Developing a 1D CA

Original Author(s): Koen Koning & Michael Lees

Note for 2018: please update the assignment file (Mac error)

Objectives
This lab exercise has three objectives:

1. Learn more features of Python and the libraries.
2. Develop a model of a 1D CA that can execute for different k and r values.
3. Explore the 1D CA parameter space and build understanding.

Outcomes
By the end of the lab you should be able to:

1. Understand more of Python and the framework.
2. Graphically show and interpret qualitative behavior of 1D cellular automata.

In this exercise a model of a 1D cellular automata (CA) will be created. You'll need to have a basic idea of how 1D CAs work; you can find an introduction in the lecture slides.

The goal

By the end of the exercise, you should have implemented a 1D cellular automata that can incorporate a range of k values (alphabet size) and a range of r values (neighbourhood), and have written code to automatically produce a graph showing average cycle lengths for (some) elementary CAs.

With $k = 3$ and $r = 1$ we have an input alphabet (or rule table size) of 3^3. The total number of possible rule tables is $3^{3^3} = 7,625,597,484,987$. Recall that a rule number specifies one instance of a particular rule table in all possible rule tables. Also recall that the rule number corresponds to the output configuration of the rule table (the input sequence being fixed in decreasing order). When running the model, the maximum rule number can be calculated (from r and k) and displayed. The user then specifies a rule number to generate in decimal format. To interpret rule 34 we need to convert it to a number in base k of length $3^3 = 27$:

34 decimal = $34_{10} = 1021_3$ = 1021 ternary = [0 0 0 0 0 . . . 0 1 0 2 1] (23 leading zeros)

If you are need more information regarding base-n numbers and how to convert to and from decimal (base 10), take a look at http://cs.umd.edu/class/sum2003/cmsc311/Notes/Data/toBaseK.html. Try to understand why this works!

Of course, you are free to design the program in any way you would like. To ease the process, some skeleton code is available on Blackboard, with some suggested structure for the code. Assuming you use the skeleton code, we'll discuss how you need to fill it in.

The design

The first thing you should implement is the function `build_rule_set`. This function should, given a decimal n, build a list of length $k^(2r + 1)$ representing the rule table for the CA. So (from above) if $n = 34$, $k = 3$ and $r = 1$ `build_rule_set` should return the list:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 2 1] (length $3^3 = 27$)

This rule set (list) will then be used to drive the CA forward. In simple terms, the first zero in the rule means:
2 2 2 $\rightarrow$ 0

Remember we always list input in decreasing order and $[2\ 2\ 2]_3 = 26_{10}$

The second to last item in the list indicated that whenever we see a neighbourhood of [0 0 1] we change our state to 2: [0 0 1] $\rightarrow$ 2.

The `step` method of the model will determine the neighbourhood for every cell, and pass this to the `check_rule` function, which should return the new state.

In summary

You have two choices for this lab – build on the skeleton code or try from scratch yourself. If you feel capable then I would suggest starting from scratch yourself – this is all about learning!

Using the skeleton code you only need to implement 4 functions to make the model work, which are hopefully the most interesting ones:

- decimal_to_base_k
- build_rule_set
- check_rule
- setup_initial_row

The comments in the code should also give you ideas regarding how to start thinking about the algorithms. Please make sure you understand all other code so you really have a clear idea about how the whole program works.

Verification and Analysis

Remember, the whole point of programming simulations is to run experiments with them. Although the type of 1D CA we have developed is not a model/simulation per se, we can still experiment with it.

We can also go some way to verifying our implementation, and you should try to do this.

There are many ways to do this, but one way is to compare the output of your CA with some output (from other implementations) you know (or assume) to be correct. If you set $k = 2$ and

$r = 1$ you have the classic wolfram rule codes. Try running the following rules for the elementary cellular automata (i.e. $k = 2$, $r = 1$):

- 30
- 110
- 184
- 220

Then you could compare your results to the examples Wolfram MathWorld[1]. The next stage would be to look at a few different rule numbers yourself and see if you can characterize them in some way? E.g. boring, interesting, chaotic/noise, ...

Measurement and Testing

Now you have verified that your code creates correct results you can again look at running some experiments.

In the lectures we examine cycle lengths in elementary (k=2, r=1) CAs. You can now try to measure average cycle lengths for all 256 Wolfram Rules. Of course, you will have to set a limit on how long to run for, or more precisely how many steps you will run for until you assume no cycles happen. This could 10^4, or 10^6, or more. While debugging, you might want to pick something shorter, so that your code runs quickly while you're testing it.

More importantly, you might want to consider smaller system sizes (i.e. smaller widths) – but make sure to mention what you tried (and try to justify why).

Obviously you will need to think about initial conditions (random vs. single seed) and how many repetitions you plan to run[2].

Please plot the rule numbers [0, 255] along the x-axis and (average) cycle length on the y-axis. Some helpful notes for this plot (and earning credit):

- It may be visually very helpful if you'd sort the rule numbers on the x-axis in the order of their Wolfram class. Do you see clear differences between the classes, and is it what you'd expect?
- Since every rule can have a multitude of different cycles, it would be good to run multiple initial conditions and compute the average cycle length for each rule. It would also be nice to show the dispersion (spread) of cycle lengths for each rule, e.g. using error bars or showing the scatter points.

[1] http://mathworld.wolfram.com/ElementaryCellularAutomaton.html

[2] It may well be that, for the same rule, one initial condition leads quickly to a short cycle whereas another initial condition takes a very long time to reach a cycle (has a long 'transient'), and this cycle may be much larger.

Grading

You should submit your program code and the resulting figure (or a screenshot) in a zip file to Google Classroom. It should be functional (include instructions if necessary), and when your code is run (provide instructions and/or a shell script, if necessary), it should produce a single figure showing your analysis, including labeled axes and a figure caption with sufficient (reproducible) description (max. 250 words). Do not include a full written report. The figure may be shown on screen or it may be written to a file in the same directory.

Your grade will primarily be based on the correctness of your code, and the validity and clarity of the resulting figure. Other influences are things mentioned above – a good choice of parameters (such as system size), whether you arranged the rules along their classes, whether you did an averaged calculation, whether you show the dispersion, and the quality of the figure caption in describing what we see and how it is computed.

Beware: Your code will be tested for plagiarism using special-purpose software.

This assignment will count 10% towards your grade for the CA practical assignments.

Deadline: Thursday, November 29, 23:59.