

# Brian's Brain

Carleton University

Department of Systems and Computer Engineering

SYSC5104

Methodologies for Discrete-Event Modeling and Simulation

Assignment 2

Cristina Ruiz Martin

# Table of Contents

---

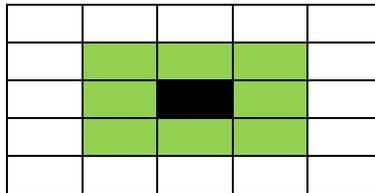
1. Conceptual Model.....	3
2. Formal specification .....	4
3. CD++ Implementation .....	4
State definition .....	4
Neighborhood definition .....	4
Rules.....	5
MA file.....	5
VAL file .....	6
4. Experiments .....	7
5. Simulation Results .....	8
6x6 Grid .....	8
20x20 Grid .....	9
100x100 Grid .....	9
500x500 Grid .....	9
400x400 Grid .....	10
6. Conclusions.....	10
7. References.....	10

# 1. Conceptual Model

---

Brian Silverman developed the Brian's Brain cellular automaton model. This model is very similar to the Seeds pattern that he developed. [1]

The model consists of an infinite two-dimensional grid of cells that can be in three states: firing or on, refractory or dying, and dead or off (in Seeds the cells can only be in two states – dead or alive). Each cell follows the Moore neighborhood, so they have eight neighbors as shown in Figure 1. [1], [2]



*Figure 1 Moore neighborhood*

The rules of the model are as follows: [2]

At each time step,

- A dead cell turns to firing if it has exactly two firing neighbors. This rule is like the birth rule for Seeds.
- A firing cell always evolves to refractory.
- A refractory cell always evolves to dead.

The refractory cells tend to lead to a spaceship pattern (a pattern that reappears after a certain amount of generations in the same orientation but in a different position). Many Brian's Brain patterns will explode messily and chaotically, and they will often contain diagonal waves of firing and refractory cells. [1]

Brian's Brain has been used to construct oscillators. [1]

## 2. Formal specification

---

Brian's Brain =  $\langle X, Y, S, N, d, \text{type}, \delta_{\text{int}}, \delta_{\text{ext}}, \tau, \lambda, \text{ta} \rangle$  for Cell-DEVS is defined as follow:

$X = Y = \{\emptyset\}$

$S = \{s | s \in \{0, 1, 2\}\}$  // where 0 means dead or off, 1 means firing or on and 2 means refractory or dying

$N = \{(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1)\}$

$d = 100\text{ms}$

$\text{type} = \text{transport}$

$\tau: N \rightarrow S$  is defined by the following rules:

cell (0,0) = 0 **if** cell (0,0) = 2

cell (0,0) = 1 **if** (cell (0,0) = 0) & ((number of neighbors with  $s = 1$ ) = 2)

cell (0,0) = 2 **if** cell (0,0) = 1

$\delta_{\text{int}}, \delta_{\text{ext}}, \lambda$  and  $\text{ta}$  are defined using CELL-DEVS specifications.

## 3. CD++ Implementation

---

I have implemented the model for four different grid sizes: 6x6, 20x20, 100x100 and 500x500. After simulating the 500x500 grid I had to change this last size of the grid to 400x400 to simulate other scenarios due to the time and the size of the log file generated.

### State definition

The states are described as followed:

$s = 0$  means dead or off

$s = 1$  means firing or on

$s = 2$  means refractory or dying

### Neighborhood definition

Each cell follows the Moore neighborhood, so they have eight neighbors. It is defined in the .ma file as in Figure 2.

```
neighbors : briansbrain(-1,-1) briansbrain(-1,0) briansbrain(-1,1)
neighbors : briansbrain(0,-1)  briansbrain(0,0)  briansbrain(0,1)
neighbors : briansbrain(1,-1)  briansbrain(1,0)  briansbrain(1,1)
```

*Figure 2 Neighborhood definition in CD++*

## Rules

The rules of the model are as follows. At each time step,

- A dead cell (0) turns to firing (1) if it has exactly two firing (1) neighbors. This rule is like the birth rule for Seeds.
- A firing (1) cell always evolves to refractory (2).
- A refractory (2) cell always evolves to dead (0).

These rules are implemented in CD++ as shown in Figure 3. The language for the rules is:

“**rule:** result delay {conditions}”

The rules are evaluated in the order they appeared. For each cell, the first one that is satisfied is the one that handles.

```
[briansbrain-rule]
rule : 0 100 { (0,0) = 2 }
rule : 1 100 { (0,0) = 0 and trueCount = 2}
rule : 2 100 { (0,0) = 1 }
rule : 0 100 {t}
```

Figure 3 Brian's Brain rules in CD++

## MA file

I have defined five .ma files, one for each grid size. The .ma file for a 20x20-grid size model is shown in Figure 4 as an example.

```
[top]
components : briansbrain

[briansbrain]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : briansbrain(-1,-1) briansbrain(-1,0) briansbrain(-1,1)
neighbors : briansbrain(0,-1) briansbrain(0,0) briansbrain(0,1)
neighbors : briansbrain(1,-1) briansbrain(1,0) briansbrain(1,1)
initialvalue : 0
initialCellsvalue : BriansBrain20.val
localtransition : briansbrain-rule

[briansbrain-rule]
rule : 0 100 { (0,0) = 2 }
rule : 1 100 { (0,0) = 0 and trueCount = 2}
rule : 2 100 { (0,0) = 1 }
rule : 0 100 {t}
```

Figure 4 Brian's Brain .ma file for a 20x20 grid

## VAL file

The cell's initial states are stored in .val files. It is important to mention that the .val files have to finish with at least one blank line. If not, the input file is not parsed properly and despite the input is read properly, the model results would not be the expected ones.

Figure 5 shows the initial state for 6x6-grid.

0	0	0	0	0	0
0	0	2	0	0	0
0	0	1	1	2	0
0	2	1	1	0	0
0	0	0	2	0	0
0	0	0	0	0	0

Figure 5 6x6 grid initial state

For 20x20-grid, 100x100-grid, 400x400-grid and 500x500-grid the initial values are generated randomly through a C++ small program –one for each grid size- (Figure 6). The initial state can be a completely random grid or a grid with random values in the centre and with the rest of its cells in the death state (0).

```
#include <fstream>
#include <stdlib.h>
#include <time.h>
using namespace std;
int main()
{
    srand(static_cast<unsigned>(time(NULL)));
    int m,n;
    ofstream fs;
    fs.open("BriansBrain20.val");
    int p=0;
    for(m=0;m<20;m++)
        {
            for(n=0;n<20;n++)
                {
                    p= rand()%3;
                    fs<<"("&<<m<<"", "<<n<<"")"<<"="<<p<<endl;
                }
        }
    fs.close();
    return 0;
}
```

Figure 6 C++ program that generates the cell's initial states (completely random grid). Adapted from [3]

## 4. Experiments

---

I run multiple experiments in order to test the correctness of the model.

I have simulated a 6x6-grid to show how oscillators can be constructed using Brian's Brain and to test the correctness of the model. I have replicated the one on [1].

Bigger grids (20x20, 100x100, 400x400 and 500x500) have been simulated to test the model and its behavior. In addition, different initial states have been considered.

The simulation of the 100x100 grid takes around 20 minutes to generate the log file and 20 minutes to generate the drawlog file. Loading the drawlog file to visualize the results takes around 40 minutes.

The simulation of the 500x500 grid takes more than 20 hours to generate the log file (I left it running the whole weekend). The generation of the drawlog file of the 1800 first steps of simulation took around 4 hours. Loading the drawlog file to visualize the results takes a couple of hours and the visualization runs slowly. The video attached is recorded at x16. The log file is not included because of its size, more than 100GB (Figure 7). I had to delete it to run another simulation. The drawlog file was only generated for the first 1800 steps due to de lack of space to store a bigger one.

BriansBrain500	30/06/2015 4:17 PM	MA File	1 KB
BriansBrain500	06/07/2015 4:02 PM	VAL File	1 KB
BriansBrain500DRW_01.drw	06/07/2015 12:21 ...	DRW File	4,384,748 KB
BriansBrain500LOG_01	06/07/2015 8:18 AM	Text Document	105,739,112 KB

*Figure 7 Size of the files for a 500x500-grid*

Due to the issues running the 500x500 grid, to run other scenarios in a big grid I had to reduce the size to a 400x400 grid.

The different input files can be easily generated and placed in the project folder running a script on Cwing. Figure 8 shows the script for generating random values in the centre of the grid to create an initial state for a 400x400 grid

```
#!/bin/bash
g++ centre_random_initial_value_generator_400.cpp -o c_inval_400
./c_inval_400.exe
mv BriansBrain400.val ../
```

*Figure 8 Script for generating the initial state in 400x400 grid*

To run the different experiments, please read *readme.txt* file, which is available in BriansBrain.zip. This *readme.txt* file is based on the *readme* file for Supercooling Cell-DEVS model implemented in CD++ [4].

# 5. Simulation Results

In this section, I explain the simulation results and I show some images. The videos are available in BrainsBrain.zip

## 6x6 Grid

In the 6x6 grid I have replicate the oscillator in [1] to test the model and show how and oscillator in Cell-DEVS can be constructed following Brian's Brain rules (Figure 9).

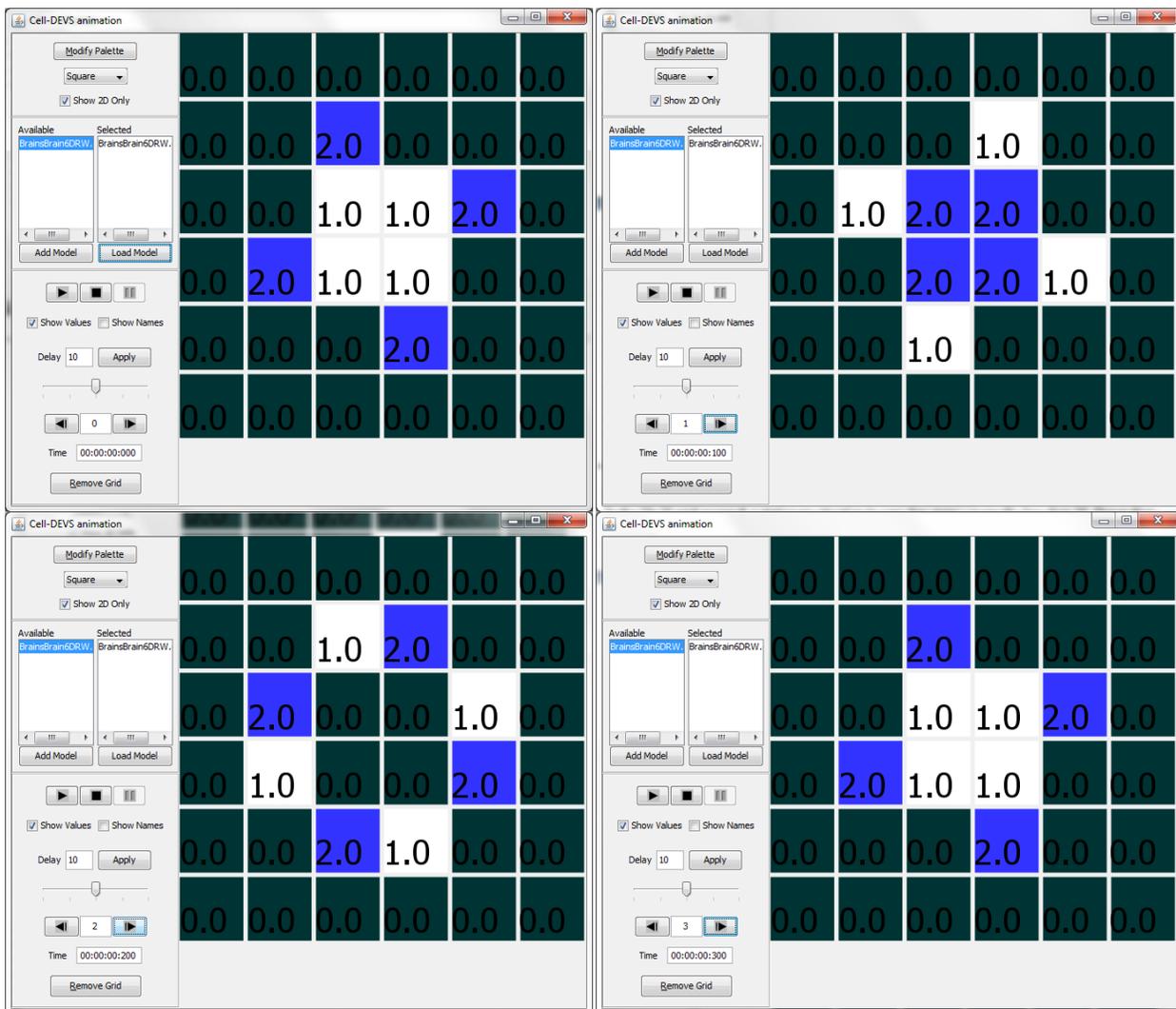


Figure 9 Oscillator following Brian's Brian rules implemented in CD++

## 20x20 Grid

In the 20x20 grid we reach a stationary situation in very few steps. How long it takes depends on the initial configuration. Depending on the initial configuration, we reach the dead state where all cells are dead or off (0) or we find an oscillator moving across the grid.

In the videos available in BriansBrain.zip we can see that different initial configurations leads to the same oscillator in different positions.

We also appreciate that the time it takes to reach a pattern depends on the initial configuration, but it does not depend on if the random values are generated in the whole grid or only in the centre.

## 100x100 Grid

Running the simulations and displaying the results in a 100x100 grid takes longer. The videos are available in BriansBrain.zip.

Analyzing the simulation results, we can reach the same conclusions as in a 20x20 grid: the time it takes to reach a pattern depends on the initial configuration, but it does not depend on if the random values are generated in the completely grid or only in the centre.

In this case, despite the whole pattern in the simulated scenarios is not the same, we can observe that the oscillators that form the pattern appear in the different scenarios no matter what the initial configuration is.

We also observe that as the grid size increases, it takes more time to reach the pattern.

## 500x500 Grid

Running a single scenario for the 500x500-grid has taken a weekend. The log file generated was more than 100GB, and the visualization tool works very slowly.

A video has been generated at x16 velocity of the real time the simulation took. It only displays a sample of the first 1780 steps. The initial state of the grid is a random value for each of the cells.

In this video, we can easily appreciate the diagonal waves of firing and refractory cells explained in the introduction.

Due to the simulation time and the size of the log file, I use a 400x400 grid to simulate another scenario with random values in the centre and the rest of the cells in the death state (0).

## 400x400 Grid

Running a single scenario for the 400x400 during 10 minutes took 23hours to generate the log file (49GB), 3hours to generate the drawlog file and more than 9 hours to load the results in the visualization tool. The visualization tool works very slowly.

A video has been generated at x16 velocity of the real time the simulation took. The initial state of the grid is a random value for each of the cells in the centre of the grid.

In this video, we can see how firing (1) and refractory (2) cells deploys in the whole network. We can also easily appreciate the diagonal waves of firing and refractory cells explained in the introduction.

In the first 6000 steps, we cannot see a pattern that repeats over time.

## 6. Conclusions

---

In the videos, we can observe how we reach different patterns and how the diagonal waves of firing and refractory cells. These were the expected results published in [1].

As bigger the grid size is, as easier is to identify the diagonal waves, but it takes longer to reach a pattern. In bigger grids, in our simulations experiments, the time was not enough to reach the pattern and I have the limitation of computer capacity.

## 7. References

---

[1] [https://en.wikipedia.org/wiki/Brian%27s\\_Brain](https://en.wikipedia.org/wiki/Brian%27s_Brain) (access date 10/07/2015)

[2] Wilensky, U. (2002). NetLogo Brian's Brain model.  
<http://ccl.northwestern.edu/netlogo/models/Brian'sBrain> Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL (access date 10/07/2015)

[3] Weiwei, L., 2D Crystal Growth Simulation with Cell-DEVS based on extension CD++  
[http://www.sce.carleton.ca/faculty/wainer/wbgraf/doku.php?id=model\\_samples:start](http://www.sce.carleton.ca/faculty/wainer/wbgraf/doku.php?id=model_samples:start) (access date 10/07/2015)

[4] Vicino, D. , Niyonkuru, D. (2014). Free Dendritic Growth in Supercooled Liquids.  
[http://www.sce.carleton.ca/faculty/wainer/wbgraf/doku.php?id=model\\_samples:start](http://www.sce.carleton.ca/faculty/wainer/wbgraf/doku.php?id=model_samples:start) (access date 10/07/2015)