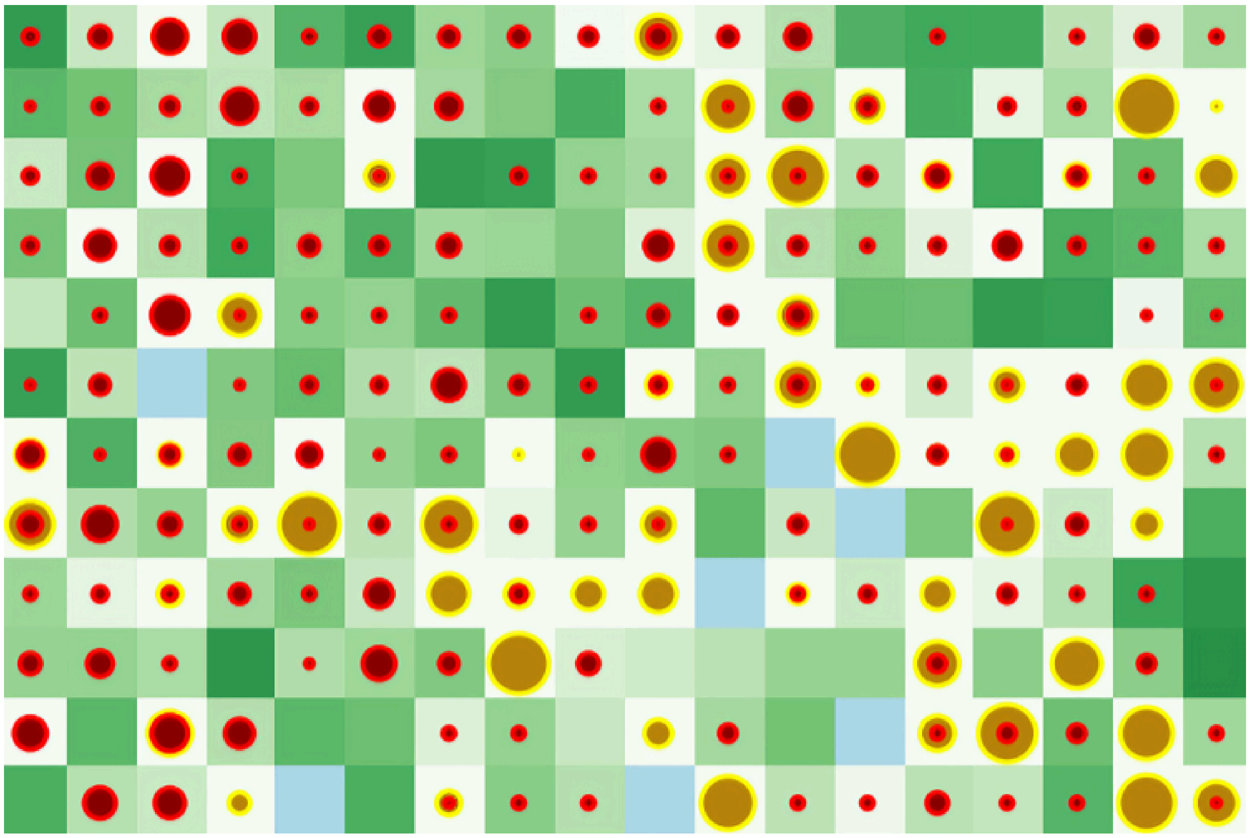

Planisuss

Ntegano Bahenda Yvon Dylan - 515657

Pietrasanta Sebastiano - 513054

509477 - Computer programming, Algorithms and Data structures., Mod. 1

Table of Contents

Introduction

The Planisuss project is a simulation designed to model a fictional ecosystem and study the interactions between different species within it. The primary goal is to define population dynamics and behavioral patterns through computational means. Using Python and specifically the Matplotlib and NumPy libraries, the project generates a dynamic and interactive simulation environment where species like Erbast and Carviz interact with each other and their surroundings.

Project Goals and Constraints

The project's goal is to build a simulation of a pseudo-naturalistic world, in the form of a grid where each cell represents a water or a land zone, and the inhabitants of this continent move following specific rules. There are three species of entities that refer to actual living beings: the Vegetob, which is a vegetable that constantly grows on each land cell; the Erbast, a herbivore that roams around the land eating Vegetob and forming herds; the Carviz, a carnivore which predates Erbast and organizes in prides.

The code's main objective is to handle this simulation by implementing a series of social behaviors for the animal species and at the same time respecting the limits posed by the ecosystem itself: the individuals have to be able to move freely on land, basing their decisions on how appealing their surroundings are, their attitude and their strength; each entity's energy must be balanced between moving or eating, and so activities like grazing and hunting have to be tuned with the energy consumption and gaining; additionally Carviz prides may end up fighting and this leads to the disappearance of one of the contender groups. All of these functionalities must take into consideration the life cycle of each individual, so that when all energy is lost or a certain age is reached they die and are replaced with their offspring, while Vegetob have to grow in a cycle every day, but must not exceed a maximum level of density.

The main problems arise from the interaction between the species, because to properly simulate an ecosystem like this one which is based on reality, it is required that there is no complete prevarication of one group on the other, and that the social interactions and attitudes are respected.

Probably the central challenge is constituted by the visualization part, which has to return a clear insight of the events, especially through the usage of graphs that update as time passes and must offer the possibility of interacting with the simulation itself, like slowing down or speeding up the process.

Assumptions, Methods, and Procedures

Design Choices and Motivations

The chosen spatial design is that of a grid that allows representing the world in a clear and effective way. The continent is divided into cells which enable us to handle movements and interactions by updating the state of individuals in each one of them. All of this proves effective in simplifying the spatial model and therefore leading to more efficient computation. The grid also provides an optimal visualization option, it is easy to understand and to be interacted with.

The time is based on discrete steps, each one representing a day: this choice reflects in some way the previous one, since it's also dictated by efficiency and effectiveness, in the sense that it facilitates the control of events by defining them in a constant but not continuous update of the cells.

Finally, the use of constants proved fundamental for flexibility, enabling easy adjustment of simulation parameters and therefore letting us perform experimentation and optimization in a better and more precise way.

Logical Structure of the Solution

The simulation's logical structure is based on the space-time settings described above. The grid initialization consists in assigning water and ground cells (being careful to have only water as border cells) and then populating land with our three species: as cited previously, Vegetob is defined by the density on each ground cell, while Erbast and Carviz possess more complex features like social attitude and can therefore decide whether to join a herd or pride or not.

The daily simulation cycle is built in the following way:

- **Growth Phase**, during which the Vegetob density in each ground cell is increased by a defined constant;
- **Movement Phase**, where the determination and execution of movements for Erbast and Carviz take place; this process considers multiple factors, like the distribution of resources in nearby cells, the choice of the herds and prides and the individuals' decision to follow the group or doing the opposite;
- **Grazing Phase**, consisting in the consumption of Vegetob by Erbast herds and individuals, a process that gives energy to the ones that decide to stay in their cell and eat for that time step;
- **Struggle Phase**, in which Carviz actions are defined; these comprehend the hunting of Erbast, giving energy to the successful predators, and the fights between prides, that lead to the complete disappearance of the loser of the two groups involved;
- **Spawning Phase**, when new individuals are introduced by reproduction mechanics based on the current population.

Implementation Details

The implementation of the elements described above is conducted through the use of classes and custom functions, all grouped in different main code blocks.

The first one defines the class `‘Cell’`, which not only specifies the nature (water or land) and content (Vegetob density, list of Erbast and list of Carviz) of the cell, but also comprehend all the actions that are carried out by the species in relation to the cell itself:

- `'grow'` function, the growing of the Vegetob which happens only on land and follows the constant value describing the amount of growth per day; it must not exceed a set limit (in our case 100);
- `'move_erbast'` function, defining the choice of Erbast individuals based on the combination of their social attitude (so relating them to the decision of the herd) and a random value; if the movement is chosen, then the individual will spend some energy to move to a neighbor cell; also in this case, the number of Erbast in one cell must not surpass the limit imposed by the `'MAX_HERD'` constant;
- `'move_carviz'` function, which follows a structure similar to the previous one, but adds the specification for prides' changes following fight results; as described above, the movement requires energy, and the prides have to accept a maximum of `'MAX_PRIDE'` individuals;
- `'graze'` function, simulating the grazing of Erbast in the cell;
- `'hunt'` function, simulating the hunting performed by Carviz;
- `'spawn'` function, defining the spawning and aging of Erbast and Carviz; the age is increased by 1 over each time step, but the individual is removed if their energy reaches zero; also the energy is reduced by the `'AGING'` constant every 10 time steps and finally if an individual's age exceeds or equals its lifetime and the current number of individuals is less than the maximum allowed (`'MAX_HERD'` or `'MAX_PRIDE'`), the individual is removed and two new ones are generated, with half the energy but same attitude value.

Following this initial block, we encounter the definitions of the `'Erbast'` and `'Carviz'` classes. Both classes are initialized with parameters for *`'energy'`* and *`'attitude'`*. Additionally, each class includes a lifetime attribute, set to the constants `'MAX_LIFE_E'` and `'MAX_LIFE_C'`, respectively, and an age attribute, which is initialized to zero.

We then find the `'fight'` function, which takes as arguments the two fighting prides and returns the winner one. The function calculates the total energy for both prides and determines the winning pride based on the ratio of their energies.

Specifically, it computes the probability of *'pride1'* winning as its total energy divided by the combined energy of both prides. A random number is generated to decide the winner, with *'pride1'* winning if the random number is less than this computed probability. If the total energy is zero, the winning probability defaults to 0.5. The winning pride's 'Carviz' instances then have their *'social_attitude'* attribute increased by 0.05. Finally, the function returns the list of 'Carviz' instances representing the winning pride.

We pass to the world itself with the 'Create_world' function, which generates a 2D array representing a world composed of cells. It iterates through each cell within a grid of a fixed size, based on the constant 'NUMCELLS', initializing each cell with its coordinates. The function then marks cells as water based on specific conditions: cells located at the edges of the grid are automatically marked as water, and this could happen to any other cell with a 5% probability. Each cell is represented by an instance of the 'Cell' class, which includes a boolean attribute *'is_water'* to indicate if the cell is water. Finally, the function returns the 2D array of initialized cells.

For the initialization we have the 'Initialize_world' function, setting up the previously created 2D array of cells by adding vegetation and populating it with Erbast and Carviz. The function iterates through each cell in the grid and for cells that are not marked as water, it sets a vegetation density value between 30 and 100. It then has a 40% chance to populate the cell with between 1 and 30 Erbast, each initialized with a random energy value between 50 and 80. Similarly, it has a 10% chance to populate the cell with between 1 and 20 Carviz, also each initialized with a random energy value between 50 and 80.

Fundamental for the movement simulation is the 'get_neighborhood_cells' function. It identifies and returns the non-water cells adjacent to a specific cell within a defined neighborhood range. To do so, it takes as input an object of the 'Cell' class, representing the central cell, and then it initializes an empty list to store adjacent non-water cells. After that, it iterates through a grid of size defined by the constant 'NEIGHBORHOOD' around the given cell's coordinates, skipping the cell itself. For each neighboring cell, it calculates the coordinates, retrieves the cell from the world array, and checks if it is not water. If that's the case, it is added to the list. Finally, the function returns the list of non-water neighboring cells.

All these parts combine in the `'simulate_day'` function to model a day's dynamics. For each cell in the world, the function performs the following actions

- **Vegetation Growth:** increments the vegetation density. If all neighboring cells have vegetation density above 100, it clears Erbast and Carviz from the current cell.
- **Erbast Movement:** Erbast move to a neighboring cell if it has space.
- **Carviz Movement:** Carviz move to a neighboring cell if it has space.
- **Grazing:** Erbast reduce the vegetation density by grazing.
- **Hunting:** Carviz hunt Erbast if both are present in the same cell.
- **Spawning:** new Erbast and Carviz are generated.

Finally we get to the visualization part, composed of the world grid, population graphs and interactive buttons.

To better grasp the density of Vegetob in each cell, we define the `'get_vegetation_map'` function, creating a 2D array representing the vegetation density across the entire world grid. It initializes a NumPy array of zeros with the same dimensions of the world. Then it iterates through each cell in the grid, setting the value to -1 for water cells and to the cell's vegetation density, described by *`'vegetob_density'`*, for non-water cells. Finally, the function returns this vegetation map.

To set up the actual visual representation of the world we use Matplotlib. This process includes all the aesthetic elements that are necessary for a clear visualization: the whole figure is set up in 2 parts, one for the grid and one for the graphs; background color is changed, a text to keep track of time is added and the vegetation map colors are decided; finally the vegetation map is plotted and the lists of Erbast and Carviz are initialized.

The `‘update’` function is the main protagonist of the visualization part, since it’s designed to refresh the animation frame by frame, simulating a day in the ecosystem and updating the representation accordingly. It starts by simulating a day using the `‘simulate_day’` function. Next, the vegetation map is updated with the latest data. The *`‘time_text’`* is updated to show the current day count and any previous scatter plots from the world grid are cleared.

The function initializes lists to store the x and y coordinates and sizes of Erbast and Carviz populations. It then iterates over each cell in the grid, counting the number of Erbast and Carviz. For cells containing these creatures, it appends their positions and sizes to the respective lists and keeps track of the total count of Erbast and Carviz. Their positions are plotted with sizes proportional to their numbers in the cell. The two creature types are distinguished by the color, with Erbast being yellow and Carviz being red.

The total counts of Erbast and Carviz for the current day are appended to the *`‘num_erbasts’`* and *`‘num_carvizes’`* lists, respectively. The population plot is cleared and the updated population data is plotted with lines for Erbast and Carviz. Titles, labels, axis limits, and ticks are set for graphs, and the legend is updated with a customized appearance. Finally, the function returns the updated grid, graphs and time for the animation framework to render.

To better interact with the simulation we set up interactive buttons to control the animation. Different callback functions for handling button actions are defined. The `‘on_click_start’` function restarts the animation by calling, while `‘on_click_stop’` pauses the animation. The `‘decrease_speed’` function decreases the animation speed by increasing the time interval between the frames and `‘on_click_slow_down’` uses this function to slow down the animation when the slow down button is clicked, adjusting the animation interval and restarting it. Conversely, the `‘increase_speed’` function increases the animation speed by subtracting 100 milliseconds from the speed variable, and `‘on_click_speed_up’` employs this function to speed up the animation when the speed up button is clicked, similarly adjusting the interval and restarting the animation.

We then generate the four buttons with specified positions and sizes on the figure. Each button is created using the `'Button'` class from `'matplotlib.widgets'`, with custom colors and hover effects, and is linked to its corresponding callback function using the `'on_clicked'` method.

The `'animation.FuncAnimation'` function is used to create the animation. It takes the following arguments: `'fig'` specifies the figure on which the animation will be drawn, and `'update'` is the function that updates each frame of the animation; the `'frames'` argument is set to the constant `'NUMDAYS'`, which defines the total number of frames in the animation, corresponding to the number of days to simulate; the `'interval'` is set to `'speed'`, which controls the time (in milliseconds) between frames, thus determining the animation's speed. The `'blit'` argument is set to `'False'`, indicating that the animation will redraw the entire frame instead of only updating parts of it, and `'repeat'` is set to `'False'`, meaning the animation will not loop once it completes.

Finally, `'plt.show'` is called to display the animation. This command opens a window showing the animated plot, allowing users to see the dynamic simulation of the ecosystem in real-time.

Tools and Resources Used

The code revolves around the usage of two libraries: Matplotlib version 3.4.3 and NumPy version 1.21.2.

Matplotlib, with its flexibility, offers lots of features for making many kinds of animated and interactive plots. It allows the detailed customization of plots, making it optimal for creating the animations and visual representations needed for this simulation. The library's support for interactive controls, like buttons for starting, stopping, and changing the speed of the animation, makes it more interactive and user-friendly.

NumPy is crucial in the efficient handling of arrays and matrices, which are the core of this simulation. Its optimized mathematical functions make complex calculations and data manipulations easier, allowing the simulation to run smoothly and efficiently.

Testing and Results

Testing consisted in running various scenarios with different initial conditions and parameters. Multiple simulations were conducted to observe how the system behaved under different setups. This included varying the starting populations of Erbast and Carviz, finding the best values of starting energy, adjusting vegetation density, and changing the probability of cells being marked as water.

Performance and accuracy were evaluated by observing the population dynamics and visual outputs of each simulation. The behavior of Erbast and Carviz, their interactions, and the resulting changes in vegetation density were taken into consideration when trying to ensure that the model was functioning as expected. Visual outputs, especially the population graphs, were used to check that the simulation produced consistent results over each run.

The results of the simulation were positive, successfully demonstrating interactions between species. The behaviors of Erbast and Carviz were accurately represented in the simulation.

The population trends and spatial distributions of both species were correctly visualized over time, with the former being tracked and plotted, showing how populations grew, declined, and moved across the grid and the latter displaying how Erbast and Carviz occupied different cells, interacted with their environment and with each other when in the same cell, and how these patterns stayed consistent over the course of the simulation.

Possible Improvements

Possible improvements for the simulation are the following:

- enhanced visualization, which could be implemented by providing more detailed and interactive visualizations, like allowing to focus on one zone of the map or one individual, and enabling the introduction of real time modifications to the ecosystem;
- optimization of the code, to improve computational efficiency, allowing for larger and more detailed simulations without compromising performance;
- addition of more complex behaviors and decision-making processes for the species, like improving the social structure of the groups and adding properties to the individuals, so as to make the interactions more realistic and dynamic.