CS1114 Section 9: The Game of Life
March 27th, 2013

1 The Game of Life

Today we'll start by building upon an idea from 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. The Game of Life is what's called a "0-player" game, meaning that its behavior is entirely determined by the start state without intervention from any player. Although this may not sound like a very fun game to play, we'll see that a lot of really interesting things can happen based only on the choice of starting state.

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 indefinitely, 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, but it gets tedious to do it by hand. So let's write a program to do the work for us.

2 Implementing the Game of Life

Your first task is to write a Game of Life simulator. You'll write a function `evolve` that takes a binary image and returns the image corresponding to the next generation. Begin by copying the files from `/courses/cs1114/section/life` into your home directory. You'll find stub file `evolve.m` as well as some other helpful code.

`evolve.m` looks like this:

```
function [next_gen] = evolve(B)
% evolve binary image B to the next generation of Conway's Game of Life
% next_gen is the binary image evolved in one generation from B

end
```

Implement this according to the rules of the Game given above. The first step is to use convolution (the `conv2` function) to count neighbors. A couple technical notes on the use of convolution:

- To get back a matrix that's the same size as the input, give `'same'` as the third argument.
- Performing convolution on an image doesn't work if the data type of the image isn't some format that supports decimal math operations. As such, you'll need to convert the input binary image to a `double` when passing it to the convolution function.

For example, if your neighbor-counting kernel is stored in a variable `k`, you would call

```
neighbor_counts = conv2(double(B), k, 'same');
```

Then, you need to create a new binary image based on the neighbor counts that implements the rules of the Game of Life.

To decide which cells are alive and which cells are dead in the next generation, you should consider making use of some of the following element-wise logical operators:

- AND: `&` (ampersand; shift + 7)
- OR: `|` ("pipe" character; shift + the key above Enter)
- NOT: `~` ("tilde" character; shift + the key to the left of 1)

AND and OR are binary operators, while NOT is a unary operator. They behave as you would expect them to, given their names:

```
  AND:                OR:
0 & 0 => 0        0 | 0 => 0
1 & 0 => 0        1 | 0 => 1
0 & 1 => 0        0 | 1 => 1
1 & 1 => 1        1 | 1 => 1

  NOT:
~0 => 1
~1 => 0
```

These operators also work elementwise on matrices. Recall also that we can use comparison operators (`<, <=, ==, >=, >`) in an elementwise fashion.

Think carefully about the rules of the game and how you can use these operators to implement the rules. Try to boil the rules down to their simplest form and write the code as concisely possible. The body of my `evolve` function (not counting header and comments) is two lines, each less than 80 characters. Can you match this?

$\Longrightarrow$Implement `evolve`.

To test your implementation, use the `gameoflife` function. This function uses `evolve` to animate the evolution of a given seed for a given number of generations. Its function header is:

```
function gameoflife( seed, generations, pause_seconds )
% gameoflife - beginning with a seed, animate a specified number of
% generations of the game of life
%   seed - binary image of any size
%   generations - number of generations to animate before stopping
%   pause_seconds - pause time between generations (animation speed)
```

Run this function on some simple inputs like the blinker from above to make sure your `evolve` is working.

2.1 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. Once again, it is possible to write this very concisely—my code is a one-liner. Resist the urge to use a `for` loop.

$\Longrightarrow$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 running `gameoflife` with different random seeds.

People have come up with starting states that exhibit particularly cool behavior. In fact, there's an entire lexicon of names for patterns that create certain behavior—spaceships, oscillators, guns, etc. We've provided you with a few interesting seeds to try out; each of the following is a function that returns a seed, so you can simply use a call to one of these functions as your seed:

```
pulsar
quasar
glider
puffer
glider_gun
```

Many other interesting patterns are described here:
`http://www.bitstorm.org/gameoflife/lexicon/`
and here:
`http://www.conwaylife.com/wiki/Main_Page`

3 If you have time...

$\Longrightarrow$Modify your `evolve` method in one or more of the following ways:

- Make the world a torus. That is, make it so the neighbor relationship wraps from one edge of the grid around to the opposite edge.
- Make the world a Möbius strip. The top and bottom should stay as they are, and the left and right edges should behave as if the world gets twisted a half-turn before joining the opposite edge.
- The original game is defined to exist on an infinite grid. If, for example, a glider hits the edge, it should disappear as if it continued traveling. This isn't what happens in our implementation. Fix ours so that it behaves like an infinite grid (you'll have to do it without storing an infinite grid, though). You may find it useful to modify `gameoflife`.