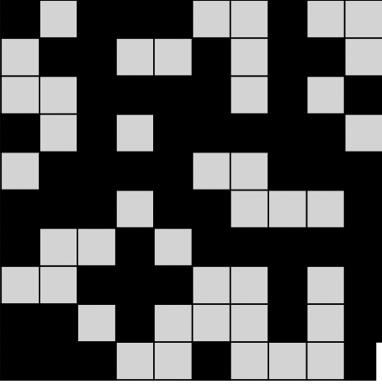
***ABSTRACT***

This project presents a comprehensive Python-based implementation of Conway’s Game of Life, a classic example of cellular automata. The implementation has been enhanced to support custom user-defined rules, enabling exploration of diverse evolutionary behaviors, and multi-state dynamics, where cells can transition among various states based on probabilistic rules. Utilizing NumPy for computational efficiency, Matplotlib for detailed visualizations, and JSON for intuitive rule customization, this project offers a flexible framework for studying emergent patterns and phenomena. Our system includes interactive features for specifying grid configurations, evolving patterns dynamically, and exporting results for further analysis. This work not only demonstrates the capabilities of cellular automata but also provides a tool for researchers and enthusiasts to experiment with complex rule systems and visualize their effects.

***PROJECT DESCRIPTION***

This project centers on Conway’s Game of Life, a cellular automaton known for demonstrating complex behaviors emerging from simple rules. The project implements the Game of Life across three levels of complexity:

1. **Level 1:** In Level 1, the basic version of the Game of Life is implemented. Each cell can either be alive or dead, and its future state depends on the number of its neighbors that are alive. A dead cell with exactly three live neighbors comes to life, while a live cell survives if it has two or three live neighbors. All other cells remain or become dead. This simple setup is used to simulate fascinating patterns like moving gliders or static blocks
2. **Level 2:** In Level 2, the rules are made flexible so that users can define how cells evolve. Instead of fixed conditions, users can set their own thresholds for when a dead cell should come to life and when a live cell should survive. These rules are saved in JSON files, making them easy to adjust and reuse. This level allows for experimenting with new rule sets and observing unique behaviors beyond the traditional Game of Life
3. **Level 3:** In Level 3, the cells have more than two states. For example, a cell can be susceptible, infected, or removed. Cells change states based on probabilities and the states of their neighbors. This setup can model real-world phenomena like the spread of a disease. Users can define these probabilities and rules in JSON, making the system versatile for different kinds of simulations.

***PROJECT HIGHLIGHTS***

**Key Features**

* **Custom Rule Support:** Users can define their own rules for how cells evolve, using JSON to set conditions for cell birth, survival, and state transitions. Enabling users to explore not only standard rules but also innovative behaviors that may lead to unexpected patterns. Allowing users to alter these rules, they can discover entirely new patterns
* A grid of squares with different colors

  Description automatically generated**Visualization**: The project includes robust dynamic visualizations that display the evolution of cellular patterns over time. Grids are color-coded to represent different states, which provides users with intuitive visual feedback. This makes it easy to analyze how small changes in rules or initial conditions can lead to drastically different outcomes

A grid of squares with red and green squares

Description automatically generated

* **Efficiency**: The implementation leverages NumPy’s powerful array operations to handle large grids efficiently. Even with complex rule sets, the system performs smoothly. This makes it practical for running simulations that require a significant number of iterations or large grid sizes

**User Interaction for Non-technical Users**

* A screen shot of a computer

  Description automatically generated**Inputs:** The system is designed with non-technical users in mind. Users can specify the grid size, initial configurations, and custom rules through a straightforward interface. Rule files are provided in JSON format with clear documentation, allowing users unfamiliar with coding to easily make modifications (see image to the right). Probabilities for state transitions can also be specified through user-friendly prompts. The interface guides users with clear instructions for every field, and all inputs are checked in real time to ensure validity. For example, if a user enters an invalid grid size or rule parameter, the system prompts them to correct their input without requiring technical knowledge. Each graph or visualization is displayed one at a time, allowing the user to review and close each before the next one appears, providing a step-by-step interaction model that is easy to follow
* **Outputs**: To accommodate non-technical users, the system produces visual representations that are both intuitive and informative. Color-coded grids clearly display state transitions, making it easy for users to track changes over time. The step-by-step visualizations ensure that users can focus on one pattern or iteration at a time, making the results digestible and straightforward to interpret. Additionally, results can be exported in widely used formats such as CSV and JSON, enabling users to share or further analyze the data without requiring technical expertise

**Challenges Overcome**

* **Efficient Grid Processing:** Handling large grids was a challenge. Using NumPy, the system has efficient grid updates, even when dealing with millions of cells. This ensures that simulations run smoothly.
* **Flexible Rule Design:** Designing a flexible rule system that allows both beginner-friendly default rules and advanced user -defined rules was a challenge. The JSON-based rule storage format successfully addresses this, enabling users to easily modify and experiment with rules.
* **Multi-State Visualization:** Representing multi-state dynamics in a clear and intuitive way posed challenges, especially for users who could be unfamiliar with the concepts. By incorporating color-coded grids and step-by-step visual updates.
* **Scalability:** Ensuring that the system scales well for larger grids and more complex simulations required optimization.