Cellular Automata

October 12, 2021

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 automata, a continuous model derived from the Lorenz 96 toy atmosphere model and Conway's Game of Life.

1 Lorenz 96

We will start with an automaton based toy model of weather patterns designed by Ed Lorenz around (who also developed a more famous model which was influential in early study into Chaos theory) around 1996, hence the choice of name.

This model assumes we have a number of cells, N, each taking a given real number value, x_i, $i \in 1, 2, \ldots, N$. At each update these values update according to the following rule:

$$x_i^{\text{new}} = \frac{1}{101}\left(100x_i^{\text{old}} + \left(x_{i-2}^{\text{old}} - x_{i+1}^{\text{old}}\right) x_{i-1}^{\text{old}} + 8\right).$$

The model uses *periodic* boundary conditions, where $x_0 = x_N$ and $x_{N+1} = x_1$. This makes it convenient to plot results graphically on a circle, where x_1 and x_N end up next to each other.

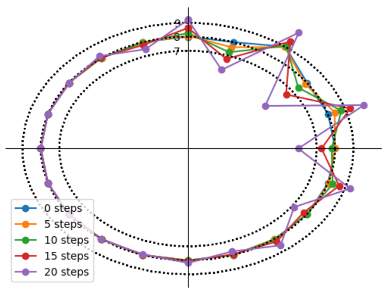
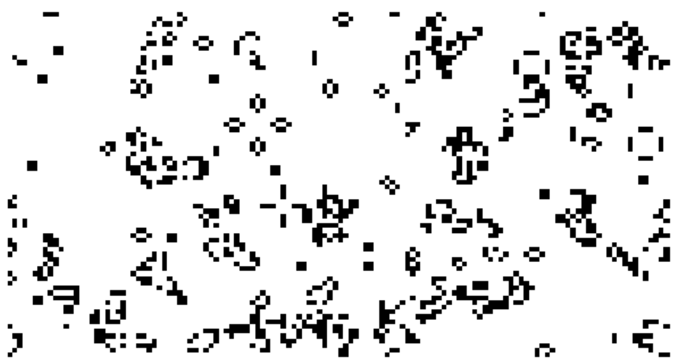


Figure 1: Several updates of the Lorenz 96 automaton

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 pentagonal 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 pentagons, along with rules:

- a live cell survives with 3 or 3 neighbours
- a dead cell turns on with 3, 4 or 6 neighbours (it stays dead with 5).

This pattern is run on the "Cairo" pentagonal tesselation, and each cell is assume to have 7 neighbours, the 5 it shares an edge with, and two more it meets at a vertex. See the image below.

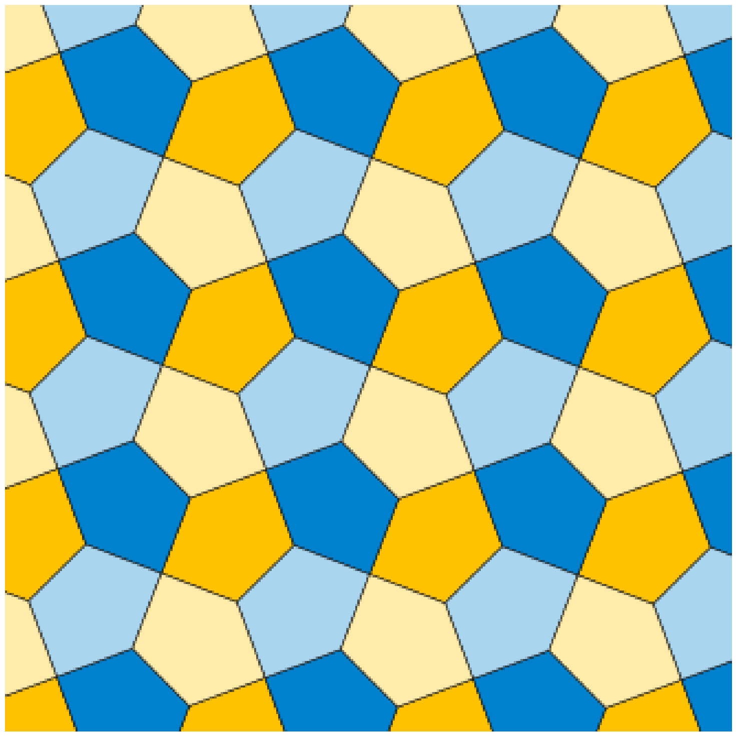


Figure 2: The Cairo tiling (image by David Eppstein, sourced from Wikipedia)

2.4 Extension 3: Two Colour Life

A final extension uses a regular rectangular grid, but has two separate "living" states (e.g. blue and red when labelling with colour). For this sytem, the rules are now:

- A living cell with 0 or 1 neighbours (of any colour) dies of loneliness.
- A living cell with 2 or 3 neighbours (of any colour) survives to the next generation, and stays the same colour.

- A living cell with 4 or more neighbours (of any colour) dies from overcrowding.
- A dead cell with 3 neighbours becomes live, with the colour of the majority of its neighbours.
- A dead cell with 0–2 or 4–8 neighbours (of any colour) stays dead.

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`, `lifepent` and `life2colour` if you attempt the extension exercises) with the following signatures:

3.1 Lorenz 96

```
def lorenz96(initial_state, nsteps):
    """
    Perform iterations of the Lorenz 96 update.

    Parameters
    ----------
    initial_state : array_like or list
        Initial state of lattice in an array of floats.
    nsteps : int
        Number of steps of Lorenz 96 to perform.

    Returns
    -------

    numpy.ndarray
         Final state of lattice in array of floats

    >>> x = lorenz96([8.0, 8.0, 8.0], 1)
    >>> print(x)
    array([8.0, 8.0, 8.0])

    >>> lorenz96([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 function.
Z = automata.rule_thirty(X, 10)
print(Z)
```

the function should execute and *return* the output of `nsteps` steps of the Lorenz96 update. 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 function.
Z = automata.life(X, 10)
print(Z)
```

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 function.
Z = automata.life_periodic(X, 10)
print(Z)
```

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.

```
    def life2colour(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 ints with value 1, 0, or 1.
            Values of -1 or 1 represent "on" cells of both colours. Zero
            values are "off".
        nsteps : int
```

```
            Number of steps of Life to perform.

        Returns
        -------

        numpy.ndarray
             Final state of grid in array of ints of value -1, 0, or 1.
        """
```

When imported and called like

```
import numpy
import automata
# Generate random initial state.
X = numpy.random.randint(-1, 2, (16, 16))
# call module function.
Z = automata.life2colour(X, 10)
print(Z)
```

the function should execute and output the result of **`nsteps`** steps of 2 colour Life on a non-periodic mesh using the specified encoding (1 and -1 on, 0 off). 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 lifepent

The `lifepent` function should have the following signature:

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

    Parameters
    ----------
    initial_state : array_like or list of lists
        Initial state of grid of pentagons.
    nsteps : int
        Number of steps of Life to perform.

    Returns
    -------
    numpy.ndarray
        Final state of tessellation.
    """
```

When imported and called like

```
import numpy
import automata
# Generate random initial state.
X = np.array([ [numpy.random.random()>0.3 for j in range(6)] for i in range(6) ])
# call module function.
Z = automata.lifepent(X, 10)
print(Z)
```

the function should execute `nsteps` steps of Life on a pentagonal 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 pentagonal 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 pentagons. You *do not* need to consider the periodic case. As an example, see the figure below

Figure 3: The pentagonal period 48 glider moves up 6 cells each cycle.

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

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

3.5 Checking your code

3.5.1 Lorenz 96

The simplest test case is the "steady state" with $\texttt{x}[:] = \texttt{8}$. Note that since we are using floating point calculations, a very small variation in the answer is to be expected. A second test case is to take $\texttt{x}[:] = \texttt{7}$ and to check that the forcing causes it to increase appropriately.

Another test case is to take $\texttt{x}[\texttt{i}] = \texttt{9}$ for one cell, with $\texttt{x}[\texttt{j}] = \texttt{8}$ for $\texttt{i} \neq \texttt{j}$. Then the new values should have $\texttt{x}[\texttt{i} - \texttt{1}] = \frac{\texttt{800}}{\texttt{101}}$, $\texttt{x}[\texttt{i}] = \frac{\texttt{908}}{\texttt{101}}$, $\mathbf{x[i+1]} = \mathbf{\frac{817}{101}}$ with $\texttt{x}[\texttt{j}] = \texttt{8}$ otherwise. This is particularly useful to check the boundary conditions.

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".

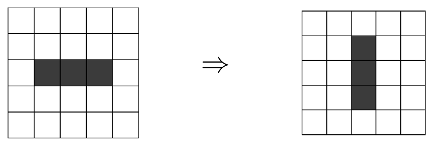


Figure 4: The blinker remains centred and rotates with period 2.

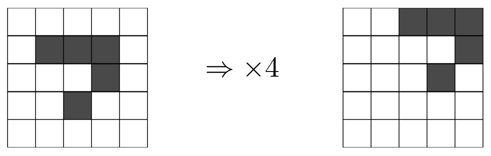


Figure 5: The glider translates itself diagonally over 4 steps.

3.5.3 The extensions

The periodic version can be well tested using gliders, or by blinkers placed at boundaries. Any object in the regular Game of Life is also a solution in the two colour game, while two colour blinkers and gliders can be constructed fairly easily. There is also a pentagonal glider (see figure 3). See the references below for other interesting patterns.

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 via upload to GitHub is 5pm BST on Friday, 15th October 2021. Your mark should be available by Friday 29th October.

4 Assessment Criteria

A majority of the marks and a good passing grade can be achieved by successfully completing the Lorenz (35%) and 2d rectangular (30%) Game of Life exercises to fully meet the specification. The 3 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 Lorenz 96 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

- A discussion on the history & mathematics of the Lorenz 96 system: van Kekem *Dynamics of the Lorenz 96 model*`https://pure.rug.nl/ws/portalfiles/portal/65106850/1_Introduction.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`