Game of life

The game of life is a simple two-dimensional cellular automaton. You can see one in action here:

https://www.youtube.com/watch?v=669LY3H24Q0&t=1s

In the game of life, there is a grid of cells. A cell can be either ‘alive’ or ‘dead’.In each iteration step, the current pattern of alive and dead cells will change according to these three rules:

- If a cell is alive and has two or three alive neighbors, it will be alive.
- If a cell is alive but has less than two or more than three alive neighbors, it will be dead.
- If a dead cell has exactly three alive neighbors, it will be alive.

Your task is to code the ‘game of life’ and display the grid with its changing pattern, just as in the video.

Before you start, create a main.m script that will execute the functions that you will create below. Remember to create a new folder for this exercise where all these functions (and the script) will be saved.

Tasks

Creating the initial state

We will create a function called init_matrix() which will take two inputs and give one output. This function will create a matrix whose elements are randomly assigned to 0 or 1. The only deterministic rule is that the outer layer must be filled with zeros.

Follow the steps below:

1. Using the function rand(), create a random matrix whose size is determined by the two inputs of the function (you can also have one input, which is a vector with two elements). The random numbers will be between zero and one.
2. To turn this matrix into a matrix of just 0s and 1s, use logical indexing. Turn every element of the matrix that is bigger than 0.7 into 1, and all the others into 0. This will yield a matrix whose elements are ~70% 0s and the rest are 1s.
3. Turn the elements of the edges of the matrix into zeros. Do this without any For-loops. The resulting matrix should be the output of the function.

Computing the new pattern

At every time step, a new matrix needs to be calculated according to the rules of the Game of Life. Create a new function compute_grid() with one input and one output.

1. Create a new empty matrix (i.e. with zeros) of the same size as the input matrix. Call this new_matrix.

2. Using two for-loops, go through every element of the input matrix, and calculate the number of neighbors of this element that equal 1. To do this, you can use the sum() function. To make sure that the outer elements of the matrix stay as 0s, exclude them from the for-loops.
3. Based on the number obtained before, apply the Game of Life rule to determine whether this element should be 1 or 0; save the result to the corresponding element of the new_matrix. At the end of these for-loops, the new_matrix will be the evolution of the input matrix; use it as the output of the function.
4. (Optional) Create a wrap-around rule. To do this, see the instructions at the end of this file.

Plotting

Create a function plot_matrix() that takes two inputs and returns nothing. In this function, we will plot the matrices obtained throughout the exercise to create an animation of the Game of Life. The inputs should be the current matrix and the current iteration. Follow these instructions:

1. Using the function imagesc(), plot the given matrix. Make sure that the axes are square and that there are no labels on the axes.
2. Set the title of the plot with the function title(). The title should be 'Iteration: # s', where # is the second input of the function. You will need the function sprintf() for this.

Creating life

You will now use all the functions you have created. In your main.m script, set a variable with the number of iterations (try 100 for test purposes; when your code is ready, change it to something bigger). Create the initial matrix with init_matrix(), then do a for-loop for all the iterations, in which you plot and compute the new pattern.

(Optional) Find equilibrium points. See the Optional Tasks section for instructions.

Importing life

You will now modify your init_matrix() function to take an additional input, which will be a string containing the name of a .mat file. The idea is that you are able to run the Game of Life with any initial matrix that you provide; with these, you can see some known automatons in action. Follow these instructions:

1. Add an additional input to your function called file_in. This will be a string with the name of a mat-file that contains an initial state for the matrix.
2. Inside the function, check whether the file exists. If it does exist, use it as output to the function. If the file does not exist, then create the random matrix as was done originally in this function.
3. (Optional). Calculate the period of recurrent automatons. See the Optional Tasks section for instructions.

Optional tasks

Wrap-around rule

So far you have ignored the cells at the borders; they are all set to zero. While this is convenient, it is not the only way of handling the border. Another way is the wrap-around rule.

The idea of the wrap-around rule is that the whole matrix is cyclic. In this case, the left-most cell is neighbors with the right-most cell, and the top-most with the lower-most. This means that if something is moving towards the left and reaches the edge, it will reappear on the right edge and continue moving.

There will be no detailed instructions to do this. In general, what you need to do is modify the neighbor-checking routine to check the borders and make it so that if an index reaches 0, it is set to N (where N is the size of the matrix's dimension in question) and if it reaches N+1, set it to 1.

Finding equilibrium points

The idea here is simple: if the updated matrix is identical to the previous one, the world has reached an equilibrium. The simulation should be stopped then.

At every time-step, before calculating the new matrix, save the old one to another name. Then, after calculating the new one, compare the two of them (using isequal()). If they are the same, print a message stating this and stop the simulation (using break).

Finding the period of automatons

There are some known initial states that lead to the so-called patterns: "creatures" that live on through iterations, often moving and spawning different creatures. Download the FILE, which contains some known patterns. You can import them and see them in action.

Some of these patterns are static, in the sense that they do not evolve. You probably saw some during the testing of your code. Some others live for some time and die. There are also those who repeat themselves: these are called oscillators. In this section, you will find how long it takes for one of these to repeat itself. This is called its period.

1. In your main script, create a variable called keep_old. Set it to 4 for now.
2. In order to find the period of an oscillator , we need to compare its current state with all the past states (ideally); when a new state is equal to one of the old ones, the time (number of iterations) that passed between those two is the period. Saving all previous states would be too costly, so for now you will save keep_old states (4 for testing):
    1. Start by creating a three-dimensional array of zeros, whose size is (keep_old, Nx, Ny), where Nx and Ny are the dimensions of your initial matrix. Call this matrix old_mats. The idea is that old_mats(1,:,:) contains the t-1 state, old_mats(2,:,:) the t-2 state, and so on.
    2. At every time-step of the main loop, move the matrices in old_mats one step backwards. This means that old_mats(1,:,:) should go to old_mats(2,:,:), old_mats(2,:,:) to old_mats(3,:,:) and so on. Hint: do this in reverse order.

    3. Save the current matrix to old_mats(1,:,:).
3. Every time you calculate a new matrix in the main loop, compare it to the previously-observed matrices in old_mats (use isequal() for this). If you find a match, the first index of old_mats that matches represents the time-steps that separate the two of them: this is the oscillator's period. As a side effect, this would also find the equilibriums found in the previous optional task.
4. If you find the period, print it to screen (with a nice text) and stop the simulation (using the command break).