CSC120H Lab 4

Objectives

1. Practice with file reading
2. Practice with numpy arrays, particularly filtering
3. Practice with if-statements
4. Introduction to discrete-event simulation

Marking

Every lab will be marked out of 4. Each lab will contain four items marked **TODO:** – completing each TODO item is worth one mark. Unlike the other labs, however, this one will not be marked for completing each TODO item; participation in this lab counts for one lecture activity credit.

Driver and navigator

Throughout the term, we will use the terms *driver* and *navigator*. Here are the definitions of the two roles:

**driver:** The person typing at the keyboard.

**navigator:** The person watching for mistakes, and thinking ahead.

Here is the most important rule for this and all future labs:

The **navigator must not touch the keyboard or mouse.** The driver’s role is to work with the computer, and the navigator’s is to think about language issues and upcoming issues related to the problem being solved. If the navigator interferes with the driver, the group loses its view of what is coming up *and* the problem may become harder to solve *and* the driver will find it harder to learn the material.

In every lab handout, we’ll call you two s1 and s2, and s1 will be the first driver.

1. Is it a leap year?

*Assign roles: s1 drives and s2 navigates.*

From [https://www.timeanddate.com/date/leapyear.html](https://www.timeanddate.com/date/leapyear.html) “In the Gregorian calendar 3 criteria must be taken into account to identify leap years:

1. The year is evenly divisible by 4;
2. If the year can be evenly divided by 100, it is NOT a leap year, unless;
3. The year is also evenly divisible by 400. Then it is a leap year.

This means that 2000 and 2400 are leap years, while 1800, 1900, 2100, 2200, 2300 and 2500 are NOT leap years.”

**TODO:** Complete the function `is_leap_year` in `lab4.py` which returns `True` if it is given a leap year, and `False` otherwise.
2 Conway’s Game of Life

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton. The “game” is actually a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours becomes a live cell.

The initial pattern constitutes the ‘seed’ of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed. Births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

2.1 The next state for a cell

Switch roles: s2 drives and s1 navigates.

TODO: Complete the function get_next_state. This function takes in the number of neighbours a cell has, and the current state of the cell. It returns what the new state of the cell should be, using the rules above. If given an invalid number of neighbours (such as -1, or 100) it should return -1.
2.2 Finding the neighbourhood of a cell

*Switch roles: s1 drives and s2 navigates.*

A vital part of the Game of Life is being able to see the ‘neighbourhood’ of a cell: the 8 cells that border it. Say we had this array:

```python
>>> import numpy as np
>>> Z = np.array([[0,0,0,0,0,0],
                 [0,1,0,1,0,0],
                 [0,1,0,1,0,0],
                 [0,0,1,1,0,0],
                 [0,0,0,0,0,0],
                 [0,0,0,0,0,0]])
```

The neighbourhood of the cell at row 2 and column 1 is:

```
array([[0, 1, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

And the neighbourhood of the cell at row 0 and column 0 is:

```
array([[0, 0],
       [0, 1]])
```

**TODO:** Complete the function `get_neighbourhood`, providing a docstring for it. This function takes in the complete array, and the row/column of the cell we want the neighbourhood of. It returns the neighbourhood.

2.3 The number of neighbours

*Switch roles: s2 drives and s1 navigates.*

We also want to be able to know how many neighbours a cell has. Complete the function `num_neighbours`, which tells you how many neighbours the cell at the provided row/column has.

Download the files `seed1.csv` through `seed5.csv`. These seed files provide the starting state of the five Games of Life we will simulate. Our first task is to import these csv files into numpy arrays.

Once you have this function working, play with the parameters in the main body. In particular, turn `animate` to True. **TODO:** Once you think you have the animation of `seed5.csv` working for 20 time steps, show your work to your TA.