SIMULATING ALLELES IN A POPULATION USING JOHN CONWAY’S GAME OF LIFE

A SEMESTER PROJECT   
PRESENTED TO

THE FACULTY OF CS 22B, SPRING 2024

BY

MAYA TALPAI-VASINTHASCHA AND SADAF SADRE ARAHAMI

MAY 2024

## ABSTRACT

John Conway was a British mathematician famous for founding “Conway’s Game of Life” which slingshot the field of cellular automata into popularity. Our project aims to simulate how a population of alleles can undergo natural selection, genetic drift, and inheritance in a metaheuristic and didactic way using John Conway’s Game of Life. There are many versions of Conway’s game of life that are inspired by biological systems, however we plan to bring a bit more complexity and realism to current models. We will push the envelope a bit by incorporating “fitness” into our model, modifying how “child” cells inherit traits from “parent”, and incorporating both beneficial and deleterious mutations into the model. Finally, we will log all the occurrences of alleles in a population over time in a data frame.

**TABLE OF CONTENTS**

[Abstract 2](#_fecq76a3q82n)

[Introduction 4](#_aqgqako9pvz7)

[Mechanisms of Evolution 4](#_qhcv3agjujfl)

[Fitness 5](#_hoz7begyey62)

[Conway's Game of Life Rules 6](#_p16c2111se87)

[Phase 1 Code: 8](#_8taj6jjmxe1i)

[Phase 2 Code: 13](#_6uvyh0qnrwwb)

[Phase 3 Code: 16](#_aco8r4bpe9bm)

[Phase 4 Code: 19](#_13movtbn5lej)

[Analysis of Code Output 20](#_og2uq685rqdy)

[Conclusions and Future Work 22](#_2cx0x0b8vmp3)

[References 23](#_jhhftkmol0d)

[Sample Outputs 24](#_31oxl4u1rc3l)

[Instructions to Run Code 25](#_v038u0xvc2i0)

## 

## INTRODUCTION

John Conway’s Game of Life is a famous cellular automaton which is commonly used to demonstrate complexity in systems that are self-organizing, dynamic, and can evolve over the course of “n” iterations or “generations”. Conway’s Game of Life can offer a foundation for modeling changes in a population’s alleles over time. In this application of the model each cell in a grid acts as a distinct organism and its alleles are represented by its state and color. Each successive generation following Conway's basic rules can model genetic drift, natural selection, inheritance, mutation and gene flow of allele frequencies across generations. This proves to be an exciting way to explore the complex processes involved in evolution and demonstrate the importance of biodiversity.

## Mechanisms of Evolution

There are numerous processes that alter the genetic makeup of a population over time. Mutations are the building blocks for evolutionary change by introducing genetic variation into a population. The exchange of genetic material between populations, also known as gene flow, promotes genetic diversity by hybridizing alleles from the different populations. Genetic drift is the occurrence of a random event, which results in variations of an allele’s frequency in a population. An example of this is a population of ants invading a home, and a homeowner indiscriminately killing the ants. By killing a portion of the population randomly, you sway the genetic diversity to be of just the ants that survived being squashed. The final mechanism of evolution is natural selection, which favors individuals that have a lot of “fitness.” Individuals that are “fit” have desirable traits that improve survival and reproduction. The complex interactions between these mechanisms shape the genetic landscape and drive biodiversity in all organisms. Our Conway Game of Life simulation is a visual representation of these mechanisms at work for a set of alleles in a relatively small population.

## Fitness

In evolutionary terms “fitness” is the capacity of an organism to endure and procreate in its surroundings and thus passing on its genetic material to succeeding generations. Fitness of an individual can be determined by physical characteristics (humans prefer to mate with attractive-looking people), behavioral adaptations (the bird that sings and dances the best will attract a mate) and other methods to ensure successful reproduction (the male elephant seal that can fight off another elephant seal for a harem or the “sneaker male” that can mate quickly without anyone noticing). In our simulation the concept of fitness is calculated quantitatively for each allele in the population based on “relative frequency” since frequency in our population is the main driver to whether an individual can successfully “mate” and produce “offspring”.

## Conway's Game of Life Rules

Conway's Game of Life starts out as a simple grid of cells that each have a binary state (“alive” or “dead”). Every Game of life’s starting grid must be deterministic such that there is an initial input that dictates the state of “alive” or “dead” for each cell in a grid. In following iterations or generations, the “state” of the cells can change and is determined by 4 simple rules:

1. Any live cell with 2 or less live neighbors dies, as if by underpopulation.
2. Any live cell with 2 or 3 live neighbors lives on to the next generation.
3. Any live cell with more than 3 live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly 3 neighbors becomes a live cell, as if by reproduction.

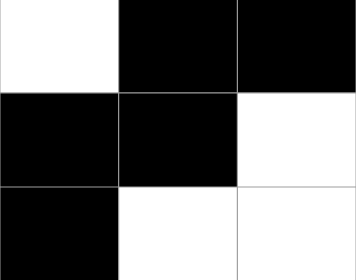
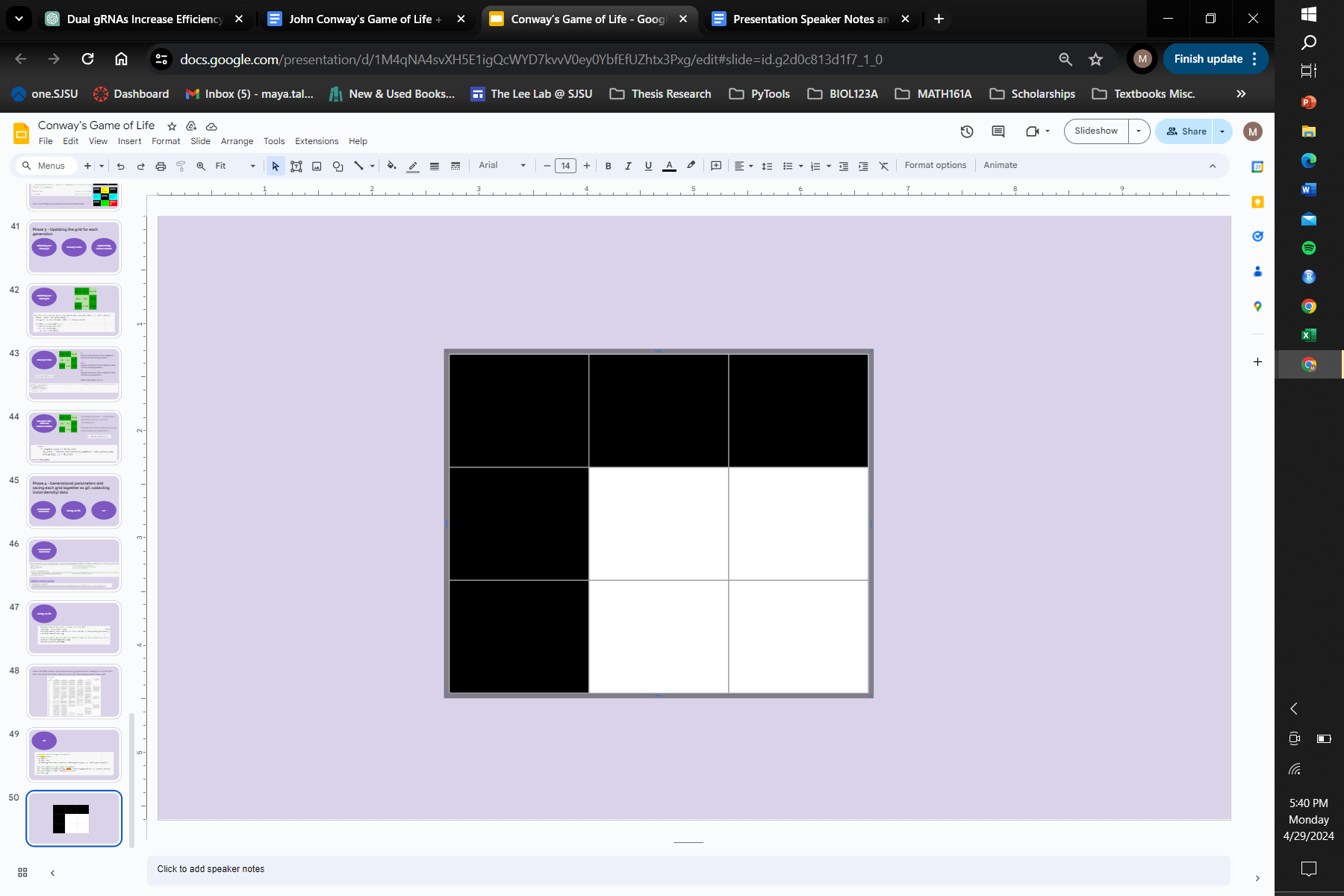
 

Fig 1. Example of a starting grid (left) where black cells are “dead” and white cells are “alive”. In the next “generation” (right) the state of these cells changes based on Conway's Game of Life Rules.

Usually cells that are either “alive” or “dead” are represented by “black” or “white” colors. However, in our simulation we add the variable of color to represent different alleles in a population. By adding colors into our simulation we can also demonstrate an organism's ability to inherit traits from parents by creating a hybrid of the two colors or alleles. Previous models determined the color of a “child” cell by the average color of the two parents. However, in our model a child’s color is determined by instead both parent cell's “fitness” represented by a weighted average of the two colors’ relative frequency. We also incorporated the aspect of a random chance mutation, where despite the parents contribution to the child color, it will generate a beneficial or harmful mutant color with an artificial positive or negative effect on fitness.

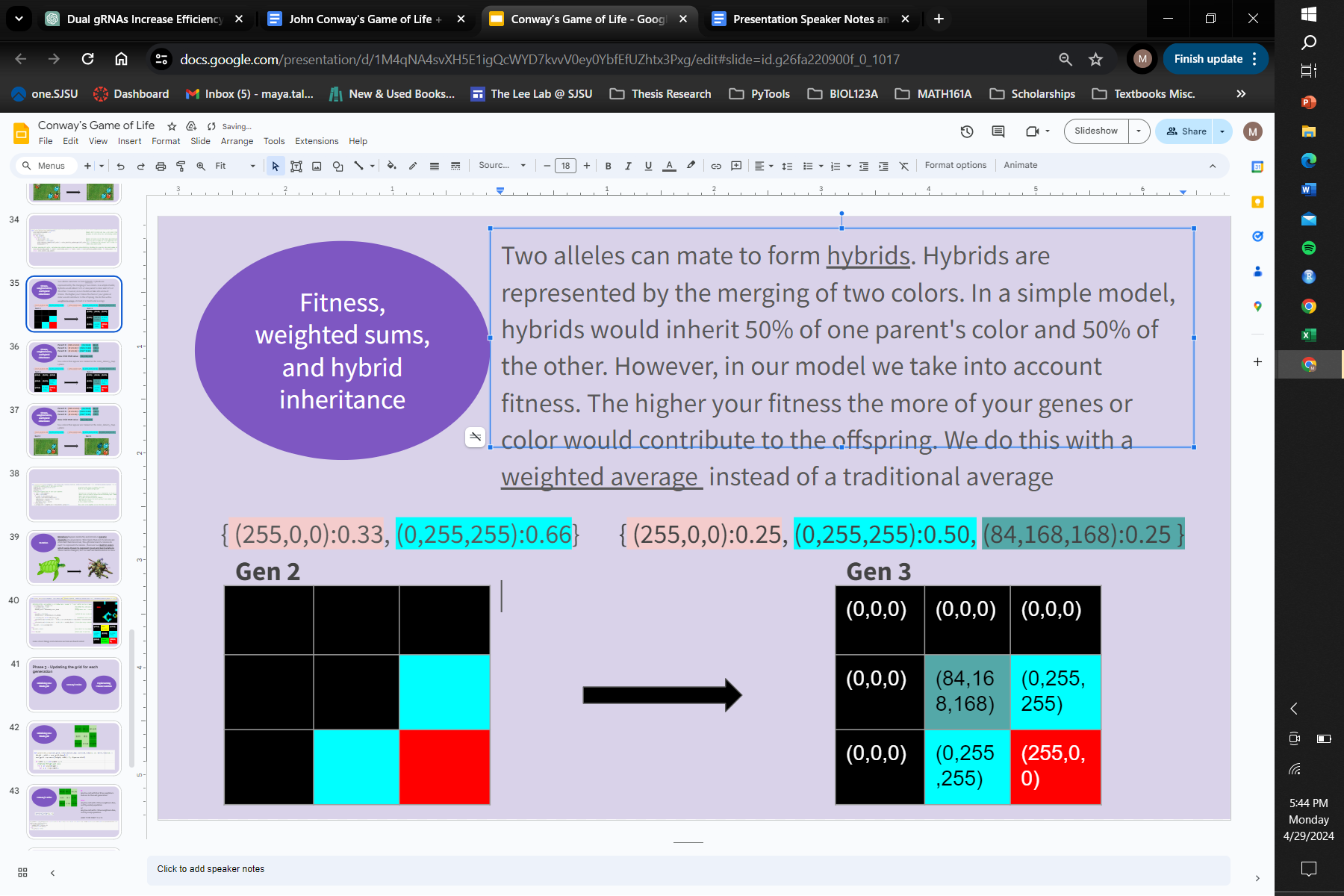
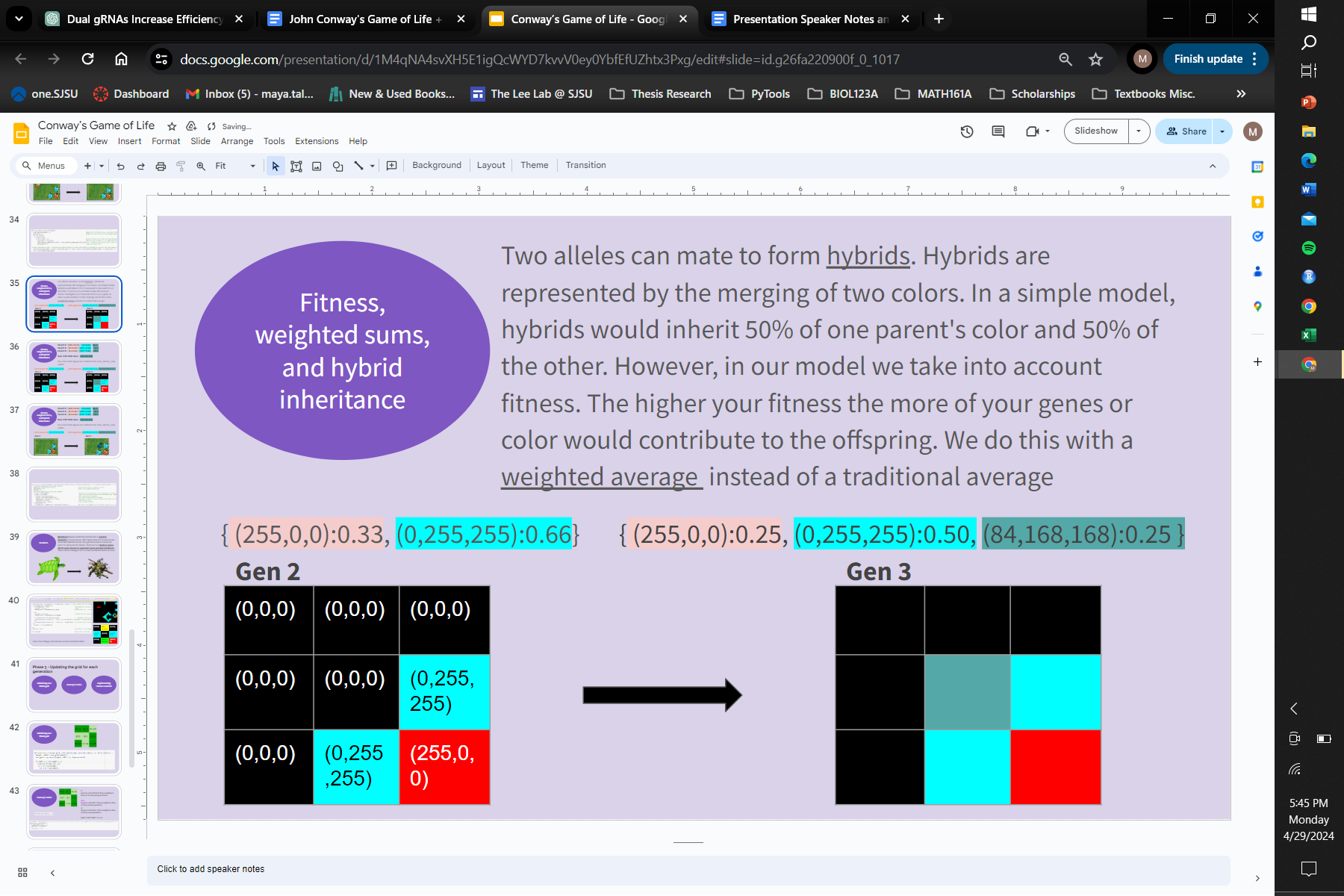
 

Fig 2. Example of a starting grid that incorporates colors to represent alleles that are alive, “red or blue” and dead cells, “black”. The next “generation” of the initial grid based on Conway's Game of Life Rules is also shown . Note the “teal” cell that was “born” representing more genetic information from the blue cell being passed down than the red cell.

To visualize genetic drift and gene flow of alleles in a population, each generation’s grid is saved as a GIF animation to easily view the changes in alleles over time. The colors and frequency of each allele is also tracked over each generation in a data frame so that further analysis can be conducted.

## **Phase 1 Code:**

There are a few built in libraries we import and utilize to perform tasks downstream of our code. We heavily rely on “numpy” for a lot of functionality within the code such as incorporating randomness, assigning cell states, checking cell states, managing numerical operations, and creating arrays. We also use “matplotlib.pyplot” and “matplotlib.animation” to visualize the grids as well as generating an animation or GIF of each grid over the course of the simulation. Finally we also use “pandas” as a way of storing color and relative frequency information of alleles in a dataframe over the course of the simulation.

Phase 1 of the code focuses on generating the initial deterministic state of the grid. The starting state of the grid is completely input dependent allowing the user a lot of freedom and customization options. There is a tutorial and example test-cases available for the user to look at associated with the document and submitted code.

The user is able to customize: 1.) the height of the grid (int) , 2.) the width of the grid (int) , 3.) the population’s density (float) , 4.) the number of alleles that occur in the population (int) 5.) the actual colors of the alleles (array) , 6.) and the frequency of the allele (float) .

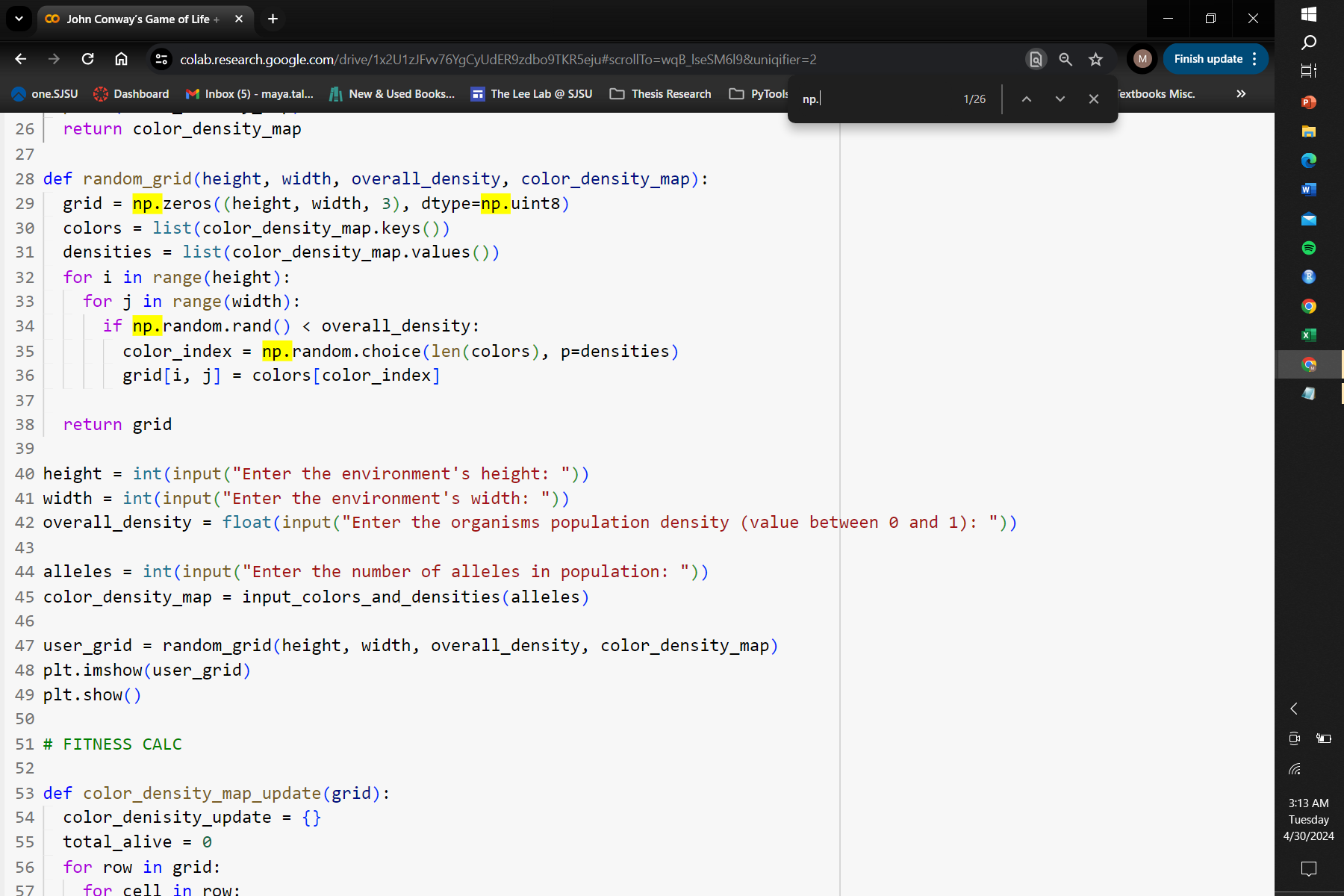


Fig. 3 Code demonstrating some of the initial user input.

The input values from the user are then used in two separate functions

1.) **input\_colors\_and\_densities**( ) and 2.) **random\_grid**( ).

When a user specifies how many alleles they wish to add to the simulation, this information is used as an argument in the input\_colors\_densities function. This function prompts the user to determine the color and frequency of their alleles. The user must input the color of the allele in (R,G,B) format and the frequency of that allele as a float (the probability that it is one allele versus another). The input\_colors\_and\_densities( ) function stores the colors and frequency of the allele into a dictionary called the color\_denisty\_map. A recurring theme within the code is to easily access both color and/or frequency data through .keys(), .values(), or .items(). Functionality like this is important later in the code when determining a child’s color from parent data stored in the dictionary. The relative frequency of an allele in a population is the probability it will occur in the grid, and thus the code is written such that the last allele frequency will always sum to 1.

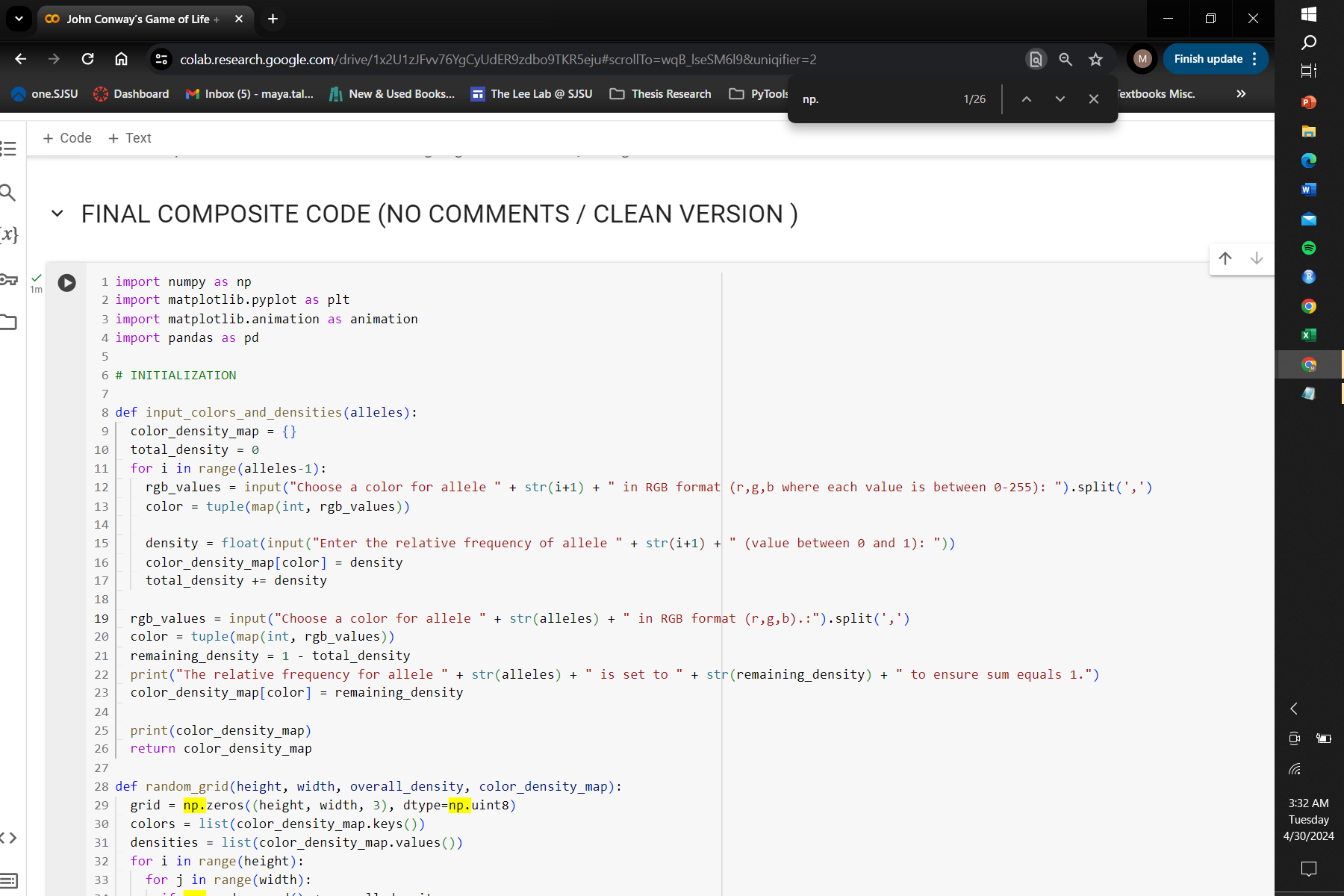


Fig 4. Code demonstration the input\_colors\_and\_densities function

The remaining user input data is used as an argument in the random\_grid() function. The grids in our simulation are made using np.zeros which creates an array of (0,0,0) in three channels of the user’s specified height and width. The three (0,0,0) in the array represent a single cell where each zero represents the three channels for RBG information.

| (0,0,0) | (0,0,0) | (0,0,0) |
| --- | --- | --- |
| (0,0,0) | (0,0,0) | (0,0,0) |
| (0,0,0) | (0,0,0) | (0,0,0) |

Fig 5. Visual representation of a initial cell with all “dead” cells

When a cell is “dead” it will look black and contain no RBG information: (0,0,0). However, when a cell is “alive” any one of the three RGB channels will contain non-zero values between 1-255 : (255,0,0) or (0,255,0) or (0,0,255) or any variation of the numbers between 1-255 therein. An initial cell is determined “alive” or “dead” using np.random.rand( ) where each cell in the grid is iterated upon and a number between 0-1 is picked. If the number is smaller than the user defined overall population density then that cell is “alive”. A similar methodology is used to then determine out of all the alive cell’s, what their color/allele will be. If a cell is determined to be alive it accesses the values and keys from the color\_density\_map dictionary and uses np.random.choice to pick a color with the density being the probability that color is randomly picked.

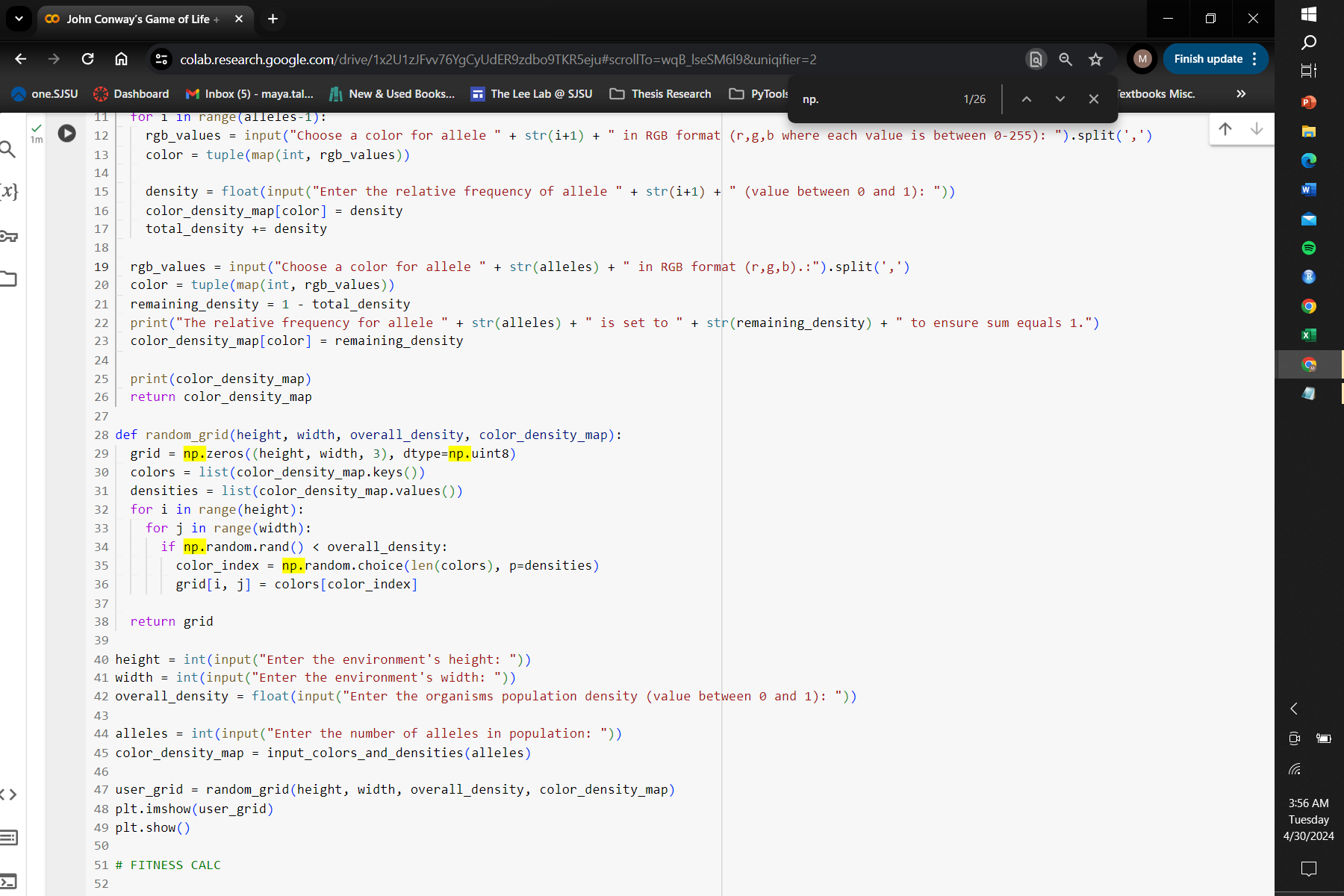
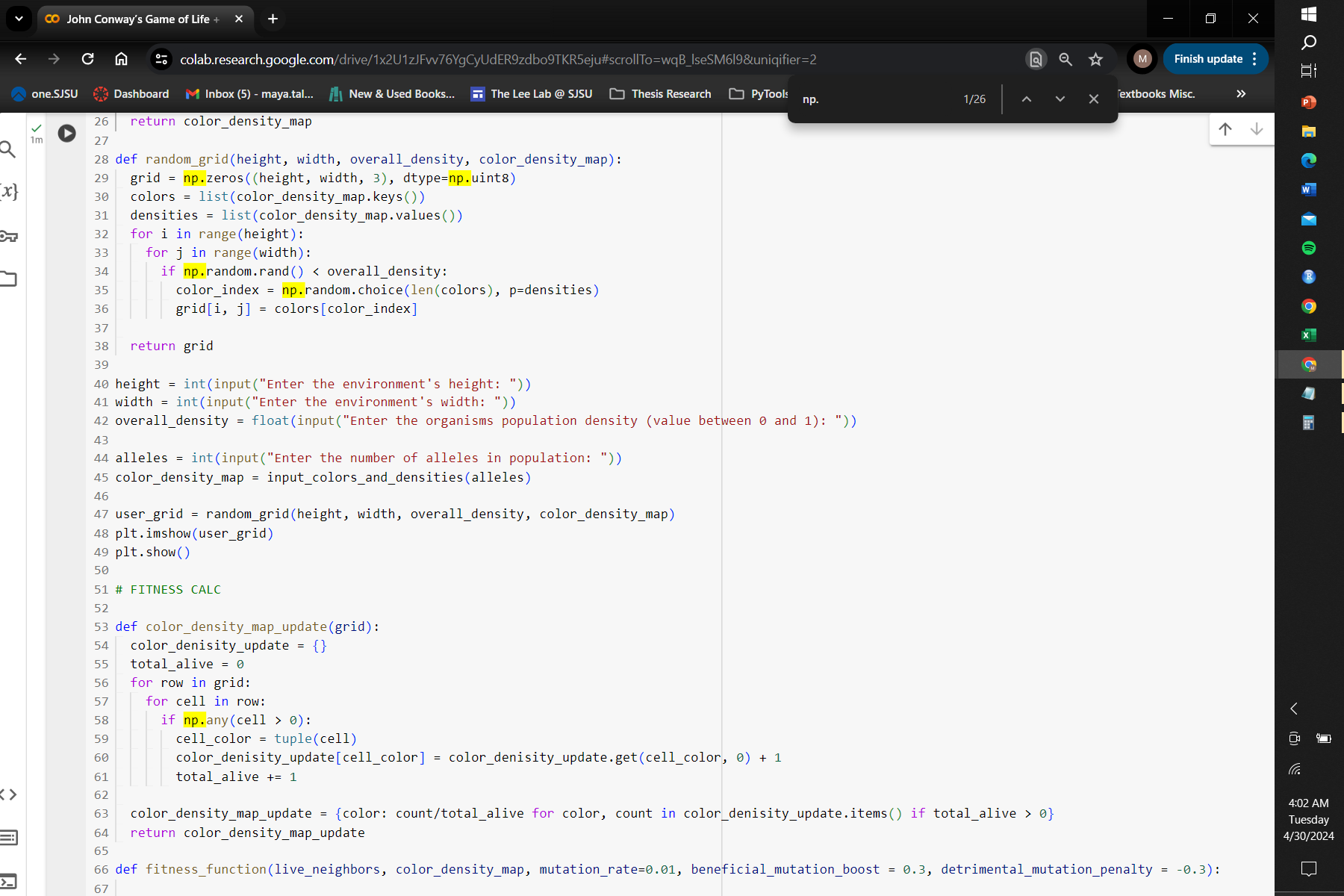


Fig 6. Code demonstration the random\_grid function

| (255,0,0) | (0,0,0) | (0,0,0) |
| --- | --- | --- |
| (0,0,0) | (0,0,255) | (0,0,0) |
| (255,0,0) | (0,0,0) | (0,0,0) |

Fig 7. Visualization of a grid with RGB color information. This demonstrates a 33% overall population density (3 of 9 cells alive). 66% of the living cells are red and 33% are blue. color\_denisty\_map = {(255,0,0): 0.66, (0,0,255):0.33}

The last part of Phase 1 is calling in the random\_grid function, giving it a variable name, and using matplot.plt to show a cursory glance at the initial grid before the simulation is run completely.

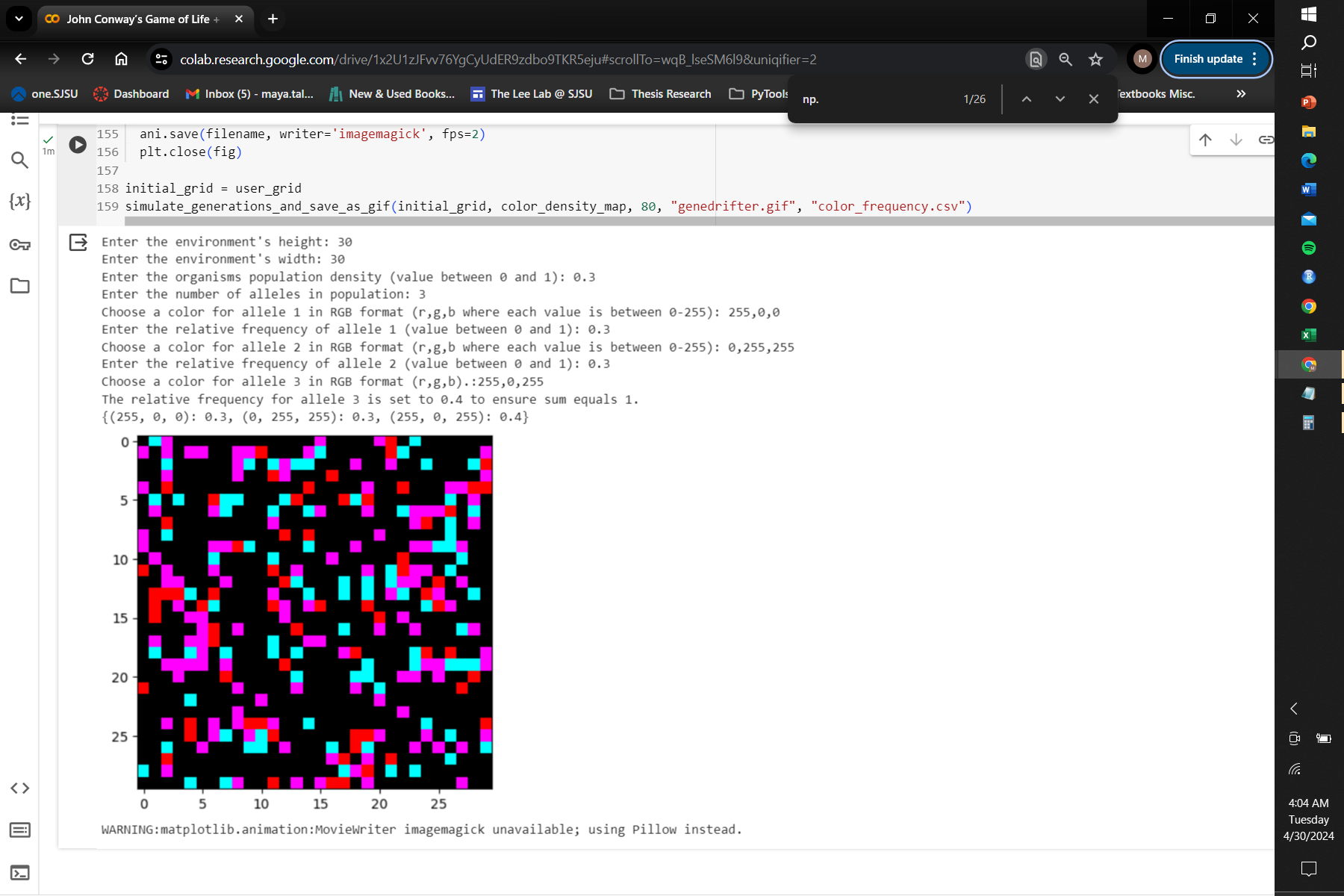


Fig. 8 The initial\_grid output and example of what the user input looks like.

## Phase 2 Code:

Phase 2 of the code focuses on managing the color\_denisty\_map dictionary over the course of the simulation. When a new cell is born or a cell dies the relative frequency of all alleles or colors changes and must be properly accounted for and updated in each generation. By proxy when the relative frequency of an allele changes, the “fitness” of every other cell sharing that same allele changes. The “color\_density\_map\_update” function updates the relative frequencies of color within the grid by iterating through each cell in the grid and extracting its (RBG) color information. It then stores this information in an empty dictionary along with a recalculated relative frequency. A dictionary comprehension is used to recalculate the relative frequency of each color it encounters by dividing its tally by the total number of alive cells. The resulting color\_density\_map\_update dictionary encapsulates the relative frequencies of colors within the population of the new generation until it is called again the next generation

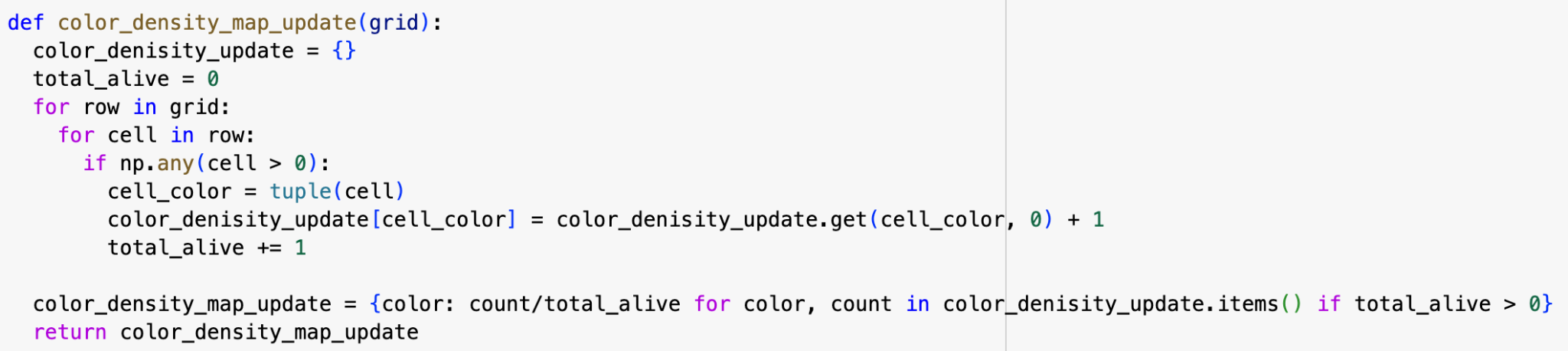
****

Fig. 9 Code demonstrating the color\_denisty\_map dictionary and how it is updated over generations

Two alleles can mate to form a hybrid child cell. Hybrids are represented by the merging of two colors. The higher your fitness or relative frequency the more of your genes or color would contribute to the offspring. We do this with a weighted average instead of a traditional average. The ‘fitness\_function’ is called when a child cell is born. The fitness\_funciton computes the weighted average from the parent cells’ three RGB values over the total density of alive cells.



Fig. 10 The fitness\_function that determines the weighted average

Mutations are also handled by the fitness function and occur when a new cell is born. More likely than not mutations are bad than good, though good mutations do exist. The rate of mutations and the boost or penalty to fitness mutations given to a child cell are defined arguments in the fitness\_fucniton. To represent mutations, there are two built in colors which were chosen to represent good(green) and bad (yellow) mutations. These are hardcoded in for now for ease.

| (0,0,0) | Bad mutant | (0,0,0) |
| --- | --- | --- |
| (0,255,255) | (0,0,0) | (0,255,255) |
| (0,0,0) | Good mutant | (255,0,0) |

Fig 11.

In this simulation, every new cell has a 1% chance of being a mutant, with 10% of those mutations being beneficial and 90% being harmful. The color of the mutant cell is incorporated into the {color:density} dictionary, accompanied by a boost or penalty to its relative frequency or fitness. Despite receiving a fitness penalty for bad mutations, mutants cannot decrease below 0.01 in fitness or become negative. These mutations adhere to the same rules as regular cells so one could visualize how a mutant can be naturally selected out or for, diluted, and persist within a population.



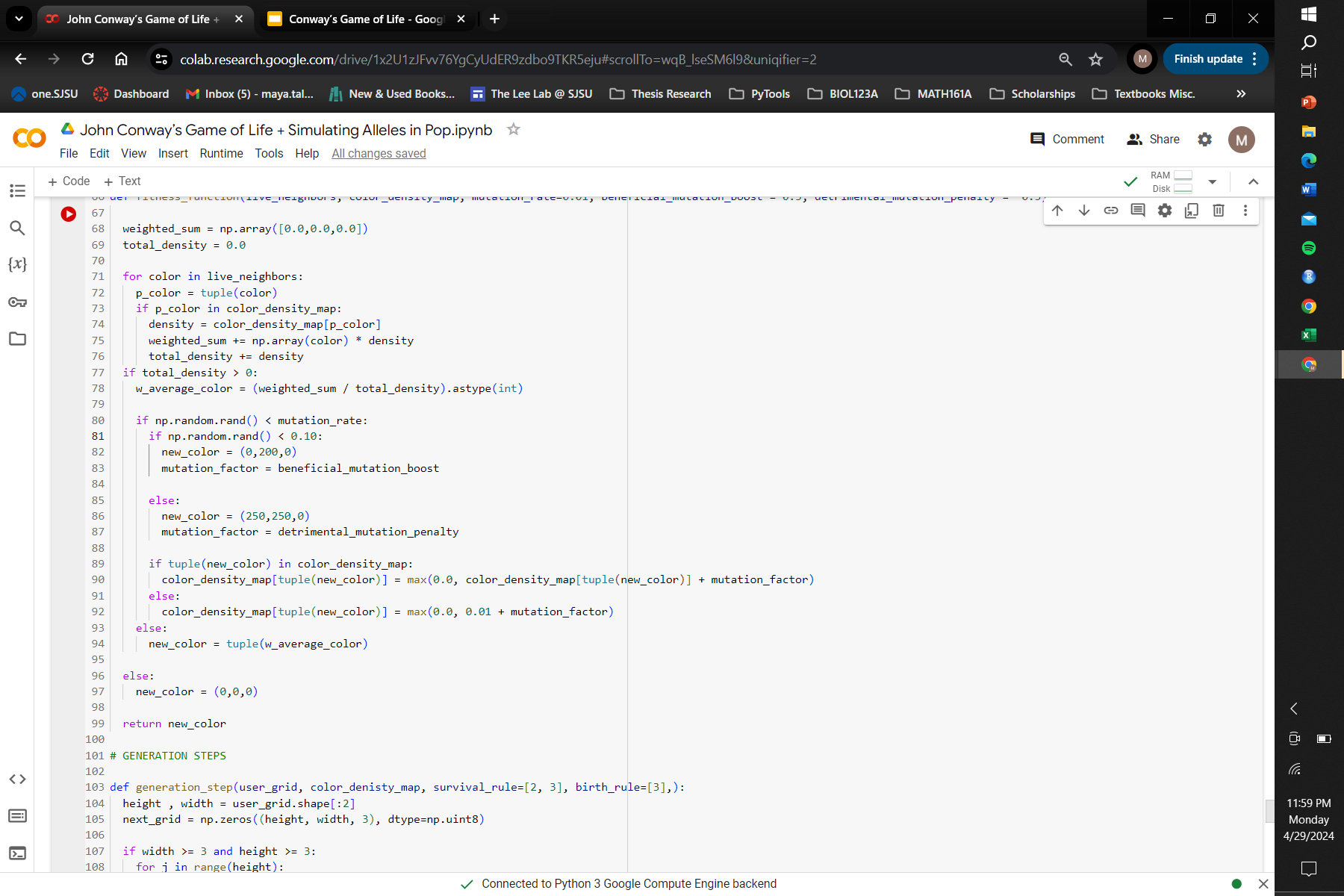


Fig 12.

## Phase 3 Code:

Phase 3 of the code focuses on implementing Conways’ rules to determine whether a cell lives or dies on a grid and generates the grid for the next generation. Conway's Rules are implemented on the current grid called user\_grid and outputs the next\_grid. Phase 3 consists of a function called generation\_step which takes in the arguments of the user\_grid, color\_denisty\_map, survival\_rule , and birth\_rule. The survival\_rule and birth\_rule are a list of numbers which define the number number of neighboring cells needed for a cell to survive (2 or 3) and the number of neighboring cells needed to birth a cell (exactly 3)

The generation\_step function generates a blank grid of an array of (0,0,0) that is the same shape as the last generation (the user defined starting grid). The function uses a list comprehension to iterate through each cell in the grid, and accounts for how many neighboring cells are alive. Again, a cell is counted as alive if any of the three values in the RGB channels are non-zero numbers. Neighbor cells are accounted for by its position where the current cell being iterated on is at position (0,0) or (i, j). The positional coordinates change in the “i” (up/down) or “j” (left/right) direction by adding “a” to “i” and “b” to “j”. The “a” or “b” can take on 3 different values (0,1,-1).

For example, to look at the cell to the left of the cell being iterated upon (0,0) the coordinates for the cell to the left would be (0, -1); where a =0 and b= -1, thus i+0 =0 and j+(-1) = -1. This method should check all 8 surrounding cells with the exception of cells on the edge. Our simulation assumes hard boundaries and no wrapping or toroidal boundaries. There are conditions defined within the list comprehension to ensure the neighbor cells are within the bounds of the grid. The resulting number of alive neighbor cells in relation to each cell is called neighbor\_count.



Figure 13. A demonstration of how neighbor cells are accounted for via position in relation to the current cell being iterated on.

A series of conditional statements determine the state of the cells in the next generation using the neighbor\_count we calculated above , the survival\_rule, and birth\_rule. If a cell is alive it will make it to the next generation if it has exactly the right number of neighbors (2 or 3). If however, it has too many or too few neighbors it will die in the next generation and the color is updated to (0,0,0).

A series of conditionals also checks the cells that are not alive to see if they meet the criteria for the birthed rule by having exactly 3 neighbor cells. If the child cell is born in the next generation, the aforementioned fitness\_function( ) is called and its RGB values are decided from its parents, also known as its live neighbors (weird to think your neighbors are your parents, but this is a weird organism that can only have children if 3 individuals are involved). This function finally returns the updated grid with the new cells. This will repeat for the entire simulation.

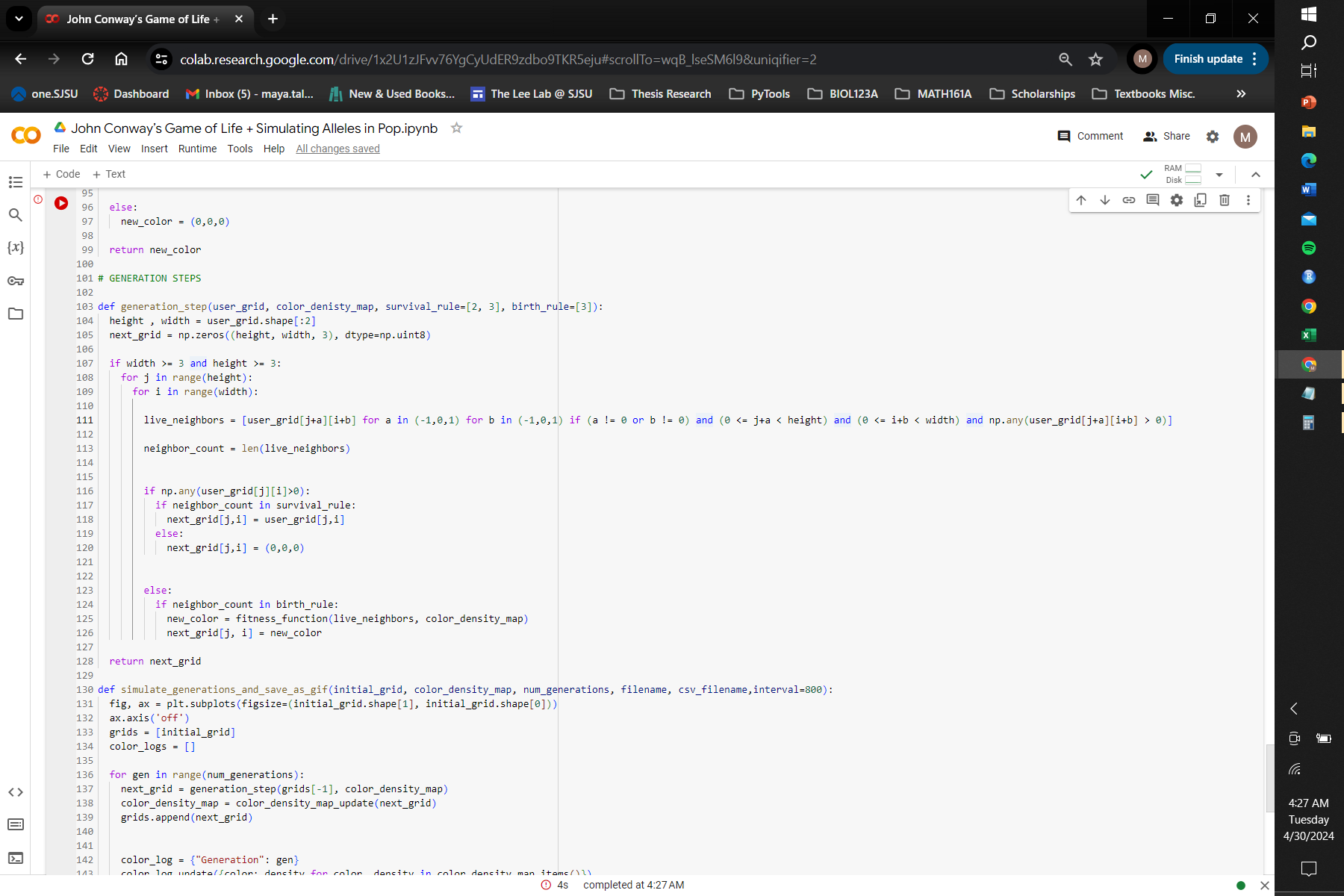


Fig 14. The generation step function

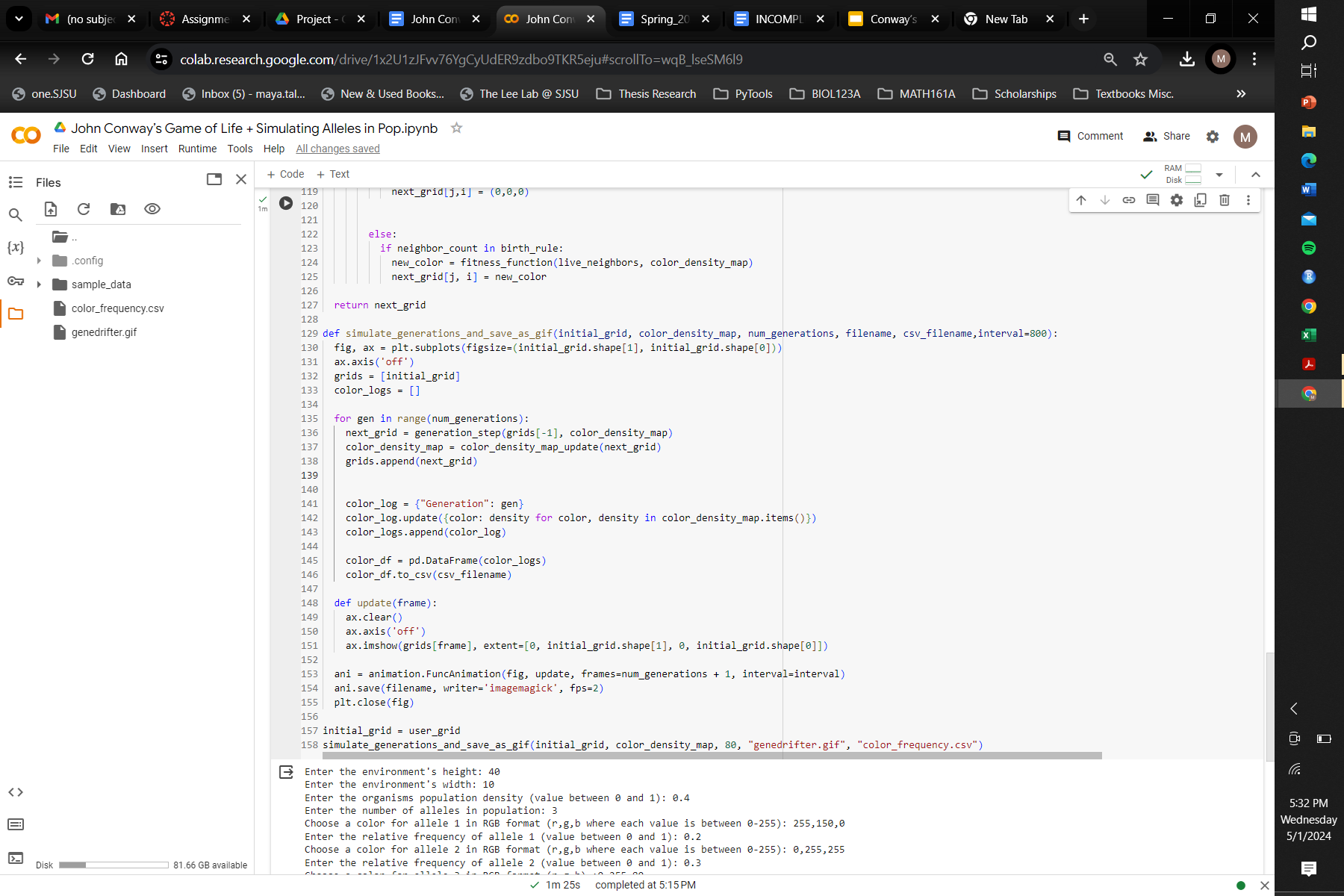
## Phase 4 Code:

Phase 4 of the code implements all the functions we have set up before, but defines for how many generations or iterations are we going to be implementing these functions. This is an important structure of the code because this implements the “time” aspect.

The simulate\_generations\_and\_save\_as\_gif function manages the simulation by iterating through multiple generations and updating the grid accordingly. The function also uses lists to store each generation's grid and color distributions.

As the function loops over each generation a specified number of times it appends the new grid to a list for further visualization. It also accumulates all the colors and densities from the color\_density\_map\_update dictionary and appends it to the color\_logs list. Lastly, it saves the color distribution logs to a CSV file using a pandas DataFrame.

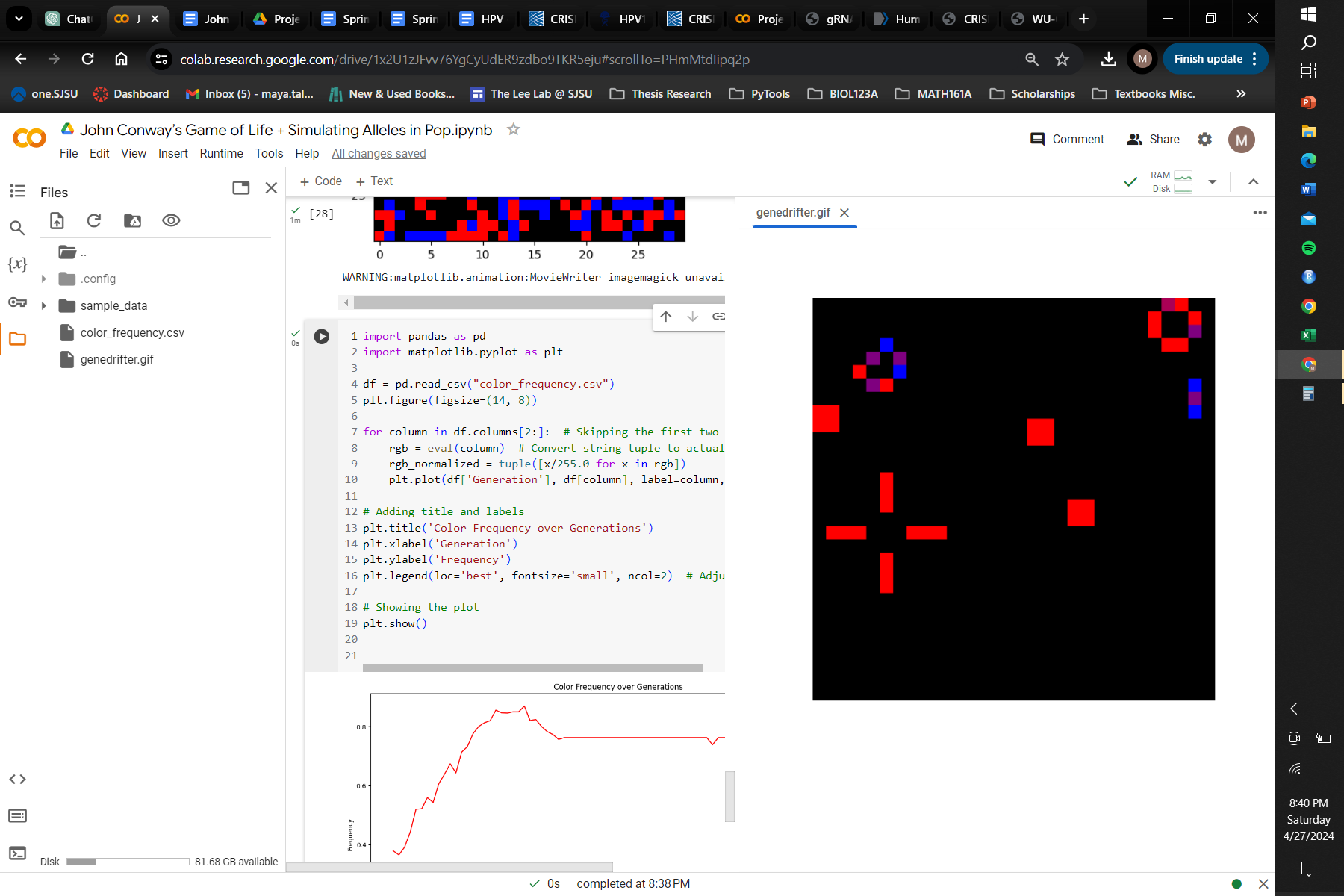
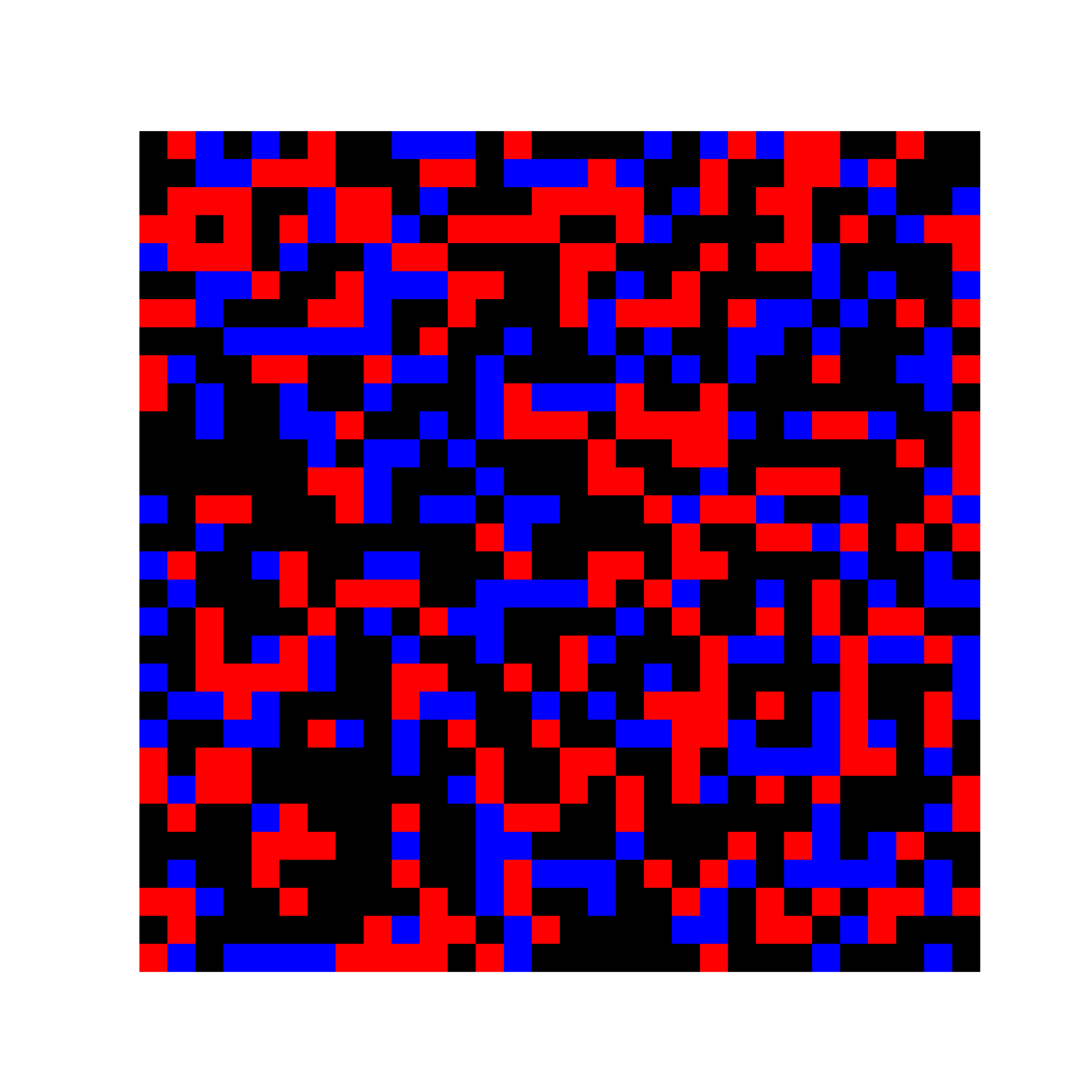
The generation-by-generation evolution is captured in an animated GIF. The update function clears the previous frame, visualizes the current grid, and sets the axis off for each frame. Using Matplotlib's FuncAnimation, it creates the animated sequence by iterating over the frames (generations) and updating the plot accordingly. The ani.save method saves the animation as a GIF file, specifying the filename, frame rate (fps), and interval between frames (interval). The parameters for the animation are hardcoded, but one could change the speed and intervals of the GIF, the number of generations, and file names in this section of code. The code finally outputs two files with the GIF and .csv file of the data frame.



## Analysis of Code Output

In this example of output below there are two alleles in one large population that is initially homogeneous across the grid. Over the course of 80 generations there are significantly less individuals, meaning the initial population was overpopulated for the size of the environment. Towards the end of the simulation the alleles form smaller and more stable populations segregated from each other. This can simulate natural selection where alleles tend to select for alleles similar to them. Red is the dominant and more fit allele in this simulation where towards the end of the simulation around 80% of the alive cells are red. The original blue allele seems to persist after 80 generations, but only because of hybridization with the red allele, based on the presence of purple in the smaller populations that contain blue. Deleterious mutations pop up briefly, but die out quickly, and beneficial mutations show up, but aren't able to persist.

First Generation from Initialization Last Iteration after 80 generations.



The relative frequency of each color mapped over time using pandas

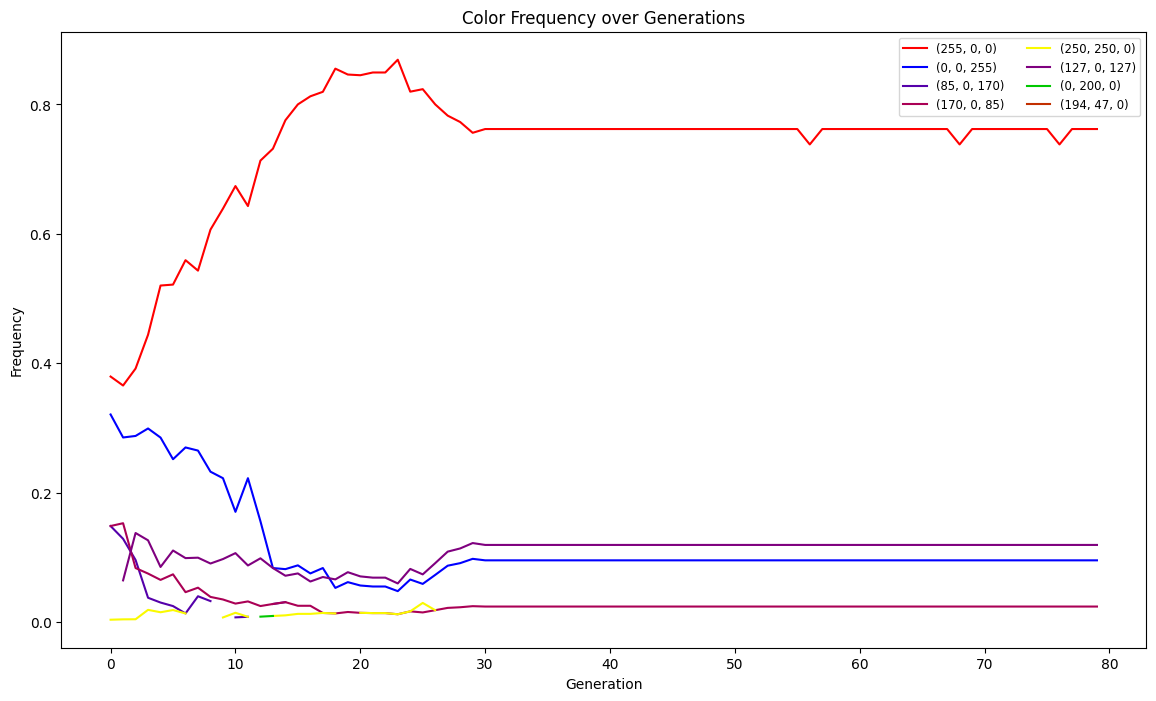


Fig 15.

## CONCLUSION AND FUTURE IMPROVEMENTS

Our program offers a fun way to simulate biological systems in silica. Though it is far from truly accurate there are a lot of interesting interpretations you can derive from our model of genetic drift, natural selection, and inheritance. For future improvements, I would redo the structure of the program to be more class oriented and use the below UML diagram as a guide to the relationships between each of the classes. This would've been a more organized way of organizing the code and more robust to make instances of each class for the simulation. Additionally a lot of things that may be biologically relevant and useful for a user to tweak such as “mutation rate” and “number of generations” are hardcoded. With already so many user inputs I would have designed either a more user friendly GUI to allow for more customization or allow for exception handling in the code.

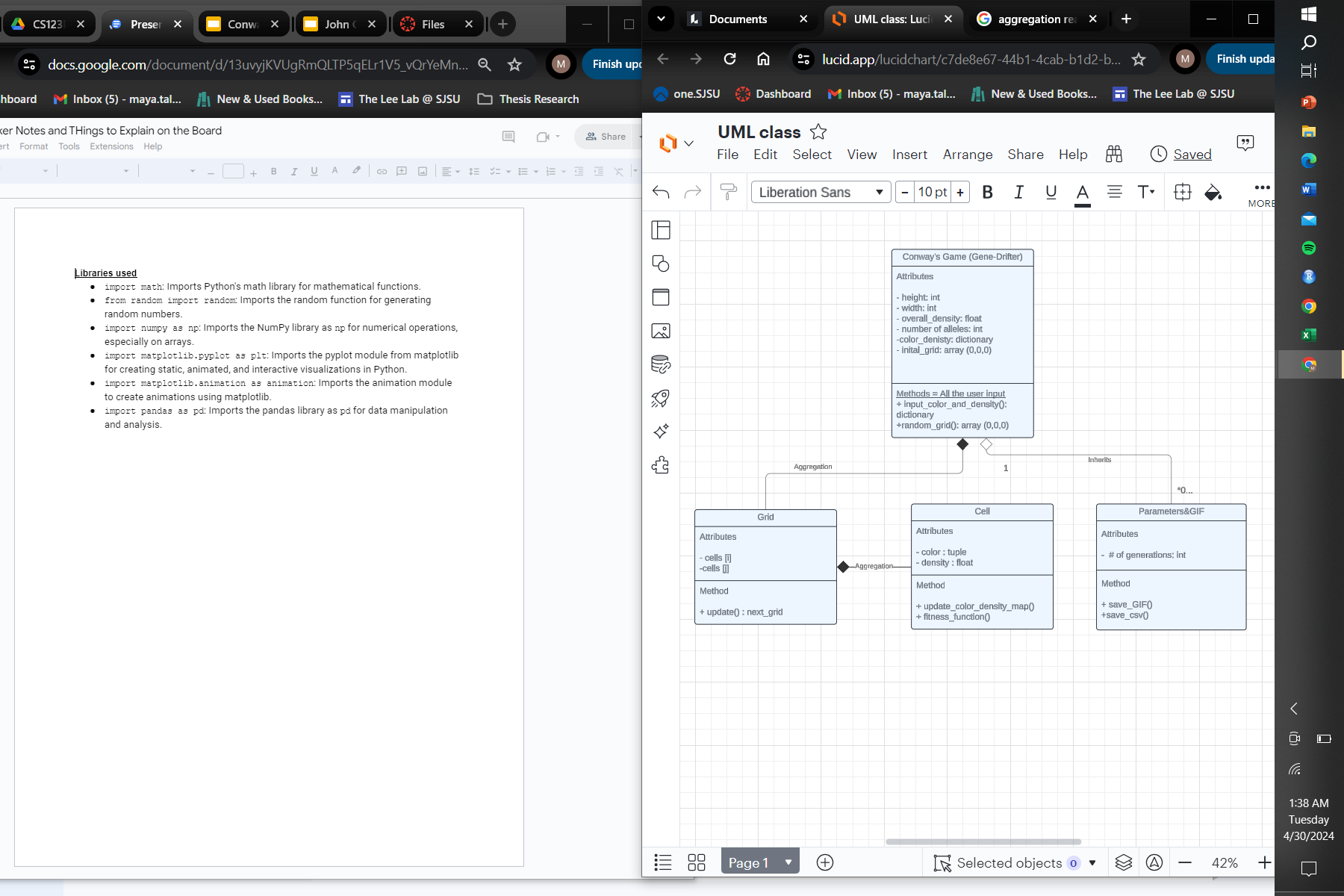


Fig 16.

## 

## 

## 

## 

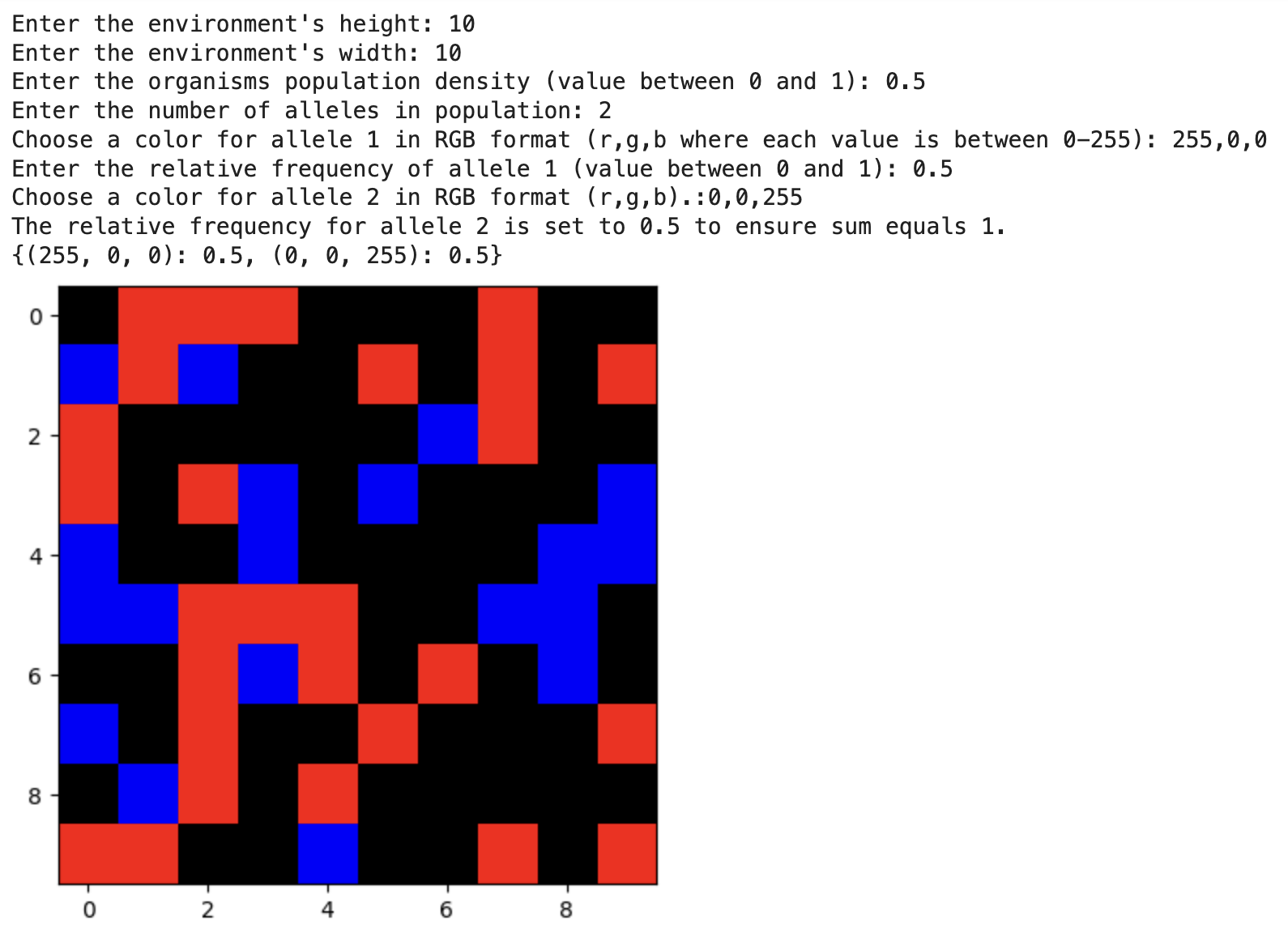
## 

## 

## REFERENCES

1. Tyshevskyi, P. (2019, April). Reversing Game of Life using genetic algorithms. *Medium*. Retrieved from <https://medium.com/@paveltyshevskyi/reversing-game-of-life-using-genetic-algorithms-45f6209e1daf>
2. Cerruti, U., Dutto, S., & Murru, N. (2020). A symbiosis between cellular automata and genetic algorithms. *Chaos, Solitons & Fractals, 134*, 109719.<https://doi.org/10.1016/j.chaos.2020.109719>
3. Fitness and its role in evolutionary genetics. (2010). *PMC*. Retrieved from<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2753274/>
4. Zheleznyak, A. (2020). colorful-life: The colorful game of life. *GitHub*. Retrieved from<https://github.com/adam-zheleznyak/colorful-life>
5. Khankasikam, K. (2013). Background research on topic. *International Journal of Computer Theory and Engineering*. Retrieved from<https://www.ijcte.org/papers/769-A009.pdf>
6. Chandrakant, K. (2024). Nature-inspired algorithms. *Baeldung*. Retrieved from<https://www.baeldung.com/cs/nature-inspired-algorithms>
7. The Game of Life. (2020). *arXiv*. Retrieved from<https://arxiv.org/pdf/2007.02894.pdf>
8. Conway’s Game of Life – Python Implementation. (2024). *GeeksforGeeks*. Retrieved from <https://www.geeksforgeeks.org/conways-game-life-python-implementation/>

## SAMPLE OUTPUT



**TEST CASES**

| **Input** | **Output \*see attached TestCases.zip file** |
| --- | --- |
| **A Standard Test Case** |  |
| **Enter the environment's height: 10**  **Enter the environment's width: 10**  **Enter the organisms population density (value between 0 and 1): 0.5**  **Enter the number of alleles in population: 2**  **Choose a color for allele 1 in RGB format (r,g,b where each value is between 0-255): 255,0,0**  **Enter the relative frequency of allele 1 (value between 0 and 1): 0.5**  **Choose a color for allele 2 in RGB format (r,g,b).:0,0,255**  **The relative frequency for allele 2 is set to 0.5 to ensure sum equals 1.** | test\_case\_1\_genedrifter.gif  test\_case\_1\_color\_frequency.csv |
| **A test case with >2 alleles** |  |
| **Enter the environment's height: 30**  **Enter the environment's width: 30**  **Enter the organisms population density (value between 0 and 1): 0.30**  **Enter the number of alleles in population: 5**  **Choose a color for allele 1 in RGB format (r,g,b where each value is between 0-255): 255,153,153**  **Enter the relative frequency of allele 1 (value between 0 and 1): 0.30**  **Choose a color for allele 2 in RGB format (r,g,b where each value is between 0-255): 204,255,153**  **Enter the relative frequency of allele 2 (value between 0 and 1): 0.30**  **Choose a color for allele 3 in RGB format (r,g,b where each value is between 0-255): 255,178,102**  **Enter the relative frequency of allele 3 (value between 0 and 1): 0.2**  **Choose a color for allele 4 in RGB format (r,g,b where each value is between 0-255): 255,255,153**  **Enter the relative frequency of allele 4 (value between 0 and 1): 0.1**  **Choose a color for allele 5 in RGB format (r,g,b).:153,255,204**  **The relative frequency for allele 5 is set to 0.09999999999999998 to ensure sum equals 1.** | test\_case\_2\_genedrifter.gif  test\_case\_2\_color\_frequency.csv |
| **A test case with a rectangular environment/grid** |  |
| **Enter the environment's height: 40**  **Enter the environment's width: 10**  **Enter the organisms population density (value between 0 and 1): 0.4**  **Enter the number of alleles in population: 3**  **Choose a color for allele 1 in RGB format (r,g,b where each value is between 0-255): 255,150,0**  **Enter the relative frequency of allele 1 (value between 0 and 1): 0.2**  **Choose a color for allele 2 in RGB format (r,g,b where each value is between 0-255): 0,255,255**  **Enter the relative frequency of allele 2 (value between 0 and 1): 0.3**  **Choose a color for allele 3 in RGB format (r,g,b).:0,255,80**  **The relative frequency for allele 3 is set to 0.5 to ensure sum equals 1.** | test\_case\_3\_genedrifter.gif  test\_case\_3\_color\_frequency.csv |

## 

## INSTRUCTIONS TO RUN CODE

1. **Set up the Environment:** 
   1. Make sure that necessary libraries are installed: numpy, matplotlib, and pandas
2. **Run Code**
3. **Follow Instructions for user input**
   1. Enter the environment's height
   2. Enter the environment's width
   3. Enter the organisms population density (value between 0 and 1)
   4. Enter the number of alleles in population
   5. Choose a color for allele 1 in RGB format (r,g,b where each value is between 0-255)
   6. Enter the relative frequency of allele 1 (value between 0 and 1)
   7. Choose a color for allele 2 in RGB format (r,g,b).
4. **View Results:** Output should include:
   1. GIF animation
   2. csv.file with color frequency data