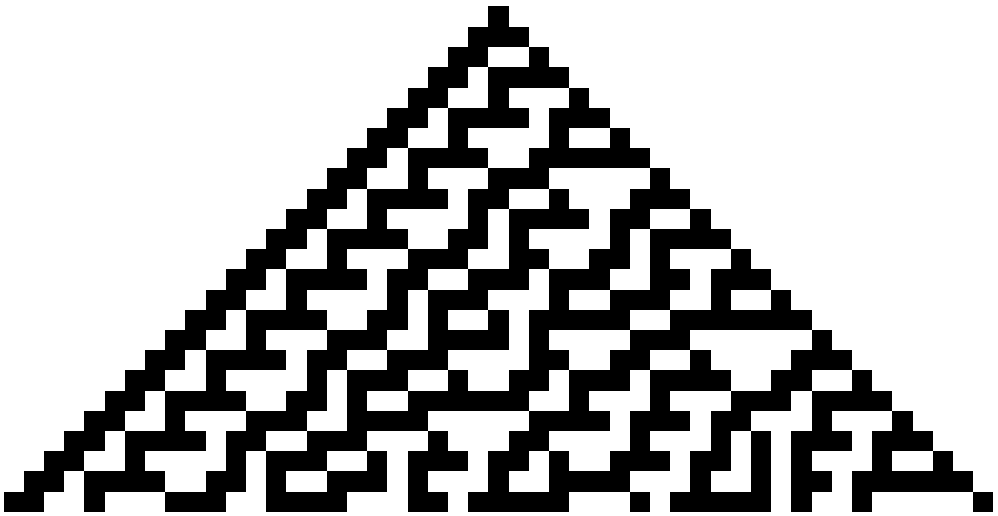
CHIC402/602 Task 3 - Cellular Automata

Barry Rowlingson

1 Introduction

In this exercise you will write some code to simulate one-dimensional cellular automata (CA). A typical example is seen here:

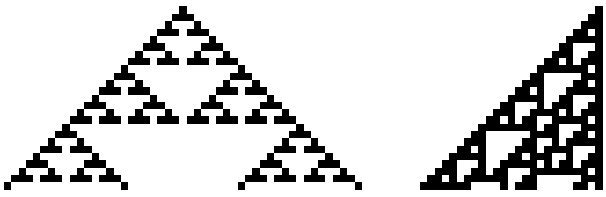


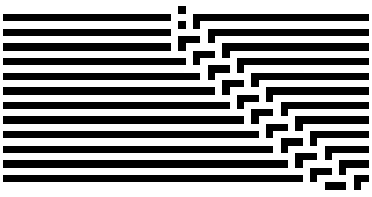
In this diagram "time" can be viewed as starting at the top, and the system evolves downwards. Initially (at the top) one "cell" (out of 25) is set to "on" and coloured black. A set of rules is applied to the pattern to determine the next row down which becomes 3 black cells below the first one. The rules are then applied again to create the next row. This is repeated *ad infinitum*, with the rules repeatedly applied to create new rows.

The rules used in these automata are *local*, in that the updated value of a cell is only dependent on the value of the cell to its left, the cell to its right, and its current value. Representing an "on" (black) cell as a one and an "off" cell as a zero, the rule table for the above pattern is:

current pattern	111	110	101	100	011	010	001	000
new state	0	0	0	1	1	1	1	0

Other rule sets can produce other patterns, a few are shown here:





For more background and explanation of the process, including a useful animation, see Cellular Automaton on Wikipedia, specifically the section on Elementary Cellular Automata.

2 Submission Requirements

Your submission for this work should be an R Markdown document written in a descriptive style, including code chunks, figure chunks, and text sections. Use section headings to match some of the headings in this document where appropriate. You may also choose to put your R code in a separate file, but make sure you include this in your submission and have a call (for example, to `source`) to read it into your R Markdown. Submit your R Markdown file, the PDF created from it, and the optional R code file. Your PDF should show the R code from the Markdown.

There is an initial blank R Markdown file included in the download.

3 Outline Approach

The ultimate goal is to write a function that fills a matrix of zeroes and ones given a function that implements a rule, an initial state, and the number of iterations (steps) to repeat. To do this we break the process down into stages:

- Write a rule function
- Write a function to update one element
- Write a function to update a complete row
- Write a function to update multiple rows and fill the matrix.

The matrix can then be drawn using the `image` function.

3.1 Writing a Rule Function

Write the rule tabulated above as an R function. It should take three arguments and return the value according to the rule. All inputs will be 0 or 1, as should be the returned value.

A number of approaches are possible:

- Write a set of "if-else" conditions that test for zeroes and ones in the input values, returning the corresponding value from the table as output.
- Paste the input values together as a string and look up the output in a named vector.
- Make a data frame with four columns - the three input values and the corresponding output value. For a given set of inputs, match the record in the data frame and return the output value.
- Note that the input values, expressed as binary numbers, count from 0 to 7. Convert the inputs to decimal and get the output value from a vector.

Check that:

```
rule(0,0,0) = 0
rule(0,0,1) = 1
rule(0,1,0) = 1
rule(0,1,1) = 1
rule(1,0,0) = 1
rule(1,0,1) = 0
rule(1,1,0) = 0
rule(1,1,1) = 0
```

3.2 Applying the Rule

Next we will write a function `new_value` that returns the updated value at position `i` for a vector of cell values and a rule function. To get the updated value for a cell at position *i*, you need to evaluate the rule for the cell values at `i-1`, `i`, and `i+1`. To update a whole row we'll eventually call this for all values of `i` in the vector but for now this function will compute the new value for a single cell.

There is a slight complication for the first and last cells, since they only have one neighbour in the vector. The solution to this is usually to imagine the cells "wrapping" around, so that the first cell is a neighbour of the last cell and vice-versa. In other words if `i+1` is larger than the length of the vector, instead use an index of 1. If `i-1` is zero, then instead use an index of the length of the vector (to get the last element). (Note there are ways to do this using the "modulo" operator "`%%`").

Create a vector called `CA` of length 11 containing zeroes. Set the 6th element to 1.

Check that the return value of your `new_value` function for cells 1 to 11 corresponds to this table:

`new_value(1, CA, rule)`	0
`new_value(2, CA, rule)`	0
`new_value(3, CA, rule)`	0
`new_value(4, CA, rule)`	0
`new_value(5, CA, rule)`	1
`new_value(6, CA, rule)`	1
`new_value(7, CA, rule)`	1
`new_value(8, CA, rule)`	0
`new_value(9, CA, rule)`	0
`new_value(10, CA, rule)`	0
`new_value(11, CA, rule)`	0

That table corresponds to the second row of the first diagram in this document where you can see three black squares in a row.

3.3 Returning a New Row

Next write a function that takes a vector and a rule, and returns the result of applying the rule over all cells from 1 to N, where N is the length of the vector.

Check that updating the `CA` vector created previously returns the following value, and then updating *that* vector produces the third output below:

```
print(CA)
```

```
##  [1] 0 0 0 0 0 1 0 0 0 0 0
```

```
CA2 = update_row(CA, rule)
print(CA2)
```

```
##  [1] 0 0 0 0 1 1 1 0 0 0 0
```

```
CA3 = update_row(CA2, rule)
print(CA3)
```

```
##  [1] 0 0 0 1 1 0 0 1 0 0 0
```

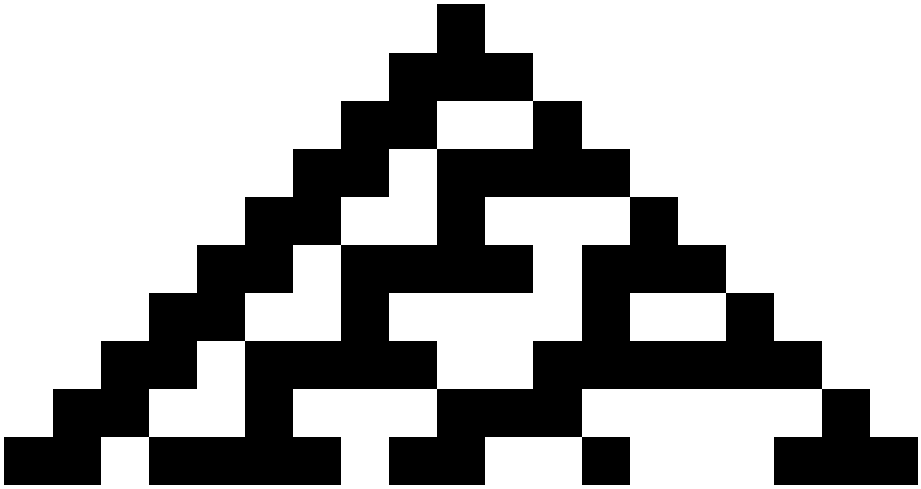
Note that `CA`, `CA2` and `CA3` correspond to the first three rows of the first figure.

3.4 Filling a Matrix

Our final function will take an initial vector of values, a rule function, and a number of iterations ("`N`") to do, and it should return a matrix. This matrix will have the same number of columns as elements in the initial state, and `N` rows.

The first row in the matrix will be the initial vector, and rows 2 to `N` will be the result of running the updating function on the previous row.

Start with an initial vector of length 21 with its middle element set to 1. Run for `N` set to 10 and the resulting output should be the first ten rows of the first figure. Use `image` to draw the figure, set the colours to black and white, and remove axes and annotation.

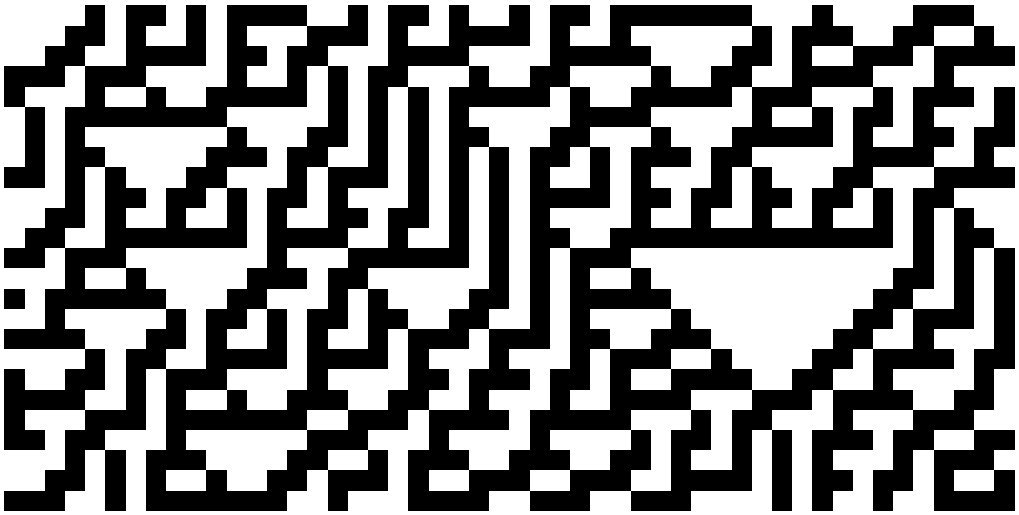


4 New Variations

Now that you have a working procedure to simulate these automata with this rule, you can try two variations - starting with a different starting vector and using a different rule set.

4.1 Random Start

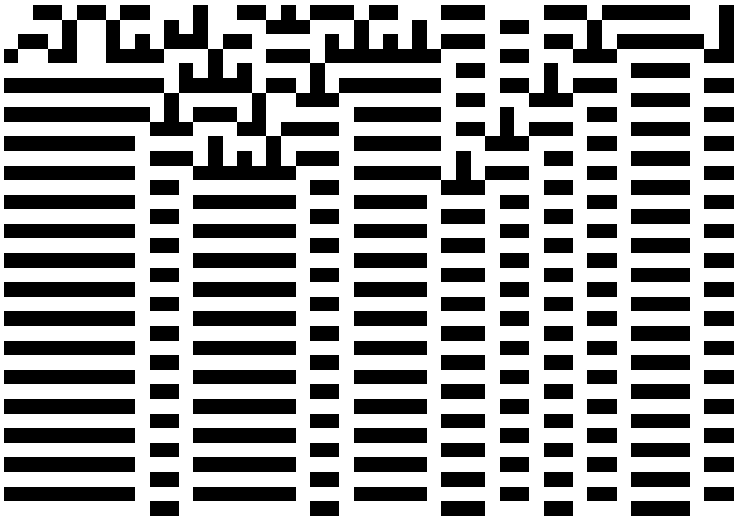
Create a vector of length 50 with random zeroes and ones with probability 0.5 each. Run the automaton for 25 steps and show the output. You should see some sort of chaotic structure:



4.2 New Rule

Create a new rule function. You could copy the code for the current one, give it a new name, and change which conditions return 0 and 1. Run this on a random starting vector of length 50 with 25 generations as in the previous section. Experiment with the rule until you discover something interesting, and plot it. Note that a rule returning lots of zeroes or lots of ones is likely to be boring, so experiment and find a rule that has some non-trivial behaviour to it.

Here's one I found, it looks like a view of city blocks.



5 Submission

Remember to include your R Markdown, generated PDF, and any R code file if you have one. Make sure your student ID is in the R Markdown metadata and appears in the PDF (you can also put it in a comment in the R code too).