Can Julia win the Game of Life?

Florian Schröder

Supervisor: Gerasimos Chourdakis

Abstract. This paper gives an overview of Conway's Game of Life (and cellular automata in general) and various implementations of it in the Julia programming language. It is mostly based on Daniel Shiffman's Nature of Code [Shi25].

1 Introduction

The study of complex systems often begins with simple rules. Cellular automata, introduced by John von Neumann and later popularized by Stephen Wolfram, are a prime example of how simple, local interactions can lead to surprisingly rich and varied global behavior. These models have become a cornerstone in the field of simulations, providing insight into phenomena ranging from biological growth to traffic flow and even the spread of diseases. Simulations are essential tools in science and engineering, allowing us to explore the behavior of systems that are too complex, dangerous, or expensive to experiment with directly. Cellular automata offer a unique approach to simulation: instead of relying on continuous equations, they use discrete states and local update rules. This makes them particularly well-suited for modeling systems where individual components interact in simple, repetitive ways. In this paper, we will explore the concept of cellular automata, focusing on Conway's Game of Life, and implement it in the Julia programming language.

2 Cellular Automata

2.1 Understanding cellular automata

A cellular automaton itself is a model consisting of cell objects, which have a few properties:

- All cells are arranged in a grid (1D, 2D, or even higher dimensions)
- Each cell has a state (the simplest case is a boolean state (1 or 0, alive or dead, ...), but it can also be an integer or even a more complex object)
- Each cell has a defined neighborhood (typically the cells directly adjacent to it, but it can also be more complex)

The last requirement is a function, that takes the neighborhood of a cell and returns the next state of the cell. Typically, this function is defined by a set of rules, which can be very simple or very complex. In a 1D cellular automaton, the neighborhood

of a cell is typically the cell itself and its two neighbors (left and right). In this case, there are $2^3 = 8$ possible configurations of the neighborhood (including the cell itself). Therefore, the ruleset can be represented by 8 bits, where each bit represents the next state of the cell for a specific configuration of the neighborhood. These 8 bits can be interpreted as a number in the range 0 to 255, which is the so-called *rule number* of the cellular automaton.

2.2 Graphical representation of 1D cellular automata

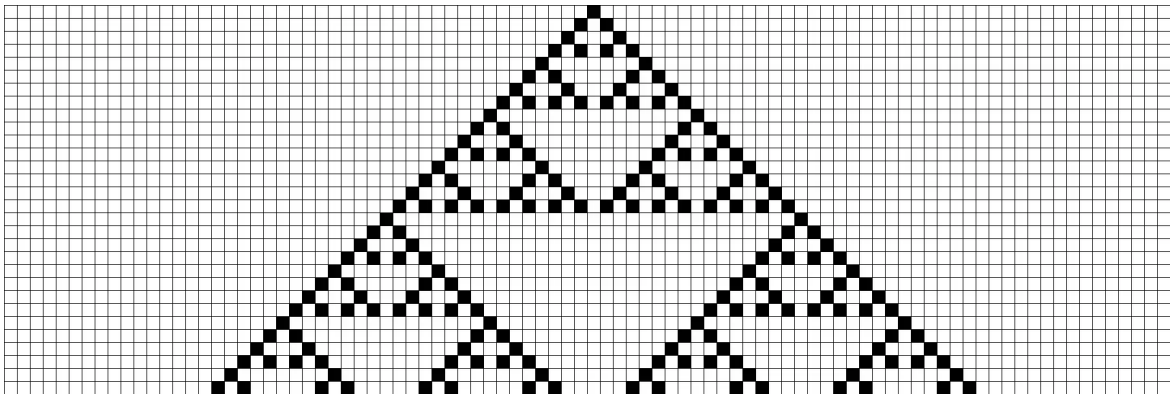

Fig. 1. Rule 90 (=1011010) visualized as a stack of generations
Source: https://natureofcode.com/static/ce7e94b17a9690f8b4182e120755263f/2c91d/07_ca_22.webp

Until now, we have only discussed the theory of cellular automata, but how can we visualize them? The simplest way is to use a 1D array of cells, where each cell is represented by a pixel. The state of the cell can be represented by the color of the pixel (e.g., black for alive and white for dead). This way, we can visualize the state of the cellular automaton at a specific time step. In a 1D cellular automaton, we can also visualize the history of the cellular automaton by displaying the state of the cellular automaton at each time step in a 2D grid. This way, we can see how the state of the cellular automaton evolves over time. An example of such a visualization is shown in Figure 1.

2.3 Jump to 2D cellular automata (Game of Life)

As already mentioned, a cellular automaton can also be defined in a grid of higher dimensions. The general idea is the same, but the neighborhood of a cell is defined a bit differently, as a cell now has 9 neighbors (the cell itself and the 8 cells surrounding it). There would be $2^9 = 512$ possible configurations of the neighborhood, which would lead to a ruleset of 512 bits. However, this would be quite complex and not very useful. Instead, most of the time, a 2D cellular automaton is defined by a set of rules that are based on the number of alive neighbors of a cell. The most famous example of a

2D cellular automaton is **Conway's Game of Life**, which is defined by the following rules:

- A cell is born (state=1) if it has exactly 3 alive neighbors.
- A cell survives if it has 2 or 3 alive neighbors.
- A cell dies (state=0) if it has less than 2 or more than 3 alive neighbors.
- A dead cell remains dead if it has less than 3 alive neighbors.

2.4 Various interesting Game of Life patterns

There are many interesting patterns (formations of alive cells) in the Game of Life, which can be classified into different categories. Examples are:

- **Stable**: These patterns do not change over time. (See Figure 2)
- **Oscillators**: These patterns change their state periodically. (See Figure 3)
- **Spaceships**: These patterns seem to move across the grid. (See Figure 4)
- **Guns**: These patterns produce ("shoot") other patterns, typically spaceships, in a periodic manner.

Whats interesting about these patterns (or generally all patterns) is that they can be used to create complex structures, but they are also very sensitive to external influences. Even if a single cell is changed, the whole pattern can change drastically (often leading to a complete extinction of the pattern).

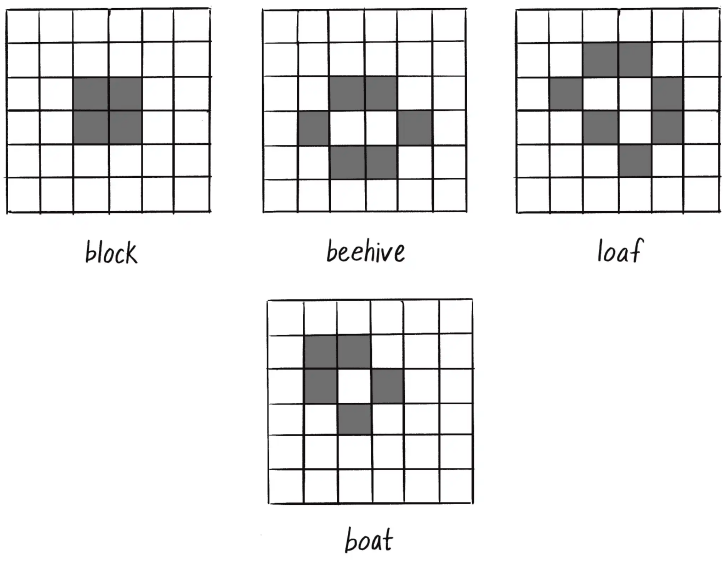

Fig. 2. Examples for stable formations
Source: https://natureofcode.com/static/4f9158d65163cbbb8fa41030a2000079/e80fe/07_ca_29.webp

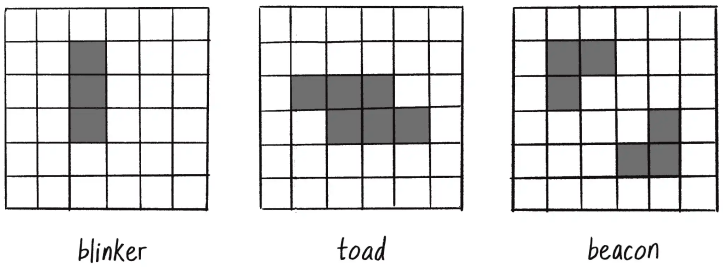

Fig. 3. Examples for oscillators
Source: https://natureofcode.com/static/04f7903e1010ffd716e12c5c86785894/d4cd6/07_ca_30.webp

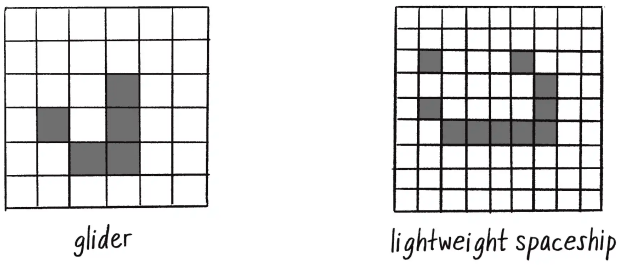

Fig. 4. Examples for spaceships
Source: https://natureofcode.com/static/bbac4c485b936395226f8b6addd24e19/46ba6/07_ca_31.webp

3 Implementation

3.1 Implementation of 1D in Julia

First, I want to implement the most basic version of a 1D cellular automaton in Julia. This will serve as a foundation for understanding the principles of cellular automata and how they can be implemented in Julia. The final code for this implementation will be available on GitHub TODO ADD LINK. As explained in subsection 2.1, a 1D cellular automaton can be represented by a 1D array of cells, where each cell is represented by a pixel. We need a function, that takes the neighborhood of a cell and returns the next state of the cell, based on the ruleset (defined by the rule number). As previously mentioned, the rulenumber can be represented by an 8-bit number, where each bit represents the next state of the cell for a specific configuration of the neighborhood. To check the next state of a cell, we convert the neighborhood of cell into a number and shift the rulenumber by this number of bits to the right. The least significant bit of the result is the next state of the cell. The main part ist the `simulate_ca` function, which takes the initial state, the rule number and the number of generations and returns all the generated states as a vector of vectors.

3.2 Implementation of 2D (Game of Life) in Julia

Here I'll differentiate between simple solutions (like a bool matrix for storing) and an object-oriented approach which may be even complexer

3.3 Comparison with other programming languages

Here I compare the performance with languages, I think I'll choose JavaScript and Python. I don't think that I'll provide implementations with these languages, rather a brief overview of them.

4 Discussion

5 Conclusions

Here I will summarize the findings and implications of the research conducted.

References

Shi25. Daniel Shiffman. The Nature of Code, Chapter 7: Cellular Automata. https://natureofcode.com/cellular-automata/, 2025. Accessed: 2025-05-12. 1