were changed to a 0.

And finally, cells containing a 1 with more than 3 neighbors become a 0. Only 1 cell fits this criteria.

X3[X==1 & neigh>3] <- 0



After the 3 rules are applied to the matrix, the resulting **X3** is set as the new **X** matrix, and this **X** is stored in the array called **storage** and the for loop is closed**.** The for loop starts storing vales at i+1, because i is occupied by the initial matrix.

X <- X3

storage[,,i+1] <- X

}

This process of shifting matrices to count neighbors and changing the values in each cell based on the given rules is done for every iteration of the for loop. In order to visual the data contained in the storage array, an animated GIF can be used to observe the changes in spatial distribution of 1’s and 0’s over time. Using the **write.gif** function and the caTools package, a GIF can be made and exported to your working directory.

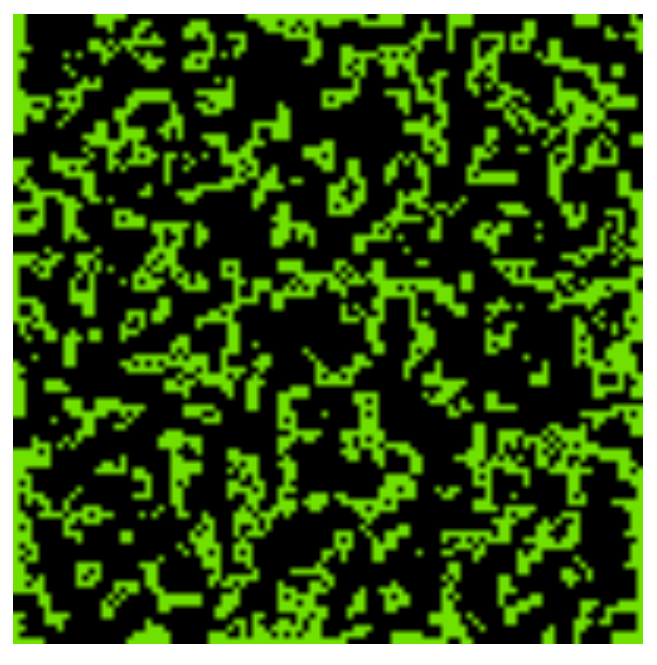
write.gif(file="gameoflife.gif", storage,col="rainbow", delay=5)

The first argument in the **write.gif** function is simply the name of the file the GIF will be saved as. In the second argument you should put the name of the array your data is stored in (**storage** in our example). The **write.gif** function needs the data to be an array of integers ranging from 0 to 255, but using the **scale= “always”** argument, you could input non-integer values and the function will internally rescale the values into integers. However, this is unnecessary in my example, because we only have integer values. The **col** argument allows you to define the colors that will be displayed in the GIF. **“rainbow”** is the color palette I chose for this example. The value in the **delay** argument determines how quickly the GIF will run. A lower delay will make the GIF display each frame for less time. Setting **delay=5** means each frame will be shown for 5 hundredths of a second.

After running the entire code, you should see a file named “conway.gif” in the folder of your working directory. Opening the file will open a tab in Internet Explorer displaying the GIF. This is what the first frame of our GIF looks like:



The black represents areas of the matrix occupied by zeroes, and the green shows cells with a 1. Typically the size of the grid will be much bigger than 5 x 5, which makes the output of this function look unusual. After scaling the simulation up to a 100 x 100 matrix by changing the value of **side** to 100, the result after 25 generations is:

****

By examining this output, we can observe the abundance and spatial distribution of 1’s and 0’s over time. We can see that the rules used in this model produced a patchy distribution with a fairly high abundance of 1’s. Due to the wide variety of applications, analysis of the results of CA can vary greatly and depend entirely on the question you are interested in. Typically, many simulations would be run with different values for key parameters and then comparisons would be made between the resulting outputs at the different parameter levels.

The code shown above is sufficient to create a simple cellular automaton, but these models allow for much more complexity. By changing the rules of the simulation, you can tailor the behavior of the simulation to match the properties of the system you wish to model.

**Optional code**

*Reflecting Boundaries*

One limitation of using the approach we used to count neighbors is that the cells at the edge of the matrix have less potential neighbors. For example, the bottom row of cells can have a maximum of 3 neighbors each since there is no row below them. You may want to define new rules for these boundary cells to avoid unusual behavior, or, alternatively, you could simulate your system as torus or open cylinder. In a toroidal grid, the bottom and top rows, and left and rightmost rows would be considered adjacent. This gives every cell the same amount of potential neighbors. A cylindrical grid would have either adjacent top and bottom rows or adjacent boundary columns but not both. This configuration can be used when you are simulating a system along a unidirectional environmental gradient, since it wouldn’t make sense to consider the cells at opposites ends of the gradient as adjacent.

To count the neighbors in a cylindrical grid, the code for shifting the matrix changes slightly. Instead of creating a new row/column of 0’s, the row from the opposite side of the matrix is replicated. For **allW**, we combine the leftmost column of the matrix with the rest of the matrix, excluding this column. This essentially moves the column from the right side of the matrix to the left side and shifts all of the other columns one position to the right. Similarly, for **allE**, the first column is moved to the rightmost position.

allW=cbind(X[ ,side] ,X[, -side])

allE=cbind(X[ ,-1] ,X[ , 1])

For a toroidal shape, we also need to make the top and bottom rows of the matrix function as if they were adjacnt. This is achieved by using the **rbind** function as we did before, except instead of shifting the matrix with a row of zeroes, we insert the top or bottom row of values.

allS=rbind (X[-1,],X[ 1 , ])

allN=rbind (X[ side ,], X[ -side ,])

*Moore Neighborhood*

In my example, I used a von-Neumann neighborhood for simplicity. If your system would be better modeled with a Moore neighborhood, there are five additional lines of code that need to be added to the example code shown above. Each line counts one of the four additional neighbors.

allNW = rbind(rep(0,side),cbind(rep(0,side-1),X[-side,-side]))

allNE = rbind(rep(0,side),cbind(X[-side,-1],rep(0,side-1)))

allSE = rbind(cbind(X[-1,-1],rep(0,side-1)),rep(0,side))

allSW = rbind(cbind(rep(0,side-1),X[-1,-side]),rep(0,side))

To count the northwest neighbor, we need to shift all the cells down and to the right to move the cell in the northwest position into the focal position. This portion of the code creates a column of zeroes in the leftmost position to shift the cells to the right.

cbind(rep(0,side-1),X[-side,-side])

The shifted matrix is then shifted down by using **rbind** to add a row of zeroes to the top of the matrix.

allNW = rbind(rep(0,side),cbind(rep(0,side-1),X[-side,-side]))

Repeating this for NE, SE, and SW, we are able to count the 8 neighbors for each focal cell instead of the 4 we previously had. The only other alteration to the code required for switching from a van-Neumann neighborhood to a Moore neighborhood is the inclusion of these new matrices into the summation formula for **neigh**.

neigh <- allW + allN + allE + allS + allNE + allNW + allSE + allSW

These are just two of the many ways you may want to alter your cellular automaton model to better suit your system. The addition of transition matrices and Markov chains, including more than one species, stochasticity in the application of the rules, parametrization based on empirical data, and asynchronous updating are other options you may want to consider when designing your CA, but are beyond the scope of this tutorial.

I would like to thank users Petrkeil and Martin Weiser from the R-Bloggers website for providing the code that I based this version of my cellular automaton off of.

**References**

Balzter, H., P. W. Braun, and W. Köhler. 1998. Cellular automata models for vegetation dynamics. Ecological Modelling 107:113–125.

Breckling, B., Pe’er, G., Matsinos, Y. 2011. Cellular automata in ecological modeling. Modeling complex ecological dynamics. 105-117.

Holmes, E. E., M. A. Lewis, J. E. Banks, and R. R. Veit. 1994. Partial differential equations in ecology: spatial interactions and populations dynamics.

Omohundro, S. 1984. Modelling cellular automata with partial differential equations. Physica 10D: 128- 134.

van der Putten, W. H., R. D. Bardgett, P. C. de Ruiter, W. H. G. Hol, K. M. Meyer, T. M. Bezemer, M. A. Bradford, S. Christensen, M. B. Eppinga, T. Fukami, L. Hemerik, J. Molofsky, M. Schädler, C. Scherber, S. Y. Strauss, M. Vos, and D. A. Wardle. 2009. Empirical and theoretical challenges in aboveground–belowground ecology. Oecologia 161:1–14.

**Full Code**

install.packages("caTools")

library(caTools)

side=5

X <- matrix ( nrow=5, ncol=5 )

X[] <- sample( c (0,1) , length(X) , replace=T, prob=c ( .25 , .75 ) )

steps = 100

storage <- array (0, c ( side, side, steps+1 ) )

storage[ , , 1]<-X

for (i in 1:steps)

{

allW = cbind ( rep ( 0 , side ) , X [ , -side ] )

allN = rbind (rep( 0 , side ) , X [ -side , ] )

allS = rbind (X [ -1 , ] , rep ( 0 , side ) )

allE = cbind (X [ , -1] , rep ( 0 , side ) )

neigh <- allW + allN + allE + allS

X3 <- X

X3[X==0 & neigh==3] <- 1

X3[X==1 & neigh<2] <- 0

X3[X==1 & neigh>3] <- 0

X <- X3

storage[ , , i+1] <- X

}

write.gif(file="gameoflife.gif", storage,col="rainbow", delay=5)