PSet1 Report

Ali Abolhassanzadeh Mahani

Sep. 28

1 Problem 1

1.1 Creations and the loops

For starters, since I know how many steps I'm going to have and the size of the row, I allocate a table of that size in memory using `np.ndarray` with `int` as my data type; This will be my canvas.

Now, I do a nested for loop for each *round* and each *element* in the row to find their situation and decide their next state. Here, some people use `(index) % len(row)` to set the boundary condition, but with this setup, we are doing a `mod operation` per element and it makes it a little inefficient. I used the fact that Python `ndarray`s support *negative indexing* and just set the upper boundary condition with an if statement.

1.2 The rule

Our rule is as follows: *"If only one of my neighbors is 1 (i.e. Has a hat), I become 1 (i.e. put on the hat), otherwise, I become 0."*

The mathematical interpretation of *"If only 1 neighbor is 1"* is that the sum of the neighbors equals to 1. Hence the `if` statement in our code.

1.3 Fancy

The style of which I have written this code is to be able to import the functions, if I need them in the future. All that the last `if` statement does, is to see if we are executing this file, and then, execute the `main()` function.

1.4 Analysis

The main part of the code is the `put_hat()` function, which consists mainly of two nested `for` loops. taking one to have m iterations and the other, n, we can see that the runtime complexity of this code is $\mathcal{O}(m.n)$. In order to analyze this code, one can run a profiler on it as follows:

```
python -m cProfile -s time Hats.py
```

Note that this profiler lists all the modules that are being called, so due to our use of `numpy` and `matplotlib.pyplot`, it will show a lot of things, but the main function `put_hats` will be at the top few.

1.5 Results

I used the `pcolormesh()` module from the `matplotlib.pyplot` library to export my canvas into a file. The resulting image is identical to rule 18, 26 or 90. (Figure 1.5)

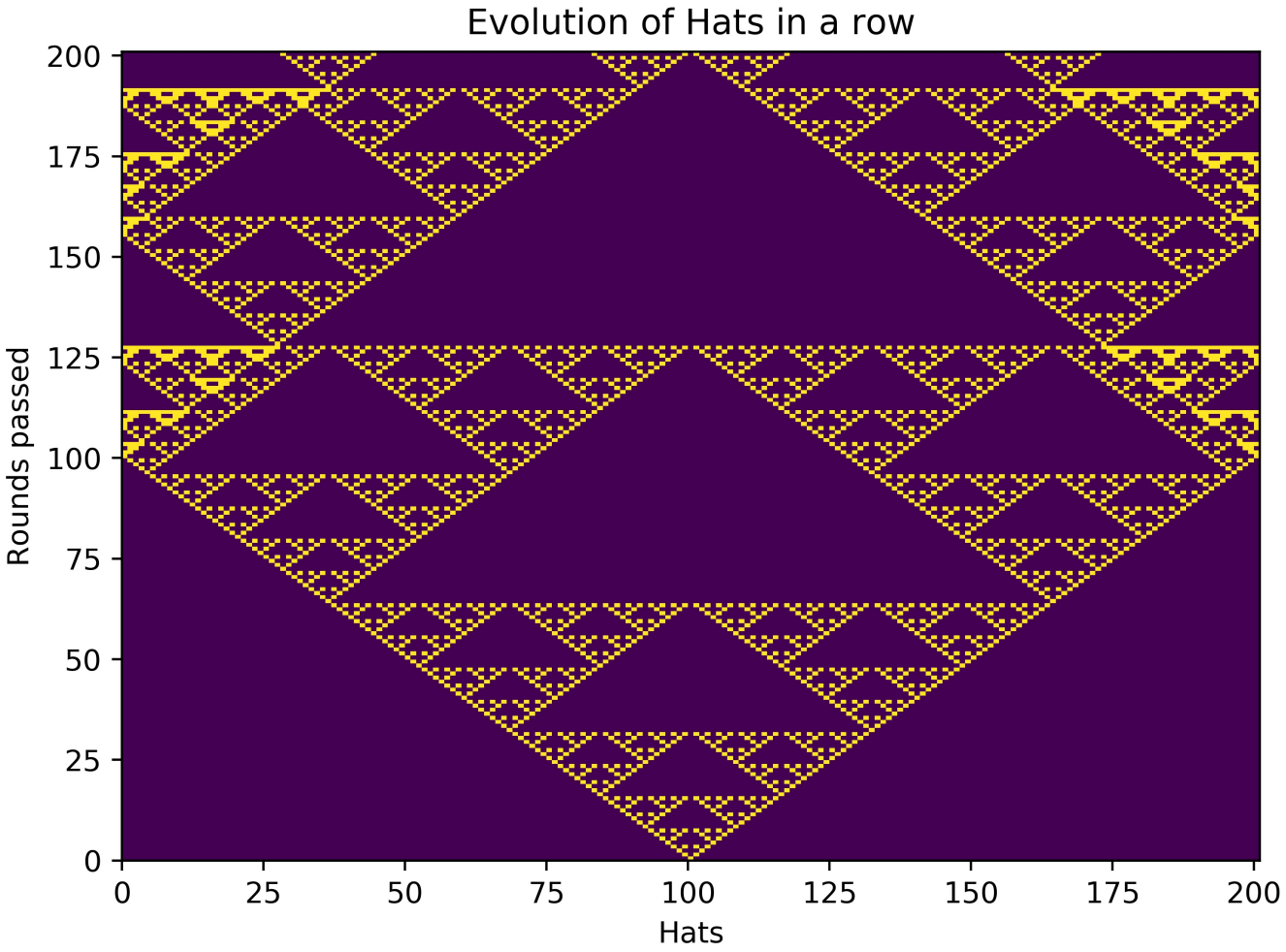

Figure 1: The evolution of the hats for 200 steps using the rule that if only one of my neighbors has a hat, I put on a hat, otherwise, I put down my hat. The yellow cells have a hat on and the purple ones don't. *Note that this rule is independent of the item in question.*

2 Problem 2

2.1 The Code

This project consists of 3 files. `256rules.py` which is the main file to be run, `cells.py` which contains the creation and evolution of cells, and finally, `graphics.py` which contains the modules to make the image files.

As for analysis, one can use the code used in section1.4 to run a profiler on the main functions. But the algorithm used is the same as the one before.

2.2 Results

Using the `graphics.py` module we output the file with its rule number in binary as its name. The result are included in Fig 2.2. As one can see, the CA rule 75 makes a chaotic-looking pattern meaning that it's classified in **Class 3** in Wolfram, while rule 110 leads to a comples pattern that seems to repeat itself locally, hence in **Class 4** in Wolfram.

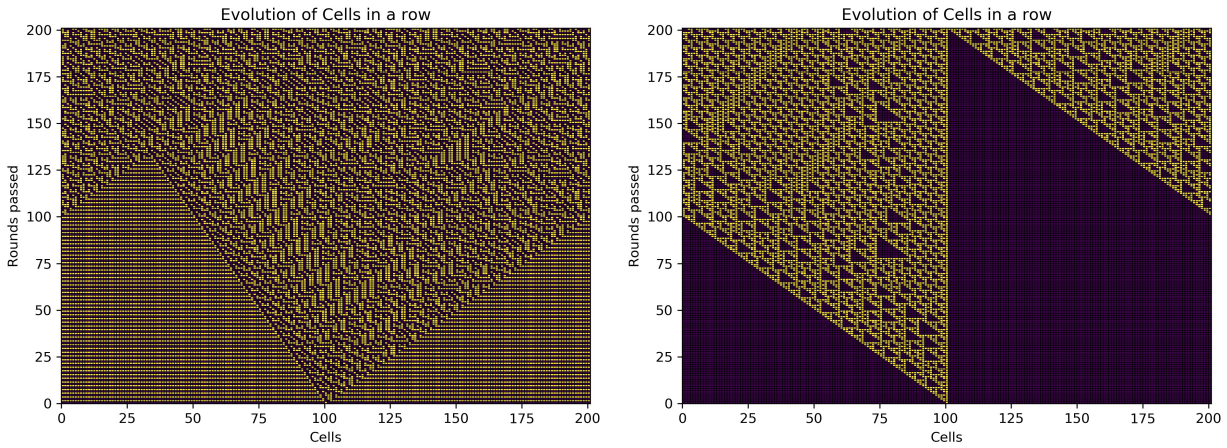

Figure 2: These images where created using rules 110 (on the right) and 75 (on the left) of the 256 Elementary Cellular Automata Rules. Color yellow means the cell is *on* and purple means that it's *off*.

3 Problem 3

3.1 The Code

This project is almost identical to the previous one. The only difference is the main file `iteration_bitflip.p` This time we take rule 110 and iterate over its digits and do a bit flip for each one. Then we make the CA for that rule and output it as before.

3.2 Results

The result are the same as the ones in question in the assignment. (Figure 3.2) As one can see, rules `00101110`, `01101010`, `01101100` are Class 2 CAs and rules `01001110`, `11101110`, `01100110` are class 4 CAs, while `01101111`, `01111110` seem chaotic and are classified as Class 3 CAs.

4 Problem 4

4.1 The Code

Here we have the main file `gol.py` which stands for Game of Life (GoL). In the subfolder `classes`, we have `types.py` which contains a set initializers for our 4 models, `operations.py` which is contains everything we need to make decisions for the model, and `graphics.py` which consists of the module `animate`. This module takes an array, makes a new one and initializes it with the input, and prints an image of it. Then, it decides the next frame and saves the new array to `canvas`, and prints it, and so on the loop goes. I chose 20 frames to be the default. (Let's be honest, I just hard coded it! ;-P)

4.2 Results

At last, I took the frames and made a gif with them using this GIF Maker. The files are available at the corresponding subdirectories. I checked them with this source and they are spot on! Yay!

5 Conclution

I Hate `matplotlib.animation.FuncAnimation` soooo much!

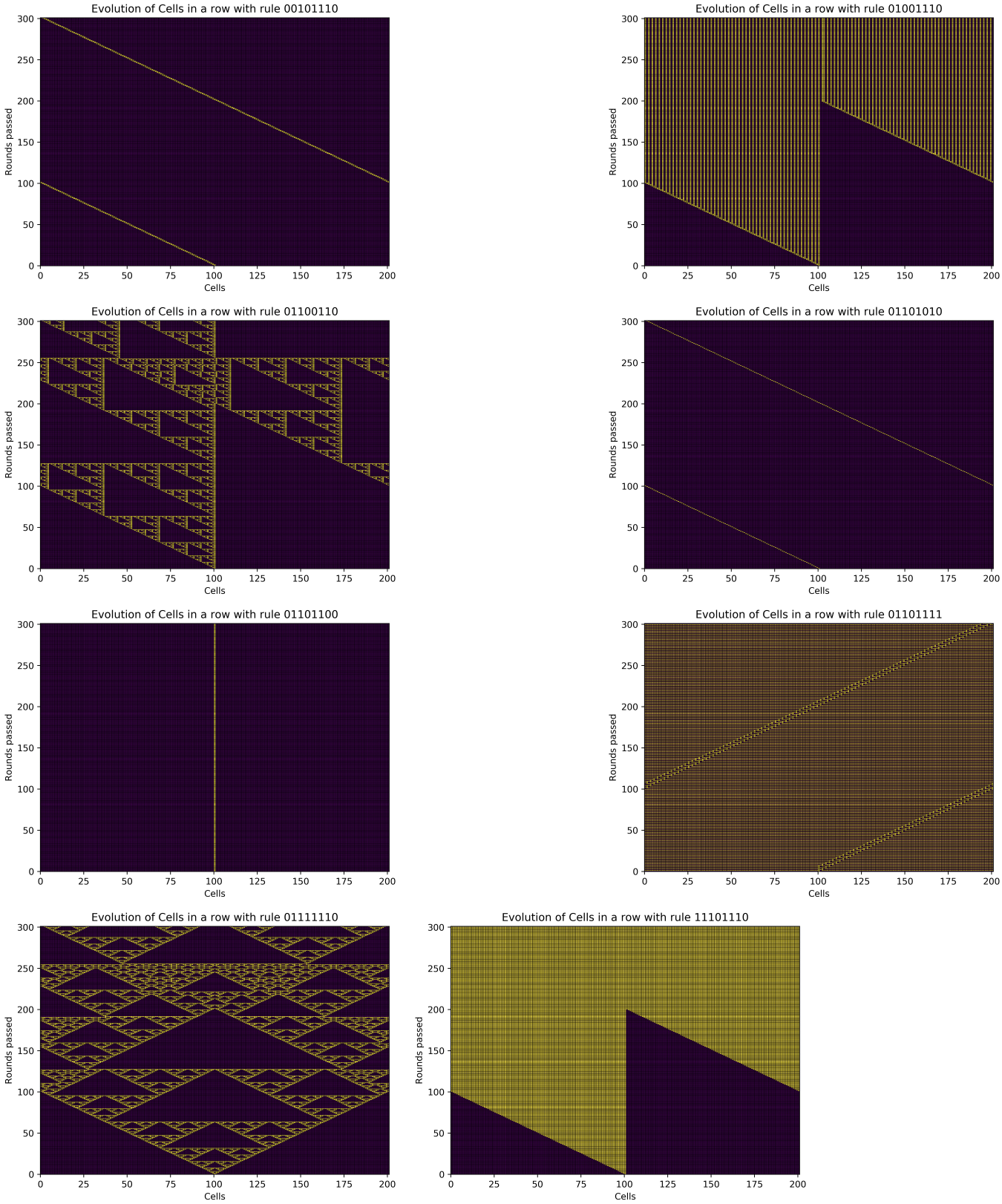

Figure 3: This collections of images is created by performing bitflips on every binary digit of rule 110. The yellow cells are *on* and the purple ones are *off*.