Matt Lund
19 March 2022

Cellular Automata Project

Cellular automata is a useful computational method of analyzing complex subatomic phenomena. Though it has uses in many other fields, it has physics applications from statistical to condensed matter physics, as well as indirect applications in quantum mechanics through the diffusion equation. The premise of cellular automata is simple: you have a grid or row of cells in which each cell contains a state such as on or off, and you apply a set of rules that alter these states over given periods of time. After taking an interest in a 1-dimensional game of life simulation I learned about through a previous lecture, I wanted to attempt to create a 2-dimensional version. Thus as a personal project, I decided to simulate the Game of Life as an animation.

The Game of Life takes place on a two-dimensional plane using a simple set of cell states in which each cell is either dead or alive. As a simple example, you can start with a square grid of randomly assigned ones and zeros as an initial state, although there are many fun patterns to start with. The grid you start with will then be updated in increments of time using a specified set of rules to determine the cell states after such time. The rules of Conway's Game of Life are as follows:

1. If a cell is alive, and has fewer than two neighbors that are alive, it dies off
2. If a cell is alive, and has either two or three neighbors that are alive, it stays alive.
3. If a cell is alive and has more than three neighbors that are alive, it dies off to "overpopulation".
4. If a cell is dead and has exactly three neighbors that are alive, it comes to life.

granted that if a cell does not satisfy any of the given rules, it is declared dead. When the whole process is animated, you can observe the cells switching states in real time, leading to some interesting patterns and formations.

In order to effectively code this simulation and animate it, I created a pseudocode in which I would create a function that initializes the matrix given dimensions. I would of course also need a function that updates the given matrix each time it was called by applying Conway's rules. Finally if I wanted to animate this, I would need a function that calls my update function at each time interval and displays the grid, although instead of creating a ton of grids, I wanted to condense it into one grid that was animated.

```
%pylab inline
import matplotlib.pyplot as plt
import matplotlib.animation as animation
set_printoptions(threshold=sys.maxsize)
```

```
Populating the interactive namespace from numpy and matplotlib
```

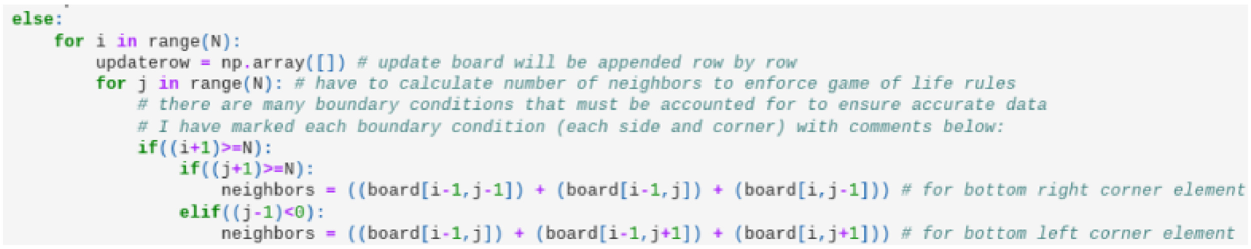
Here is the header for my code, where I imported both the numpy and matplotlib libraries, and also set print options to avoid truncation of my arrays.

```
N = 64 # size of game board

np.random.seed(123456789) # establish seed for consistent results

def random(N): # creates random matrix of size N with equal probability of cells being 1(alive) or 0(dead)
    board = np.random.choice([0,1],size=(N,N),p=[.5,.5])
    return board
```

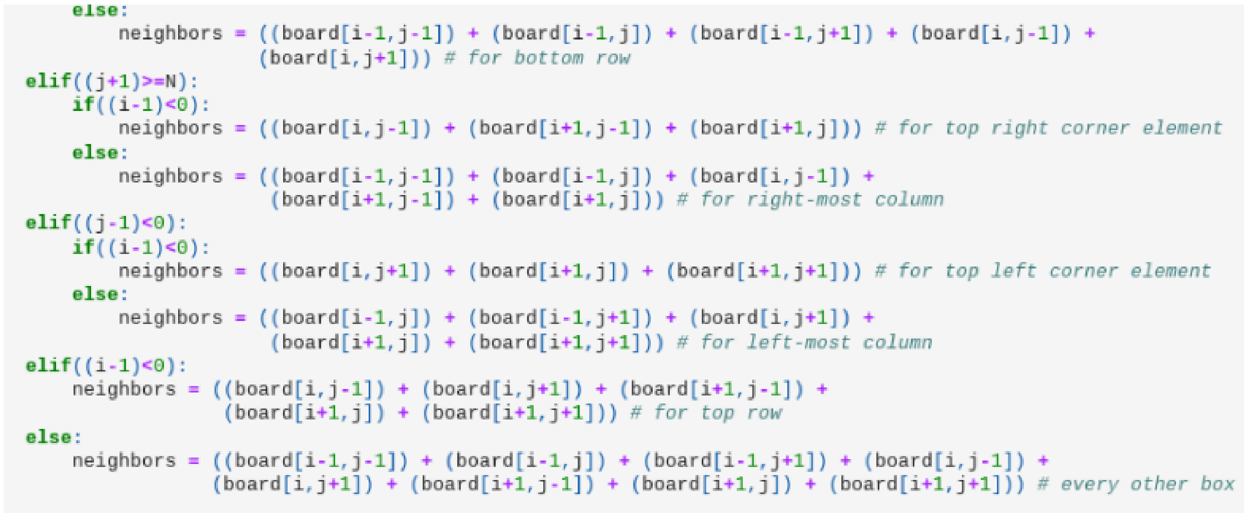
This part of the code initializes the size of the square grid being used, and sets a seed for the simulation to ensure consistent results for troubleshooting. I also defined the function I will use to initialize a square array of size N using numpy random generation. With that function set up, I began writing the function that would update a given grid and return it as an illustrated frame for my animation function.

```
def update(frame): # function that updates game of life board when called
    global board,img_plot # defines board and image frame as globabl variable for coding simplicity
    updateboard = np.array([]) # creates new board used to update existing board
    if(frame < 5): # pauses board for roughly half a second before animation starts (helpful visually)
        updateboard = board
```

In order to fix issues I encountered with referencing a single array through multiple functions, I decided to declare the main board as a global variable. Furthermore, this function revolves around declaring a new array that serves as an update to the array called, hence the declaration of ‘updateboard’. The next order of business is to look at each individual entry within the square grid and determine its updated state using the rules. Of course, this comes with its boundary conditions:

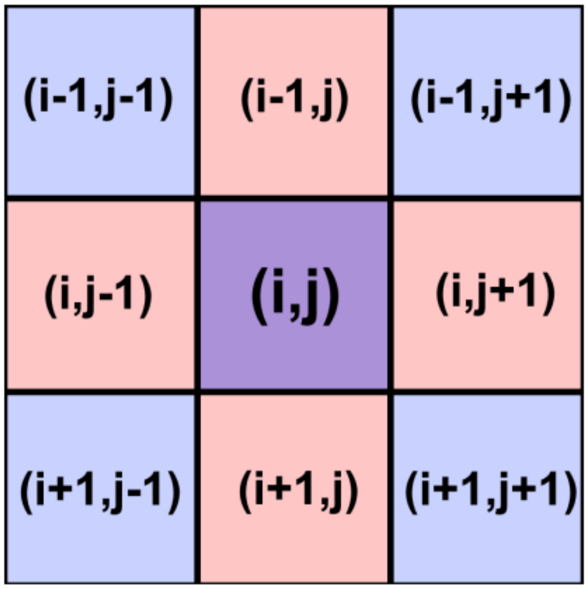


```
else:
    for i in range(N):
        updaterow = np.array([]) # update board will be appended row by row
        for j in range(N): # have to calculate number of neighbors to enforce game of life rules
            # there are many boundary conditions that must be accounted for to ensure accurate data
            # I have marked each boundary condition (each side and corner) with comments below:
            if((i+1)>=N):
                if((j+1)>=N):
                    neighbors = ((board[i-1,j-1]) + (board[i-1,j]) + (board[i,j-1])) # for bottom right corner element
                elif((j-1)<0):
                    neighbors = ((board[i-1,j]) + (board[i-1,j+1]) + (board[i,j+1])) # for bottom left corner element
```

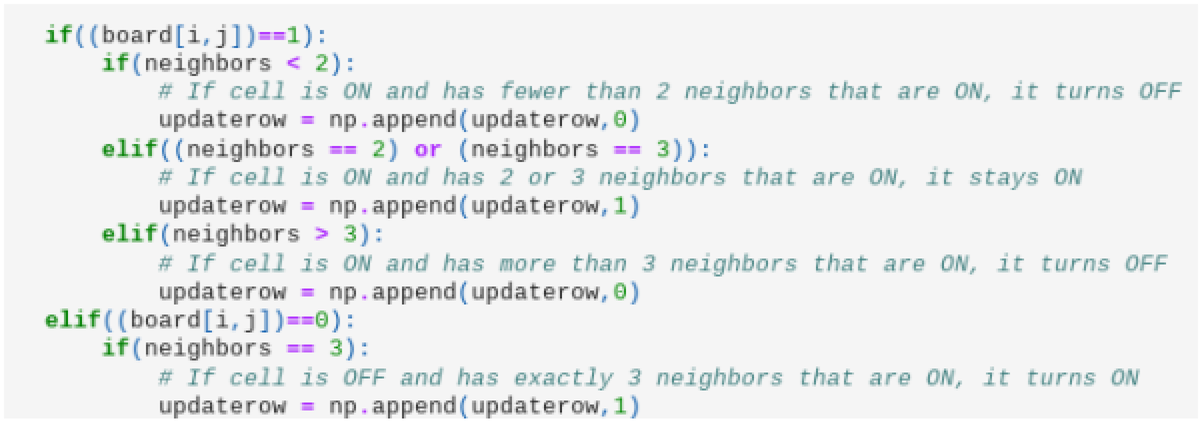


```
    else:
        neighbors = ((board[i-1,j-1]) + (board[i-1,j]) + (board[i-1,j+1]) + (board[i,j-1]) +
                    (board[i,j+1])) # for bottom row
elif((j+1)>=N):
    if((i-1)<0):
        neighbors = ((board[i,j-1]) + (board[i+1,j-1]) + (board[i+1,j])) # for top right corner element
    else:
        neighbors = ((board[i-1,j-1]) + (board[i-1,j]) + (board[i,j-1]) +
                     (board[i+1,j-1]) + (board[i+1,j])) # for right-most column
elif((j-1)<0):
    if((i-1)<0):
        neighbors = ((board[i,j+1]) + (board[i+1,j]) + (board[i+1,j+1])) # for top left corner element
    else:
        neighbors = ((board[i-1,j]) + (board[i-1,j+1]) + (board[i,j+1]) +
                     (board[i+1,j]) + (board[i+1,j+1])) # for left-most column
elif((i-1)<0):
    neighbors = ((board[i,j-1]) + (board[i,j+1]) + (board[i+1,j-1]) +
                 (board[i+1,j]) + (board[i+1,j+1])) # for top row
else:
    neighbors = ((board[i-1,j-1]) + (board[i-1,j]) + (board[i-1,j+1]) + (board[i,j-1]) +
                (board[i,j+1]) + (board[i+1,j-1]) + (board[i+1,j]) + (board[i+1,j+1])) # every other box
```

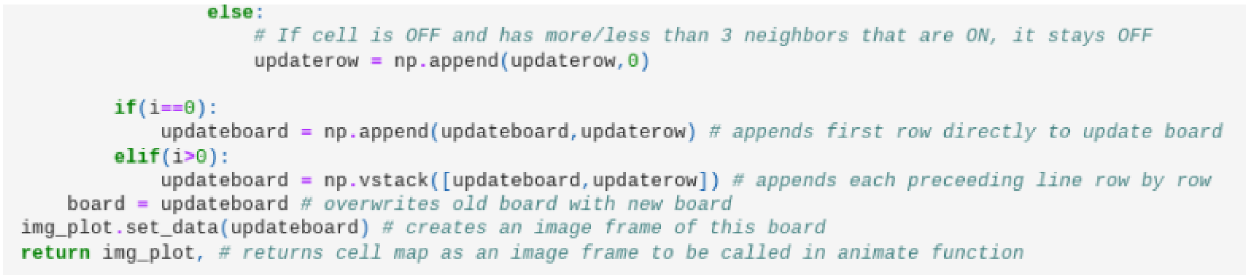
In this case, I have commented each boundary condition, one for each side of the box, and one for each corner. The problem with just adding up all 8 neighbors for each box, is that if there is no physical box to the side of a given cell, python decides to essentially surround the array in duplicates of itself and take the value from the other side of the array instead of defaulting to zero.



In other words, if looking at this diagram I've made, if (i,j) corresponds to the top right corner of the array for example, then (i-1,j-1) shouldn't exist, but python tiles the array diagonally up and left, and (i-1,j-1) becomes (i+1,j+1) instead of zero. This logic applies similarly in this case to any tile above or to the left of (i,j). With this part of the code working properly, I can then compare each cell to the rules to determine its final state.

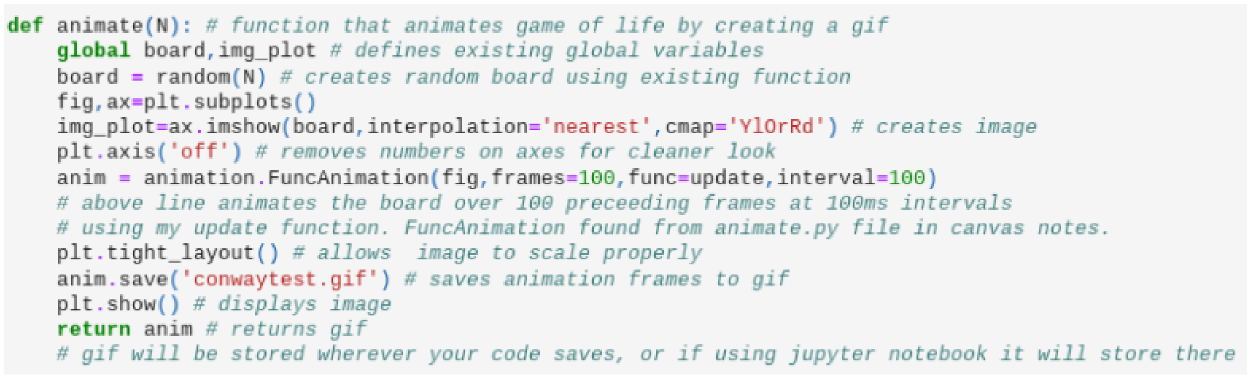


```
            if((board[i,j])==1):
                if(neighbors < 2):
                    # If cell is ON and has fewer than 2 neighbors that are ON, it turns OFF
                    updaterow = np.append(updaterow,0)
                elif((neighbors == 2) or (neighbors == 3)):
                    # If cell is ON and has 2 or 3 neighbors that are ON, it stays ON
                    updaterow = np.append(updaterow,1)
                elif(neighbors > 3):
                    # If cell is ON and has more than 3 neighbors that are ON, it turns OFF
                    updaterow = np.append(updaterow,0)
            elif((board[i,j])==0):
                if(neighbors == 3):
                    # If cell is OFF and has exactly 3 neighbors that are ON, it turns ON
                    updaterow = np.append(updaterow,1)
```



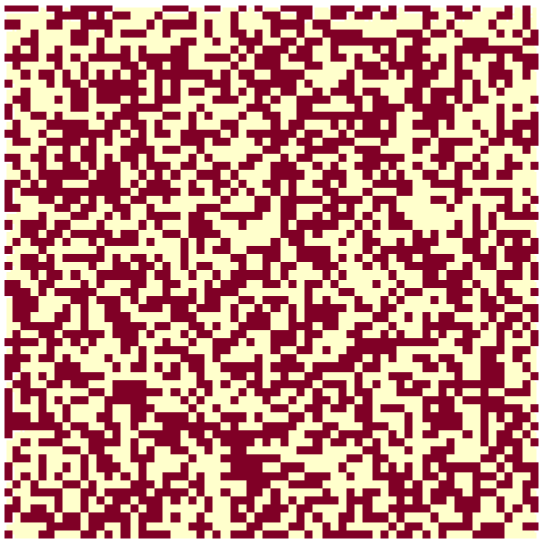
```
                else:
                    # If cell is OFF and has more/less than 3 neighbors that are ON, it stays OFF
                    updaterow = np.append(updaterow,0)

        if(i==0):
            updateboard = np.append(updateboard,updaterow) # appends first row directly to update board
        elif(i>0):
            updateboard = np.vstack([updateboard,updaterow]) # appends each preceeding line row by row
    board = updateboard # overwrites old board with new board
img_plot.set_data(updateboard) # creates an image frame of this board
return img_plot, # returns cell map as an image frame to be called in animate function
```

Here, I use the same rules I outlined above my code and build the updated array row by row, then overwrite it to the original array. Finally, this function returns an image plot of the new array, or a frame in time that is used in the animate function below:



```python
def animate(N): # function that animates game of life by creating a gif
    global board,img_plot # defines existing global variables
    board = random(N) # creates random board using existing function
    fig,ax=plt.subplots()
    img_plot=ax.imshow(board,interpolation='nearest',cmap='YlOrRd') # creates image
    plt.axis('off') # removes numbers on axes for cleaner look
    anim = animation.FuncAnimation(fig,frames=100,func=update,interval=100)
    # above line animates the board over 100 preceeding frames at 100ms intervals
    # using my update function. FuncAnimation found from animate.py file in canvas notes.
    plt.tight_layout() # allows  image to scale properly
    anim.save('conwaytest.gif') # saves animation frames to gif
    plt.show() # displays image
    return anim # returns gif
    # gif will be stored wherever your code saves, or if using jupyter notebook it will store there
```

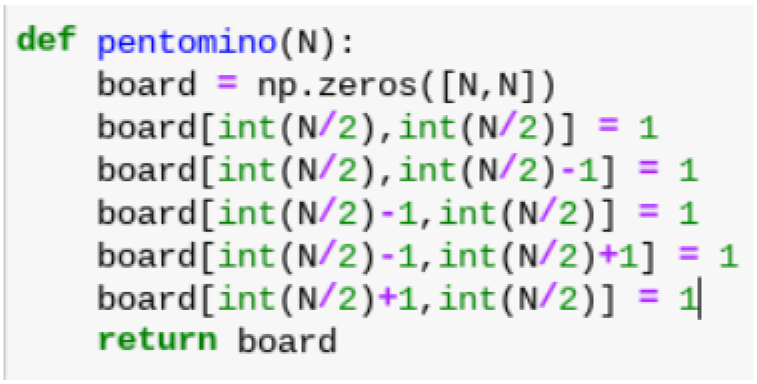
This animate function makes use of both previous functions, by initializing a board from the ‘random’ function, and applying the update function many times through the ‘FuncAnimation’ command, which actually creates the animation. Other lines of code are included to make the output as clean and visually appealing as possible. Instead of displaying the gif as output, the program actually saves the gif to your computer/notebook. Here is what a call of animate(N) displays using the entirety of the code covered so far (With N as 64 and 100 total frames):



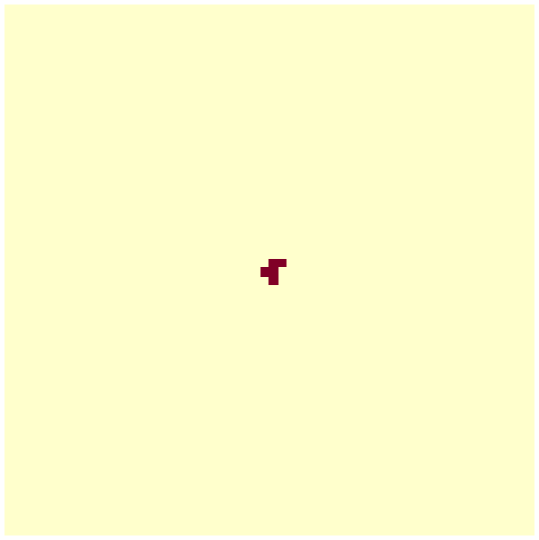
(Figure 1: conwaytest.gif)

The gif displays as it should, and as you may have noticed at the beginning of my ‘update’ function, I added a couple lines of code to freeze the first 5 frames of the animation so it is possible to see the starting arrangement.

As seen in the output, after long periods of time, patterns start to emerge in the form of 2x2 squares, oddly shaped circles, and rotating 3x1 lines that don't seem to change unless acted on by an outside cluster. Next, I want to see the long-term effects of different, less random starting arrangements. Here is what appears to be a simple starting arrangement of 5 'alive' cells, that actually evolves over time:

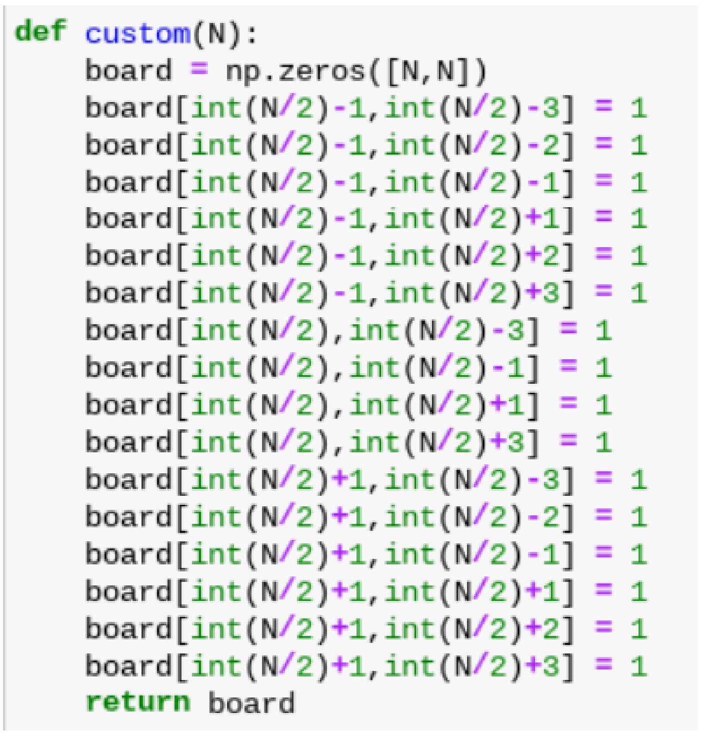


```python
def pentomino(N):
    board = np.zeros([N,N])
    board[int(N/2),int(N/2)] = 1
    board[int(N/2),int(N/2)-1] = 1
    board[int(N/2)-1,int(N/2)] = 1
    board[int(N/2)-1,int(N/2)+1] = 1
    board[int(N/2)+1,int(N/2)] = 1
    return board
```

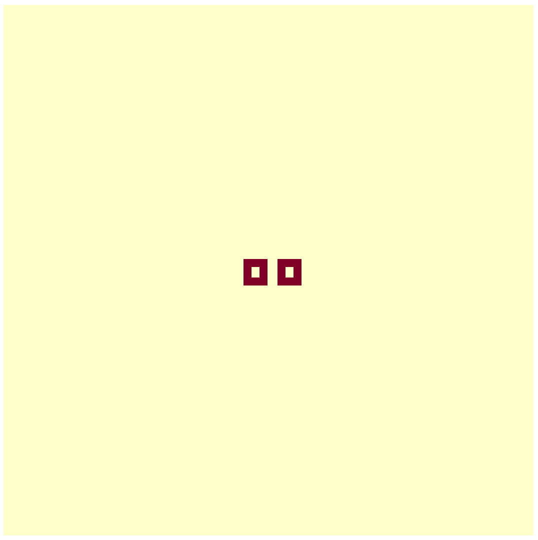


(Figure 2: pentomino.gif)

This is actually a famous starting arrangement nicknamed the *R-Pentomino* and though it is not finite, it does leave quite an impression on the grid for 5 simple tiles. Lastly, I wanted to try making a pattern on my own and testing it. I was aiming for some sort of aesthetically pleasing symmetrical pattern, trying different size circles and squares, and ended up coming across a cool looking animation that results from putting 2 squares 1 cell away:



```python
def custom(N):
    board = np.zeros([N,N])
    board[int(N/2)-1,int(N/2)-3] = 1
    board[int(N/2)-1,int(N/2)-2] = 1
    board[int(N/2)-1,int(N/2)-1] = 1
    board[int(N/2)-1,int(N/2)+1] = 1
    board[int(N/2)-1,int(N/2)+2] = 1
    board[int(N/2)-1,int(N/2)+3] = 1
    board[int(N/2),int(N/2)-3] = 1
    board[int(N/2),int(N/2)-1] = 1
    board[int(N/2),int(N/2)+1] = 1
    board[int(N/2),int(N/2)+3] = 1
    board[int(N/2)+1,int(N/2)-3] = 1
    board[int(N/2)+1,int(N/2)-2] = 1
    board[int(N/2)+1,int(N/2)-1] = 1
    board[int(N/2)+1,int(N/2)+1] = 1
    board[int(N/2)+1,int(N/2)+2] = 1
    board[int(N/2)+1,int(N/2)+3] = 1
    return board
```



(Figure 3: explosion.gif)

The animation from such a starting pattern almost resembles that of a symmetrical explosion that disappears at a certain time.

Some methods I used to test my code along the way were by testing function by function. I used a set seed to test the random array generator, and altered my update function to return just the updated array so that I could call it once and cross-reference values to make sure the bulk of the function was working properly. After making those work properly, the biggest challenge was animating the simulation. I’m not the most savvy coder so a lot of it came from looking at the animate.py file and using similar code until I could get tangible gif output. With some form of output, I was able to tweak the animate function until it produced a nice looking animation that I was happy with.

All in all, this cellular automata project was a fun topic to learn about. Although I ran into coding challenges along the way (mostly from trying to animate the simulation), the output was interesting to look at and play around with. Conway’s Game of Life is just one example of the many applications of cellular automata that exist, and although it isn’t used as a direct real-life application, there are still other forms of cellular automata that help us make more sense of the physical world around us.