Aisha Thermidor

Dr. Drummond

Chem 274B

Final Project

Summary of the project:

At the core of our library is the CellularAutomata class, defined in the header file CellularAutomata.h. Here, we chose to use a vector of vectors for our 2D cellular automata grids and a single vector for the 1D automata. The library features include the implementation of parity, totalistic, and majority rules, depending on the user's choice of grid type and neighborhood conditions. We also incorporated functions for initializing the 1D or 2D grids and for displaying and updating the grid. The accompanying cellular_automata.cpp and neuron2neuron files contain the implementation of these functions, enabling initialization, computation, and display based on user inputs.

3.1.

As Member 3 in our project, my primary responsibility was to design and implement the output features for individual steps and data analysis. This role was pivotal in ensuring accurate and insightful visualization of our simulation results. Alongside this, I played a part in setting up and implementing the overall project, with a particular focus on visualizing the neuron2neuron, test_cellular_automata, and cellular_automata C++ files.

3.2.

I organized regular team meetings and maintained our project timeline, ensuring efficient progress and communication. In addition to managing these logistical aspects, I tackled the challenge of static results in the neuro2neuron and cellular_automata files by introducing dynamic grids and randomness, enhancing the realism and utility of our simulations. Using Jupyter Lab as my primary mode for graphing, I created visual representations of our data. For cellular_automata, I parsed data through text files to create GIFs, showing the evolution of the automata over time. The code involved reading iterations from a file, converting grids to images, and then compiling these images into GIFs using different color schemes to represent various states.

For neuron2neuron, I visualized the data differently due to the range of neighborhoods from 0-3. I assigned distinct colors to each neighborhood state, creating a clear and informative visual representation. The process involved reading grid states from a file, converting these states into images, and then creating a looping GIF to depict the progression of states over time.

Additionally, I developed the void update2dgrid function to maintain the dynamic nature of our simulations and contributed significantly to the neuron2neuron file, enhancing its

3.3

Header File Management

- Challenge: The primary issue was with management of the CellularAutomata.h header file. We initially struggled with redundancy in function implementations across different C++ files. To overcome this, we created public wrappers for private classes and wrote functions that called the header file instead of rewriting the code. This not only reduced redundancy but also made the code more maintainable.

Dynamic Grid Creation

- Challenge: Our grid was static and unchanging, resulting in repeated, predictable patterns. This was counterproductive, as we needed a dynamic system to simulate neuron behavior accurately. We introduced randomness and probabilities in the grid initialization and update functions, like initNeuronGrid and addRandomActivity. This randomness ensured that each simulation was unique and more representative of real neuron behavior.

Premature Grid Convergence

- Challenge: A significant problem was the grid converging too quickly to all 0s or all 1s. This convergence was happening within a few steps, making the simulation short-lived and less informative. We altered the rules in the weightedFiringRule function to add more complexity and variability. By introducing randomness and periodic changes, we managed to prolong the life of each simulation, making it more useful for studying dynamic systems.

Additional Considerations

- Separation of concerns was crucial. We made sure that each C++ file like neuron2neuron.cpp and test_cellular_automata.cpp had clear, distinct responsibilities. Error handling was also improved in scenarios where grid configurations did not match expected parameters.

3.4:

In terms of algorithmic efficiency, one-dimensional (1D) cellular automata generally outperform their two-dimensional (2D) counterparts. This difference stems from the nature of their initialization and update processes. For a 1D grid, we initialize a linear array of 'n' cells, but for a 2D grid, the initialization expands to 'n^2' cells due to its matrix-like structure. Similarly, updating the grid state in each iteration for 2D requires nested loops, in contrast to a single loop needed for 1D. This observation holds true irrespective of the chosen neighborhood type or boundary condition. Spatially, 2D operations are less efficient compared to 1D, as 2D utilizes a vector of vectors, leading to increased memory usage on the user's computer. The complexity of operations in our cellular automata library can be summarized as O(n) for 1D and O(n^2) for 2D.

3.5:

Our project leveraged the foundational cellular automata class to simulate neuronal activities in the brain. To mimic the multifaceted nature of neurons, we allowed for more than two states in each cell of the automata. The concept that neurons firing together increases the likelihood of future joint activation, akin to the formation of habits and learning processes, was a key aspect of our model. The repeated activation of certain cells in our simulation echoes this principle, as active cells tend to get activated again through successive iterations. While the model has shown promising results, its accuracy could be further enhanced by incorporating a three-dimensional (3D) aspect. Since neurons function in a 3D space, a 3D model would offer a more realistic portrayal of neuronal behavior.

3.6. Items we could improve:

3.6.1 Library Development:

Making the Code More Modular and Efficient

If I could redo this project, one of the first things I would focus on is making our library more modular. In our initial design, functions like grid initialization (**initNeuronGrid** and **initGrid2D**) and rule applications (**weightedFiringRule**) were tightly coupled with the data they operated on. This made the code less flexible and harder to maintain. I would restructure the library by separating functionalities into distinct classes or namespaces. For example, one class could handle grid initialization, another for grid operations, and a third for rule definitions. This separation would make the code more organized, easier to understand, and more adaptable to changes.

Efficient Data Handling

Regarding our grid representation, while nested vectors were a straightforward choice, they might not be the most efficient, especially for large grids with sparse data. A better approach could have been using a more memory-efficient data structure that can handle sparse data well. For instance, implementing a custom grid structure that only stores active cells and their states, while implicitly treating all other cells as inactive, could drastically reduce memory usage and improve performance, especially in scenarios where active cells are few.

3.6.2 Software Project Management: Emphasizing Version Control and Testing

Rigorous Version Control

We initially underutilized version control, which led to some challenges in tracking changes and understanding the evolution of the codebase. I would emphasize using a version control system like Git more rigorously. Each feature or bug fix should be a separate commit or branch, with clear, descriptive messages. This practice makes it easier to track changes, understand what each part of the code does, and facilitates collaboration.

Comprehensive Testing

Our testing was not as thorough as it could have been, which sometimes led to bugs slipping through. Implementing a comprehensive suite of automated tests, including unit tests for individual functions and integration tests for the entire system, is crucial. This not only ensures that the code works as intended but also helps in quickly identifying and fixing bugs.

3.6.3 Product Improvements: Enhancing User Interface and Customization

User-Friendly Interface

The library was primarily designed for users comfortable with C++, which limited its accessibility.

- GUI Implementation: Developing a graphical user interface (GUI) would make the library more user-friendly. This GUI could visually represent the grid, allow users to modify parameters (like grid size, rules, etc.), and observe the simulation in real-time, making the library accessible even to those without a programming background.

Our library was not as flexible as it could have been in allowing users to define their own rules or configurations.

- Customization Features: I would add features enabling users to easily customize rules and grid configurations without deep code modifications. This could involve a simple scripting interface or a set of APIs that let users define their own rules and interactions. It would extend the utility of the library to a broader range of scenarios and research fields.