A CONTINUOUS APPROXIMATION OF CONWAY'S GAME OF LIFE USING SAGE

Z. FOLWICK, S. GANGULY AND G. SISODIA

1. Introduction

1.1. **What is cellular automata?** A cellular automaton (pl. cellular automata, abbrev. CA) is a discrete model consisting of a regular grid of cells, each in one of a finite number of states, such as "On" and "Off". The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighborhood (usually including the cell itself) is defined relative to the specified cell. For example, the neighborhood of a cell might be defined as the set of cells a distance of 2 or less from the cell.

An initial state (time t=0) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood.

For example, the rule might be that the cell is "On" in the next generation if exactly two of the cells in the neighborhood are "On" in the current generation, otherwise the cell is "Off" in the next generation. Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously, though exceptions are known.

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician **John Horton Conway** in 1970. The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Date: March 9, 2012.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed-births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

Ever since its publication, Conway's Game of Life has attracted much interest, because of the surprising ways in which the patterns can evolve. Life provides an example of emergence and self-organization. It is interesting for computer scientists, physicists, biologists, biochemists, economists, mathematicians, philosophers, generative scientists and others to observe the way that complex patterns can emerge from the implementation of very simple rules. The game can also serve as a didactic analogy, used to convey the somewhat counter-intuitive notion that "design" and "organization" can spontaneously emerge in the absence of a designer. For example, philosopher and cognitive scientist Daniel Dennett has used the analogue of Conway's Life "universe" extensively to illustrate the possible evolution of complex philosophical constructs, such as consciousness and free will, from the relatively simple set of deterministic physical laws governing our own universe.

1.2. **Life patterns.** A good way to get started in Life is to try out different patterns and see what happens. Even completely random starting patterns rapidly turn into Life objects recognizable to anyone with a little experience. In this section, we follow a simple-looking pattern called the R-pentomino. It starts out with just five cells, but gets complicated very fast. The R-pentomino is the first pattern Conway

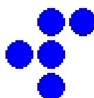
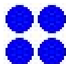


FIGURE 1. R-pentomino

found that defied his attempts to simulate by hand. In fact, the pattern eventually becomes "stable" or easy to predict, but this does not happen until 1103 steps have passed. Some of the earliest computer programs for Life were written to determine the fate of this small pattern. This was a challenging problem for many computers of that time, but a modern PC can run the complete sequence of steps many times in one second.

1.3. **Still life objects.** Some of the most common objects in Life remain the same from step to step. No live cells die and no new cells are born. Conway, who is fond of making puns, called this kind of object a "still life." For an object to be a still life, every live cell most have 2 or 3 live neighbors, and every dead cell may have any number of neighbors except 3. The most common still life is called the block. It is simply a 2x2 square of live cells:



You will see it appear many times as you run the R-pentomino. Every live cell has exactly three neighbors, but no dead cell has more than two neighbors. Some other still lifes you will see are:

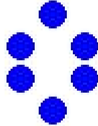


FIGURE 2. beehive

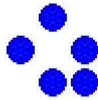


FIGURE 3. boat

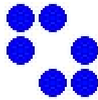


FIGURE 4. ship

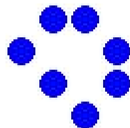


FIGURE 5. loaf

1.4. **Oscillators.** Oscillators are objects that change from step to step, but eventually repeat themselves. The simplest kind are period-2 oscillators, or those that repeat themselves after two steps. The most common is the blinker, which consists of three cells.

1.5. **Gliders.** These common moving patterns, called gliders, consist of just 5 cells.

You will see that after four steps, it looks just like it did when it started. The only difference is that it is shifted along a diagonal. Repeat this process, and you have a moving Life object, which in general we call a spaceship. Remember, the rules said nothing about movement; moving patterns just appear. This is a simple but convincing demonstration of emergent complexity.

FIGURE 6. oscillator

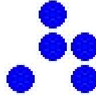


FIGURE 7. glider

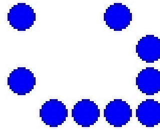


FIGURE 8. light weight

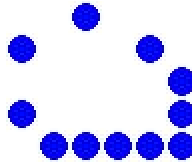


FIGURE 9. medium weight

1.6. **Other interesting objects.** There are some early discoveries that do not show up in the R-pentomino, but which sometimes show up from random starting states. The orthogonal spaceships move left, right, up, or down instead of on diagonals like gliders. These are much less common than gliders, but very important in certain kinds of patterns. They come in three sizes:

Finally, here are some simple starting patterns that develop into oscillators. A row of 10 live cells becomes the period-15 oscillator called the pentadecathlon:

1.7. **Do all patterns stabilize eventually?** We say a pattern has stabilized when it becomes obvious that the population has stopped growing. For example, we say the R-pentomino stabilizes at step 1103 because at this point it consists of just gliders and oscillators, and it is clear that the gliders will never collide with each other or with any oscillators. It was observed early on in the study of Life that random starting states all seem to stabilize eventually. Conway offered a prize for any example of patterns that grow forever. Conway's prize was collected soon after its announcement, when two different ways were discovered for designing a pattern that grows forever. The first of these patterns is the period-30 glider gun, which is based on the interaction of two queen bee shuttles. Where these shuttles collide, instead of producing beehives, they produce a new glider. This glider moves away in time for the process to repeat itself 30 steps later

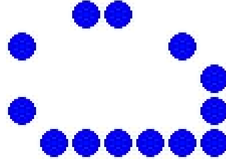


FIGURE 10. heavy weight

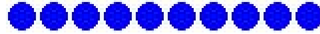


FIGURE 11. pentadecathlon

Another kind of pattern that grows forever is called a puffer. Unlike the glider gun, which remains stationary and produces moving objects, puffers move while producing objects, which may be either moving or stationary. Here is one of the simplest, and earliest puffers.

2. The algorithm and code

2.1. **Continuous approximation.** Let M be a Game of Life universe (realized as a two-dimensional matrix of 0s and 1s) and let M' be the the universe after one generation. We view the Game of Life rules in the following way: let

$$A(x) = \begin{cases} 1 & \text{if } x \in \{2,3\} \\ 0 & \text{if } x \in \{0,1,4,5,6,7,8\} \end{cases}$$

and

$$D(x) = \begin{cases} 1 & \text{if } x = 3 \\ 0 & \text{if } x \in \{0,1,2,4,5,6,7,8\}. \end{cases}$$

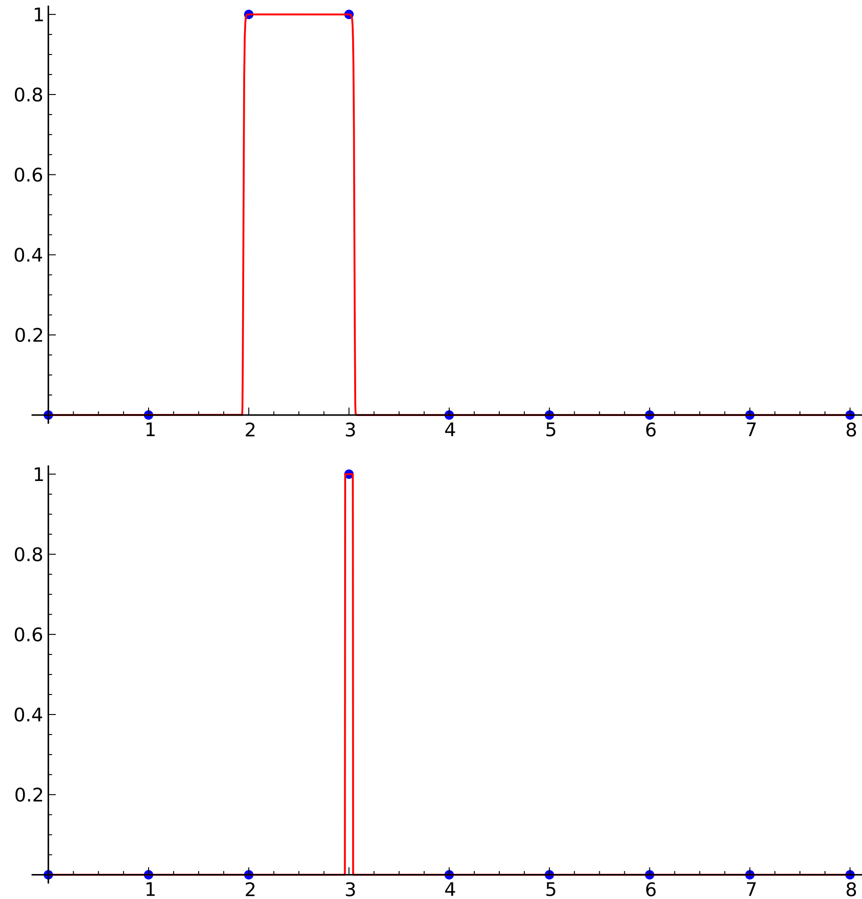
Then for N the neighbors matrix of M,

$$M'_{i,j} = \begin{cases} A(N_{i,j}) & \text{if } M_{i,j} = 1 \\ D(N_{i,j}) & \text{if } M_{i,j} = 0. \end{cases}$$

We approximate $A(x)$ and $D(x)$ continuously:

$$A(x) \approx \exp\left[-(1.8(x-5/2))^{100}\right], \qquad D(x) \approx \exp\left[-(25(x-3))^{100}\right].$$

The approximations are graphed below in red ($A(x)$ and $D(x)$ in blue):



We now allow real-number entries between 0 and 1. We try to answer three questions:

(1) How does our approximation affect computation time?
(2) How good is our approximation?
(3) What happens if we (slightly) deform the rules?

2.2. **The `twoDCA` class.** We develop the `twoDCA` class to encapsulate the universe and rules of a Life-like CA. It has as class variables

- the current universe `M` as a two-dimensional numpy array, and the shape (`r`, `c`) of `M`,
- the original universe (or seed) `orig`,
- a function `Alive` to apply to "alive" cells,
- a function `Dead` to apply to "dead" cells, and
- a function `neighbors` to return the neighbors matrix.

The neighbors matrix is the matrix to which the functions `Alive` and `Dead` are applied. For Life, we take the neighbors matrix N of a matrix M to be

$$N_{i,j} = \# \text{ of neighbors of cell } (i,j) \text{ in } M. \tag{1}$$

We call the function which returns the Life neighbors matrix `LifeNeighbors`.

The class methods are

- `nextMatrix`, which updates `M` to the next generation by applying the prescibed rules,
- `gener(m)`, which updates `M` by `m` generations (by calling `nextMatrix` `m` times),
- `npcount`, which returns the population of `M` (the sum of all entries),
- `resetM(N)`, which sets `M` to `N`,
- `mplot`, which returns a `matrix_plot` of `M`, and
- `anigen(n)`, which returns an animation of `matrix_plot`s of the next `n` generations of `M`.

2.3. **The algorithm in detail.** In `nextMatrix`, we update `M` to the next generation via

$$\texttt{M} = \texttt{M} * \texttt{Alive}(\texttt{N}) + (1 - \texttt{M}) * \texttt{Dead}(\texttt{N}) \tag{2}$$

where `N = neighbors(M)`, `Alive` and `Dead` are applied entry-wise, and $*$ is entry-wise multiplication. For Life, we (naively) define

```
def LifeAlive(n):
    if n == 2 or n == 3:
        return 1
    else:
        return 0

def LifeDead(n):
    if n == 3:
        return 1
    else:
        return 0
```

Along with the `LifeNeighbors` function described in (1), the instance

```
Life = twoDCA(M, LifeAlive, LifeDead, LifeNeighbors)
```

represents classical Life.

We use cython for a fast implementation of `LifeNeighbors` (we name the cython function `cyneighbors`).

2.4. **The boundary.** Currently, the `cyneighbors` function computes the neighbors of a cell on the boundary of the array to be zeros. This means that the universe of a `twoDCA` instance using `cyneighbors` will maintain a boundary of zeros throughout its generations. When using `cyneighbors`, leave healthy buffers of zeros on all sides for accurate results.

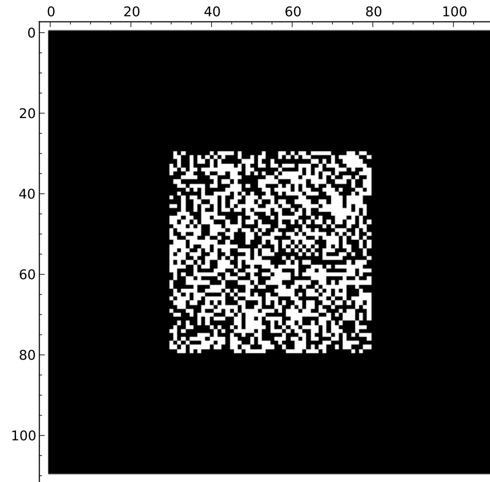
2.5. **Read from file.** There are many libraries of interesting Game of Life seeds. One such library, at

`http://www.radicaleye.com/lifepage/patterns/contents.html`

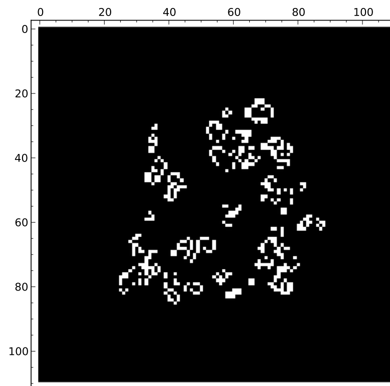
stores the seeds as files in the .LIF format. We implement a function which takes as input a file in the .LIF format and returns the corresponding seed as a numpy matrix. In the next section, we analyze how our code behaves on some classical Game of Life seeds.

3. Analysis

We show that our approximation is a good one. We start with a 50×50 matrix with random entries (either 0 or 1) with a border of zeros on all sides, pictured below as a `matrix_plot`.



The matrix after 40 generations is



We notice various still lifes, blinkers and gliders.

Let's test our code on some classical patterns from the library mentioned above, like the Gosper glider gun (Figure 12).

Figure 13 shows the universe after 180 generations. We observe that our approximation behaves very much like classical Game of Life with respect to this pattern.

We can input larger patterns, like the prime number computer (Figure 14), which sends a light weight spaceship to the left after $120n + 100$ generations if n is prime. However, our code is too slow to do serious analysis of such large patterns. The prime number computer builds to the right indefinitely, and even at this fixed size

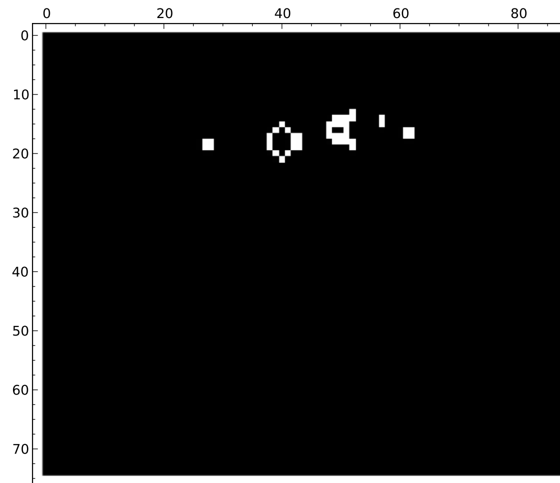


FIGURE 12. Glider gun

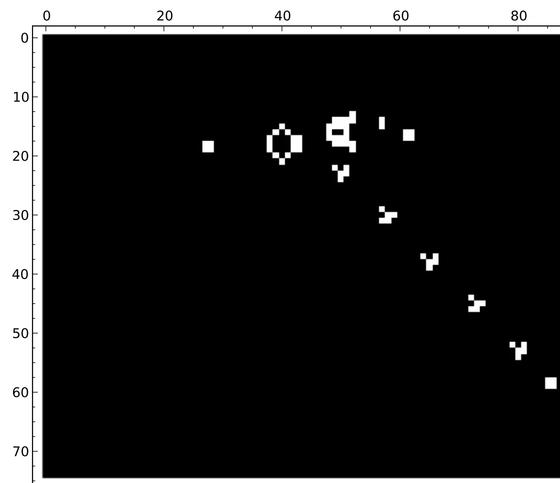


FIGURE 13. Glider gun after 180 generations

(about 650×450), 100 generations takes about 7.5 seconds, so one would see the spaceship corresponding to 5 after more than 53 seconds.

3.1. **Vastly different rules.** Our framework for two-dimensional CA can handle a wide variety of rules. As an example, let's implement the rule which allows entries strictly between 0 and 1 and replaces a cell with the average of the cells around it. Then our alive (and dead) function would be

```
def aliveAvg(n):
    return n/8
```

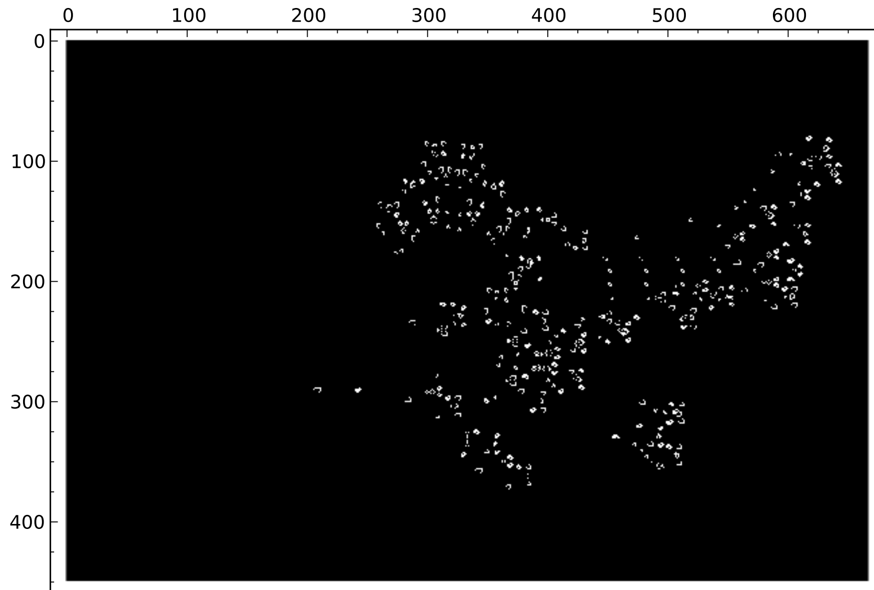


FIGURE 14. Prime number computer

and an instance with a random seed M

```
avgCA = twoDCA(M, aliveAvg, aliveAvg, cyneighbors).
```

Figure 15 shows M, 16 the universe after 1 generation and 17 the universe after 2 generations.

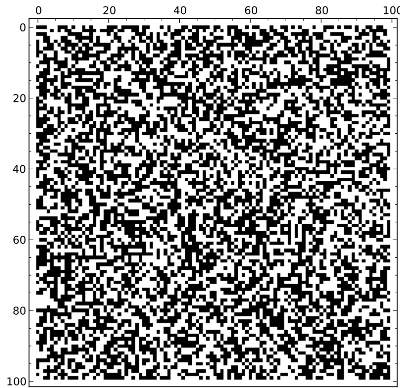


FIGURE 15. M

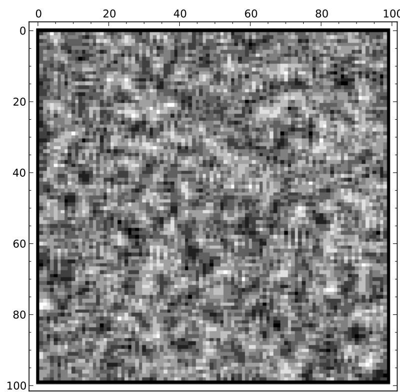


FIGURE 16. M after 1 generation

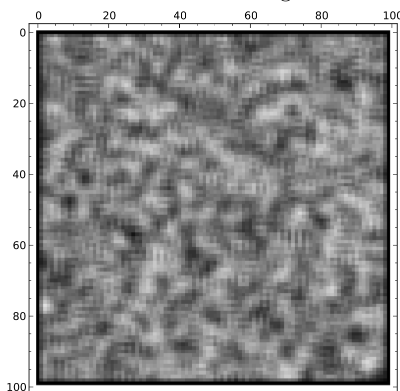


FIGURE 17. M after 2 generations

References

[1] Gardner, M. (1970). The fantastic combinations of John conway's new solitaire game. *Scientific American, 224,* 112-117.