Cellular Automata

October 13, 2020

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, Wolfram's Rule 30 and the Game of Life.

1 Wolfram's Rule 30

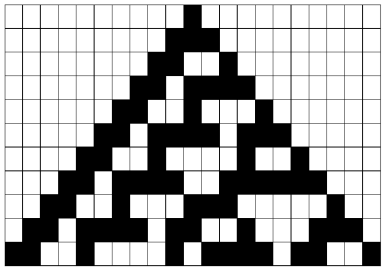


Figure 1: The onset of chaos in Rule 30

Wolfram's Rule 30 is possibly the most famous one-dimensional cellular automaton. At each time level the state of a cell depends on its value at the last level and that of its neighbours according to the following pattern

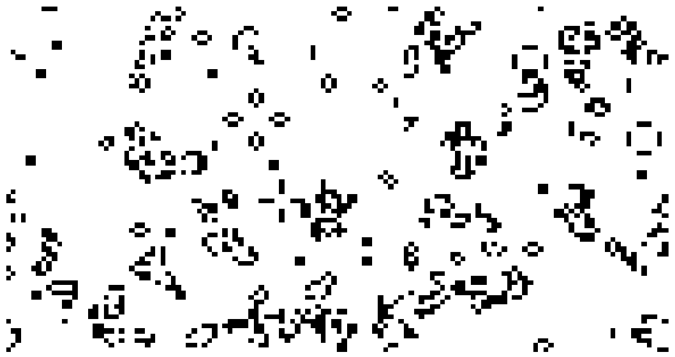
previous pattern	next centre cell state
on, on, on	off
on, on, off	off
on, off, on	off
on, off, off	on
off, on, on	on
off, on, off	on
off, off, on	on
off, off, off	off

The figure plots the progression of the state with one cell on initially with time increasing downward. Note that the table above has the left column ordered by the equivalent binary value, 111, 110, 101 etc. The right hand column then translates to 00011110, or in decimal 30, hence the choice of name.

Looking at the values of the centre cells, they are hard to predict, indeed Wolfram himself proposed they could be used in pseudorandom number generators.

Boundaries in a finite lattice can either be dealt with by assuming missing cells are treated as off, or by treating the lattice as periodic, with the left edge connecting to the right one.

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.

You may treat the boundary as if the grid were surrounded by an outer circle of "always dead" cells.

2.2 Extension 1: The periodic game of life

One method to extend Life is to apply periodic boundary conditions. Since we have two boundaries, the mesh is *doubly* periodic, as though the mesh were surrounded on all sides by exact copies (including in diagonal directions). Otherwise, all rules operate the same as in the non-periodic case.

2.3 Extension 2: Life on a hexagonal tessellation

Another method to extend Life is to use a mesh pattern which does not map nicely to multidimensional 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 hexagons, along with rules:

- a live cell survives with 3 or 5 neighbours (note it dies with 4),
- a dead cell turns on with 2 neighbours.

Obviously, each cell neighbours only the six other hexagons it touches.

2.4 Extension 3: 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 (also known as an adjancy matrix) to identify which cells neighbour each other (i.e. a symmetric matrix which has value 1 for element ij if cell number i neighbours cell number j or value 0 otherwise); 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 rule_thirty and `life` (plus `life_periodic`, `lifehex` and `life_generic` if you attempt the extension exercises) with the following signatures:

3.1 Rule 30

```
def rule_thirty(initial_state, nsteps, periodic=False):
    """
    Perform iterations of Wolfram's Rule 30 with specified boundary
    conditions.
```

```
    Parameters
    ----------
    initial_state : array_like or list
        Initial state of lattice in an array of booleans.
    nsteps : int
        Number of steps of Rule 30 to perform.
    periodic : bool, optional
        If true, then the lattice is assumed periodic.

    Returns
    -------

    numpy.ndarray
         Final state of lattice in array of booleans

    >>> rule_thirty([False, True, False], 1)
    array([True, True, True])

    >>> rule_thirty([False, False, True, False, False], 3)
    array([True, False, True, True, True])
    """
```

When imported and called like

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

the function should execute and return the output of `nsteps` steps of Rule 30. The input array should not be modified inside your function. You may import any module in the standard Python 3 libraries, as well as `scipy`, `numpy` and `matplotlib`. No other nonstandard modules should be imported.

3.2 life

```
def life(initial_state, nsteps):
    """
    Perform iterations of Conway's Game of Life.

    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.

    Returns
    -------

    numpy.ndarray
         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 and output the result of `nsteps` steps of Life. The input array should not be modified inside your function. You may import any module in the standard Python 3 libraries, as well as `scipy`, `numpy` and `matplotlib`. No other nonstandard modules should be imported.

3.3 life_periodic

```
def life_periodic(initial_state, nsteps):
    """
    Perform iterations of Conway's Game of Life on a doubly periodic mesh.

    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.

    Returns
    -------

    numpy.ndarray
         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_periodic(X, 10)
```

the function should execute and output the result of `nsteps` steps of Life on a periodic mesh. The input array should not be modified inside your function. You may import any module in the standard Python 3 libraries, as well as `scipy`, `numpy` and `matplotlib`. No other nonstandard modules should be imported.

3.4 lifehex

The `lifehex` function should have the following signature:

```
def lifehex(initial_state, nsteps):
    """
    Perform iterations of Conway's Game of Life on
    a hexagonal tessellation.

    Parameters
    ----------
    initial_state : list of lists
        Initial state of grid on hexagons.
    nsteps : int
        Number of steps of Life to perform.

    Returns
    -------

    list of lists
         Final state of tessellation.
    """
```

When imported and called like

```
import numpy
import automata
# Generate random initial state.
X = [ [numpy.random.random()>0.3 for j in range(6+i%2)] for i in range(6) ]
# call module.
Z = automata.lifehex(X, 10)
```

the function should execute `nsteps` steps of Life on a hexagonal mesh. The input data should not be modified inside your function. You may import any module in the standard Python 3 libraries, as well as `scipy`, `numpy` and `matplotlib`. No other nonstandard modules should be imported.

3.4.1 The hexagonal 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 hexagons and that every second row (i.e. the 2nd, 4th 6th etc) contains one more hexagon than the first row. Obviously, grids of this form cannot be periodic.

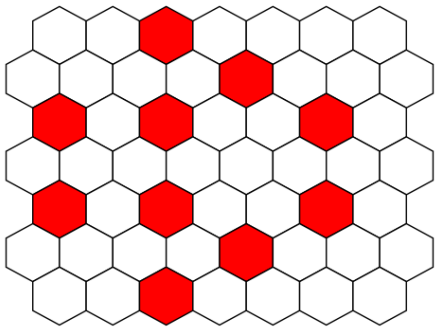


Figure 2: The hexagonal period 7 glider.

The figure above corresponds to the following code (when 0 is converted to False and 1 to True):

```
[[0,0,1,0,0,0,0],
 [0,0,0,0,1,0,0,0],
 [1,0,1,0,0,1,0],
 [0,0,0,0,0,0,0,0],
 [1,0,1,0,0,1,0],
 [0,0,0,0,1,0,0,0],
 [0,0,1,0,0,0,0]]
```

3.5 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 ints (0 or 1)
         a symmetric boolean matrix with rows indicating neighbours for each cell
    initial_state : 1d array_like or list of bools
         Initial state vector.
    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
    -------

    numpy.array
         Final state.
    """
```

When imported and called like

```
import numpy
import automata
M = numpy.zeros((8*8, 8*8), int)
for i in range(1,7):
    for j in range(1,7):
        M[8*i+j, 8*i+j+1] = 1
        M[8*i+j, 8*i+j-1] = 1
        M[8*i+j, 8*(i+1)+j] = 1
        M[8*i+j, 8*(i-1)+j] = 1
# 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 `scipy`, `numpy` and `matplotlib`. No other nonstandard modules should be imported.

3.6 Checking your code

3.6.1 Rule 30

Some simple patterns are easy to run through the algorithm by hand. Examples can be taken from the figure at the beginning of this document.

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

3.6.2 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".

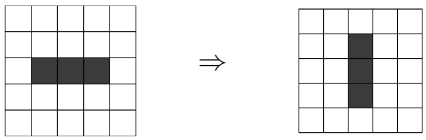


Figure 3: The blinker remains centred and rotates with period 2.

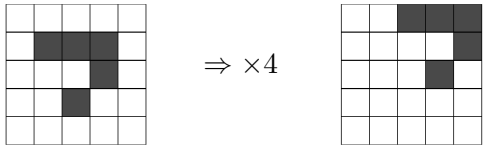


Figure 4: The glider translates itself diagonally over 4 steps.

The periodic version can be well tested using gliders, or by blinkers placed at boundaries. There is also a hexagonal glider (see figure 2)

The generic system can (sometimes somewhat slowly) mimic any of the Life 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 5)

.

See the references for other interesting patterns.

3.7 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 via upload to GitHub is 4pm BST on Friday, 16th October 2020. Your mark should be available by Friday 30th November.

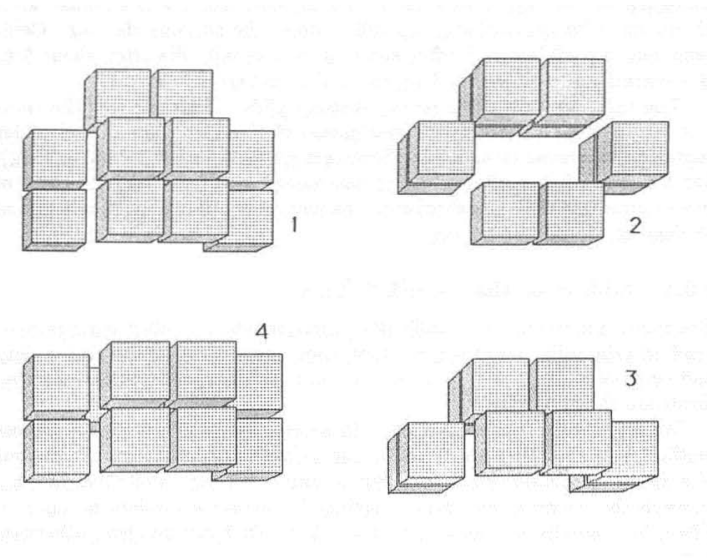


Figure 5: The glider translates itself up and forward over 4 steps. Note that there are 7 other possible orientations. Picture is taken from Bays (1987).

4 Assessment Criteria

A majority of the marks and a good passing grade can be achieved by successfully completing the Rule 30 (35%) and 2d rectangular (30%) Game of Life exercises to fully meet the specification. The extensions are collectively worth a maximum of 25%. The final 10% will depend on the performance of your code (see below).

- Your module will be tested by running it in a Python virtual environment and asking it to calculate and output the result of a fixed number of steps of Rule 30 and of Life starting from known initial conditions on specified grids. These results will then be compared to a reference solution.
- A code style checker (i.e. pycodestyle and a full run of pylint) checker will be run against your submitted module with marks deducted for any linting errors discovered.
- Your code should include at least one `pytest` test for each of the automata you choose to implement (see Thursday'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.
- Your module functions will be timed repeatedly on large grids (of order 1024×1024 cells for the Game of Life), with bonus marks awarded, to a maximum of 10% of the total mark, based on runtime compared with a reference implementation.

5 Further Reading

- Cellular Automata: Is Rule 30 Random: Gage, Laub and McGarry **Cellulara** (2005) https://legacy.cs.indiana.edu/~dgerman/2005midwestNKSconference/dgelbm.pdf
- 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 hexagons: Bays. *A Note on the Game of Life in Hexagonal and Pentagonal Tessellations.* **Complex Systems** (2005) http://wpmedia.wolfram.com/uploads/sites/13/2018/02/15-3-4.pdf