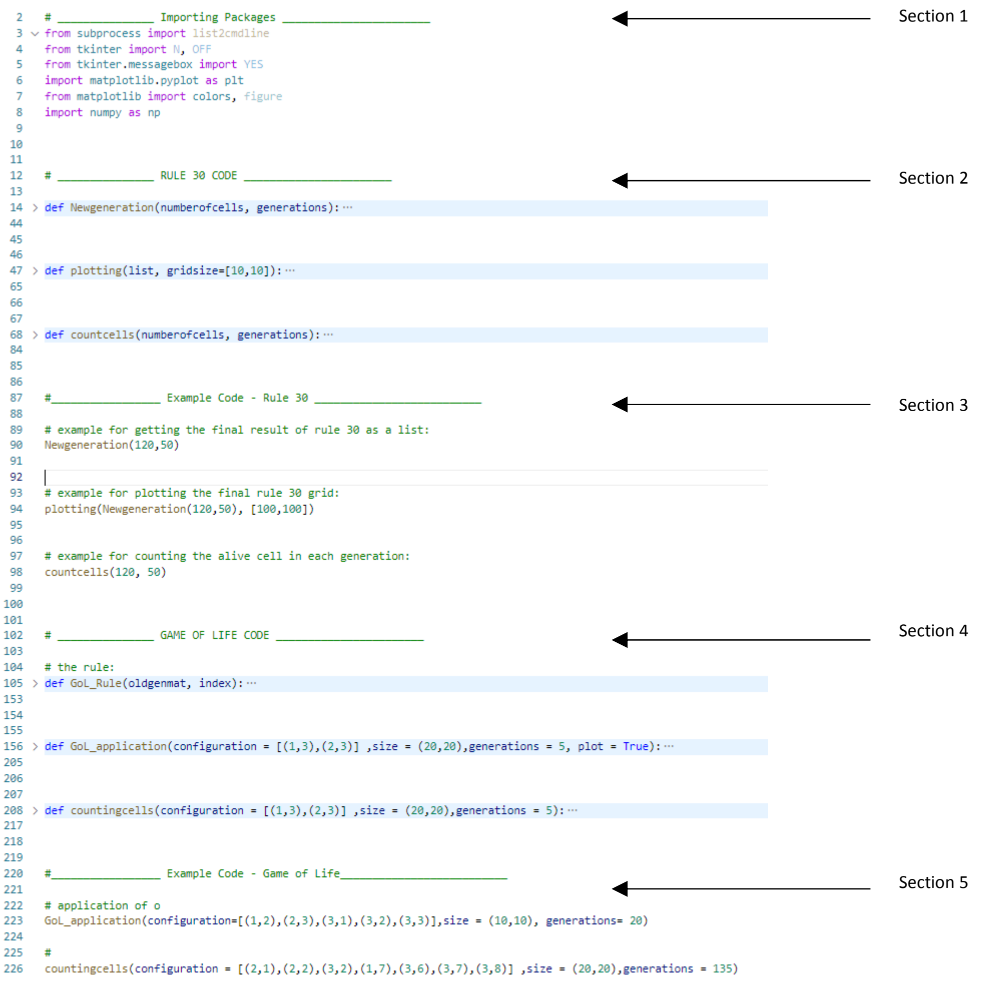
Students:
Mustafa Ramada – Bhavish Sirvi

Computational Many-Body Physics – Sheet 1

To solve this sheet we have wrote the attached code that is divided into five main sections (as we can see in figure 1). The first one is only there to import the important packages that we need. Section 2&3 are there to solve Rule 30. And finally, section 4&5 are there to solve the game of life.



```
# ______________ Importing Packages ______________________
from subprocess import list2cmdline
from tkinter import N, OFF
from tkinter.messagebox import YES
import matplotlib.pyplot as plt
from matplotlib import colors, figure
import numpy as np



# ______________ RULE 30 CODE ______________________

def Newgeneration(numberofcells, generations):···



def plotting(list, gridsize=[10,10]):···



def countcells(numberofcells, generations):···



#________________ Example Code - Rule 30 _________________________

# example for getting the final result of rule 30 as a list:
Newgeneration(120,50)


# example for plotting the final rule 30 grid:
plotting(Newgeneration(120,50), [100,100])


# example for counting the alive cell in each generation:
countcells(120, 50)



# ______________ GAME OF LIFE CODE ______________________

# the rule:
def GoL_Rule(oldgenmat, index):···



def GoL_application(configuration = [(1,3),(2,3)] ,size = (20,20),generations = 5, plot = True):···



def countingcells(configuration = [(1,3),(2,3)] ,size = (20,20),generations = 5):···



#________________ Example Code - Game of Life__________________________

# application of o
GoL_application(configuration=[(1,2),(2,3),(3,1),(3,2),(3,3)],size = (10,10), generations= 20)

#
countingcells(configuration = [(2,1),(2,2),(3,2),(1,7),(3,6),(3,7),(3,8)] ,size = (20,20),generations = 135)
```

Figure 1: Showing the different sections of our code

Exercise 1: Rule N:

a) Writing a code that produces rule 30 for 120 generations:

To do so, we have defined in section 2 of our code two functions: “New_generation()” and “plotting()”.
The first function will create a list of lists, each represent a generation. And we do so by the following steps:

- Define the rule 30 condition as a list.
- Creating the starting point (all zero except the middle cell)
- Create a loop that loop through all cells, and check if the condition is satisfied. And we wrap this rule with a loop for the amount of generations that we should create.
- And saving the end result in a list of lists called “endplot”.

And then the second function is just for representing the list of lists as grid with blue cells representing the alive cells and each row representing a new generation.

Running the code:

you can run the code by simply running the function "Plotting()" with two inputs: the first being our "new_generations(cell numbers, #generations)" function, which gives us the list of lists that we want. And the second input is the size of our figure. When we ran it for our specific example, we got the results that are shown

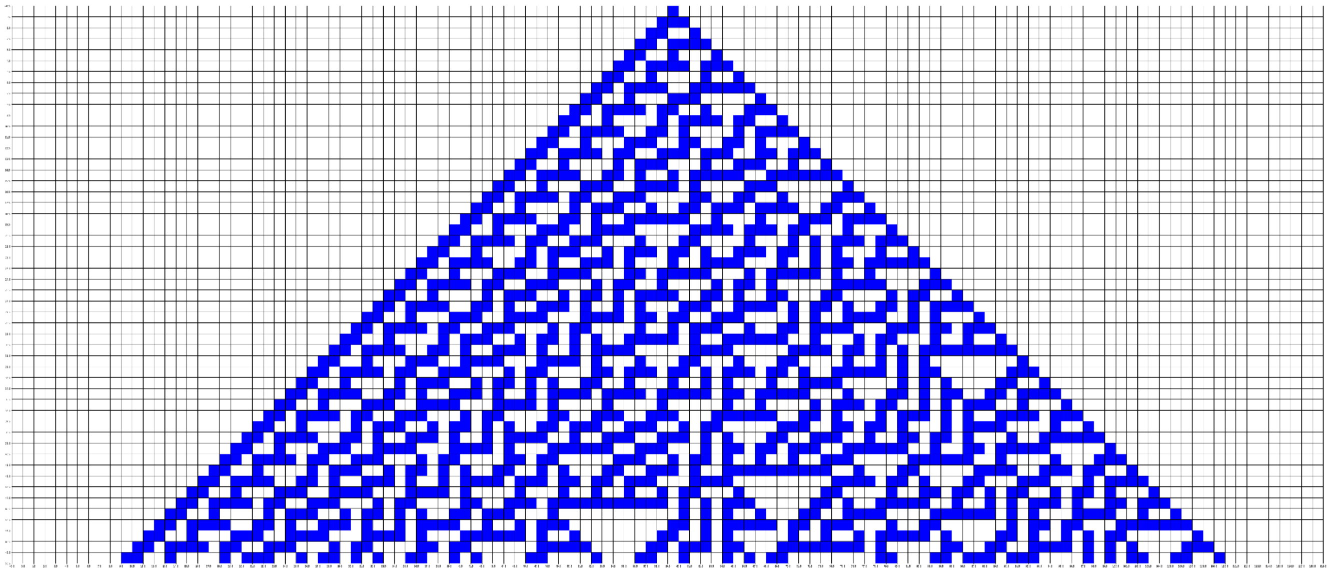


Figure 2: Showing Rule 30 applied for 50 generations.

b) Counting the alive cells in each generation:

To do this, we have created a function called "countcells()" which takes two inputs: number of cells and the number of generations. At the end of this function we will have a list of the alive cells at each generation. and to represent that, we thought it would be best to do a simple scatter plot of that list (figure 3).

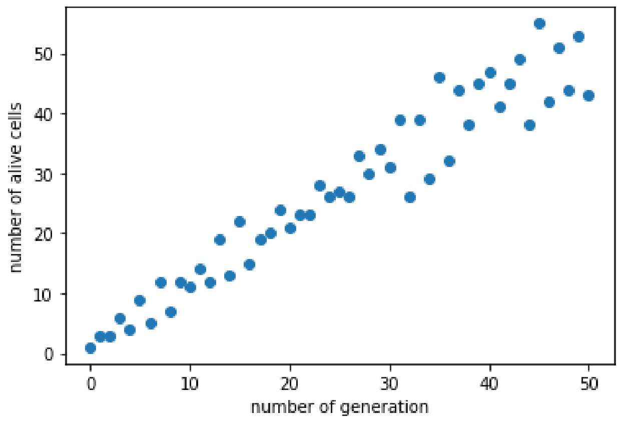


Figure 3: Showing the number of the alive cells in each generation.

c) The rule that reproduces the exact result for different configurations:

After discussing with our classmates, we didn't really understand the question. But if by configuration is meant the initial configuration where the state begins, then we believe it is the 256 rule, where all the cells will turn alive on the first generation.

Exercise 2: Game of life:

This exercise is solved using section 4&5 of our code.

a) **Creating the game of life code for different initial configurations:**
To do so, we have defined two functions: “GoL_Rule()” and “GoL_application()”. Let’s start with the former.
“GoL_Rule()”: This function returns a Boolean value depending if a certain cell will be dead or alive in the next generation and we did so in the following way:
- Create an array (under the name “effectmat”) that represents the effective cells that would affect a certain cells (its immediate 8 neighbors).
- To count for boundary conditions in this part, we were able to do a modular addition to the indices so that it would wrap around the matrix.
- Summing over this effectmat, and checking if the value follows the game of life rules. Depending, of course, on the state of the cell beforehand (if it was dead or alive).

Now, this function doesn’t do anything but check if the input cell is going to be dead or alive in the next generation.
To do the actual loop over all cells and over all generations, we have created the second function “GoL_application()”. We did so in the following manner:

- Creating a for loop that goes through all the cells and checking if “GoL_Rule()” will be True or False.
- After checking that for each cell. We have a created a list for the alive cells (aliveindices) , and a list for the dead cells (deadindices) .
- Each of those cells contain only the indices of the cell in question.
- After having this list. We create a new array out of these two lists and then plot it (using our old plotting function).
- And then we loop over all the required generations.

And as a result of this function, we were able to apply it to the following initial configuration and get the following results (row 1: initial configuration. Row 2: first generation):

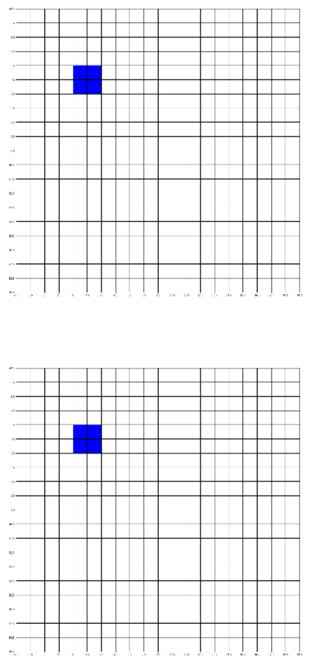


Figure 4: after running this for five generations nothing changed in the configuration

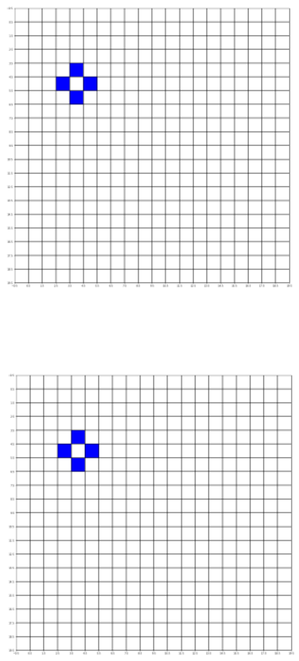


Figure 5: after running this for five generations nothing changed in the configuration

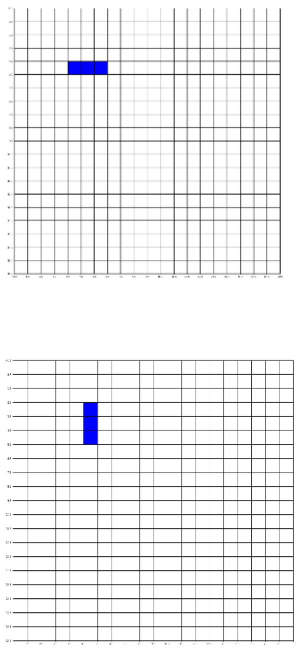


Figure 6: after running this for five generations, we noticed the periodic flipping between those two configs.

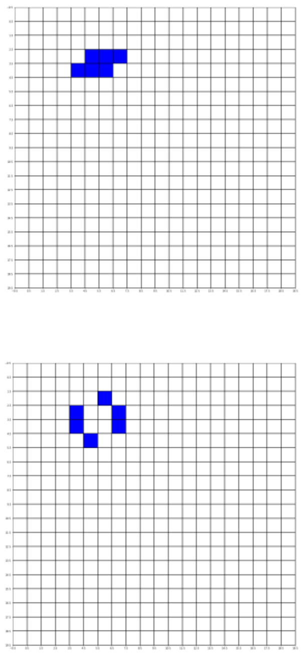


Figure 7: after running this for five generations, we noticed the periodic flipping between those two configs.

b) **The glider**:

For this question, we had everything ready. We just needed to run our function, and see what happens. As we can see from the picture, the initial shape is repeated every 4th generation with a sliding toward the right bottom corner.

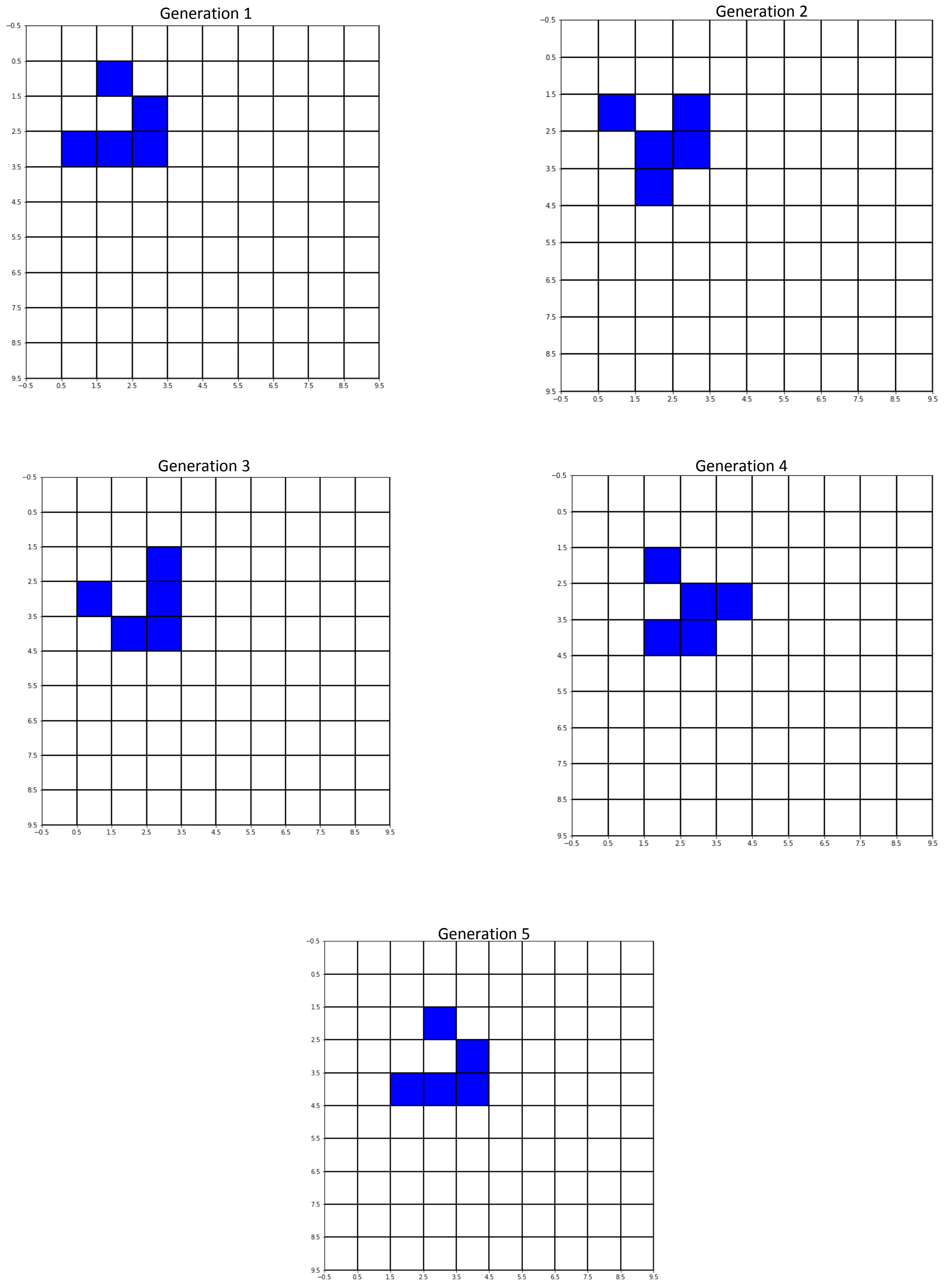


Figure 7: Showing the first 5 generations of the glider.

c) Calculating the number of alive cells in a “diehard”:

Just like part b in exercise 1, we will represent this using a scatter plot where we have used our game of life functions to check the alive cells at each generation. and we got the following result:

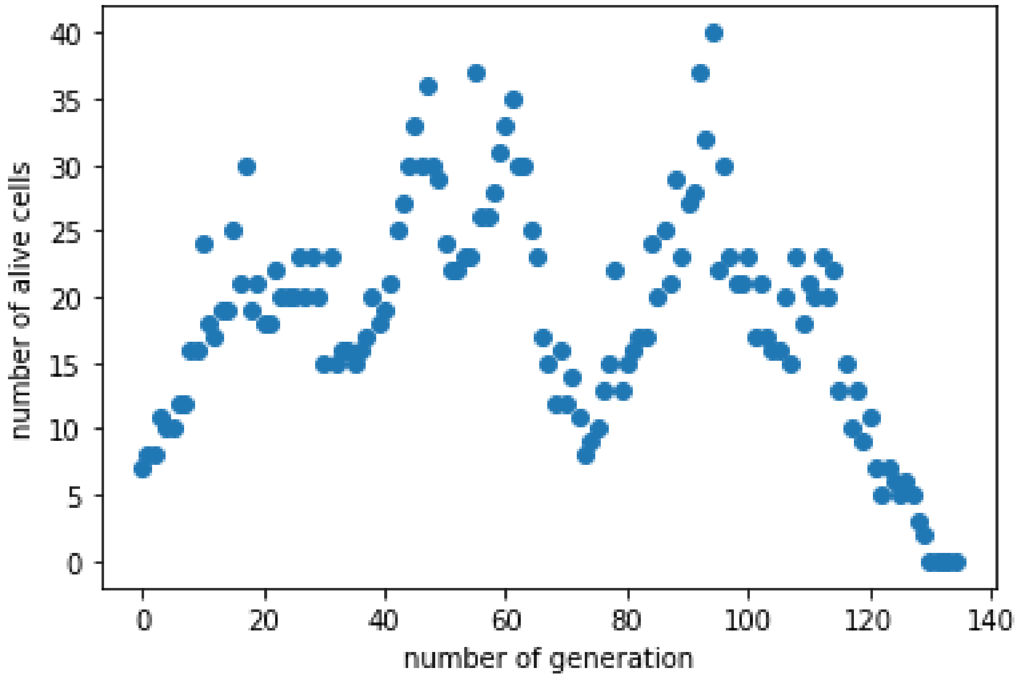


Figure 8: showing the number of alive cells at each generation for a diehard initial configuration.

As we can see, the number of alive cells goes to zero at the 130th generation.