Cellular Automata

11th of October, 2023

1 Introduction

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. In this assessment, with the 'aid' of ChatGPT 3.5, you will implement, test and profile two automata: (i) Lorenz 96, a toy atmosphere model derived by Lorenz and (ii) Conway's Game of Life.

Over the past year, LLM based chat bot's such as ChatGPT have received a great deal of media coverage. Utilizing LLM based tools will likely become an increasingly important part of many tasks we carry out - writing code being no exception. Harnessing the power of these tools will however require us to use them *responsibly*; that is, we need to ensure we understand their output and know how to test it for correctness. Further, whilst this situation will continually change as new LLMs trained on ever larger data sets are released, the solutions they provide will often be sub-optimal and require some improvements.

Along with testing the contents of this weeks lectures, this assessment is also designed for you to learn how to use ChatGPT (or any similar tool) responsibly. Below, we will first introduce the automata you will be required to build, and then specify the criteria for this assignment.

1.1 Lorenz 96

We first introduce an automaton based toy model of weather patterns designed by Ed Lorenz (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 the T-th iteration. At the $(T+1)$-th iteration, the values are updated according to the following rule:

$$x_i^{(T+1)} = \alpha \left(\beta x_i^{(T)} + \left(x_{i-2}^{(T)} - x_{i+1}^{(T)} \right) x_{i-1}^{(T)} + \gamma \right),$$

where α, β and γ are real valued parameters. 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. See Figure 1.

The behaviour of this automata will vary depending what values are chosen for α, β and γ. If we choose

$$\begin{aligned} \alpha &= \frac{1}{101}, \\ \beta &= 100, \\ \gamma &= 8, \end{aligned}$$

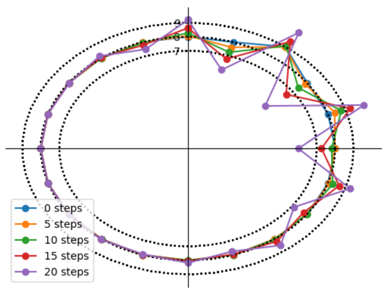


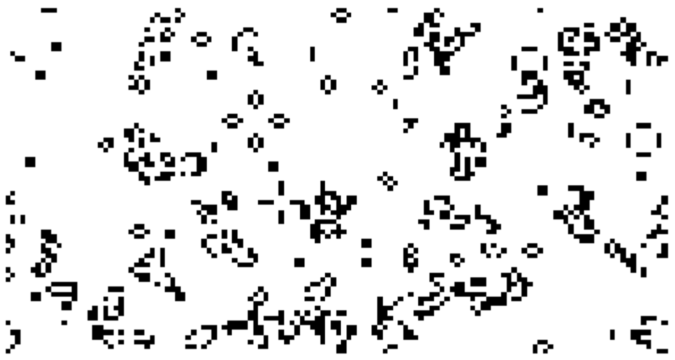
Figure 1: Several updates of the Lorenz 96 automaton

then we can define some simple test cases. The simplest test case is the "steady state" with x[:] = 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 x[:] = 7 and to check that the forcing causes it to increase appropriately.

Another test case is to take x[i] = 9 for one cell, with x[j] = 8 for i $\neq$ j. Then the new values should have x[i − 1] = $\frac{800}{101}$, x[i] = $\frac{908}{101}$, x[i + 2] = $\frac{816}{101}$ with x[j] = 8 otherwise. This is particularly useful to check the boundary conditions.

For different values of α, β and γ, similar test cases can be established.

1.2 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).

1.2.1 Rules of the basic 2D game

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 "always dead" cells.

1.2.2 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.

1.2.3 Extension 2: Two Colour Life

Another extension uses a regular rectangular grid, but has two separate "living" states (e.g. blue and red when labelling with colour). For this system, 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.

1.2.4 Extension 3: 3D Life

The final extension we will consider is the addition of the 3rd dimension. Since any cube in a 3D mesh has many more neighbours than a square in a 2D mesh (26 versus 8) we cannot keep the same survival rules. There are a variety of rules which give interesting behaviour, but here we will specify that

- A live cell survives with 5 or 6 neighbours.
- A dead cell turns on with 4 neighbours.
- All other cases result in the cell dying/remaining dead.

Again, you may treat the boundary as if the grid were surrounded by "always dead" cells. As an additional challenge, you may wish to consider how to implement periodic boundary conditions in 3D.

2 Assessment specification and criteria

For this assessment you should complete the following:

1. (a) Enlist the aid of ChatGPT 3.5 to write code for the Lorenz 96 system. You may need to iterate with several prompts to get to a suitable solution. You should add URLs to the final states of any conversations to the `References.md` located in this directory. To do this, use the "share chat" button to show us your prompts and ChatGPT's responses.
   (b) Add a suitable set of additional tests to the `TestLorenz96` class in `test_automata.py` (located in the `tests` folder). Tests should cover the values of α, β and γ introduced above and **one** additional set of your choice. Further, you should add unit tests for any helper functions you write.
   (c) Next, profile your code using `timeit`. Make any improvements you can to the ChatGPT solution. Comment on how well ChatGPT did, what improvements you made and why. Add these profiling results and the discussion under the **Lorenz 96** section of the `discussion.ipynb` notebook. Note that you only need to include your final profiling results in your submission, although in your discussion you should comment on how modifications improved the efficiency of your code.
2. Repeat the steps of part `1` for the Game of Life. Note however that here, you do not need to enlist the aid of ChatGPT if you do not wish to do so. If you do, include the logs as with part `1`. If you do not, for the final discussion, comment on how you profiled and improved on your code (without the aid of ChatGPT).

Note that the specifications for each function are given below. You should follow these specifications exactly. For this assessment you do not need (and probably should not) add any new files to your repository. Simply modify the various required files included with the repository.

The Lorenz 96 section will be allocated 30% of your grade, and the Game of Life and its various extensions 50%. Marks will be awarded according to the quality of your code, the associated tests and the discussion you present. The final 20% will depend on how well your code performs on a series of tests we will run it on. Note that:

- 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.
- `flake8` will be run against your submitted module with marks deducted for any linting errors discovered.
- Your module functions will be timed repeatedly on large grids (of order 1024×1024 cells for the Game of Life 2D). Tests will time out and fail after a short period of time, hence ensure your code is efficient to pass any many tests as possible!

2.1 Function specification and some notes on testing

You should modify the python module file called `automata.py`, which when imported exposes functions called `lorenz96` and `life` with the following signatures:

2.1.1 Lorenz 96

```
def lorenz96(initial_state, nsteps, constants=(1/101, 100, 8)):
    """
    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.
    constants : tuple of floats
        Values of the contents alpha, beta and gamma.

    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])

    """
```

It should be possible to import and run the function in the following manner:

```
import numpy
import automata
# Generate random initial state.
X = numpy.random.random(16)
# call module.
Z = automata.lorenz96(X, 10)
```

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.

2.1.2 life

```
def life(initial_state, nsteps, rules="basic", periodic=False):
    """
    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 ints.
    nsteps : int
        Number of steps of Life to perform.
    rules : str
        Choose a set of rules from "basic", "2colour" or "3d".
    periodic : bool
        If True, use periodic boundary conditions.

    Returns
    -------

    numpy.ndarray
         Final state of grid in array of ints.
    """
```

It should be possible to import and run the function in the following manner:

```
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, using the `rules` set - by default basic, non-periodic. 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.

2.1.3 life (periodic)

It should be possible to 'play' with these rules in the following manner:

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

the life function should execute and output the result of `nsteps` steps of Life with the basic rules 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.

2.1.4 life (two colour)

It should be possible to 'play' with these rules in the following manner:

```
    import numpy
    import automata
    # Generate random initial state.
    X = numpy.random.randint(-1, 1, (16, 16))
    # call module.
    Z = automata.life(X, 10, rules="2colour")
```

the life 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. To run with periodic boundary conditions, we should simply add the `periodic=True` keyword argument to the function call:

```
Z = automata.life(X, 10, rules="2colour", periodic=True)
```

2.1.5 life (3D)

It should be possible to ‘play’ with these rules in the following manner:

```
import numpy
import automata
# Generate random initial state.
X = numpy.random.random((16, 16, 16))>0.3
# call module.
Z = automata.life(X, 10, rules="3d")
```

the function should execute `nt` steps of 3D Life on a non-periodic mesh. 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.

To run with periodic boundary conditions, we should simply add the `periodic=True` keyword argument to the function call:

```
Z = automata.life(X, 10, rules="3d", periodic=True)
```

2.1.6 Notes on testing

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”.

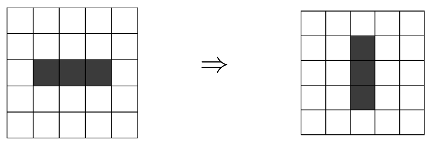


Figure 2: The blinker remains centred and rotates with period 2. Note that only a subsection of a ‘larger’ grid is shown in the above illustration.

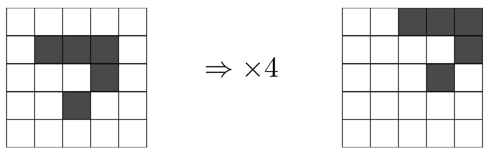


Figure 3: The glider translates itself diagonally over 4 steps. Note that only a subsection of a ‘larger’ grid is shown in the above illustration.

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. See the references below for other interesting patterns from which test cases can be constructed.

The 3D version is a little trickier, and we’ll leave it to you to devise a test case!

3 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`