Abstract

The purpose of this project was to get more experience working with array lists and two-dimensional arrays in java. This project uses multiple classes in order to properly set up our data structures, simulate interactions between elements in our data structures, and visualize those interactions using cellular automata. As a result, we were able to simulate and visualize Conway's Game of Life in order to see how the rules created by Conway impacted interactions between our cells. From there, we took the basic idea of cellular automata, and modified our approach for Conway's Game of Life so that we can simulate a complicated system known as the reaction-diffusion system. This implementation required us to use a continuous cellular automata model to properly simulate a famous chemical reaction known as the Belousov–Zhabotinsky reactions. At the end of the project, we saw that we were able to successfully simulate the Belousov–Zhabotinsky reactions and recreate patterns that have been found in a laboratory setting.

Results

The Cell class is a class that was made to represent a cell in Conway's Game of Life. The cell only has states, dead or alive, and the Cell class takes advantage of this binary state by using Boolean values to keep track of the cell's state. This class represents an alive cell with the Boolean value true and a dead cell with the Boolean value false. By default, the class initializes a cell to be in the dead state, but there is also an option to initialize a cell with a given state. The class also has methods which allow you change the state of the cell, return the state of the cell, and also reset the cell. There are more methods which I will talk about when graphics come up. We also have a toString method which overrides the inherited toString method so that it can output a 1 if the cell is alive or a 0 if the cell is dead.

The next class is the Landscape class. This class represents a grid of cells that we will be using to simulate Conway's Game of Life. This class uses a 2D array to hold a cell at some ijth position in the grid, the reason we opted for a 2D array is because we are planning on visualizing our cells as time goes on. The Landscape initializes a grid with row and column values as its given parameters. The grid is then filled with cells each initialized in their default state. The class also has a reset method which goes through every cell in the grid and resets it to its default state. Additionally, this class has methods which returns the number of columns and rows of our grid as well as a method that returns a cell at a given row and column in our grid. There are also more methods which I will talk about in the next section. This class also has a toString method which prints out our grid in a nice format, but this format breaks down when the grid is huge.

The next class we have is the LandscapeDisplay class which was originally written by Professor Bruce A. Maxwell! This class has methods which will draw our cells and grid using Java's Abstract Window Toolkit as well as methods that allow us to save our Landscape's display.

Now that we have talked about the basic methods in all of the classes in this project, I will talk about the drawing methods and the methods responsible for running the Conway's Game of Life simulation. The first important method is from the Landscape class, which is the getNeighbors method. This method returns the neighbors of a cell at a given row and column value in our grid. This method is extremely important because if our cells can't recognize their neighbors, then we won't be able to properly simulate Conway's Game of Life. I designed this method to follow the

Moore Neighborhood algorithm! Essentially, we have a cell at some ijth position and we pick its neighbors to be the 8 closest cells. This creates a box around our ijth cell which looks like the following:

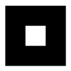

Figure 1: Moore Neighborhood set up

In order to deal with the boundary, I decided to take a wrapping approach as opposed to a border approach. The reason I went with a wrapping approach is because I wanted to prepare for an extension that I was planning to do, and the extension wouldn't look aesthetically pleasing if I took a border approach. The way the wrapping approach works is by recalculating our indices by using modular arithmetic! I used the following expressions to calculate my indices:

$$i_{wrap} = (i + r) \bmod r$$
$$j_{wrap} = (j + c) \bmod c$$

Here, r is understood to be the number of rows in the grid and c is understood to be the number of columns in our grid. The powerful thing about this approach is that we don't need to check any extra conditions since it is impossible to go out of bounds. By that, I mean that i_{wrap} will always be less than the number of rows in our grid and if we happen to be at a corner, this expression will take care of the wrapping. This is how the neighborhood of cell at the bottom left corner looks like with this method:

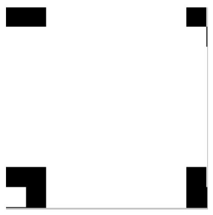

Figure 2: Wrapping approach to the boundaries

The next important method is the updateState method in the Cell class. This method is responsible for checking the state of a cell and then determining the cell's new state by checking the states of its neighbors. The updateState method uses the rules defined by Conway:

1. If the cell is alive and has exactly 2 or 3 alive neighbors, keep it alive. Otherwise, kill it.
2. If a cell is dead and has exactly 3 alive neighbors, bring it back to life. Otherwise keep it dead.

This method has an ArrayList of the neighbor cells as an input. First it checks if the cell is alive or dead, and then it counts the amount of alive neighbors. After that, it checks the conditions set by Conway to determine if the cell should be dead or alive.

The next group of methods are responsible for visualizing our simulation. First, the draw method in the cell class! This class inputs a graphics object, positions, and a scale which will be used to draw our cell. I opted to visualize the cells as squares since using ovals had this weird gap that got distracting and often times hard to stare at for a long period of time. I also opted to overwrite the LandscapeDisplay class' default color since gray and black don't mix very well. The method uses the position parameters to put our cell in a position that mirrors its position in the 2d grid array. I decided to go with the traditional black and white color for the cells, opting to visualize alive cells with black and dead cells with white. Finally, we have the draw method from the Landscape class, which just goes through all the cells in our grid and calls on their draw method.

The last class in this project is the LifeSimulation class! This class does not have any methods and it just uses the main method to run our simulation for 300 generations. A notable thing about this class is that it initializes the cells in a random state, and it only initializes a percentage of them. By default, the percentage is 30%, but that could changed to higher percentages.

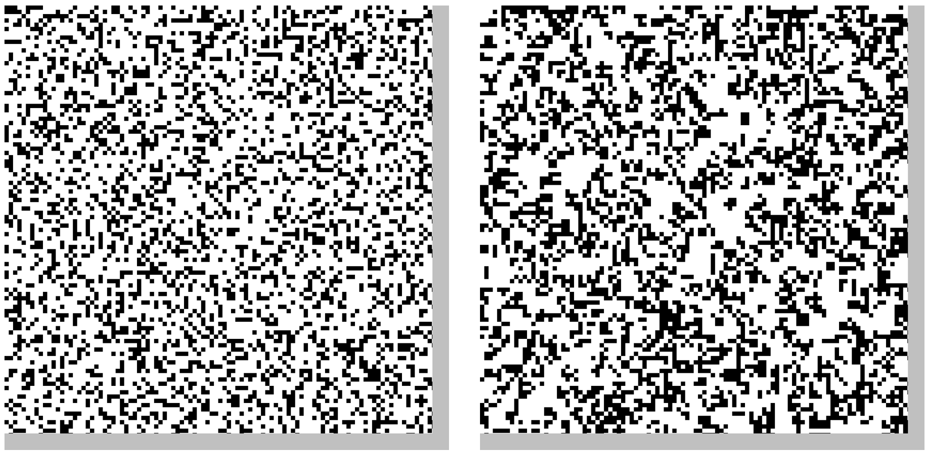
(Generation: 1) (Generation: 2)

Figure 3: First two generations of my Conway's Game of Life simulation

The results of the project were pretty interesting! Even though we started from a completely random configuration, and the next step also looked random as seen in Figure. 3, the simulation got some form to it after a couple of generations went by. This evolution to a less random state is shown in the mySim.gif file in the google drive, but here is a neat photo of a random generation!

Overall, Conway's Game of Life is pretty interesting and can be used to simulate population dynamics. An interesting thing that I played around with was the density! Each population has its own carrying capacity and depending on their population size, it can either grow towards that capacity or decay down to that capacity. I didn't include that as an extension though, because I wanted to focus on the Cellular Automaton aspect of the project.

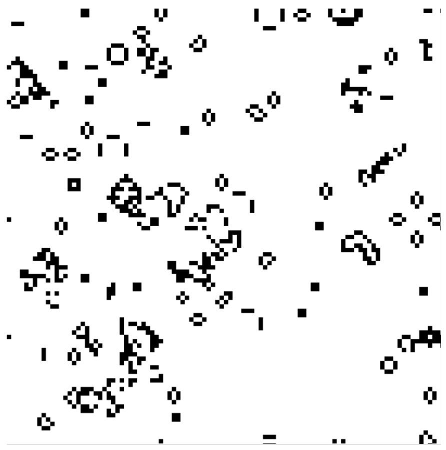

Figure 4: A random generation from the Conway's Game of Life simulation. The straight lines are known as oscillators since they spin!

Extension

For my extension I played around with the aspect of Cellular Automatons (CA). CA is very powerful because we can use it to model a lot of complicated things by using a couple of simple rules. Conway's Game of Life for example has the potential to simulate the spread of disease with a couple of modifications to its rules. I did not go for that extension though because it didn't sound that exciting.

Instead, I decided to model Turing Patterns! Specifically, I decided to model a famous type of chemical reaction known as the Belousov–Zhabotinsky reaction. First, a little context before moving into the code! Belousov–Zhabotinsky reactions are a special class of reactions that happen outside a reaction's equilibrium point. These reactions are also known as reaction-diffusion systems which were initially proposed to be the main mechanism of pattern formation in animals by Alan Turing. [1] Alan Turing's Paper, " The Chemical Basis of Morphogenesis " essentially proposed a system in which a there are different competing chemicals whose interaction results in a pattern forming as opposed to the diffusion of all the chemicals involved. [1] The Belousov–Zhabotinsky reaction is an example of such a system which was accidentally discovered and actually has a pretty depressing story behind it.

Cellular Automatons make a great fit for modeling Belousov–Zhabotinsky reactions, since chemical reactions can basically be broken down into a set of rules. Unlike Conway's Game of Life, modeling the Belousov–Zhabotinsky reactions takes a bit of work; instead of a binary state, we have a continuous state determined by the interactions of three chemicals. This means that I would need to modify the Cell class so that it can hold information about the concentrations of three chemicals, follow the Belousov–Zhabotinsky reaction, and draw the cells in accordance to the concentrations. I would also need to modify the Landscape class so that it can initialize the cells properly and also pick out the neighbors properly.

First, Cell class modifications. I opted to remake the Cell class and call it the TuringCell class in order to avoid headaches but also to rewrite the Cell class code with auxiliary methods and smaller lines. The TuringCell class has a similar structure to the Cell class, but has some major modifications including having 3 chemicals represent the state of the cell. The first significant modification is the draw class, and it happens to be the first class I was worried about. Since we are dealing with continuous values, I needed a way to draw the cell to represent the concentration of one of the chemicals as accurately as possible. To do this, instead of using RGB values to get a color, I used HSB values which makes things easier. The reason I used HSB values is because they are limited to values between 0 and 1, and I had the ability to keep two values constant while only changing one of the values. This allowed me to draw the cell with a color directly determined by the concentration of a chemical. In my code, I decided to let chemical A determine the color, but this can be changed to any other chemical.

The next issue was initializing the cells properly. Since the concentrations of the three chemicals in my cells were limited to values between 0 and 1, I had to come up with a way to make sure that those were the only values allowed. Using if statements is great if a user is setting the values, but when it comes to the values being calculated, if statements can be a pain. Instead I created an auxiliary method called limit! This method took in a value and limited it to values between 0 and 1. If a value of 1.1 got passed, the method will return a value of 1, similarly the if a value of -1 got passed, the method will return a value of 0. If any value between 0 and 1 got passed, the method will just leave that value alone. Initially this worked perfectly, but there some issues with the color gradients, so I lowered the limit from 1 to 0.8 to have a smoother gradient.

The next significant change I made was to the update method. Since we are dealing with three different concentrations, 8 neighbors and the current cell, the update method became huge and hard to follow. Here I broke the method up into auxiliary methods that did things a step at a time. First, I needed to average all the concentrations of the chemicals between the 8 neighbors and the cell we are looking at. This is where the average method comes in! The average method iterates through all 9 cells, gets their concentrations, adds them all up, and then averages them. The output of this method is an array which contains the averaged concentrations of all three chemicals.

The next part involves calculating the new concentrations based on the average concentrations. The rules that I used can be easily derived from the Belousov–Zhabotinsky reaction equations, but I opted to use the rules derived by Alasdair Turner [2]. The rules are given as the following:

$$A_{t+1} = A_t + A_t \cdot (\alpha B_t - \gamma C_t)$$
$$B_{t+1} = B_t + B_t \cdot (\beta C_t - \alpha A_t)$$
$$C_{t+1} = C_t + C_t \cdot (\gamma A_t - \beta B_t)$$

Here, α, β, and γ are parameters that descirbe how quickly chemicals in our mixture are being created or used up. Calculating the new concentrations is where the BZ method comes into play! This method has the neighbors ArrayList as its input so that it can call on the average method to get the average concentrations. It then uses the equations above to calculate the new concentrations which it then allocates to an array that will be the output of the method.

Finally we have a simplified update method which just calls on the BZ auxiliary method and assigns the cells concentrations to the new calculated concentrations.

The Landscape class also had a modification, but it was only for the getNeighbors class which I modified to include the cell we were checking. I also created an auxiliary method called the scanRow method. This method scans the row starting at some initial column value and ending at a final column value, this method also calculates the wrapped indices before getting the neighbors, and it returns an ArrayList. The creation of this auxiliary method shortened the length of the getNeighbors method and it also made it easier to organize the neighbors into one final ArrayList which would be the output.

The toString methods for both classes were also modified! For the TuringCell class, the toString method returned the concentrations of all three chemicals whereas the toString method for the Landscape class only returned the concentration of chemical A.

Finally, I made a class called BZSimulation which is exactly the same as the LifeSimulation class except it optionally takes in command line arguments. If there aren't any command line arguments, the class just runs the simulation with default values for α, β, and γ which is $\alpha = \beta = \gamma = 1$. The class only accepts three values which are strictly for α, β, and γ. This allows someone to play around with the simulation and see how the parameters impact the types of patterns that form.

The results for this extension were awesome, and pictures do not do them any justice. Comparing the initial state to the final state helps, but watching the system get to the final state is the fascinating part of this extension!

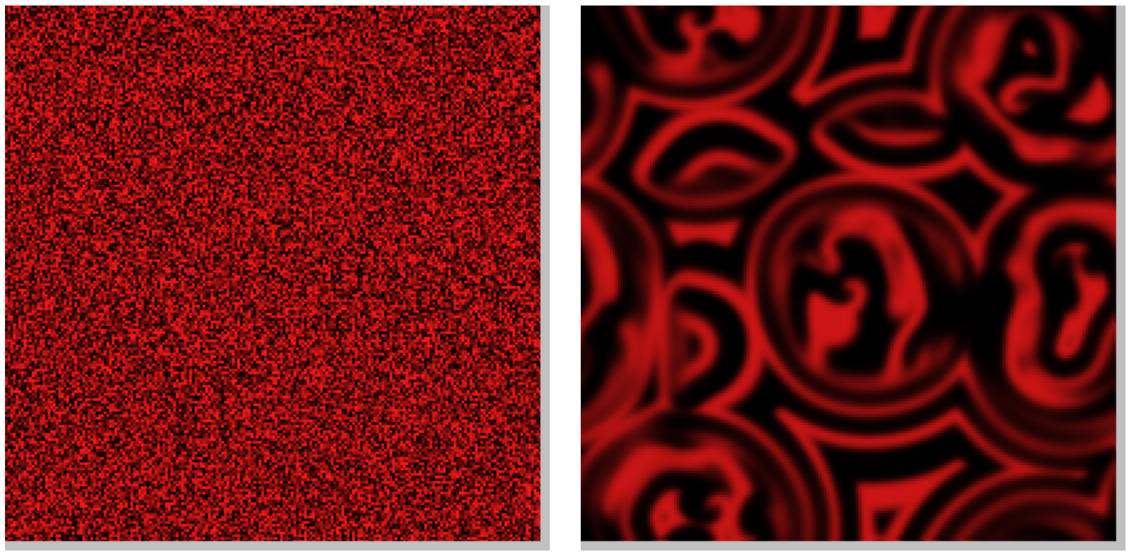
(Iteration: 1) (Iteration: 150)

Figure 5: Running BZSimulation with $\alpha = 1.0$ and $\beta = \gamma = 1.2$

Figure. 5 shows one of the many different types of patterns we can get from Belousov–Zhabotinsky reactions! The amazing thing to notice is the difference between the first iteration and the 150th iteration. This significant difference is what makes the discovery of Belousov–Zhabotinsky reactions depressing since no one wanted to believe Belousov when he attempted to publish his papers and his recipe! The mindblowing thing about this is that we started a chemical mixture from a completely random state, but instead of ending in a diffusive state, we ended in state that has patterns! In the different gifs I made you will see that these patterns are periodic! That is, these patterns repeat themselves over time and they originate from the same locations.

Overall, I am pretty happy that this extension worked properly without any bugs! The next possible steps would be to add a GUI that could not only let people change the parameters, but also the initial conditions, since they could absolutely impact the patterns that appear and their periodicity. The other change I would like to implement is removing the limit mechanism and using normalization. I think that normalizing the values of the concentrations after every update will result in a better color gradient, which will make the simulations a bit more realistic.

Finally, I noticed that the gifs for my extension were pretty big. I am still going to upload them to the google drive, but I also put them on a website I have been working on since Saturday! I will put the link in the references!

References

[1] Alan Mathison Turing. "The chemical basis of morphogenesis". In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 237.641 (1952), pp. 37–72. DOI: 10.1098/rstb.1952.0012. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rstb.1952.0012. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rstb.1952.0012.

[2] Alasdair Turner. "A Simple Model of the Belousov-Zhabotinsky Reaction from First Principles". In: *Bartlett School of Graduate Studies* (2009). eprint: https://discovery.ucl.ac.uk/id/eprint/17241/1/17241.pdf. URL: https://discovery.ucl.ac.uk/id/eprint/17241/1/17241.pdf.

I also want to reference Professor Jonathan McCoy from the Physics department! I took his Experimental Soft Matter class last semester which is where I became aware of Belousov–Zhabotinsky reactions and the history behind their discovery!

link to gifs: https://hpache.github.io/sims/BZgifs/