CHIC402/CHIC602 Coursework 3 - Cellular Automata

36112985

The focus of this programming project is Cellular Automata, or more specifically Elementary Cellular Automata. In *Statistical Mechanics of Cellular Automata*, Stephen Wolfram states that[1]

> *Cellular automata are mathematical idealizations of physical systems in which space and time are discrete, and physical quantities take on a finite set of discrete values. A cellular automaton consists of a regular uniform lattice (or "array"), usually infinite in extent, with a discrete variable at each site ("cell").*
>
> — Stephen Wolfram, Statistical Mechanics of Cellular Automata

Additionally

> *A cellular automaton evolves in discrete time steps, with the value of the variable at one site being affected by the values of variables at sites in its "neighborhood" on the previous time step.*
>
> — Stephen Wolfram, Statistical Mechanics of Cellular Automata

In general, after randomly or deterministically initialising a grid of cells, a cellular automaton determines the state of a new cell via **(a)** a defined rule, and **(b)** the state of a specific set of cells.

In the case of elementary cellular automata. The initial grid is a row vector, and each cell has only two possible states: 0 or 1. The initial row vector can be a randomly or deterministically created grid of zeros & ones. Thereafter, the state of each next generation cell, i.e., each cell of the next row down, depends on a defined elementary cellular automaton rule & specific cells in the current/active row. This project creates functions for exploring elementary cellular automata, including

- Elementary cellular automata rules functions.
- A generic cell updating function that updates cells according to a provided rule.
- A generic row updating function that creates a next generation row w.r.t **(a)** a current/active row, and **(b)** a provided rule.
- A generic automata matrix function that creates elementary cellular automata matrices.

The exploration of elementary cellular automata is via **(a)** Wolfram's Rule 30, and **(b)** newly created custom rules.[2]

[1] Statistical mechanics of cellular automata, Stephen Wolfram, Reviews of Modern Physics 55, 601

[2] Rule 30 is decscribed in Statistical mechanics of cellular automata

1 A Reference Frame

In this document, each outlined elementary cellular automaton rule determines the state of a next generation cell via the same set of active/current row vector cells. In brief, each new generation cell depends on the state of three cells of the preceding generation, as illustrated in *Fig. 1*.

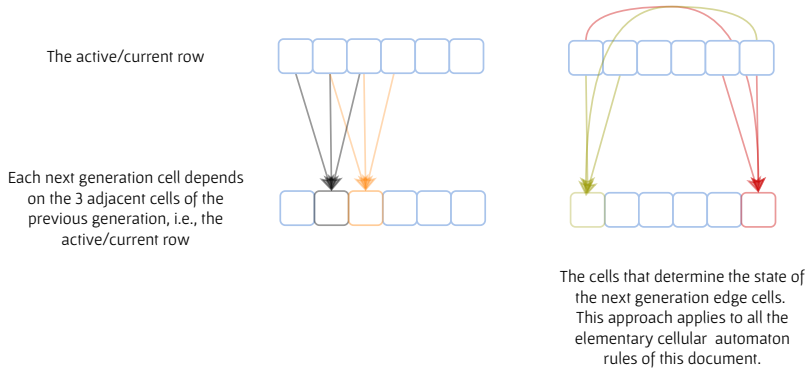


Figure 1: The cells that determine the state of a next generation cell

Bearing *Fig. 1* in mind, and noting that an elementary automaton cell only has 2 possible states: 0 or 1. Then there are 2^3 possible combinations of zeros & ones w.r.t. 3 contiguous cells. These 2^3 state combinations form the reference frame of each elementary cellular automaton rule in this document. *Table 1* outlines the combination; note that contiguity w.r.t. edge cells is as illustrated in *Fig. 1*, and

- i: the state of the first of three contiguous cells of a row vector.
- j: the state of the second of three contiguous cells of a row vector.
- k: the state of the third of three contiguous cells of a row vector.

Table 1: The eight possible state combinations of 3 contiguous cells. Each cell has 2 possible states: 0 or 1.

i	j	k
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

The document's examples illustrate that the rules can be differentiated in terms of the set of states they create in relation to the reference frame.

2 Automata Matrix Fundamentals

This section describes a set of functions that are used to create the matrix of an elementary cellular automaton.

2.1 Automaton Rule 30 Function

We need a rule to determine the state of a next generation cell, based on the cells of the active/current row, as illustrated in *Fig. 1.* Wolfram's Rule 30 is one of a number of elementary automaton rules for determining the state of a next generation cell. Herein, Rule 30 is encoded in the function *AutomataRule( )*. The state of a new generation cell due to Rule 30, and the reference frame, is

```
# Applying Rule 30 to the states outlined in the reference
# frame; Rule 30 is encoded in the AutomataRule( ) function.
states <- reference %>%
        mutate(state = mapply(FUN = AutomataRule, i = i, j = j, k = k))
```

Table 2: The state of a new cell w.r.t. Wolfram's Rule 30

i	j	k	state
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Now, we have a rule for a cell updating function.

2.2 Update Cell Function

The cell update function *UpdateCell( )* uses an elementary cellular automaton rule function to assign a state to a new generation cell. In this example, a starting row of length 11 is created, and the *UpdateCell( )* function sets the state of each next generation cell via the Rule 30 function, i.e., via the *AutomataRule( )* function.

```
# A starting row of an elementary cellular automaton
N <- 11
element <- floor(median(c(1, N)))
tensor <- numeric(length = N)
tensor[element] <- 1

# Then, the state of the cells of the next generation, i.e., row 2, w.r.t.
```

```
# starting row 'tensor' and rule function 'AutomataRule( )'
latest <- mapply(UpdateCell,
       index = seq(from = 1, to = N),
       MoreArgs = list('tensor' = tensor, 'FUN' = AutomataRule))
```

The starting row:

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

The next generation

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

2.3 Row Updating Functions

A row updating function sets the state of an entire new generation row. There are 2 row updating functions available

UpdateRowIndependent( )

This function does not depend on the external *UpdateCell( )* function.

and

UpdateRowDependent( )

This function depends on the *UpdateCell( )* function.

Regardless, their results are the same. Both are tested here via *UpdateTensor( )*, which is used to create an automaton matrix as instructed in the project brief. In-line with the project brief,

- the matrix is an $iterations = 3$ matrix, i.e., it has 3 rows
- the starting row is

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

The following code sections illustrate that the 2 row updating function give the same results.

```
#  Setting the state of the cells of a new generation row via UpdateRowIndependent( )
UpdateTensor(iterations = iterations,
             tensor = tensor,
             FUN = UpdateRowIndependent,
             rule = AutomataRule)
```

```
        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
pattern    0    0    0    0    0    1    0    0    0     0     0
T          0    0    0    0    1    1    1    0    0     0     0
T          0    0    0    1    1    0    0    1    0     0     0
```

```
# Setting the state of the cells of a new generation row via *UpdateRowDependent( )*
UpdateTensor(iterations = iterations,
             tensor = tensor,
             FUN = UpdateRowDependent,
             rule = AutomataRule)
```

```
        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
pattern    0    0    0    0    0    1    0    0    0     0     0
T          0    0    0    0    1    1    1    0    0     0     0
T          0    0    0    1    1    0    0    1    0     0     0
```

2.4 The Matrix Function

The function *AutomataMatrix( )* creates elementary automata matrices, and it builds on the preceding functions. In brief

- One of its parameters is an automaton rule function parameter.
- It implicitly uses a row updating function to build a matrix.

Additionally, it requires a starting row vector, and it needs to know the number of matrix rows required. The code below tests the *AutomataMatrix( )*. Thus far in this document only one rule function has been created → *AutomataRule( )*, hence it is used for the test.

```
# Starting row length > N
N <- 21

# Starting row vector > tensor
tensor <- numeric(length = N)
element <- median(c(1, N))
tensor[element] <- 1

# Testing AutomataMatrix: iterations is the number of matrix rows
pattern <- AutomataMatrix(iterations = 10, tensor = tensor, FUN = AutomataRule)
```

The resulting numeric pattern, stored in variable *pattern*, is

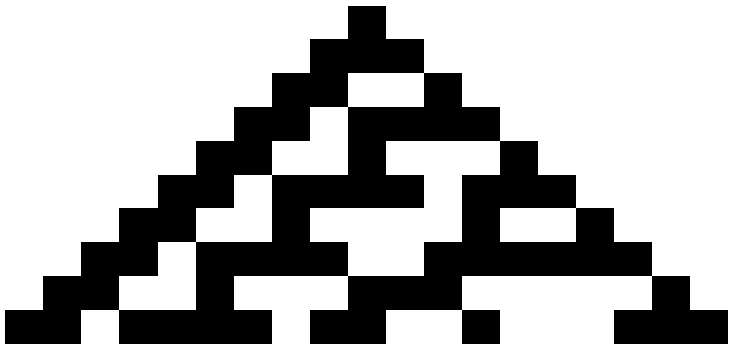
```
# Previewing the automaton matrix
prmatrix(pattern, rowlab = rep('', nrow(pattern)), collab = rep('', ncol(pattern)))
```

```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 1 1 1 0 1 1 1 0 0 0 0
0 0 0 0 0 1 1 0 0 1 0 0 0 0 1 0 0 1 0 0 0
0 0 0 1 1 0 1 1 1 1 0 0 1 1 1 1 1 1 0 0 0
0 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 0 0
0 1 1 0 1 1 1 1 0 1 1 0 0 1 0 0 0 1 1 1 0
```

3 The Image Of a Matrix

It is easier to observe the pattern of an elementary cellular automaton matrix via an image of the matrix. The image of the preceding automaton matrix is

```
# The image of cellular automaton 'pattern'
image(pattern, col = c('white', 'black'), axes = FALSE)
```



3.1 Random Start

Herein, an exploration of cellular automata due to starting row vectors of randomly placed zeros & ones. The probability of a zero or one is 0.5 each. The code below illustrates the creation of automaton matrices due to such starting row vectors; the starting row vector length is 50, and the number of automaton matrix rows is 25.

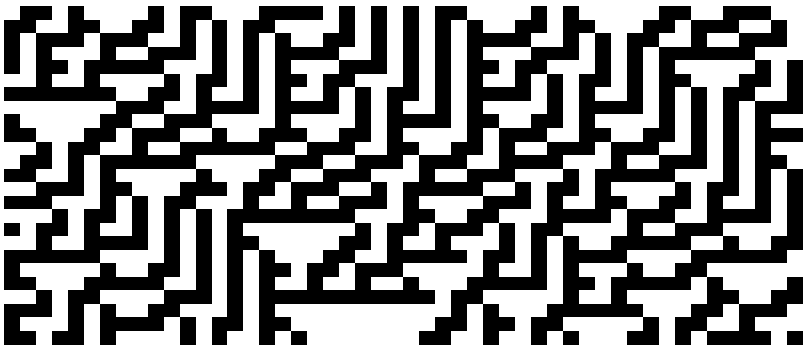
```
# Creating cellular automata matrices based on starting row vectors with
# randomly placed zeros & ones.  Matrices are created via the
# AutomataMatrix( ) function, and using the AutomataRule( ) function.

# The length of the starting row vector > N
N <- 50

# The starting row vector of randomly placed zeros & ones > tensor
tensor <- sample(x = 0:1, size = N, replace = TRUE, prob = c('0' = 0.5, '1' = 0.5))

# An automaton matrix named 'randomstart'; 'iterations' is the number
# of automaton matrix rows.
randomstart <- AutomataMatrix(iterations = 25,
                              tensor = tensor,
                              FUN = AutomataRule)
```

The resulting image is



4 A New Rule

Herein, an alternative - alternatives - to the Rule 30 function *AutomataRule( )*. Readers have access to three of the tried rules via the functions

- *AutomataStateRule( )*
- *AutomataRandomRule( )*
- *AutomataModuloRule( )*

which are listed within the *functions* directory. However, the focus herein is *AutomataStateRule( )*.

4.1 The Automata State Rule

The *AutomataStateRule( )* is based on the sum of the 0/1 states of the three cells that dictate the state of a new cell. It is

$$sum(i, j, k)\ mod\ 2$$

The state of a new generation cell w.r.t. the Automata State Rule, and w.r.t. the reference frame combinations, is outlined in *Table 3*.

Table 3: The state of a new cell w.r.t. Automata State Rule

i	j	k	state
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

4.2 Exploring the Automata State Rule

In addition to a rule, the pattern of an automata matrix, and the richness or peculiarity of the pattern, also depends on the starting row's

- pattern of zeros & ones
- length

Noting that the extent to which a pattern/anything can be discerned depends on the number of automata matrix rows, i.e., the number of iterations. Hence, herein a rule's automata matrix/image depends on, and is created w.r.t

- Starting row length
- The distribution of zeros & ones in the starting row . . . example -> c('0' = 0.95, '1' = 0.05)
- The number of automata matrix rows, including the starting row
- An automata rule

The *AutomataExperiment( )* function encodes this set-up. Now, an Automata State Rule image example:

```
# Trying the AutomataStateRule
AutomataExperiment(N = 50,
                   prob = c('0' = 0.90, '1' = 0.10),
                   iterations = 25,
                   FUN = AutomataStateRule)
```