# Cellular Automata

October 13, 2019

Cellular automata are a set of mathematical models consisting of a lattice of cells, each in a given state, along with a set of rules for how the cell states will change, based on their own conditions and the conditions of their neighbours. You will implement two well known automata, the Abelian Sandpile and the Game of Life.

## 1 The Abelian Sandpile model

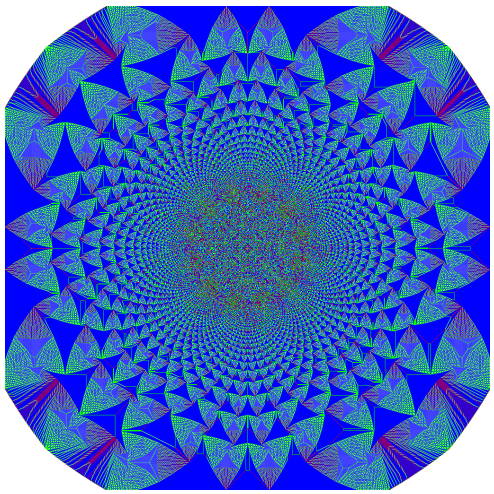


Figure 1: By Claudio Rocchini [CC BY 3.0]

A sandpile model consists of a 2d lattice of heights, $h_{i,j} \in [0, 1, 2 \ldots]$ (i.e positive integers), with $\sum h_{i,j} < \infty$. These may be thought of as the heights of piles of sand, which may eventually topple, passing on sand particles to other cells. A given initial configuration specifies the number of grains dropped in each cell, which evolve over time based on the rules of the model towards a final steady state.

### 1.1 The rules

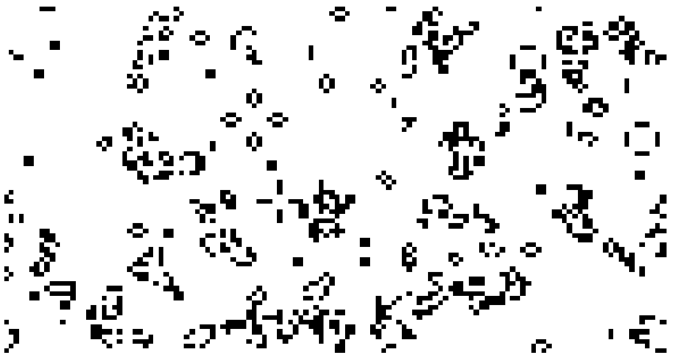
Any cell with $h_{i,j} \geq 4$ will eventually topple, sending particles out to its neighbouring cells

$$\begin{aligned} h_{i,j} &\rightarrow h_{i,j} - 4 \\ h_{i+1,j} &\rightarrow h_{i+1,j} + 1 \\ h_{i,j+1} &\rightarrow h_{i,j+1} + 1 \\ h_{i-1,j} &\rightarrow h_{i-1,j} + 1 \\ h_{i,j-1} &\rightarrow h_{i,j-1} + 1 \end{aligned}$$

where the states of the cell and its neighbours immediately post topple only depend on the values before and the fact that the topple occured.

With "sink" boundaries that simply absorb particles and never release them, these topplings will continue until every cell is of height 3 or below. It turns out that this end state is independent of the order in which the cells topple, so that there is a unique mapping from initial to final, stable configuration (hence the use of the term Abelian, which implies that the order of topplings commute). In particular, this means the same outcome is achieved whether only one toppling, or many, are assumed to occur each iteration.

# 2 Conway's Game of Life



The Game of Life is an iterative system of rules for the evolution of a 2d cellular automaton devised by the British mathematician John Horton Conway in 1970 (`http://www.math.com/students/wonders/life/life.html`). At each step, each cell may take two states, "alive" or "dead", and may change its state on the subsequent step depending on its own state and that of its 8 neighbours (including diagonals).

## 2.1 The rules

Conway's game is for 0 players. That is to say, it takes an initial state of living and dead cells on a two-dimensional grid and then works forward in time automatically. The rules are:

### For living cells

For each stage:

- A living cell with 0 or 1 neighbours dies of loneliness.
- A living cell with 2 or 3 neighbours survives to the next generation.
- A living cell with 4 or more neighbours dies from overcrowding.

The state of the mesh of cells at time $n + 1$ thus depends only on their states at time $n$.

**For dead cells**

For each stage:

- A dead cell with 3 neighbours becomes live.
- A dead cell with 0–2 or 4–8 neighbours stays dead.

Boundaries in a finite mesh can commonly be dealt with in two ways. Either an outer circle of "always dead" cells, similar to the sink cells in the sand pile model, or assuming doubly periodic bounaries, in which cells on the left edge are assumed to neighbour those on the right, and those on the bottom neighbour those on the top.

### 2.2 Extension 1: Life on a triangular tessellation

A second method to extend Life is to use a mesh pattern which does not map nicely to multi-dimensional arrays. There are a number of patterns which tessellate to cover a two dimensional plane, some regular, some irregular. An interesting choice is to use equilateral triangles, along with rules:

- a live cell survives with 4, 5 or 6 neighbours,
- a dead cell turns on with 4 neighbours.

The figure below shows the neighbours for an upwards pointing and a downwards pointing triangle.

### 2.3 Extension 2: Generic Life

The Game of Life formula can be abstracted down to four things: (1) a vector of Boolean variables representing the current state of the cells; (2) a connectivity matrix to identify which cells neighbour each other; and a couple sets of integers, (3) the environment rule, which specifies the neighbour counts where live cells survive; and (4) the fertility rule, which specifies neighbour counts where dead cells come to life.

## 3 Problem specification

You must create a single python module file called `automata.py`, which when imported exposes functions called sandpile and `life` (plus `lifetri and life_generic` if you attempt the extension exercises) with the following signatures:

### 3.1 sandpile

```
def sandpile(initial_state):
    """
    Generate final state for Abelian Sandpile.
```

```
    Parameters
    ----------
    initial_state : array_like or list of lists
        Initial 2d state of grid in an array of integers.

    Returns
    -------

    array_like
        Final state of grid in array of integers.
    """
```

When imported and called in the following manner

```
import numpy
import automata
# Generate random initial state.
X = numpy.random.randint(0, 6, (16, 16))
# call module.
Z = automata.sandpile(X)
```

the function should calculate the final sandpile state `Z` for initial state `X`. You may import any module in the standard Python 3 libraries, as well as `numpy` and `matplotlib`. No other nonstandard modules should be imported.

### 3.2 life

```
def life(initial_state, nsteps, periodic=False):
    """
    Perform iterations of Conway's Game of Life with specified boundary
    Conditions.

    Parameters
    ----------
    initial_state : array_like or list of lists
        Initial 2d state of grid in an array of booleans.
    nsteps : int
        Number of steps of Life to perform.
    periodic : bool, optional
        If true, then grid is assumed periodic.

    Returns
    -------

    array_like
```

```
        Final state of grid in array of booleans
    """
```

When imported and called like

```
import numpy
import automata
# Generate random initial state.
X = numpy.random.random((16, 16))>0.3
# call module.
Z = automata.life(X, 10)
```

the function should execute `nsteps` steps of Life. You may import any module in the standard Python 3 libraries, as well as `numpy` and `matplotlib`. No other nonstandard modules should be imported.

### 3.3 lifetri

The `lifetri` function should have the following signature:

```
def lifetri(initial_state, nsteps, periodic=False):
    """
    Perform iterations of Conway's Game of Life on
    a triangular tessellation.

    Parameters
    ----------
    initial_state : array_like or list of lists
        Initial state of grid on triangles.
    nsteps : int
        Number of steps of Life to perform.
    periodic : bool, optional
        If true, then grid is assumed periodic.

    Returns
    -------

    array
         Final state of tessellation.
    """
```

When imported and called like

```
import numpy
import automata
```

```
# Generate random initial state.
X = numpy.random.random((16, 16))>0.3
# call module.
Z = automata.lifetri(X, 10)
```

the function should execute `nsteps` steps of Life on a triangular mesh. You may import any module in the standard Python 3 libraries, as well as `numpy` and `matplotlib`. No other nonstandard modules should be imported.

### 3.3.1 The triangular data structure

As a note on the input data structure, you may assume that the data represents a set of arrays of the Boolean (true/false) states of rows in a mesh of regular tiangles and that, just as in the figure given, the mesh has an upward pointing triangle in its top left corner, and the same number of triangles in each row. For the periodic case, you may assume that the area given is sensible (i.e with an even number of triangles given in each row and an even number of rows).

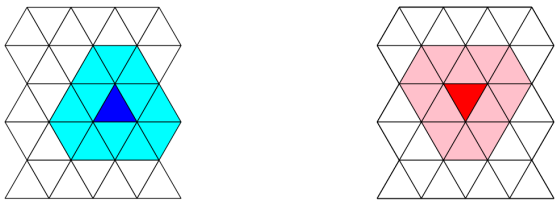


Figure 2: The 12 neighbours of upwards and downwards pointing triangles.

## 3.4 life_generic

The `life_generic` function should have the following signature:

```
def life_generic(matrix, initial_state, nsteps, environment, fertility):
    """
    Perform iterations of Conway's Game of Life for an arbitrary
    collection of cells.

    Parameters
    ----------
    matrix : 2d array of bools
         a boolean matrix indicating neighbours for each row
    initial_state : 1d array_like or list of bools
         Initial state vectr.
    nsteps : int
```

```
        Number of steps of Life to perform.
    environment : set of ints
       neighbour counts for which live cells survive.
    fertility : set of ints
       neighbour counts for which dead cells turn on.

    Returns
    -------

    array
        Final state.
    """
```

When imported and called like

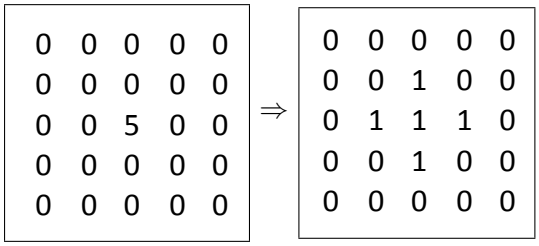
```
import numpy
import automata
M = numpy.zeros((8*8, 8*8), bool)
for i in range(1,7):
    for j in range(1,7):
        M[8*i+j, 8*i+j+1] = True
        M[8*i+j, 8*i+j-1] = True
        M[8*i+j, 8*(i+1)+j] = True
        M[8*i+j, 8*(i-1)+j] = True
# Generate random initial state.
X = numpy.random.random((8*8))>0.3
# call module.
Z = automata.life_generic(M, X, 10, {2,3}, {3})
```

the function should execute `nsteps` steps of Life for the specified connectivity. You may import any module in the standard Python 3 libraries, as well as `numpy` and `matplotlib`. No other nonstandard modules should be imported.

## 3.5 Checking your code

### 3.5.1 The Sandpile

Some simple patterns are easy to run through the algorithm by hand.



For more complicated series, two examples are available in the `tests/` folder.

### 3.5.2 The Game of Life

There are some well known initial conditions which either remain constant, or follow short periodic patterns. For the purposes of checking your code, the most relevant ones are the 2d "blinker" and the "glider".

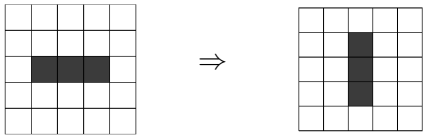


Figure 3: The blinker remains centred and rotates with period 2.

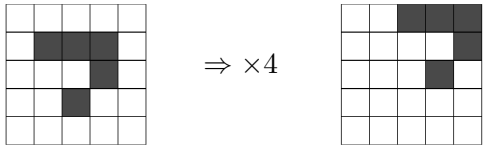


Figure 4: The glider translates itself diagonally over 4 steps.

there is also a triangular glider (see figure 5)

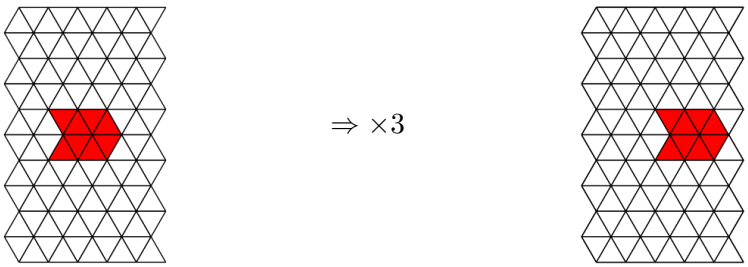


Figure 5: The glider translates itself horizontally over 3 steps. Note that there are 5 other possible orientations.

The generic system can (sometimes somewhat slowly) mimic any of the systems above, and can thus be tested using any of these patterns, as well as the 3D glider for the rule `environment={4,5}`, `fertility={5}` (see figure 6)

.

See the references for other interesting patterns.

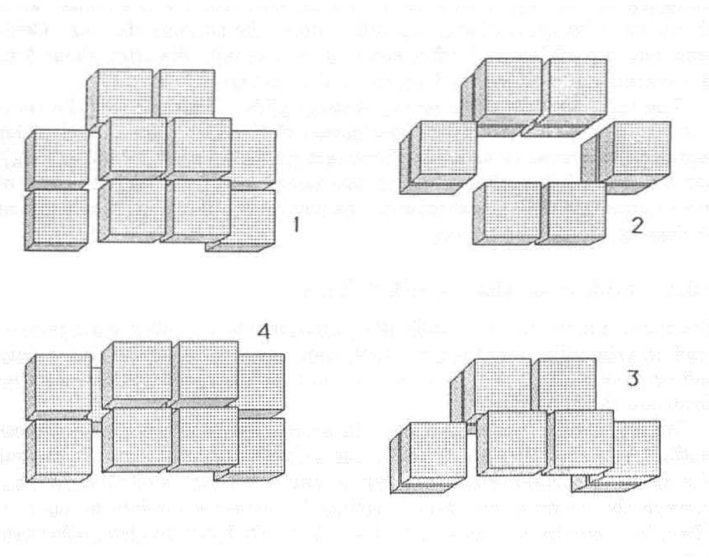



Figure 6: The glider translates itself up and forward over 4 steps. Note that there are 7 other possible orientations. Picture is taken from Bays (1987).

### 3.6 Submission Guidelines

You should have received a GitHub Classrooms invitation for this individual assessment. Please accept this invitation and upload your 'automata.py' file (and any further tests) to the repository this creates. The deadline for final submission is 12 noon on Friday, 18th October 2019, at which point your work will be collected. Your mark will be available before Friday 1st November.

## 4 Assessment Criteria

A majority of the marks and a good passing grade can be achieved by successfully completing the sandpile (40%) and 2d rectangular (30%) Game of Life exercises. The extensions are collectively worth a maximum of 30%.

- Your module will be tested by running it in a virtual environment and being asked to calculate and output i) the final sandpile conditions for a specified initial condition and ii) a fixed number of steps of Life starting from known initial conditions on specified grids. These results will then be compared to a reference solution.
- A code style (i.e. pep8/pylint) checker will be run against your submitted module, with marks deducted for any linting errors discovered.
- Your module functions will be timed repeatedly on a large grid (of order 1024×1024 cells), with marks awarded, to a maximum of 20% of the total mark, based on runtime compared with a reference implementation.

- Your code should include at least one `pytest` test for each of the automata you choose to implement (see Tuesday's lecture). Each test must pass successfully. You are free to include a reasonable number of additional tests, but no failing tests should be included.

# 5 Further Reading

- The original Abelian sandpile paper: Bak et al. *Self-Organized Criticality: An Explanation of 1/f Noise.* **Physical Review Letters** (1987) `https://journals.aps.org/prl/pdf/10.1103/PhysRevLett.59.381`
- Early article on Life: Gardner *The fantastic combinations of John Conway's new solitaire game "life"* **Scientific American** (1970) `http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm`
- A discussion on possible rules for 3D Life: Bays. *Candidates for the Game of Life in Three Dimensions.* **Complex Systems** (1987) `http://wpmedia.wolfram.com/uploads/sites/13/2018/02/01-3-1.pdf`
- A discussion on possible rules for Life on triangles: Bays. *Cellular Automata in the Triangular Tessellation.* **Complex Systems** (1994) `https://pdfs.semanticscholar.org/4b4f/fa2475e323b51ecdf6039aa71dcb616b3779.pdf`