PLANISUSS

Report by Ashley Andrea Squarcio

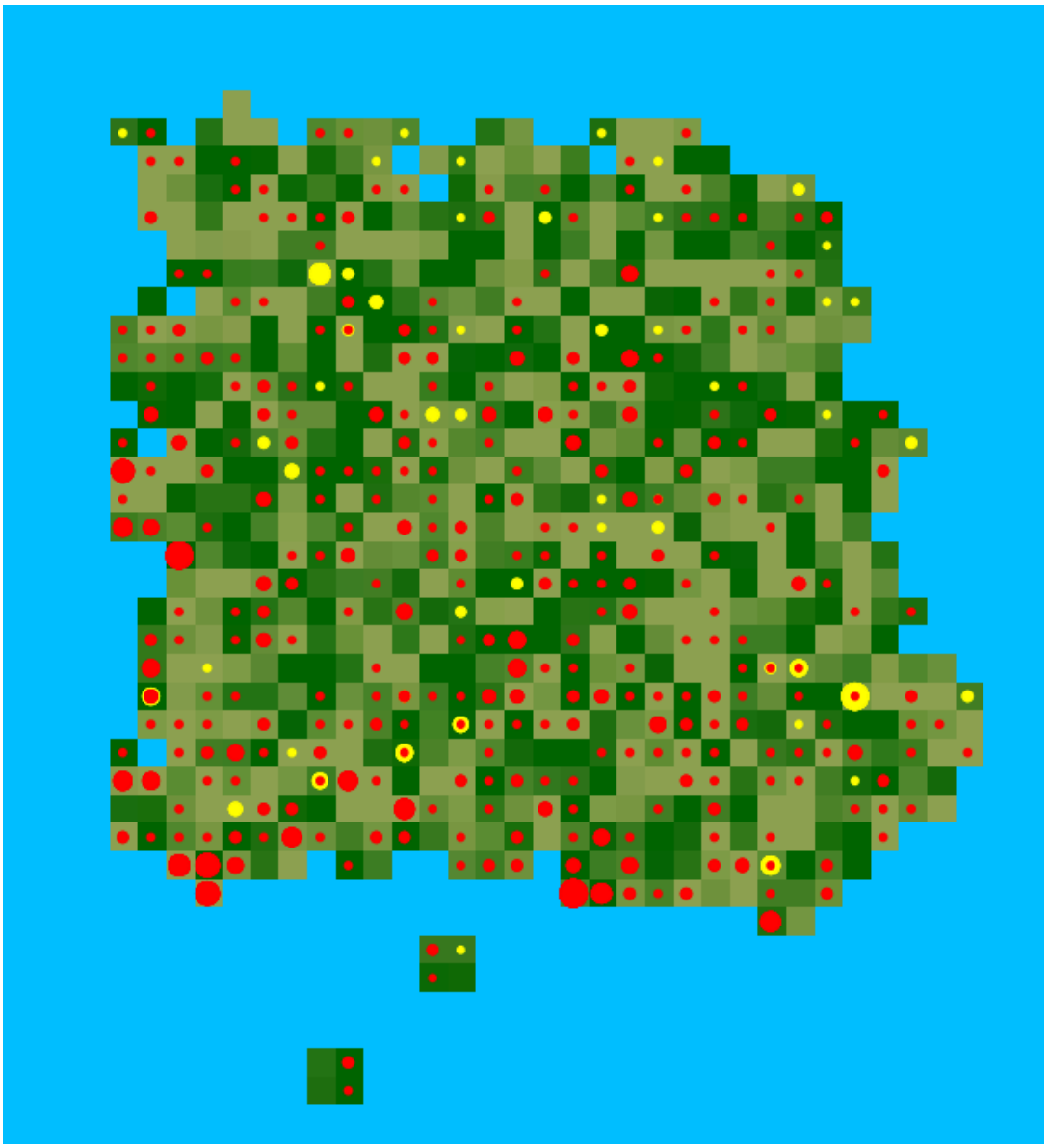

Table of Contents

1. Purpose of the Project

The Planisuss world is an interactive ecological model designed to explore and demonstrate the complex dynamics of a simplified ecosystem. Freely inspired by Wa-Tor and Conway's Game of Life, the simulation will leverage the *matplotlib* library for data visualization, providing insights into the intricate relationships between different species and their environments.

In this fictitious world, three main entities coexist:

- ➔ Vegetob, the primary producers, representing vegetation that grows and spreads across the landscape;
- ➔ Erbast, herbivores that graze on Vegetob and more in herds;
- ➔ Carviz, carnivores that hunt Erbast and move in prides.

The simulation models various ecological processes, like population growth and decline, predator-prey relationships, resource competition and social behaviors within species. Users are able to interact with the simulation through a graphical interface, allowing them to observe the real-time evolution of the ecosystem, adjust parameters to build different scenarios and analyze population trends and other key metrics through dynamic graphs.

2. Building the Environment

The foremost task to undertake in the creation of Planisuss was building a suitable environment for the creatures to inhabit. In my project, this aspect is primarily managed within the 'World' class. What follows is a detailed explanation of the process.

Firstly, the class initializes a grid of size 'NUMCELLS x NUMCELLS', which is represented by a NumPy array where 0 indicates water and 1 indicates ground. The 'ground_ratio' parameter determines the initial target number of ground cells.

The 'initialize_grid' method is then responsible for creating the terrain, following a multi-step process. It starts by setting the boundary cells to water, creating a water border around the world. After this operation, since I wanted to give the environment a more realistic feel, it goes on to shape the world as a natural-looking island. The method does so by creating a central block of ground

cells to form the core of the island, and thus expanding ground cells outward from the center, becoming more sparse as it moves towards the borders. Finally, it ensures that there are no completely isolated ground or water cells by calling the 'ensure_neighbors' method.

The 'ensure_neighbors' method is crucial for creating a cohesive landscape. It checks for ground cells with fewer than three neighbors and relocates them to areas with more neighbors, ensuring cell connectivity and creating a more realistic terrain.

After terrain generation, the constructor places the different species in the world at random positions. Vegetob is placed on 80% of the ground cells, while Erbast and Carviz are initialized randomly and their original amount oscillates between the 20% and the 50% of the ground cells..

As for world visualization, the 'create_world_image' function, while not part of the 'World' class, is essential for creating an image representation of the world, coloring water cells blue, ground cells brown, and varying the intensity of green for Vegetob-covered cells.

The final crucial function involved in the process is 'display_world', which works in conjunction with 'create_world_image' to provide a comprehensive visual representation of the Planisuss world. It does not only show the terrain and vegetation distribution, but also the current state of the Erbast and Carviz populations. The function begins by setting up a matplotlib figure and axis for displaying the map, then it calls 'create_world_image' to generate a base image of the world, eventually plotting the positions of the creatures. Erbast and Carviz are represented by yellow and red dots respectively, whose size corresponds to the density of creatures in the cell, calculated through the 'erbast_density' and 'carviz_density' methods. The 'display_world' function finally returns the figure, axis, image and scatter plot objects for Erbast and Carviz, setting up the foundation for dynamic updates to the visualization as the simulation progresses. This enables real-time observation of changes in the ecosystem, including, as we are about to see, population movements, growth of vegetation and interactions between species.

3. Life on Planisuss

3.1. Populating the World

Life on Planisuss begins with the initialization and positioning of creatures. The 'Species' class provides a base 'place_randomly' method that all species inherit and customize.

Vegetob is initialized with a random 'intensity', varying from 0 to 100. Intensity and position are stored in a dictionary thanks to the 'create_vegetob_data' method.

Erbast are created with an initial random 'energy', ranging again from 0 to 100, a fixed 'lifetime' defined by 'LIFETIME_ERBAST', 'age' 0 and a 'social_attitude' varying from 0 to 1. Carviz have the same attributes, except for their lifetime, which is defined by 'LIFETIME_CARVIZ', plus one more, 'move_time', to keep track of positions. All these attributes are stored in dictionaries through the 'create_erbast_data' and 'create_carviz_data' methods respectively.

At last, the general 'add_creature' method is overridden to actually append creatures to the population, 'creature_pop', and, in the case of Vegetob, to keep a record of intensities by position in the 'intensity' dictionary.

Since both species often prefer not to act as loners, when necessary, group actions are handled by processing herds and prides as whole entities. This is achieved by the 'group_by_positions' method, which, as the name suggests, groups creatures by their positions, creating a dictionary where each key represents an occupied cell in the map, and each value is a list of creatures located there. This approach allows for complex group behaviors like collective decision making, group fusion and coordinated actions, while still maintaining individual attributes for each creature.

3.2. Movement

The logic behind creatures' movements is complex, as both species consider both group and individual decisions and value their memories to make thoughtful decisions.

Herds make collective decisions based on their environment: they evaluate the nearby cells, retrieved through the 'get_neighborhood_positions' method defined in the 'World' class, considering Vegetob density and avoiding recently visited or dangerous areas. The herd's social attitude determines whether they move together or split.

Carviz use the same method to look for potential neighbor cells to move to, but also take into account potential prey-rich locations. They too exploit 'memory' to remember successful hunting grounds and avoid dangerous areas. The pride's social cohesion can influence their movement decisions.

For both species, memories can be accessed using the 'evaluate_cell_with_memory' method to make informed decisions about where to move, balancing immediate needs with past experiences.

The lower likelihood of individual movement following a group's decision to remain in the current cell is reflected in the multiplication factor added to the energy and social attitude thresholds for individual choices.

An interesting point to make is that, while movement is always limited to immediate neighbors, by controlling the 'NEIGHBORHOOD' parameter, which is made possible by the GUI, the user may let creatures make more informed decisions by being able to "see" a larger neighborhood. This way, they can look for the most suitable cell nearby, trace a trajectory in their minds and follow the path towards that best position.

3.3. Peaceful Soul or Bellicose Nature?

The other actions performed during a day on Planisuss heavily depend on the creatures' nature.

Herbivore Erbast show their peaceful nature by grazing, when movement doesn't occur, or by fusing with other herds in case multiple groups reach the same position. Grazing, that is, consuming Vegetob, provides energy to the Erbast, while decreasing the vegetation intensity in the cell. Herds' merging is only limited by available space, and is indeed favored, as it represents a convenient opportunity to combine memories and collaborate in a more considerate decision making process.

Carnivore Carviz, on the other hand, are of bellicose nature. When they meet, they may merge or fight depending on their sizes and social attitudes, and when they fight, matches are last-blood, seeing leaders from the two teams attack each other until annihilation. Prides also hunt Erbast present in their cell. The success probability of the hunt depends on the pride's size and their cumulative energy with respect to that of the strongest Erbast at that position. If successful, the energy of the dead Erbast is distributed among the pride members and their social attitude consequently increases. Failed hunts, on the contrary, reduce social attitudes.

3.4. The Circle of Life

At the end of the day, we have a showdown. Creatures that have reached their maximum age, as defined by 'lifetime', are terminated and removed from the population. However, every cloud has a silver lining: before removal, the creature spawns two offspring, distributing its energy between them. Children inherit a copy of their parent's memory, but a modified version of their social attitude, ensuring population renewal with a slight variation in traits.

Another common way for creatures to be terminated is being overwhelmed by Vegetob. This occurs when all neighboring cells contain vegetation at their maximum intensity and simulates death by overcrowding.

3.5 Helper Functions

Several helper functions support the functioning of the world, and I believe they are worth a brief mention. They are called inside various creatures' methods and help drive their decisions and actions.

'get_neighborhood_positions' finds surrounding cells within a specified neighborhood size, defined by the 'NEIGHBORHOOD' parameter at the outset of the project.

'evaluate_cell' determines the desirability of a cell based on the species we are interested in: Erbast look for vegetation-rich cells, while Carviz prefer positions with high prey density.

'erbast_density' and 'carviz_density' calculate the number of creatures in a given cell.

'check_cell_limits' ensures that cell populations do not exceed the specified 'MAX_HERD' and 'MAX_PRIDE' limits.

'is_valid_move' checks if a move is within the grid and on a ground cell.

4. A Typical Day of Planisuss Inhabitants

The core of the simulation progress through time is handled by the 'update' method in the 'World' class. This function shapes how one day on Planisuss looks like, performing the following steps: incrementing the day counter by one, updating Vegetob growth, moving herds and prides, allowing stationary Erbast to graze, fusing herds and managing merging or fighting for prides, allowing Carviz to hunt Erbast, spawning new offspring and finally checking for potential overwhelmed creatures. The 'validate_creature_positions' call at the end of the 'update' method guarantees that all creatures are situated at valid positions after all updates, maintaining the integrity of the simulation.

The visualization update process, which the 'update_world' function is responsible for, translates daily progress into visual elements that the user can observe. The function calls 'world.update()' to advance the simulation by one day, then creates a new image of the world using 'create_world_image(world)' and updates the image data of the plot. New creatures' positions are extracted from the updated world state, new sizes calculated based on the density of Erbast and Carviz in each cell, and the related scatter plot data is updated using 'set_offsets' and 'set_sizes'. Thanks to the repeated call of 'update_world' as part of the animation, each cell renovates the plot, creating the illusion of movement and change over time.

The separation of concerns between simulation logic and visualization lets the code structure be clear and maintainable, while providing users with intuitive visual representation of the evolving ecosystem.

5. Let's Play!

One of the most important personal goals for this project was to transform what could be a complex and potentially vapid simulation into an interactive, visually engaging and user-friendly experience. With this in mind, I realized a **GUI** implementation that significantly enhances the usability, accessibility and analytical capabilities of the Planisuss World simulation. Let's delve together into its various, valuable components.

The 'PlanisussGUI' class constitutes the core of the graphical user interface. It sets up the main window and manages the overall simulation display and control. The 'setup_gui' method creates the principal layout, including title, the simulation display area and buttons for pausing and resuming, terminating, restarting, speeding up or slowing down the simulation, displaying graphs and changing parameters. Canvas are employed for displaying the world map using matplotlib. Additionally, when the simulation is paused, any cell in the map can be clicked and the 'show_cell_zoom' method will display a zoomed version of the selected cell. This method employs .png images for both background materials and creatures, which I personally designed and implemented in the simulation with the aim of making the GUI more appealing and informative. Below the zoomed image, details of the cells are displayed, including the type, Vegetob intensity and number of Erbast or Carviz, if present, along with their attributes.

Methods in the 'GraphWindow' class create separate windows for displaying various statistics, such as the numerosity of Erbast and Carviz populations through time, the evolution of their energy levels and the Vegetob rate of growth. Hence, trends and patterns arising in the ecosystem can be analyzed and monitored. To access those graphs, the user can click on the **'Display Graphs'** button, which employs the 'GraphSelectionGUI' class to create a window for graph selection.

The 'ParameterChangeGUI' class builds a separate window for changing simulation parameters: video game looking sliders are implemented for dynamic adjustment of parameters, whose values are updated by the 'apply_changes' method. This concept enables experimentation with different scenarios without the need to restart the simulation: if one realizes a species is clearly

disadvantaged at a certain point in the simulation, they can decide to modify the parameters in an attempt to balance things out, and the outcome may change. This allows for major user participation, letting them decide how deterministic they want the process to be.

The 'update' method of the 'PlanisussGUI' class is crucial for reflecting the simulation's current state and managing its progression and termination. It updates the world state if not paused or terminated, renovates the data for real-time graphs and redraws the world map and creature positions.

Finally, the **main** block, which initializes the world and GUI, encapsulates the simulation setup and execution.

Other than promoting engagement and experimentation, the GUI, with its components' modular design, allows for easy addition of new features or modifications to existing ones.

6. Key Design Choices and Their Advantages

6.1 Superclasses

The simulation I developed makes use of a variety of strategies, designed to provide benefits of various nature.

Let's start with the use of superclasses. Their employment allows for the definition of common attributes and methods, which can be inherited by subclasses. This promotes code reuse by reducing redundancy, ensures consistent behavior across different child classes and simplifies maintenance, reducing the effort required for changes in common functionalities. Blatant examples of this are the 'Vegetob', 'Erbast' and 'Carviz' classes, all inheriting different attributes and methods from the superclass 'Species'.

6.2 Object-Oriented Design

When not exploiting superclasses, the code still relies on classes, employing an object-oriented design. This approach allows for modularity, as each class is responsible for a specific aspect of the simulation, making the code easier to understand and manage. In particular, the 'World' class models the virtual ecosystem, the 'Vegetob' class represents the food resources for herbivores and

the 'Erbast' and 'Carviz' classes give birth to the creatures in the simulation, 'PLanisussGUI' handles the main GUI, while 'GraphWindow' manages the creation and display of graphs. Another advantage of object-oriented programming is reusability, both across different parts of the code and in other projects. A clear example of this is the 'GraphWindow' class, which can be reused to create different types of graphs with minimal changes. Finally, the modular structure shows its worth even during the debugging and updating phases: if a bug is found in one part of the code, it can be easily fixed without affecting other parts, just like new features can be added by extending existing classes.

6.3 Real-Time Updates

Another valuable tool I opted for is using real-time updates and animations, which allow the user to see the immediate effects of changes in the ecosystem. This is achieved through the use of the 'animate' method in the 'GraphWindow' class and the 'update' method in the 'PLanisussGUI' class, updating the graphs and simulation state respectively. By observing how changes unfold over time, the dynamics of the ecosystem become more understandable, while the user interactivity and control is also enhanced through the ability to pause, resume, speed up, and slow down the simulation, making it all the more engaging.

6.4 Customizable Parameters

The choice to leave certain parameters customizable was key to make the user feel like they are really playing an active role in the simulation. They can experiment with different scenarios by adjusting parameters at runtime, exploring the effects of different variables on the ecosystem and enhancing their understanding of ecological dynamics.

By clicking on the **Change Parameters** button, after starting the runtime, the user is presented with a game-like window with the following parameters: 'Neighborhood', 'Max herd size', 'Max pride size', 'Erbast aging', 'Carviz aging', 'Vegetob growth', 'Erbast lifetime', 'Carviz lifetime', 'Erbast energy threshold', 'Carviz energy threshold', 'Erbast social attitude threshold' and 'Carviz social attitude threshold'. Each of these has a specific range of values to choose

from, allowing for a wide range of simulations while still adhering to some constraints. For more precision, social attitude thresholds for movement are displayed with a custom scale that includes decimal points, while the other attributes employ an integer scale.

7. Future Improvements

Overall, I believe the robustness and flexibility of this code makes it well suitable for modeling quite complex ecological systems like Planisuss. However, there are a few improvements I thought of while developing this project that I deem worth mentioning.

The first, more driven by the environments' aesthetics, would be to introduce 3D structures to enhance realism, provide a more immersive and realistic representation of the ecosystem and improve spatial understanding, providing a better grasp at the relationship between entities in the environment. Complex terrain features like hills, valleys and water bodies could be implemented, as well as vertical movement, which would make room for flying creatures or multi-layered vegetation. This would of course come at a cost, necessarily implying increased computational complexity, which may slow down the simulation.

The second, and personally more compelling, potential improvement consists of implementing machine learning and genetic algorithms for decision making and simulating evolution over generations. More complex and adaptive behaviors would be created and unexpected but effective emergent strategies could develop in a race for survival. The dynamics could also become more unpredictable and interesting to observe over time, as creatures learn from their environment and past experiences, mimicking real-world scenarios. Such an objective would equate to increased complexity in understanding and debugging creature behavior, as well as a significant rise in computational requirements, especially for large populations. Nonetheless, I'm convinced that, given the necessary resources, this would be a valuable opportunity to see advanced AI techniques first-hand, potentially facing unintended consequences and learning to deal with them.

8. References

The simulation draws inspiration from various ecological models and agent-based simulations. The implementation utilizes Python's scientific computing ecosystem, including **NumPy** for efficient numerical operations and **Matplotlib** for visualization. The graphical user interface is built using **Tkinter**, providing a cross-platform solution for user interaction.

The NumPy, Matplotlib and Tkinter documentations serve as valuable resources for understanding the technical implementation details.