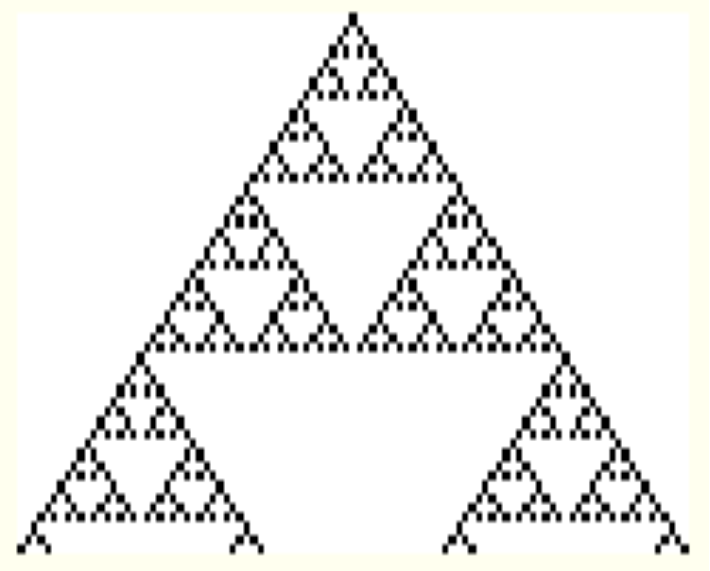
PHAS2443 PRACTICAL MATHEMATICS II

CELLULAR AUTOMATA: SEA SHELLS, LIFE, AND FOREST FIRES

Introduction

The cellular automaton is a very simple idea. Space and time are divided into discrete units (cells) and the behaviour of each cell is governed by very simple rules, which tell the cell what to do at the next time step on the basis of what it and its neighbours are doing at the present time. Each cell has a 'state', which may be represented by a number. In the simplest automata, the state is just 0 or 1, on or off, alive or dead, but is it obviously possible to have cells with more than two states. The figure below shows the evolution of a simple system, with time running down the page, which stared from two 'live' cells. The rule is giving rise to a tree-like structure (in fact, a fractal structure). Cellular automata have been explored as the origin of the patterns on sea-shells, as models of forest fires spreading and crystals growing, and for calculating the flow of liquids. If we are to believe Stephen Wolfram (the architect of Mathematica), cellular automata are fundamental to the structure of nature (see, for example, an interview with him in New Scientist, August 2001). John Holland (also known for his work on genetic algorithms) shares this view (see J. Holland, *Emergence: from chaos to order*, Helix Books 1998).



Project

The aim of this project is to write Python programs to follow the behaviour of simple cellular automata. Start with a simple system, with one spatial dimension and one time dimension. Before discussing that, we should establish the nomenclature: by 'rule' we mean the whole description of how a cell will be updated based on the configuration of its neighbours: each such rule could be implemented as a number of Mathematica Rules or functions. For example, a two-state automaton which depends only on the state of a cell, not its neighbours, has 2 possible states, so the CA rules might be:

1: 0 stays as 0, 1 stays as 1;
2: 0 stays as 0, 1 changes to 0;

3: 0 changes to 1, 1 stays as 1;
4: 0 changes to 1, 1 changes to 0,
And each of those CA rules might be implemented as one or more Python functions.

Basically, the number of components of each CA rule is the number of states, and the number of CA rules is 2 to the power of the number of states. If we enumerate all the possible configurations of three cells we get

(0,0,0) total 0
(1,0,0) total 1
(0,1,0) total 1
(0,0,1) total 1
(0,1,1) total 2
(1,0,1) total 2
(1,1,0) total 2
(1,1,1) total 3

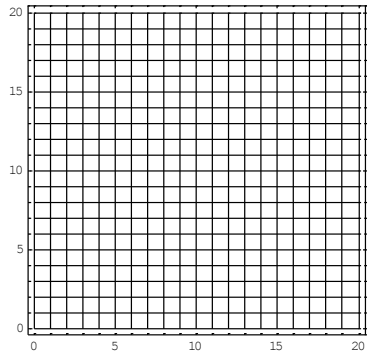
So if we use the cell values separately we have 8 different patterns, each of which we can use to set the new value of the central cell to 0 or 1, giving 2^8=256 rules.

On the other hand, if we just look at the totals, we have 4 possible totals. If we set the next value according to the total only, we get 2^4=16 rules. If, on the other hand, we use the total and take into account the current value of the central cell, to get ($2*2^4$) possible initial states, we get 2^5=32 rules.

The first step is to write an update function, which will take the list of cell values at time step n and return the values at time step n+1. Start with a two-state automaton, and base the value of cell p on the sum of the values of cells p-1, p and p+1 at the previous step. You may find it convenient to define a function to implement the rule and then loop over entries, using np.roll. Explore the 16 possibilities.

Now extend the system in two ways: either allow it to look more than one step back in time, or allow it to involve more distant neighbours. Try to find examples that show very regular behaviour and examples that look almost random.

One of the most famous examples of a two dimensional cellular automaton was the so-called Game of Life, invented by the mathematician John Horton Conway (who seems to figure in almost every modern book on recreational mathematics). This involves a square array of cells, so that each cell has eight neighbours (two vertically above and below, one to left and one to right, and four diagonal neighbours). The update rule is that a dead cell which has exactly three neighbours will come to life at the next step, a live cell with two or three neighbours will stay alive at the next step, but a live cell with more or fewer neighbours will die (of overcrowding or loneliness respectively). Set up a Python code to show the pattern of cells as a function of time. Start with a random collection of live cells: you should find that it evolves into a relatively sparse pattern. Compare configurations at successive time steps when the density of live cells is low: can you identify any small stable groups of cells. You might like to check your program by starting with a block of cells that is empty except for a block near the centre like this: you should find that it ‘glides’ towards the bottom left (it alters its shape on the way).



That's enough fun and games. Now set up a simulation of a forest fire. Now we have a three-state two-dimensional automaton: the states, of course, are green forest (1), burning forest (2), and burnt-out forest (0). Assume that the state of a cell only affects the cells above, below, to right and left. Burnt-out forest sprouts green forest with a probability p per cell per time step; if a cell is alight, then on the next step any neighbouring cell that was previously green forest will catch, but the cell that was alight will have burnt out. In addition, lightning may strike, allowing any green forest cell to catch fire with probability f per cell per timestep. Take initial values p=0.05 and f=0.00025, and explore how fires propagate through a 100 by 100 cell forest for 100 time steps, starting with a forest containing no fires but with 30% of the cells occupied by green forest, the rest empty (equivalent to being burnt out). Explore what happens as p and f are varied.

A.H. Harker
UCL
September 2005, February 2010, January 2013
David Bowler, 2019