Cellular Automata: Game of Life & Parity Rule

BY MARTINO FERRARI

1 Introduction to Cellular Automata

Cellular Automata (**CA**) are very simple discrete models introduced in the 40s by Stanislaw Ulam and John von Neumann.

A Cellular Automaton consists in a matrix of *cells*, each one in certtain status. At time $t=0$ the automaton is at initial status $s(0)=s_0$ (fixed by the user), at each iteration the status of the cells will evolve according to some fixed *rules* $s(t+1)=r(s(t))$.

In particular the rules define the future status of each cell on two factors: the current status of the cell and the current status of it's neighborhood. The most common types of neighborhood are the following:

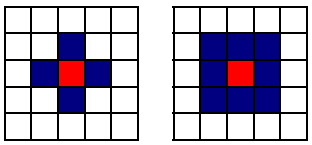

Figure 1. most common type of neighborhood

For this TP we were asked to implement two different rule-sets for theCellular Automaton, the *Game of Life* (based on the Moore neighborhood) and the *Parity Rule* (based on the von Neumann neighborhood)

1.1 Game of Life

The *Game of Life* was proposed by J. H. Conway in the 70s and is possibly the more knwon CA, and yet very simple. Each cell can be alive (1) or dead (0) and has 8 neighboors (Moore neighborood); this means that there are 2^9 (8 neighboors + 1, the cell) possible state of the neighboorod. The rules are very simple as well:

- if the cell is alive:
  - if 2 or 3 neighbors are alive it stays alive
  - else it dies
- if the cell is dead:
  - if there are exactly 3 living neighboors it lives
  - else it stays died

Even with this very simple rules this automata it's capable of many interesting thinks and it is *Turing complete*. With complex configuration is possible to generate counters, timers, primes number and much more.

1.2 Parity Rule

The *Parity Rule* CA was proposed by E. Fredkin as well in the 70s. As in the *Game of Life* the state $\psi_t(i,j)$ of each cell (i,j) as only two possible values. However this time the neighborhood is composed by only 4 cells (van Neumann neighborood) and so there are 2^5 possible configuration. The rule this time can be represented as a mathematical formula:

$$\psi_{t+1}(i,j)=\psi_t(i+1,j)\oplus\psi(i-1,j)\oplus\psi(i,j+1)\oplus\psi(i,j-1)$$

Even this simple automata can produce interesting result and patterns during its evolution

1.3 Border condition

Due to the neture of this model the border condition of the simulation is very important as we need to represent the neighboors of the cell close to the limit of the matrix, the possible configurations are mainly 3:

- Periodic: the matrix repeat it self at the border creating a torus
- Fixed value: the neighboors outside the matrix as a fixed value (ex: 0)
- Dynamic matrix: the simulation matrix expand itself when some life cells touch the border (often used for comple Game of Life configuration)

2 Implementation

We were asked to implement the Cellular Automaton model, the Game of Life rules and the Parity Rule in Python (version 3).

I implemented the rules as *look up tables*, avoid all the spatial loop and using the periodic border condition as requested.

To implement a single algorithm for the two rule set demanded I decided to first indexing the neighborhood:

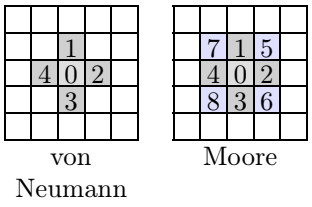

Figure 2. neighborhood indexing

Then I represented the status of the neighborhood as a chain of 9 bits (represented using an *integer*):

$$N(i,j) = \boxed{n_8^{i,j}}\boxed{n_7^{i,j}}\boxed{n_6^{i,j}}\boxed{n_5^{i,j}}\boxed{n_4^{i,j}}\boxed{n_3^{i,j}}\boxed{n_2^{i,j}}\boxed{n_1^{i,j}}\boxed{n_0^{i,j}}$$

Figure 3. binary representation

For the *parity rule* the state of $n_0^{i,j}$ (the current status of the cell) is not taken in account while computing the new state, however for compatibility reason I choose to represnt it anyways.

The rules will be represented in the form of a dictionary where the key is a possible state of the neighborhood and the value is 1 or 0 (possible state of the cell):

```
# Possible Parity Rule
rule = {
        0b11110: 0,
        0b11111: 0,
        0b10110: 1,
        0b10111: 1,
        ....
      }
```

To summirize the system will do the following (for the parity case):

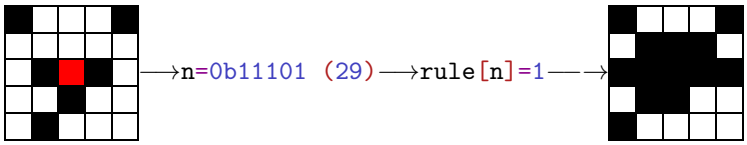

Figure 4. example of iteration

To avoid spatial loop I used first the function *numpy.roll* to compute the neighborhood status and then a *lambda* function vectorialized using the function *numpy.vectorize()* to apply the rules.

Moreover the python code suport both the *lif* and the more complete *rle* file format (as most of the existing configuration are in this format).

To configurate the simulator I chose to use a *json* file on this form:

```
{
  "config":"configurations/UnitCell.rle",
  "config_version":"rle", # or lif
  "plotmode":"stored", # or animated
  "iterations":500,
  "output_path":"plots/",
  "mode":"gameOfLife" # or parityRule
}
```

3 Results

3.1 Game of Life

After implementing the CA and the presented rules, I tested it on a known stable configuration of the *Game of Life*:

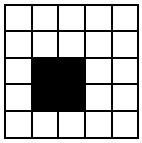

Figure 5. a stable configuration for Game of Life rules

The results of this first simulation (configuration: *configurations/tests/stable.rle*) is stable as predicted and the central square does not evolve in time:

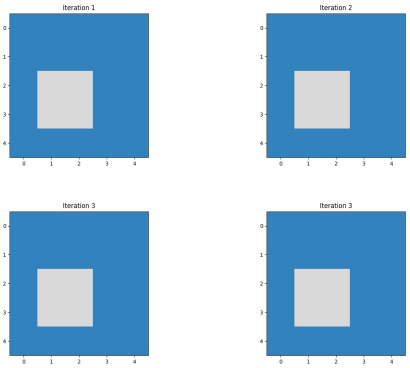

Figure 6. first 4 iterations of the simulation of a stable configuration

After this first succesfull test I wanted to a more interesting and dynamic configuration, in particular a classical *Game of Life* configuration is the *glider gun*:

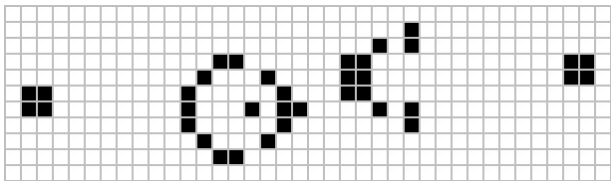

Figure 7. *Glider Gun* configuration

The results of this simulation (configuration: *configurations/tests/glidergun.rle*) is as expected a periodical structure (the cannon) firing periodically the gliders (gif: *gifs/glidergun.gif*):

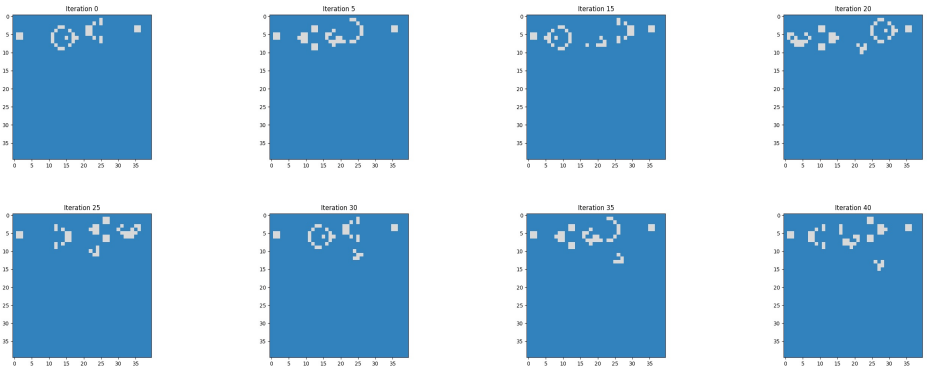

Figure 8. first 40 iterations of the simulation of a *glider gun* (5 steps per image)

Using a combination of *cannons* and other oscillator is possible to create very complex structures, like clocks, prime numbers calculator and more. However to simulate this kind of structure huge matrix are demanded (or better dynamic matrix, not implemented in my code) and due the relativly poor display (using *matplotlib*) performance is impossible to run those.

However I execute some smaller (500×500, 4ms per step) interesting configurations as the *UnitCell* (*configuration/UnitCell.rle*) that with an infinite matrix is able to copy itself (gif: *gifs/unitcell.gif*):

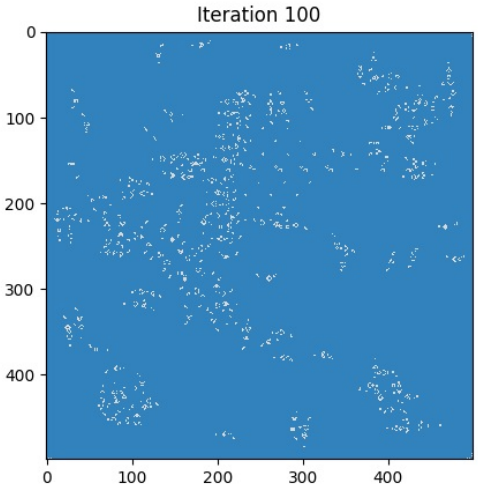

Figure 9. *UnitCell* configuration

3.2 Parity Rule

Also using the *Parity Rule* generate interesting results, in particular it creates many periodical structure like (*configurations/tests/letter.rle)*:

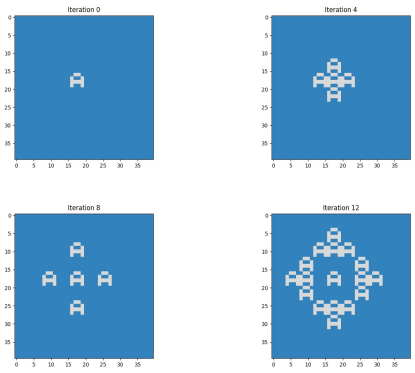

Figure 10. first 12 iterations (4 steps per image) of the simulation of a simple configuration

A more interesesting and bigger simulation (200×200) give the following pattern (*configurations/tests/hugeparity.rle*):

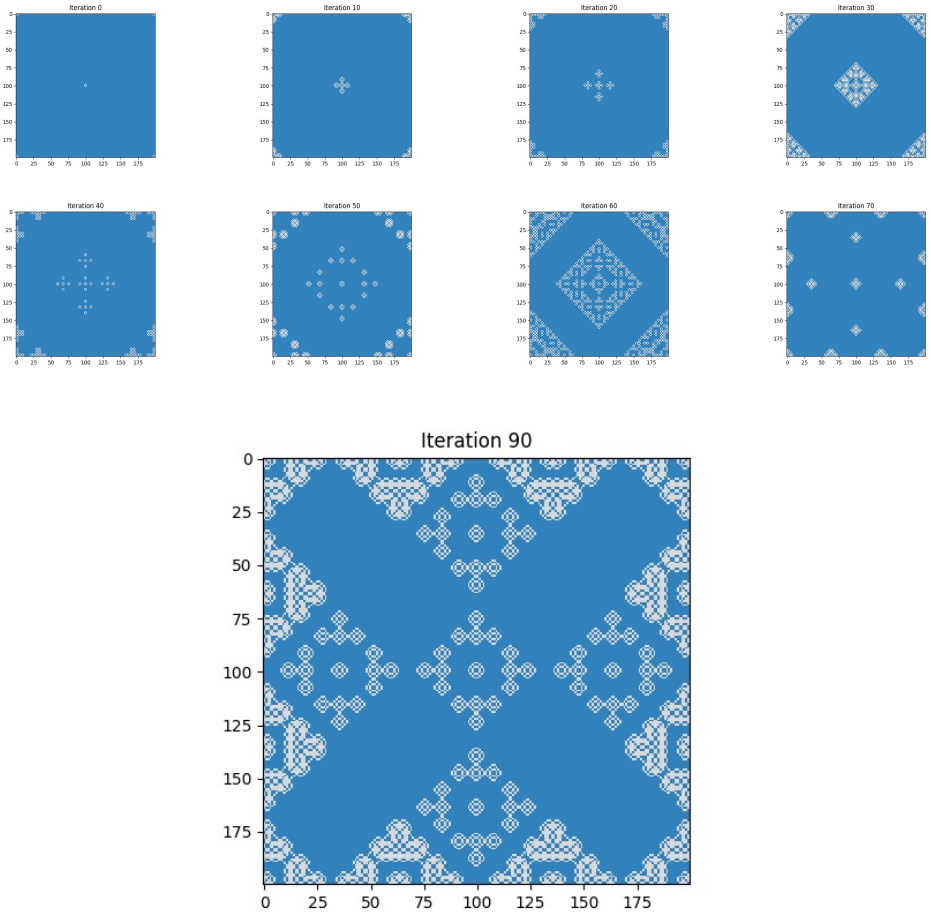

Figure 11. first 90 iterations of the simulation of a *glider gun* (10 steps per image)

This results are obtained beacouse in 2 iterations (without interferences) the pattern is copied in 4 new positions (gif: *gifs/parity4.gif*). This results can be mathematically proved, with a empty matrix with a only cell alive in the position i, j at the second iteration:

$$\psi_2(i,j) = 0_{i+1,j} \oplus 0_{i-1,j} \oplus 0_{i,j+1} \oplus 0_{i,j-1} = 0$$

$$\psi_2(i+1,j) = 0_{i+2,j} \oplus 1_{i,j} \oplus 0_{i+1,j+1} \oplus 0_{i+1,j-1} = 1$$

$$\psi_2(i-1,j) = 1_{i,j} \oplus 0_{i-1,j} \oplus 0_{i-1,j+1} \oplus 0_{i-1,j-1} = 1$$

$$\psi_2(i,j+1) = 0_{i+1,j+1} \oplus 0_{i-1,j+1} \oplus 0_{i,j+2} \oplus 1_{i,j} = 1$$

$$\psi_2(i,j-1) = 0_{i+1,j-1} \oplus 0_{i-1,j-1} \oplus 1_{i,j} \oplus 0_{i,j-2} = 1$$

This pattern is repeated at every iteration producing the result expected. Using this result it could be possible to compute in advance the status of the automaton, however this is more complex with the periodic border condition we implemented as when the border are reached the structure will start to present some interferences (as in figure 11) and it will start to repeat (regressing to the initial status) itself.

3.3 Conclusion

In conclusion the overall performance of the implementation are acceptable, and the bottleneck of the system is due to the poor graphic performance of *matplotlib* and can be solved using opengl or other graphical library.

The two rules implemented have both interesting behaviors but due to the lack of the implementation of the *infinite matrix* border condition is not possible to simulate many of the more interesting configurations.