CS1114 Lab: Game of Life

1 Game of Life

Today we'll start by building upon an idea you used in Assignment 4 to count the number of neighbors of a pixel in a binary image, for which you used (in part) the convolution operator. It turns out that this simple idea of counting neighbors—along with some simple rules—can be used to build very complex, emergent behavior. We will look at this in the context of the **Game of Life**. The Game of Life (invented by John Conway in 1970) is extremely simple, consisting of a binary image that evolves over time. Consider a binary image B_0, of some size. We will think of each element of the image as a "cell," where a 1 indicates a live cell, and a 0 indicates a dead cell. The Game of Life takes B_0 and produces a new image B_1 according to simple rules. In the rules below, a *neighbor* to a cell is an 8-connected neighbor.

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

For instance, consider the following simple 5×5 binary image B_0 (here, somewhat confusingly, a '.' symbol indicates a 0 (dead cell), and a 'O' symbol indicates a 1 (live cell)):

```
.....
..O..
..O..
..O..
.....
```

(This corresponds to the following matrix:

```
B0 = [ 0 0 0 0 0 ;
       0 0 1 0 0 ;
       0 0 1 0 0 ;
       0 0 1 0 0 ;
       0 0 0 0 0 ]
```

.)

After applying the rules of the Game of Life (on all cells at the same time), we get a new binary image B_1:

```
.....
.....
.OOO.
.....
.....
```

If we repeat the same rules again on B_1, then we get another image B_2:

```
.....
..O..
..O..
..O..
.....
```

In this case, $B_2 = B_0$. We could repeat this infinitely, running this evolution through more and more time steps. For this particular example, the evolution simply repeats between the two different states. (This is known as a "blinker".)

Here's another example—in this case, B_0 (which we will call the *seed*, or starting configuration), is a 5×9 image:

```
.........
..O......
....O....
.OO..OOO.
.........
```

After applying the rules, we get a new binary image B_1:

```
.........
.........
.OOO.OO..
.....OO..
......O..
```

We could then repeat and see what happens to this after some number of rounds.

Your first task is to write a Game of Life simulator. In the

`~cs1114/student_files/sections/gameoflife/`

directory you will find a stub file called `gameoflife.m`. The `gameoflife` function takes in two arguments, a binary image (`seed`) and an integer (`rounds`). This function should do the following: starting with the given seed, run the game of life for the given number of rounds, displaying the current state of the game at each point. You are to use convolution (recall the `conv2` function), with a particular kernel that you will come up with, in your solution—and remember to use the 'same' argument. Can the rules of the Game of Life be implemented using only convolution? That is, is there some convolution kernel that will take a binary image, and produce a new binary image representing the next state?

If not, you might need more operations. Some possible operations you can use are the '`&`' (logical and) operator from Assignment 4, as well as the '`|`' operator (logical or), and the '`~`' operator (logical not). As you might expect, the `|` operator takes two operands, and returns 1 if either is non-zero, and 0 if both are 0, which the `~` operator takes a single operand, and "negates" it (in a logical sense)—it returns 0 if the operand is non-zero, and 1 otherwise. For instance:

```
>> 1 | 1

  ans =
        1

>> 1 | 0

  ans =
        1

>> 0 | 0

  ans =
        0

>> ~5

  ans =
        0

>> ~0

  ans =
        1
```

You should try and write the code that takes the current state of the game and produces the next state in as few lines of code as possible (the instructor was able to do it in three lines of code—of course, the entire `gameoflife` function was a bit longer). See if you can improve on this.

After you compute each state, you should display it. You can either use the `imshow` command for this, or the `spy` command, which visualizes the non-zero elements of a matrix. Note that you might need to pause a bit between rounds of the simulation, using the `pause` function. You can also use the `figure` command at the beginning of your function to make sure a window actually displays the results of the simulation.

$\Longrightarrow$Implement `gameoflife`.

You will want to test your routine with some simple seeds, such as the blinker above (note that for small seeds, you may prefer to use `spy` to show the results, as it will display in a bigger window).

1.1 Random seeds.

Interesting things happen when you start the Game of Life with a random seed, with different "densities" (i.e., different initial ratios of live to dead cells). To play with this, you will write a function called `random_seed` (whose stub is provided for you in the above directory), which takes three parameters: the number of rows and columns in the initial seed, and the probability p that each cell is alive to begin with. Using the `rand` function (and other tricks), you must compute and return a binary image of the given size where each element is randomly on or off depending on the given probability.

$\Longrightarrow$Implement `random_seed`.

Now try running `gameoflife` with different random seeds of different densities. You'll find that if the density is too high or too low, almost all of the cells will die off immediately. But if the density is somewhere in between, interesting behaviors start to emerge. For instance, you might try densities of 0.01, 0.1, 0.5, and 0.9. A good seed size might be 100×100 or 500×500, and you'll want to run the simulation for at least 200 iterations. Note that after a while some parts of the space will converge to patterns which are static or oscillate ever two time steps, which other parts of the space may never seem to converge.

$\Longrightarrow$Try `gameofline` with different random seeds.

There are a number of cool starting seeds that exhibit interesting behavior. A few are included in the directory above. For instance, the `gosper` function will return a certain binary image. If you feed this into your `gameoflife` function, you will see an interesting, periodic behavior, in which an infinite number of small *spaceships* (technical term) are spawned, fly off, and crash into the side of the image. Also try the `puffer` image, to see a rather large spaceship-like object fly along, spawning off random garbage.

Many other interesting patterns are described here:
`http://www.bitstorm.org/gameoflife/lexicon/`.