

Intro to cellular automata

A one dimensional cellular automaton consists of a linear array of cells, each of which may be in any of finitely many states, together with an update rule specifying how the states evolve over time. The array can be finite or infinite, although, we will typically take it to be infinite for application to fractal geometry. Thus, we can assume that the cells are in one-one correspondence with \mathbb{Z} and index the cells with the integers. We can then let x_i^t denote the state of the cell at position i and time t . An arbitrary configuration might look like

$$\langle \dots, x_{-2}, x_{-1}, x_0, x_1, x_2, \dots \rangle.$$

Typically, we'll assume that the possible states are indexed by the integers mod n . Thus, a two state cellular automata would have states 0 and 1. An example of a configuration for a two state cellular automata, that we will frequently use as an initial state, is

$$\langle \dots, 0, 0, 0, 1, 0, 0, 0 \dots \rangle.$$

The r neighborhood of the cell at position i consists of all cells at position j such that $|i - j| \leq r$. We'll assume that the cells can interact with one another up to a finite distance. Thus, the update rule for cell i will depend on the r neighborhood of cell i for the appropriate choice of r .

Example 1: Suppose our update rule is

$$x_i^t = x_{i-1}^{t-1} + x_{i+1}^{t-1} \pmod{2}$$

and our initial configuration consists of a single 1 embedded in an infinite row of zeros. Then the first ten steps in the evolution looks like so:

```
Grid[CellularAutomaton[Mod[#[[1]] + #[[3]], 2] &, {}, 1], {{1}, 0}, 9]] /.  
0 -> Style[0, Lighter[Gray]]  
  
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0  
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0  
0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0  
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0  
0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0  
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0  
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
```

This is easy enough to verify by hand!

Exercise 1: Suppose our update rule is

$$x_i^t = x_{i-1}^{t-1} + x_i^{t-1} + x_{i+1}^{t-1} \pmod{2}$$

and our initial configuration again consists of a single 1 embedded in an infinite row of zeros. Determine the first few steps in the evolution of the CA.

The elementary cellular automata

The two state cellular automata that operate with neighborhoods of size $r = 1$ are called the *elementary cellular automata*. There are 256 such cellular automata and they may be numbered $0, \dots, 255$ in a way that sheds light on the subject. Since $r = 1$, the state of a cell at time t will depend on its own state, together with states of its left and right neighbors at time $t - 1$. Thus, each possible neighborhood state can be written as an ordered triple of zeros and ones. The set of all possible neighborhood states is

000 001 010 011 100 101 110 111

Note that each of these may be thought of as the binary representation of an integer; in fact, these are exactly the integers zero through seven in increasing order. In order to specify an elementary cellular automaton, we need only specify the rule value for each of these eight neighborhood configurations. Our first example cellular automaton may be specified as

000 001 010 011 100 101 110 111
0 1 0 1 1 0 1 0

Alternatively, since we know the top row, we could simply write 01011010. Viewing this as a binary expansion, we can simply specify any elementary cellular automaton by its rule number; this is rule 90, for example. Furthermore, there are only two possible rule values for each neighborhood configuration, either a zero or a one. Thus, there are $2^8 = 256$ total possible rules numbered 0 through 255.

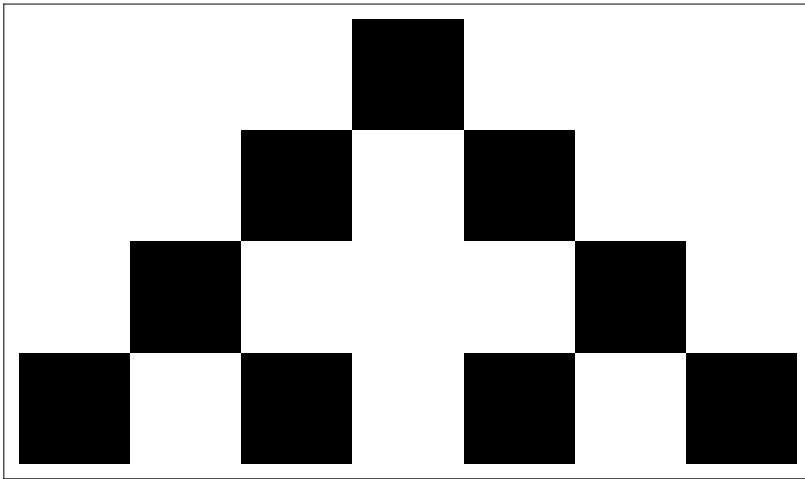
Exercise 2: Find the rule number for the elementary CA from exercise 1.

Example 2: *Mathematica* has a built in command `CellularAutomaton` that is designed to study cellular automata. Generally, you call `CellularAutomaton` in the form

`CellularAutomaton[rule, init, depth]`

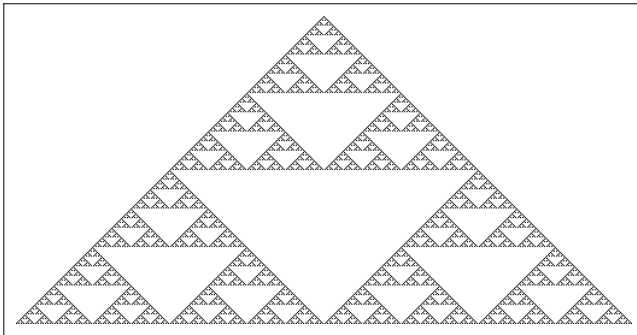
`rule` is the rule specification, `init` specifies the initial configuration, and `depth` indicates how long you want the cellular automaton to run. The simplest rule specification is a rule number between 0 and 255, as above. The simplest initial configuration is `{{1}, 0}`, which specifies that a single 1 should lie on a background of zeros. Thus, you can generate the first 16 rows (0 through 15) of Pascal's triangle using rule 90 as follows.

`CellularAutomaton[90, {{1}, 0}, 15]`



This is particularly useful if we want to run the CA for a long time, since we'll generate a much larger matrix.

```
ArrayPlot[CellularAutomaton[90, {{1}, 0}, 255]]
```



Exercise 3: Use the `CellularAutomaton` command to visualize the CA from 1. Remember that you already found the rule number.

Exercise 4: Evaluate the self-similar structure of the limiting picture generated by the CA from 1. To really do this, you'd need to find an IFS or digraph IFS that generates the same picture. This might be hard.

There are several other interesting classes of cellular automata that can be numbered like we numbered the elementary cellular automata. (Some of my favorites are the *totalistic* CAs.) The *Mathematica* `CellularAutomaton` command allows you to refer to these by number. A much more general approach is to simply specify the transition rule explicitly. In this case, the `rule` has the form `{f, {r}}`, where `f` is a function and `r` is the radius of the neighborhood. Thus, we can specify rule 30 as follows.

```
Clear[f];
f[{a_, b_, c_}, _] := Mod[a + c, 2];
Grid[CellularAutomaton[{f, {1}}, {{1}, 0}, 8]] /.
  0 -> Style[0, Lighter[Gray]]
```

```

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0
0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

```

Now, we can generalize. For example, here's an *infinite* state CA.

```

Clear[f];
f[{a_, b_, c_}, ___] := a + c;
Grid[CellularAutomaton[{f, {}}, 1], {{1}, 0}, 8] /.
  0 -> Style[0, Lighter[Gray]]

```

```

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 2 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 3 0 3 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 4 0 6 0 4 0 1 0 0 0 0 0 0
0 0 0 1 0 5 0 10 0 10 0 5 0 1 0 0 0 0
0 0 1 0 6 0 15 0 20 0 15 0 6 0 1 0 0 0
0 1 0 7 0 21 0 35 0 35 0 21 0 7 0 1 0 0
1 0 8 0 28 0 56 0 70 0 56 0 28 0 8 0 1

```

It looks a little strange when passed to `ArrayPlot`, since the numbers vary so much.

```

Clear[f];
f[{a_, b_, c_}, ___] := a + c;
ArrayPlot[CellularAutomaton[{f, {}}, 1], {{1}, 0}, 32]

```

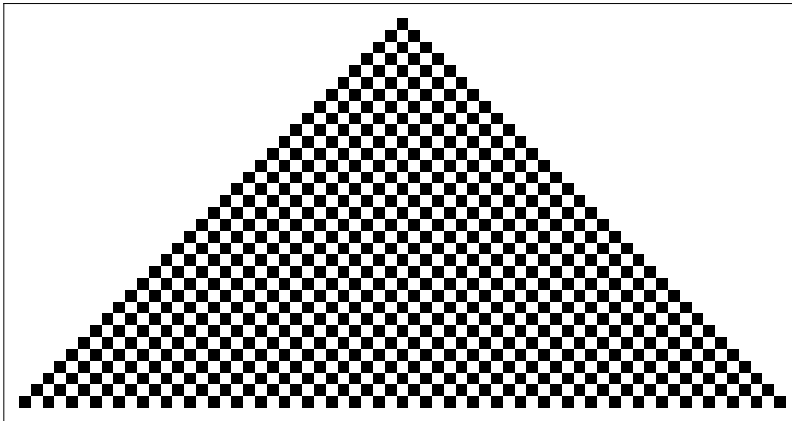


For this reason, it might make sense to use the `Sign` function, to change all the non-zero terms to 1.

```

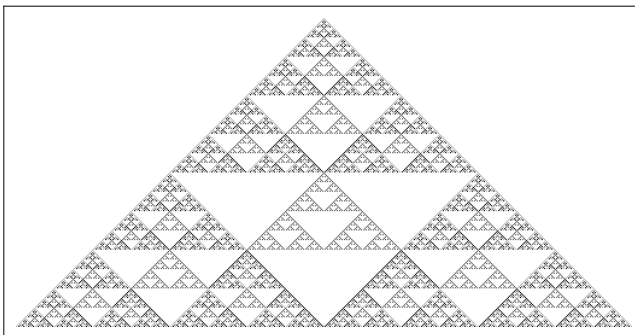
Clear[f];
f[{a_, b_, c_}, ___] := a + c;
ArrayPlot[Sign[CellularAutomaton[{f, {}}, 1], {{1}, 0}, 32]]

```



Or, better yet, we can use any modulus we like.

```
Clear[f];
f[{a_, b_, c_}, ___] := a + c;
ArrayPlot[Mod[CellularAutomaton[{f, {}}, 1], {{1}, 0}, 255], 4]]
```



Exercise 5: Specify elementary rule number 101 as a function and visualize.

Exercise 6: Suppose we have a three state CA with $r = 2$ following the rule $f(a, b, c, d, e) = (a + b - 2c^2 + d + e) \bmod 3$. Implement and visualize.

Consider our fundamental example, the rule 90 CA:

```
Grid[CellularAutomaton[90, {{1}, 0}, 8]] /.
  0 -> Style[0, Lighter[Gray]]
```

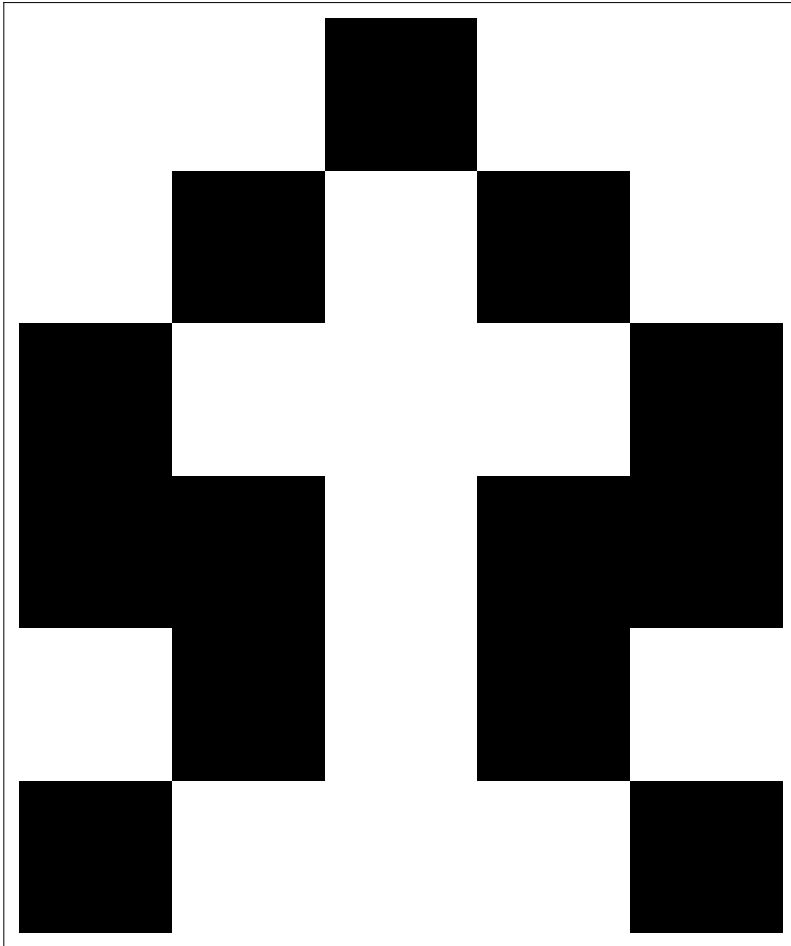
```
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0
0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

How many ones are there through level $2^n - 1$?

Note the picture above shows levels 0 through 8.

We can run a CA on a finite grid as well. We typically assume periodic boundary conditions. For example,

```
ArrayPlot[CellularAutomaton[90, {0, 0, 1, 0, 0}, 5]]
```



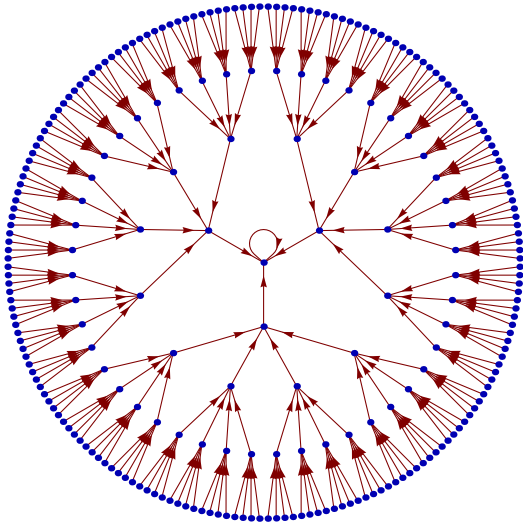
Note that an n -state CA run on a finite grid of size m has only n^m possible state vectors. For example, the overall state of 2-state CA on a grid of length 4 must be one of these:

```
Tuples[{0, 1}, 4]
```

```
{0, 0, 0, 0}, {0, 0, 0, 1}, {0, 0, 1, 0}, {0, 0, 1, 1}, {0, 1, 0, 0},
{0, 1, 0, 1}, {0, 1, 1, 0}, {0, 1, 1, 1}, {1, 0, 0, 0}, {1, 0, 0, 1},
{1, 0, 1, 0}, {1, 0, 1, 1}, {1, 1, 0, 0}, {1, 1, 0, 1}, {1, 1, 1, 0}, {1, 1, 1, 1}
```

Thus, we can understand the overall behavior of such a CA using its *transition diagram*. This is a directed graph whose vertices are the state of the CA; we place an edge from state vector u to state vector v if u transitions to v under the rule. For example, $\{1, 0, 0, 1\} \rightarrow \{1, 1, 1, 1\}$. We can visualize the long term behavior of the CA by plotting this graph:

```
GraphPlot[# -> CellularAutomaton[90][#] & /@ Tuples[{0, 1}, 8],
DirectedEdges -> True, VertexLabeling -> Tooltip]
```



So, what does this tell us?