Automata user guide

Alex Richardson

April 1, 2021

Contents

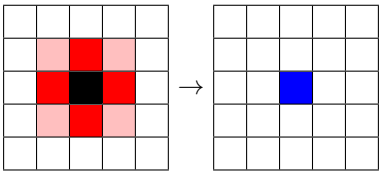
1 Introduction

This document contains a user guide to software submitted for the MPhys project and presentation public summary, as well as a brief summary of the main points of the project.

1.1 What are cellular automata and how many are there?

Cellular automata are a general class of discrete dynamical system, defined as a lattice of cells equipped with an update rule. Table 1 shows a graphical representation of the update rule. The state of the blue cell on the right is defined purely by the states of the same cell (black) and its neighbours (red/pink) at the previous timestep.

Table 1: Local update rule example



By applying this update rule in parallel on all cells, the system is evolved forwards in time. This is referred to as running a cellular automata. Depending on the update rule chosen, some very complex emergent behaviour can occur. To enforce spacial symmetries (rules must behave the same under rotation and reflections), only the number of each cell state in the neighbourhood affects the update, not where within the neighbourhood they are located. With this constraint, the number of unique 8 cell neighbourhoods $\mathbf{N}$ is given by the multiset coefficient:

$$\mathbf{N} = \binom{S+8-1}{8} = \frac{(S+8-1)!}{8!(S-1)!} \tag{1}$$

Where S is the number of states. As each update rule maps every unique neighbourhood and central cell state combination to a new cell state, the number of possible update rules is given by:

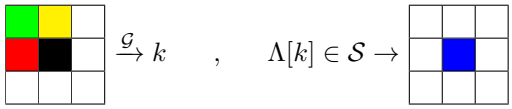
$$\|\mathcal{R}_s\| = S^{S\times\mathbf{N}} \tag{2}$$

Where $\mathcal{R}_s$ denotes the set of all possible s state 2D cellular automata rules. Plugging in some numbers as an example shows just how many rules there are:

s	2 state	4 state	8 state
$\|\mathcal{R}_s\|$	2^{18}	4^{660}	8^{51480}

The way these update rules are actually encoded in software is with a function $\mathcal{G}$ and a rule array Λ. The function $\mathcal{G}$ maps every unique neighbourhood and central cell combination to a unique integer, which in turn indexes the array Λ. By keeping $\mathcal{G}$ constant and varying Λ, any possible cellular automata rule can defined. A schematic is shown in Table 2, where $\mathcal{S}$ denotes the set of cell states:

Table 2: Schematic of how update rule f is composed of $\mathcal{G}$ and rule array Λ



Encoding cellular automata rules as an array of cell states has some major advantages. It simplifies the generation of random rules to simply generating a random array of states. The array can also be thought of as analogous to a genome, and by 'mutating' or 'breeding' rules, more interesting behaviours can be explored. The majority of randomly generated cellular automata are boring, but it is fairly easy to explore and 'evolve' more interesting ones.

2 Running the code

2.1 Setup

To setup the code, make sure you have downloaded the compressed folder for your OS. Once you have extracted this folder, simply run the executable titled `automata`. The directory `2D_rules` is where `automata` saves and loads rule arrays, the directory `text_resources` contains some necessary `.txt` files. Upon startup, the program will prompt you for a number of states, a grid size and a number of iterations. These are all integers. I typically find 128 works nicely for grid size and number of iterations - a big and long enough simulation to see some structure, but small enough to run very quickly. The number of states is some integer ≥ 2. **Be aware** that for numbers of states beyond 8, the RAM usage starts to explode - with 12 states using more than 32 GB of RAM.

2.2 Basics

2.2.1 Hitting enter (default)

If you type nothing into the prompt, the program just generates, runs and displays a new random cellular automata.

2.2.2 r

Entering `r` reruns the current cellular automata on a new initial condition.

2.2.3 c

Entering `c` continues running the current cellular automata from the previous simulation end state.

2.2.4 s

Entering `s` saves the current rule array to the `2D_rules` directory. It prompts the user for a name.

2.2.5 l

Entering `l` shows the list of currently saved rules and prompts the user to enter the name of which one to load, run and display.

2.2.6 g

Entering `g` changes the algorithm by which the rule array is generated. The two options are sampling states binomially, or sparsely sampling from a uniform distribution (meaning that most cell states are 0, but those that are not are uniformly random).

2.2.7 y

Entering `y` prompts the user for a `mu` and `sigma` parameter that defines how rules are randomly generated.

2.3 Rule combination

The following methods combine two rules together. When selected, they will show the list of saved rules and prompt the user to select which to combine. If the user enters nothing, then the currently loaded rule is used in place of loading a saved one.

2.3.1 +,-,*

Entering +,- or * performs element-wise arithmetic between two rules, modulo the number of states.

2.3.2 a

Entering a performs the element-wise mean between two rules.

2.3.3 z

Entering z performs a zip, where the new rule array is made of alternating odd and even indexed elements of the two chosen rules.

2.3.4 b

Entering b splices two rules together, where the new rule array has the first half of rule 1 and the second half of rule 2.

2.4 Rule manipulation

The following methods just change the currently loaded rule.

2.4.1 p

Entering p perturbs the current rule array, randomly changing 5% of the entries.

2.4.2 m

Entering m creates 4 mutations of the current rule, based on the mutation amount prompt to the user (I find about 0.1 to work best). Once calculated, they are displayed in parallel in the 4 quadrants of the simulation grid. The ordering is top left, top right, bottom left, bottom right. After closing the visualisation pop up the user is prompted to choose the best mutation, at which point that rule is rerun in full.

2.4.3 x

Entering x splits the rule array into as many equally spaced chunks as there are states, shuffles these chunks, and sticks them back together.

2.4.4 v

Entering v rolls the rule array, shifting all the entries along by an integer amount prompted from the user. Entries that roll over the edge come back at the other end of the array.

2.4.5 w

Entering w performs something analogous to smoothing the rule array, by which randomly selected sites in the array have their state copied onto a block of their neighbours. This prompts the user for an integer, higher number means more smoothing.

2.4.6 i

Entering i inverts the rule array, performing $s - \Lambda$ along the array.

2.5 Visualisation

The following commands change how the cellular automata simulations are visualised.

2.5.1 sm

Entering `sm` performs a gaussian blur on the grid.

2.5.2 t

Entering `t` transposes the simulation output, so that rather than viewing 2 spacial dimensions changing with time, you can view 1 spacial and 1 time dimension changing with the other spacial dimension. A bit weird but it makes more sense if you just try it.

2.5.3 fft

Entering `fft` performs a 3D fast fourier transform on the simulation data, and shows the spacial 2D slices animated along the temporal frequency axis. Looks best on rules with some periodic structures.

2.5.4 2d

Entering `2d` performs a sum along a given axis of the 3D simulation data, and displays the now stationary output. Useful if you want to easily save a visual result.

2.5.5 stat

Entering `stat` displays slices of the simulation at 1,10 and 100 timesteps. Will probably crash if the number of iterations is less than 100.