Systems Biology – Exercises

Week 6: Cellular Automaton

Cellular Automaton

One of the most straightforward ways to simulate and visualize emergent behavior is using *Cellular Automata* (CA). We have investigated *Conway's Game of Life. W*e could observe emergent behavior on a two-dimensional lattice.

Another important CA is one-dimensional. Stephen Wolfram did an extensive work on one-dimensional CA, investigating a set of 256 different models [1]. There are four categories grouping these models:

- Class 1: rules lead to homogenous states
- Class 2: rules lead to stable and/or periodic patterns
- Class 3: rules lead to seemingly random behavior
- Class 4: rules lead to complex patterns, locally propagating structures

[1] http://new.math.uiuc.edu/im2008/dakkak/papers/files/wolfram.universityofca.pdf

Rule 90 to Produce a Sierpiński triangle

- We have investigated Rule 90, which can be used to produce a Sierpiński triangle

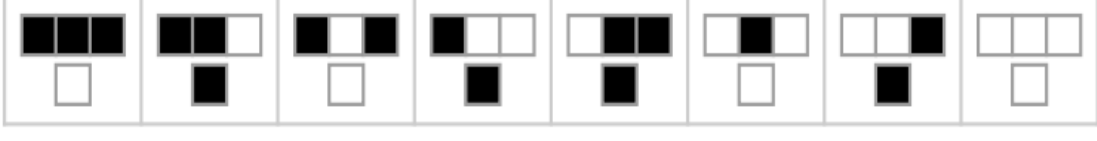


Figure 1: Wolfram Classification Scheme: Rule 90

- Figure 1 is a visualization of the cellular automaton rule of Rule 90. This can be effectively represented in Python by using the dict type.

Code 1: Python's dict data type to represent Rule 90.

```
Rule90 = {"111":"0", "110":"1", "101":"0", "100":"1",
          "011":"1", "010":"0", "001":"1", "000":"0"}
```

Flexible way to apply rules

- There is a more flexible way to apply rules, particularly when testing many different ones. Examine the following code with wolfram_rule_string returning as a string an 8-bit binary integer number. Each element out of that string is assigned to each number of the Wolfram classification (wolfram_rule_dict)

```
# 8-bit binary integer number (here 90) as a string
wolfram_rule_string = "{0:{fill}8b}".format(int(90), fill="")
#print (wolfram_rule_string)
# apply the 8-bit binary number to create the Wolfram rule
wolfram_rule_dict = {"111":wolfram_rule_string[0], "110":wolfram_rule_string[1],
"101":wolfram_rule_string[2], "100":wolfram_rule_string[3],
"011":wolfram_rule_string[4], "010":wolfram_rule_string[5],
"001":wolfram_rule_string[6], "000":wolfram_rule_string [7]}
```

Code 2: Code example to manage rules more flexibly.

Iteration for Wolfram rule

- To check each 3-bit value of the n-bit rows, let's consider the following function, which returns a new string according to the Wolfram rule.

Code 3: Function to return the next iteration of a Wolfram rule.

```
def wolfram_fkt():
    x = ""
    for i in range(len(iterstr)-2):
        x += wolfram_rule_dict[iterstr[i:i+3]]
    print (x)
    return x
```

Initialization

- To initialize the one-dimensional CA, we need a starting condition. This line is written so that a variable size can be assigned a value to represent the size (length) of the CA.

Code 4: Starting condition of the Wolfram Rule CA.

```
# seed with center "1" on a background of "0"
iterstr = "0"*math.floor(size/2) + "1" + "0"*math.floor(size/2)
```

- Hint! The number of iterations to print the output should be half the size of the initialization (iterstr). Write a function to print wolfram_fkt() and update iterstr for its next iteration. Also: have you imported necessary libraries?

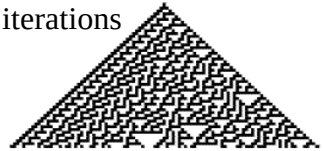
```
import string , math , random
```

Exercises

- **Exercise 1**: Combine and extend the code provided, so it prints the Wolfram rule (iteratively from “Starting condition”) in the console.
- **Followup Exercise 1**: Your console output is shrinking—why? What is the simplest way to fix this issue?
- **Exercise 2**: Write another function that replaces “0” and “1” with a character of your choice when printing the output.
- **Exercise 3**: Write a short code that starts the CA with a random sequence.

Homework

Figure 2: Wolfram's Rule 30 with 50 iterations



Due next Wednesday (17:00, 10th, Nov.) electronically to manaba+R.

• File format: YourStudentID_Week06_n. py (ID without hyphen, e.g., 12345678901_Week06_1.py).

• Your code must include your own comments for all code sections. Go line-by-line. Comments in your program must be full sentences and reflect your understanding of the code.

Q1. Complete your program so that the user will be prompted with three inputs:

• an integer number between 0 and 255 to choose different Wolfram rule for CA

• a number for the iteration size (the number of lines printed)

• initial condition random or non-random (latter a single "1" on a background of "0")

Check for invalid input! Print the output to the console.

Q2. Conway's Game of Life can easily be changed to create other different dynamics [3]. Extend your existing program to run the other two different versions:

• Maze (http://www.conwaylife.com/wiki/Maze)

• Move (http://www.conwaylife.com/wiki/Move)

In the random grid population, there is a value to skew either alive or dead cells, e.g., p=[0.1, 0.9]) in Conway's Game of Life. You have to change the ratio to emphasize the dynamics of Maze and Move.

Q1 → YourStudentID_Week06_1.py,
Q2 → YourStudentID_Week06_2.py

[3] List CA similar to Conway's Game of Life: http://www.conwaylife.com/wiki/List_of_Life-like_cellular_automata

Systems Biology – Exercises

Week 7: Course review for week 2-6