Can Julia win the Game of Life?

Florian Schröder

Supervisor: Gerasimos Chourdakis

Abstract. Numerical simulation is a fundamental tool in science and engineering, enabling the study of complex systems that are difficult or impossible to analyze analytically. Conway's Game of Life is a well-known example of a cellular automaton, illustrating how simple, local rules can generate complex, emergent behavior, and thus serves as a valuable study case for understanding simulation, complexity, and emergent phenomena. This paper gives an overview of Conway's Game of Life (and cellular automata in general) and some 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

The cellular automata we are going to discuss in this paper are based on the concept of Stephen Wolfram. A cellular automaton itself is a model consisting of cell objects, which have the following 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 (update 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. Conventionally, the leftmost bit represents the state if neighborhood is 111 (all three neighbors are alive), the next bit represents the state if the neighborhood is 110, and so on, until the rightmost bit represents the state if the neighborhood is 000 (all three neighbors are dead).

2.2 Graphical representation of 1D cellular automata

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.

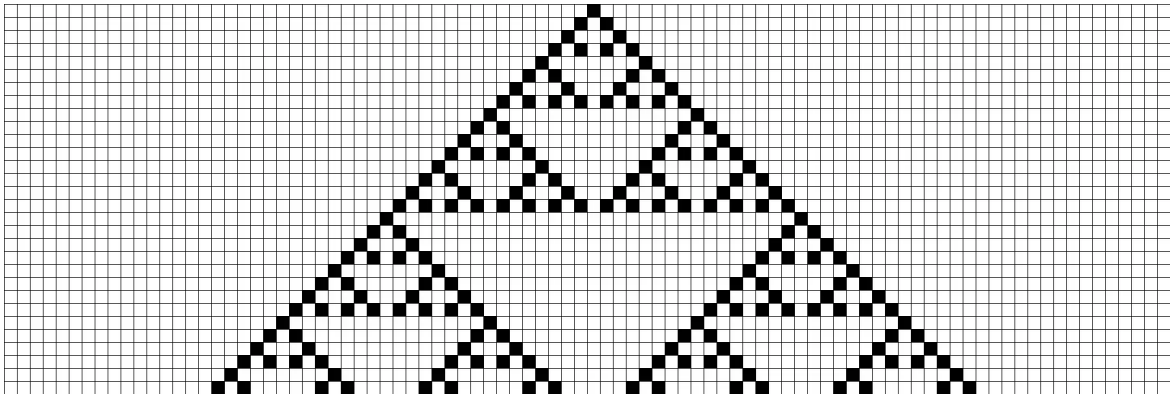

Fig. 1. The first 30 generations of Rule 90 (=1011010) visualized as a stack of generations (if the starting configuration is just a single alive cell in the middle of the grid) beginning at the top of the grid
Source: https://natureofcode.com/static/ce7e94b17a9690f8b4182e120755263f/2c91d/07_ca_22.webp

2.3 Wolfram Classification

We now have a basic understanding of cellular automata and how they can be visualized. By providing various rule numbers, we can see different behaviors of the cellular automaton. Stephen Wolfram classified the behavior of cellular automata into four classes:

- **Class 1: Uniformity** - The system evolves to a stable state, where all cells are in the same state (e.g., all dead or all alive).
- **Class 2: Repetition** - The cells evolve to a stable pattern that repeats over time.
- **Class 3: Random** - The system evolves to a chaotic state, where the cells change their state in a seemingly random manner.
- **Class 4: Complexity** - Sort of a mix between Class 2 and Class 3, where the system exhibits both periodic and chaotic behavior.

Figure 1 is an example of a Class 4 cellular automaton.

2.4 2D cellular automata (Game of Life)

As already mentioned, a cellular automaton can also be defined in a grid of higher dimensions. When going to two dimensions, the general idea stays the same, but the neighborhood of a cell is defined a bit differently, as the neighborhood of a single cell consists of 9 cells (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 (if we would also use a rule number here, the maximum number would be $2^{512} - 1 \approx 1.34 \times 10^{154}$). However, this is not practical, as this number is way too large to be used in practice. 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 or more than 3 alive neighbors.

2.5 Graphical representation of 2D cellular automata (Game of Life)

The graphical representation of a 2D cellular automaton is similar to the one of a 1D cellular automaton, but now we have a grid of cells instead of a 1D array. Therefore, we can't really visualize the history of the cellular automaton in a 2D grid. That's why we typically show the state of the cellular automaton over time as a sequence of frames, where each frame represents the state of the cellular automaton at a specific time step.

2.6 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). In reality, it is just a periodic change of the pattern, which gives the impression of movement.
- **Guns**: These patterns produce ("shoot") other patterns, typically spaceships, in a periodic manner. They are designed in such a way that they create a new pattern at regular intervals while maintaining their own structure.

An interesting observation 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). This matches the sensitivity of complex systems in the real world, where small changes can lead to large effects.

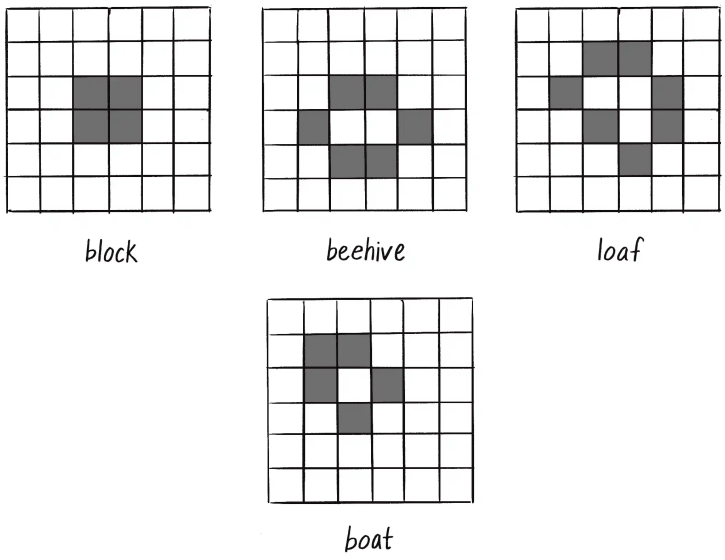

Fig. 2. Examples for stable formations
Source: https://natureofcode.com/static/4f9158d65163cbbb8fa41030a2000079/e80fe/07_ca_29.webp

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

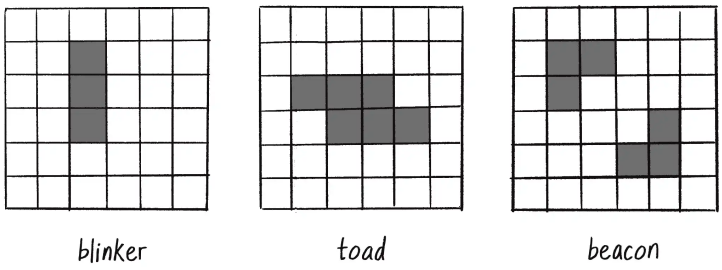

Fig. 3. Examples for oscillators
Source: https://natureofcode.com/static/04f7903e1010ffd716e12c5c86785894/d4cd6/07_ca_30.webp

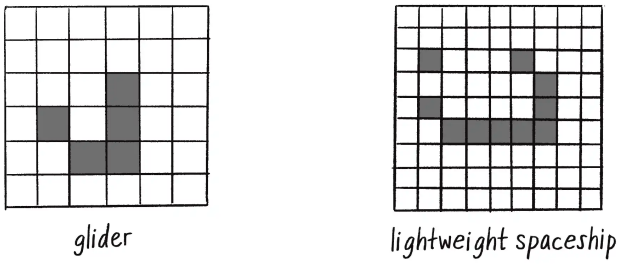

Fig. 4. Examples for spaceships
Source: https://natureofcode.com/static/bbac4c485b936395226f8b6addd24e19/46ba6/07_ca_31.webp

and how they can be implemented in Julia. The graphical representation of the cellular automatons is done using the `Plots` package [Jul25b], using the `GR` [Jul25a,GR25] backend. The final code for this implementation will be available on GitHub here[1]. 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 a 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 is 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. This history can then be visualized by displaying the state of the cellular automaton at each time step in a 2D grid, which can be done using the `plot_ca_history` function. Figure 5 shows the result of the implementation of Rule 89 with 101 cells and 50 generations, where the starting state is a single alive cell in the middle of the grid.

[1] https://github.com/TecToast/JuliaGameOfLife/blob/main/code/cellular_1D.jl

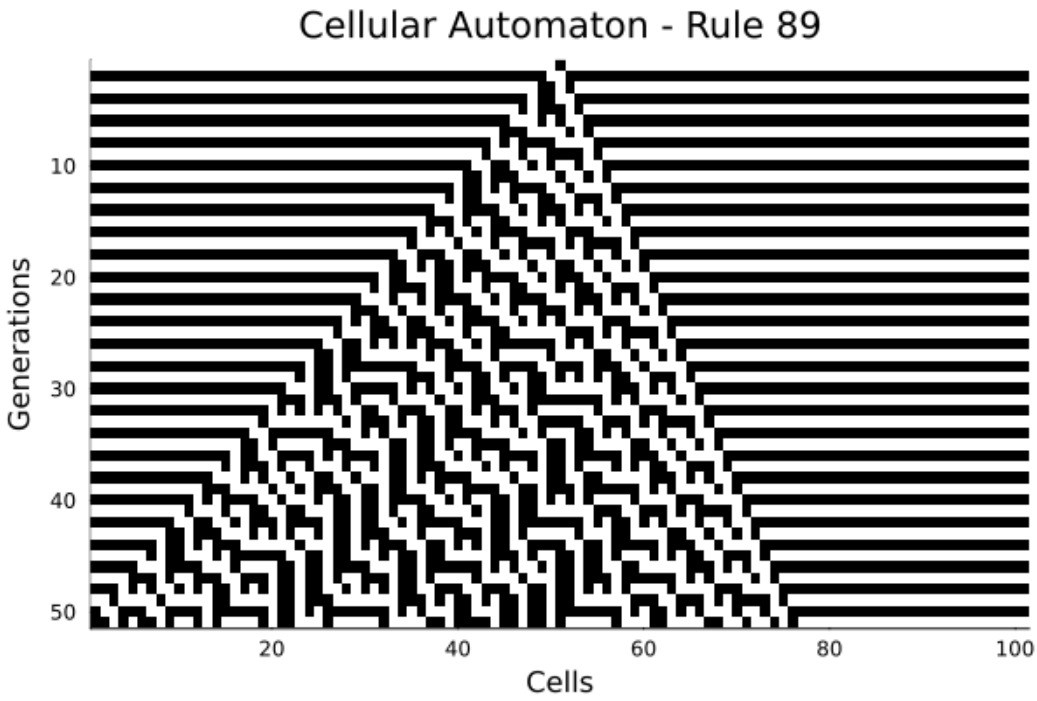

Fig. 5. Generated diagram of Rule 89 with 101 Cells, 50 Generations

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

The core idea of the 2D cellular automaton is the same as for the 1D cellular automaton, but the neighborhood of a cell is defined differently. As explained in subsection 2.5, we can visualize the state of the cellular automaton over time as a sequence of frames. The Julia implementation differentiates between two rendering modes:

- **Directly** (using the `run_game_of_life_live` method): The frames are rendered directly to the screen (using the GR backend). This is the most straightforward way to visualize the cellular automaton and it's easy to view the state of the cellular automaton at each time step.
- **GIF** (using the `create_game_of_life_gif` method): The frames are saved as a GIF file, which can be viewed later. This is useful for longer simulations or for sharing the results with others.

The most simple way to implement the Game of Life is to use a boolean matrix, where each cell is either alive (1) or dead (0). This approach can be found under the url in the footnote[2]. First, we initialize the grid with a random state, where each cell has a default chance of 30% to be alive. In each iteration, we iterate over all cells and check the number of alive neighbors and set the next state of the cell based on the default rules of the Game of Life. The neat thing about this implementation is that it can be easily extended to support different rulesets, by simply changing the `update_cell`

[2] https://github.com/TecToast/JuliaGameOfLife/blob/main/code/gameoflife.jl

function. There is also no reason why the cells have to be boolean, they can also be integers or even more complex objects. The most common starting configuration is a random state, where the user can define the chance of a cell to be alive (also called density). Here is a Game of Life simulation with a density of 0.25 and 1000 generations, generated the Julia implementation in the footnote[3].

3.3 Application: Infection Simulation

In the previous section, I implemented a simple version of the Game of Life using a boolean matrix. However, this implementation is quite limited, as it only allows for two states (alive and dead) and does not allow for more complex interactions between cells. We have multiple options to extend the Game of Life to support more complex interactions between cells. One option is to use a more complex data structure for the cells, which allows for more properties and behaviors. Another option is to bring in randomness into the ruleset (e.g., by using probabilities for the next state of the cell), as the neighborhood "function" doesn't have to be deterministic. I created a more complex implementation, which uses a custom `Cell` type, which currently only has a color property, but can be extended to have more properties in the future. The code for this implementation can be found here[4]. The previous implementation has to be adjusted (especially the rendering to make use of multiple colors) to use the `Cell` type, but the core idea is the same as for the boolean matrix. In contrast to the default Game of Life, which was rather abstract, this implementation allows for more real simulations of complex systems, such as biological systems, where each cell can have different properties and behaviors. One example of such a simulation is an *Infection Simulation*, where we have two types of cells: infected and healthy. The infected cells spread the infection to the healthy cells, while the healthy cells can reproduce, leading to a complex interaction between the two types of cells. In my example, I initialized the grid with 35% of healthy cells (blue) and 15% of infected cells (red). I specified the rules to be as follows:

- **Healthy Rules:**
  - If a healthy cell has 1 infected neighbor, it has a 20% chance of becoming infected.
  - If a healthy cell has 2 infected neighbors, it has a 40% chance of becoming infected.
  - If a healthy cell has more than 2 infected neighbors, it is guaranteed to become infected.
  - If a healthy cell has less than 2 or more than 5 infected neighbors, it dies.
  - Otherwise, it stays healthy.
- **Infected Rules:**
  - If a cell is infected, it has a 1% chance of dying.
  - If an infected cell has less than 3 or more than 6 infected neighbors, it dies.
  - Otherwise, it stays infected.

[3] https://github.com/TecToast/JuliaGameOfLife/blob/main/gifs/game_of_life.gif
[4] https://github.com/TecToast/JuliaGameOfLife/blob/main/code/gameoflife_extended.jl

– **Dead Rules:**
  - If a cell is dead and has exactly 3 healthy neighbors, it becomes healthy.
  - Otherwise, it stays dead.

If you run the simulation, you will see how the cells seem like they are fighting each other, with the infected cells spreading the infection to the healthy cells, while the healthy cells try to survive and reproduce. The simulation for 1000 generations can be found here[5]. The end frame of this specific simulation can be seen in Figure 6. We can observe that the healthy cells (blue) form a pipe like structure, while the infected cells (red) form are more clumped together and are almost always connected to healthy cells (as soon as this is not the case, it is just a matter of time that they die, as they can't reproduce themselves). The start configuration of the Infection Simulation can be seen in Figure 7.

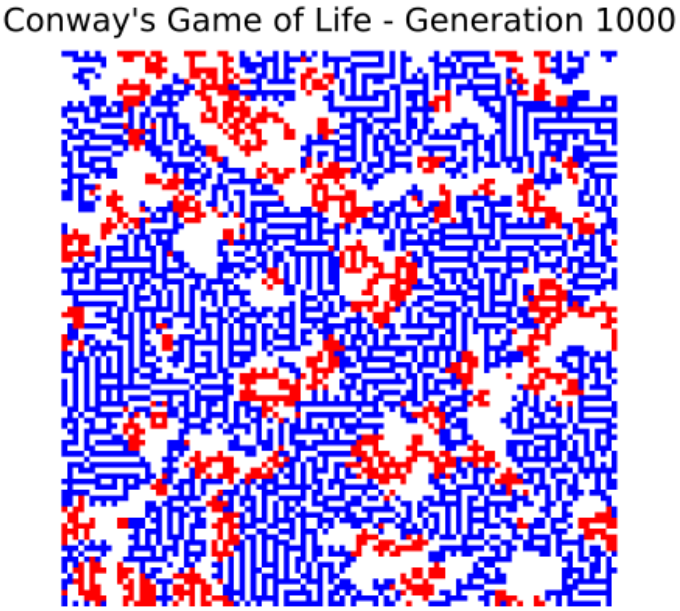

Fig. 6. Infection Simulation after 1000 generations

3.4 Comparison with other programming languages

When comparing the implementation of the Game of Life in Julia to other programming languages such as Python or JavaScript, several objective differences can be observed:

– **Performance:** Julia is designed for high-performance numerical computing and can approach or match the speed of C for array and matrix operations, which is beneficial for large-scale simulations.

[5] https://github.com/TecToast/JuliaGameOfLife/blob/main/gifs/infection_simulation.gif

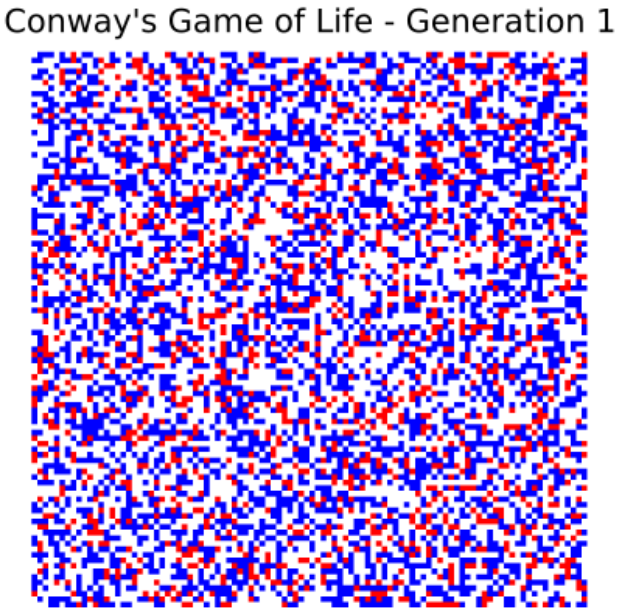

Fig. 7. Initial state of the Infection Simulation, with 35% healthy cells (blue) and 15% infected cells (red) at the beginning. The rest of the cells are dead (white). The rules are explained in the text.

- **Syntax:** Julia's syntax is concise and expressive, allowing for mathematical operations and array manipulations to be written in a style similar to mathematical notation, which can reduce code verbosity compared to languages like JavaScript.
- **Ecosystem:** Julia provides a growing ecosystem of packages for scientific computing, visualization, and data processing, such as `Plots.jl`, which facilitate rapid prototyping and visualization of simulation results.

Especially the performance aspect is important for simulations, as they often involve large arrays and matrices, which can be computationally expensive to process, if you want to simulate thousands or even millions of generations.

4 Discussion

In this section, we will discuss how simulations using cellular automata relates to PDE-based simulations.

Partial Differential Equations (PDEs) are a powerful tool for modeling continuous systems, such as fluid dynamics, heat transfer, and wave propagation. However, they can be difficult to solve analytically and often require numerical methods for their solution. Cellular automata, on the other hand, are discrete models that can be used to simulate systems with local interactions. While they are not a direct replacement for PDEs, they can be used to model systems that exhibit similar behavior.
This section will be extended

5 Conclusions

In this paper, we explored the fundamental concepts of cellular automata, with a particular focus on Conway's Game of Life, and demonstrated how these systems can be implemented and visualized using the Julia programming language. Through both simple and extended examples, we showed how local rules can give rise to complex and sometimes unpredictable global behavior. Julia's strengths in numerical computing and visualization make it a suitable choice for simulating and experimenting with cellular automata. The flexibility of the language also allows for easy extension to more complex models, such as those simulating biological processes or infections. Overall, cellular automata remain a powerful tool for understanding emergent phenomena, and Julia provides an accessible and efficient platform for their study and application.

References

GR25. Gr framework documentation. https://gr-framework.org/, 2025. 5

Jul25a. Gr.jl repository. https://github.com/jheinen/GR.jl, 2025. 5

Jul25b. Julia plots documentation. https://docs.juliaplots.org/stable/, 2025. 5

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