SCI1022 session 10 in-class tasks: Conway's Game of Life

Welcome to session 10 of SCI1022!

Some miscellaneous instructions to begin with:

- Use your two hours in this workshop to work through this Jupyter notebook.
- Read the instructions as you go. Execute the example code cells provided, and look carefully at the output.
- Recall that, in order to execute a code cell, then you can left-click on it (either on the interior or the left-hand-side margin of the cell), and then hit `Shift-Enter`, which stands for: "While holding pressed the `Shift` keyboard key, press the keyboard key labelled as `Enter`".
- **Tasks** are marked in **red** and displayed in indented blocks.
- When you finish all tasks, **upload this Jupyter notebook in Moodle with the tasks solved**. Before submitting, **we highly encourage that you restart the Jupyter Kernel, and then execute all code cells in the search of unnoticed errors.** In order to restart the Jupyter kernel and run all cells in sequence afterwards you can use the `Kernel->Restart & Run All` option in the menu at the top of the browser page.
- Ask your instructors for help at any time.

This week we are going pan-science. We will in particular explore a famous mathematical "game": the **Game of Life** (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life), also known as **Life**. It was devised by the British mathematician John Horton Conway (https://en.wikipedia.org/wiki/John_Horton_Conway) in 1970.

While formal study of games (or *cellular automata* to use the correct term) is a branch of mathematics, the original idea came from physics. This game is inspired by biology, and there are even unusual chemical reactions which behave like these games.

In a previous workshop, *The Collatz conjecture*, you saw that a simple rule applied over and over to an integer, could lead to a complicated sequence.

Now that you are acquiring more fluency with computer programming (we hope!), we can explore applying a simple rule over and over, but to a 2D (two-dimensional) grid of numbers. It turns out that using just two numbers - `0` and `1` - is sufficient to implement the Game of Life.

We will use *2D Numpy arrays* in order to store such 2D grid of numbers in the computer. These were already introduced in the previous workshop, but this week the main goal is to acquire even more fluency with handling 2D data.

We stress that a 2D Numpy array is not the only data structure that one may use to implement the Game of Life. One may use, e.g., nested lists (i.e., a list of lists), instead. Indeed, this is a decision that you will have to frequently face while coding: which is the data structure best suited for my program?

The answer to this question is not trivial. As you might have guessed, it highly depends, among others, on the problem to be solved, and the algorithm devised to solve that problem. In general, you should choose the data structure that strikes the best balance among different factors, including code performance, code readability, and memory consumption, to name a few.

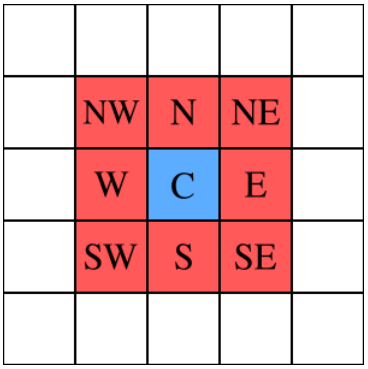
Implementing the Game of Life

Understanding the rules of Life

The Game of Life is "played" on a board, which is just a square grid of cells. Cells are either dead (represented as `0` s in the computer) or alive (represented as `1` s in the computer). Life runs across several steps or iterations (also known as generations). At each step of Life, we start from a given configuration of the board (i.e., distribution of live and dead cells), and we generate the one of the next step (i.e., we decide which cells become alive or die) according to the following four rules:

1. A live cell with fewer than two (i.e., zero or one) live neighbours dies (underpopulation).
2. A live cell with two or three live neighbours survives (i.e., it will remain alive in the next generation).
3. A live cell with more than three live neighbours dies (overpopulation).
4. A dead cell with exactly three neighbours becomes a live cell (reproduction).

In order to aid you in understanding the rules above, the 8 neighbours of the cell in blue, are shown in red in the following picture:



Development plan

Before writing any code, *no matter how simple*, it is essential that you devise a development plan. In the procedural programming paradigm, this essentially amounts to sketch how you are going to divide your program into smaller functions, and decide how they are going to interact with each other. Let us make explicit the development plan for Life. **It is best our functions to be as concise as possible**. In particular, recall our mantra, **a function must do only one thing**!

We are going to split the implementation of the Game of Life into three (main) functions:

1. A function to draw the Game of Life board.
2. A function which, given the current generation of cells, returns the next generation.
3. A function which combines 1., and 2. to evolve Life a given number of steps, given an initial configuration.

Drawing the board

As mentioned above, we are going to represent the board of the game of life with a 2D Numpy array made of `0` s and `1` s. Evaluate the code cell below to make a random array of cells that are either `0` or `1`. This code creates a 10×10 board of random integers (using the `randint` function as provided by Numpy), chosen between `0` and `2`. Because Python does not include end points, that means the integers randomly generated are either `0` or `1`. That it is precisely what we want.

In [1]:
```
from numpy.random import randint
board = randint(2, size=(10, 10))
print(board)
```

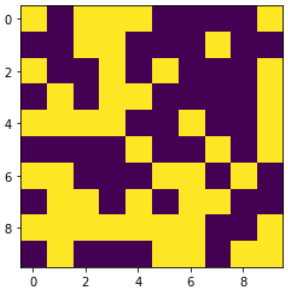
```
[[1 0 1 1 1 0 0 0 0 1]
 [0 0 1 1 0 0 0 1 0 0]
 [1 0 0 1 0 1 0 0 0 1]
 [0 1 0 1 1 0 0 0 0 1]
 [1 1 1 1 0 0 1 0 0 1]
 [0 0 0 0 1 0 0 1 0 1]
 [1 1 0 0 0 1 1 0 1 0]
 [0 1 1 0 1 0 1 1 0 0]
 [1 1 1 1 1 1 1 0 0 1]
 [0 1 0 0 0 1 1 0 1 1]]
```

That is fine, but we would like it to look a bit nicer. The function in the cell below draws a `board` (a 2D Numpy array as above) using the `matshow` function of the `matplotlib.pyplot` Python library.

In [2]:
```
import matplotlib.pyplot as plt
%matplotlib inline
def draw_board(board):
    fig = plt.figure()
    plot= plt.imshow(board)
    return fig, plot
```

In [3]:
```
draw_board(board)
```

Out[3]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.image.AxesImage at 0x7fc684ff1d90>)
```



Computing the next generation

Accordingly to the rules of Life introduced above, a basic operation that we will need in order to determine the next generation (i.e., to move Life one step forward) is to count the neighbours (see picture above) of a cell which are alive (Think why!). Let us write such a function in baby steps.

Task 1. Write a function `cell_exists(board,I,J)` that lets you double-check whether a given cell exists in the `board`, given its the cell coordinates, i.e., its row identifier `I`, and column identifier `J`. Use the `test_cell_exists()` function in order to test the correctness of `cell_exists`.

In [4]:
```
# Code to solve Task 1 goes here (function definition statement)
```

In [5]:
```
# Task 1 verification function
from numpy.random import randint
def test_cell_exists():
    board = randint(2, size=(10, 5))
    assert not cell_exists(board,-1,0), "Test failed!"
    assert not cell_exists(board,10,0), "Test failed!"
    assert cell_exists(board,1,1), "Test failed!"
    assert cell_exists(board,9,4), "Test failed!"
```

In [6]:
```
# Code to test Task 1 goes here
test_cell_exists()
```

Task 2. Write a function `cell_is_alive(board,I,J)` that returns `True` if the cell with coordinates `I`, `J` is alive, and `False` otherwise. Use the verification function `test_cell_is_alive()` in order to test the correctness of `cell_is_alive`.

In [7]:
```
# Code to solve Task 2 goes here (function definition statement)
```

In [8]:
```
# Task 2 verification function
import numpy as np
def test_cell_is_alive():
    board = np.array([[0,0,0,0,0,0],[0,1,1,0,0,0],[0,1,1,0,0,0],[0,0,0,1,1,0],[0,0,0,1,1,0],[0,0,0,0,0,0]])
    assert cell_is_alive(board,1,1)      # Must return True (why?)
    assert not cell_is_alive(board,3,5)  # Must return False (why?)
```

In [9]:
```
# Code to test Task 2 goes here
test_cell_is_alive()
```

Task 3.

- Write a function `count_live_neighbours(board, I, J)` in the cell below. This function computes the number of **live** neighbours of a cell, given a `board` and the coordinates of a cell in the board, i.e., its row identifier `I`, and column identifier `J`. Use the verification function `test_count_live_neighbours()` in order to test the correctness of `count_live_neighbours`.

Some hints:

- The function might be implemented either with multiple `if` statements (i.e., one per neighbour) or two nested `for` loops. Both solutions are valid, but the latter is much more concise in terms of lines of code.
- You have to be able to determine the coordinates of the neighbours of a cell. For example, the North-West neighbour (see NW in the picture above) is the one at row `I-1`, and column `J-1` (think why!).
- Some of the neighbours of those cells which are located at the boundary of the board (e.g., the 4 corner cells of the box) are not actually cells of the board. If you try to access them, then you will get a run-time error. You must skip those neighbours.
- Use `cell_exists` (Task 1) to skip them!

In [10]:
```
# Code to solve Task 3 goes here (function definition statement)
```

In [11]:
```
# Task 3 verification function
import numpy as np
def test_count_live_neighbours():
    board = np.array([[0,0,0,0,0,0],[0,1,1,0,0,0],[0,1,1,0,0,0],[0,0,0,1,1,0],[0,0,0,1,1,0],[0,0,0,0,0,0]])
    assert count_live_neighbours(board,1,1) == 3
    assert count_live_neighbours(board,3,5) == 2
```

In [12]:
```
# Code to test Task 3 goes here
test_count_live_neighbours()
```

Now that we are able to determine the number of live neighbours of a cell, then can write a function to move Life forward towards the next generation.

Task 4. In the cell below, complete the body of the `generate_next_generation(current_generation)` function that, given the current generation, generates a new one. Both, `current_generation` and the returned value, are Life boards, i.e., Numpy 2D arrays made of `0` s and `1` s. Some hints and clarifications:

- Your code should **NOT** modify the entries of `current_generation` . Only the entries of `next_generation` can be modified. (Think why!)
- Your code should have two nested `for` loops.
- The body of the innermost loop should count the number of live neighbours of the current cell in the current generation (Task 3), and apply the rules of Life in order to decide whether the current cell will be a live or dead cell in the next generation.

In [13]:
```
# Code to solve Task 4 goes here (complete the body of the function definition statement below)
```

In [14]:
```
# Task 4 verification function
import numpy as np
def test_generate_next_generation():
    current=np.array([[0,0,0,0,0,0],[0,1,1,0,0,0],[0,1,0,0,0,0],[0,0,0,0,1,0],[0,0,0,1,1,0],[0,0,0,0,0,0]])
    next_known=np.array([[0,0,0,0,0,0],[0,1,1,0,0,0],[0,1,1,0,0,0],[0,0,0,1,1,0],[0,0,0,1,1,0],[0,0,0,0,0,0]])
    next_computed=generate_next_generation(current)
    assert np.all(next_known == next_computed), "Test failed!"
```

In [15]:
```
# Code to test Task 4 goes here
test_generate_next_generation()
```

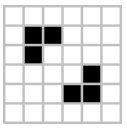
Evolving Life

We now have all building blocks in order to evolve (i.e., **iterate**) the Game of Life. We are almost there! The code cell below is in charge of doing so. The `evolve_life` function uses `Matplotlib` in order to create an animated plot of the game of life. For those interested, click here (https://towardsdatascience.com/animations-with-matplotlib-d96375c5442c) for a revealing article on how to create animations with `Matplotlib` . In any case, the cell below provides all which is needed to visualize Life.

In [16]:
```
from IPython.display import HTML
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation
import matplotlib
%matplotlib inline
def evolve_life(initial_generation, number_steps):
    fig,plot=draw_board(initial_generation)
    generations=[initial_generation]
    for i in range(1,number_steps):
        generations.append(generate_next_generation(generations[i-1]))
    def animate(i):
        plot.set_data(generations[i])
        return [plot]
    anim = FuncAnimation(fig, animate, frames=number_steps, interval = 100, blit=True)
    return HTML(anim.to_jshtml())
```

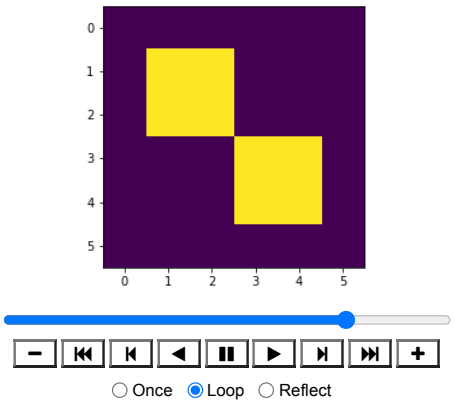
We are going to test our implementation of Life using an "oscillator". Oscillators are Life patterns which return to their initial state after a finite number of generations. They are, thus, periodic. The period of an oscillator refers to the number of steps a pattern must iterate through before returning to its initial configuration. The pattern we are going to test has Period 2, and is referred to as "Beacon".

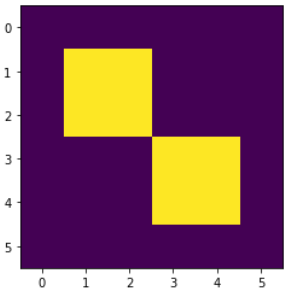
The code required in order to generate "Beacon" is available at the cells below. If you succeeded, you should see the following periodic pattern:



In [17]:
```
import numpy as np
beacon=np.array([[0,0,0,0,0,0],[0,1,1,0,0,0],[0,1,0,0,0,0],[0,0,0,0,1,0],[0,0,0,1,1,0],[0,0,0,0,0,0]])
evolve_life(beacon,10)
```

Out[17]:



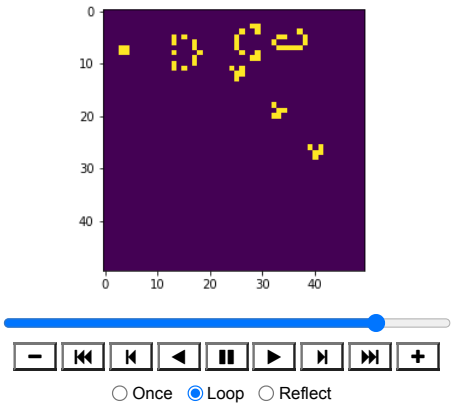


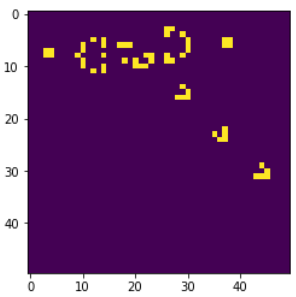
A much more interesting pattern: the "Glider Gun"

In [18]:
```
glider_gun_inner =\
[[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
 [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1],
 [1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
 [1,1,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]
glider_gun = np.zeros((50, 50))
glider_gun[3:12,3:39] = glider_gun_inner
```

In [19]:
```
board = np.copy(glider_gun)
evolve_life(board,100)
```

Out[19]:





Learning outcomes

In this workshop we have written in Python a cell automaton known as the Game of Life. Cellular automata have found application in various areas, including physics, theoretical biology and microstructure modeling.

One interacts with the Game of Life by creating an initial configuration and observing how it evolves. We did it with an oscillator known as Beacon, and with a much more interesting pattern, the Glider Gun. If you want to know more about Life patterns, you might want to read the excellent Wikipedia article on Life (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life). You can indeed try any of the patterns shown in the article with your program!

Along the way, we have acquired much more fluency with 2D array manipulation. The array access patterns (i.e., an element and its immediate neighbours) of the Game of Life are indeed ubiquitous in numerical methods for the solution of problems of scientific and engineering relevance, like the finite element method (https://en.wikipedia.org/wiki/Finite_element_method).

Finally, we have realized about: (1) the importance of devising a development plan whenever one considers the solution of a task of a certain complexity on a computer; (2) the convenience of writing small and concise functions.