Modelling ecosystem dynamics

10457059, Pau Petit Rosàs
Department of Physics and Astronomy, The University of Manchester
(Dated: April 24, 2024)

Using the external libraries CImg and Gnuplot an ecosystem is modeled. Evolution and mutations are simulated with the help of three rules that the ecosystem has to follow. The relations between species are calculated with the help of quadtrees. Once the simulation finishes, three plots are created displaying the change in the number of creatures, their average maximum health, and their average speed. Evolution is observed.

I. INTRODUCTION

An ecosystem model aims to recreate the behaviors present in nature, from the interaction of prey and predator to evolution itself [1]. In this model, each creature is given health, maximum health -the one which they are born with- and speed. For each iteration, their health diminishes by one, until they die. Additionally, they follow these straightforward rules:

1. If two creatures of the same species intersect, they both die. However, they also reproduce, passing their genes to the next generation.
2. If a predator intersects with a prey and the prey's speed is lower than the predator's, the predator will eat the prey and reproduce.
3. With more speed, the creature has to eat more frequently. If they do not eat in time, they lose health proportional to their maximum health.

These rules are designed to encourage eating and can be easily modified.
In addition, to generate the new attributes of the offspring, their speed and maximum health are generated following a Gaussian distribution that takes as the mean the parent's attributes. This aims to simulate mutations. The chance of mutating will be defined by the standard deviation of the Gaussian distribution.
Finally, plants are the only species that follow their own rules. When a creature dies from old age, a plant is born. There is no other way a plant can spawn. The ecosystem recreated is depicted in Figure 1.

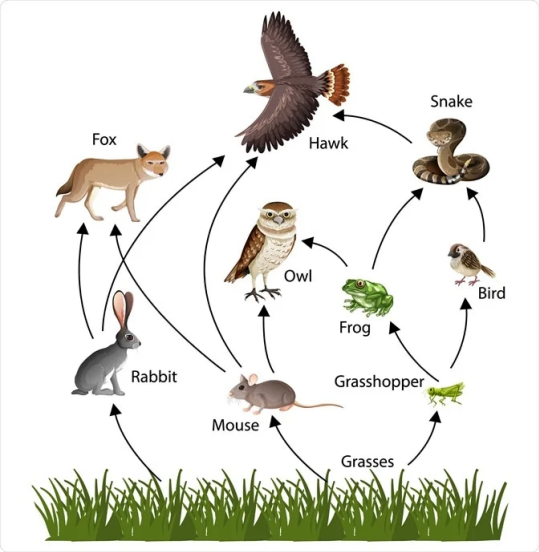

FIG. 1: Foodchain recreated.The arrows indicate the relation between predators and preys.

II. CODE DESIGN

A. Classes

The code is supported by four main classes: creature, ecosystem, rectangle, and quadtree. The last two will be discussed in the next subsection.
The creature class is an abstract class from which all the species of the ecosystem will be derived. Each creature has eleven protected variables, among which we find a pair of integers for the position, a vector of bools to check if the creature has eaten, and a static integer that tracks the number of creatures. Additionally, the class contains functions such as *move*, which randomly displaces the creature each generation, or *starvation*, which calculates whether the creature needs to lose health because it has not eaten.
The ecosystem class consists of two protected data members. Firstly, an integer that counts the number of deaths per generation. Secondly, a vector of smart pointers, pointing at different creatures -all of which are in the ecosystem-. In this case, the shared smart pointers are used. This class contains many functions to display the ecosystem. Nevertheless, the main functions that add functionality are *evolve* (which relies on *reproduce* and *createtree*) and *plantgeneration*. The latter ensures the plants are born following the rules described in the introduction. The former, on the other hand, iterates over all the creatures in the system, making them move, age, and triggering the starvation function. Furthermore, it removes the creatures with no health and calls the functions *createtree*, *reproduce* and *plantgeneration*.
To follow the rules described in the introduction, there is a need to check the position of each creature and compare it to all other positions. This would require a for loop -iterating over all specimens- inside a for loop -which,

again, would iterate over all animals-. In this case, the computational work is proportional to the number of creatures squared, and so it quickly blows up. This is a well-known problem in computer science, and the simplest solution is to implement a quadtree.

B. Quadtrees

The idea of a quadtree is simple. Divide the space into four sub-spaces and, instead of comparing each position against each other, iterate only for those creatures that are in the same subspace. This dividing into four sub-spaces can be done recursively, so that the final product is a tree data structure, with four children per node. This algorithm reduces the computational effort from n squared to a logarithmic time [2]. To implement it, a quadtree class is created. It contains a function to insert new points to the tree, a function to query for points in the same sub-space, and a function to draw the quadtree -which is not used in the final result, but was used for debugging-. Furthermore, each quadtree contains four smart pointers to other quadtrees to implement recursion.

Finally, to set the boundaries for each quadtree, a rectangle class is defined. The rectangle contains a position, its center, a width, and a height. In addition, it contains two main functions. One checks if the rectangle contains a point and the second verifies if a rectangle intersects with another rectangle.

C. Input validation, CImg and Gnuplot

The code asks the user for an initial configuration of creatures, and so the inputs need to be validated to be integers. To do so, a template is used. By comparing the type of the variable introduced and the input of the template, the integer type can be verified.
For the output, there is a need to convert integers to string. To do so, a namespace and a template are called, as can be seen below.

```
namespace patch
{
        template < typename T > std::string
        to_string(const T& integer)
        {
                std::ostringstream ostring;
                ostring << integer;
                return ostring.str();
        }
}
```

This is used to output the number of creatures.
The library CImg[3] is used to display the ecosystem. Each creature has a color associated and it is represented in a CImg display. Finally, the Gnuplot[4] library is used to plot the evolution of the number of creatures, the average maximum health of each specie, and its average speed.

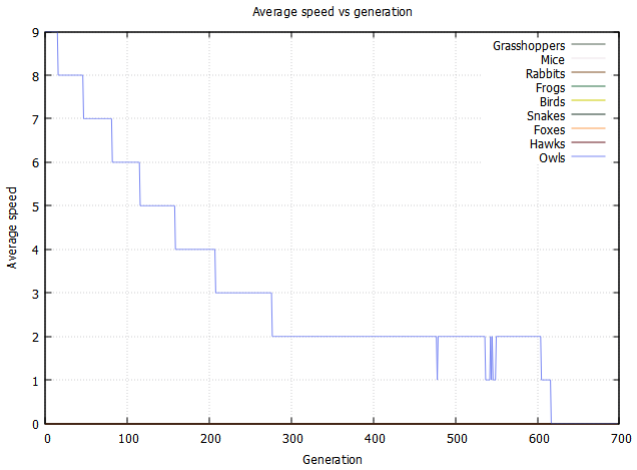

FIG. 2: Plot of the average speed with initial configuration of a thousand owls

D. Functionality and organization

The code is split into three cpp files and two h files. The file utilities.h contains the declaration of global variables, such as screen height and screen width, and the declaration of all the functions. These are defined in utilities.cpp. The second header file, foodchain.h, contains all the class declarations, which are defined in foodchain.cpp. Finally, main.cpp contains the code to run.
Compile-time polymorphism is used to overload the operator !=, to check if two pointers point at the same creature. In addition, runtime polymorphism is used to declare pure virtual functions in the base class creature, and inheritance let us define all the derived classes from creature.

III. RESULTS

Once an initial number of each creature is inputted, the simulation will start running. In case of overpopulation -and the computer not being able to process all the creatures in a sensible amount of time- there is the option to press E to kill half of the population.
All the initial variables such as each species' characteristic initial health, speed, and size are hard-coded, but can be modified at user's will. With the hard coded values, these are some cases and suggested configurations:

1. To check evolution, start with a thousand carnivore creatures of the same specie and let them die. The average speed is expected to drop, as the creatures

will live longer with less speed. This is seen in Figure 2.

2. To see predator-prey behaviour[5], start with plants and a plant eating species. Figure 3 is an example of this configuration.

3. Finally, a run with all the creatures and features is represented in Figure 4. It is recommended that, when suggesting the initial configuration, it contains more plant-eating creatures than predators.

It is clear that the model has a big factor of randomness involved, as repeating the simulation does not always give the same results.

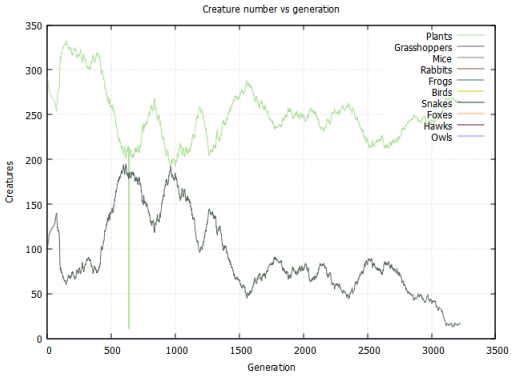

FIG. 3: Plot of the number of creatures versus generation. Initial configuration of 300 plants and 150 grasshoppers.

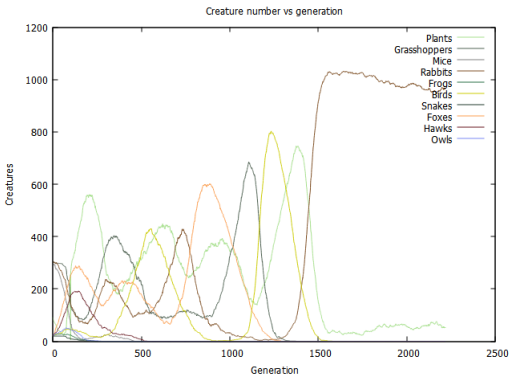

FIG. 4: Plot of the number of creatures versus generation. Initial configuration of 100 plants, 300 grasshoppers and mice and 20 creatures of the other type.

IV. CONCLUSION

In conclusion, the presented project, written in C++, uses the concepts of inheritance and polymorphism to model an ecosystem. It also uses two external libraries, Gnuplot and CImg, to display the simulation in an appealing manner and plot the information obtained from it.

There are many possible extensions for this project. For instance, an alternative nearest neighbour algorithm could be implemented to look for intersecting creatures. The rules of interaction could be modified and other variables such as visibility range or desirability could be introduced.

[1] G. William, *A guide to ecosystem models and their environmental applications* (Nature Ecology and Evolution, 2020).
[2] S. Har-Peled, *Geometric Approximation Algorithms* (American Mathematical Society, 2006).
[3] D. Tschumperlé, *CImg Library* (http://cimg.eu/index.html, 2003).
[4] Unknown, *Gnuplot* (http://www.gnuplot.info/, 2022).
[5] S. Kumar, *Chaotic behaviour of fractional predator-prey dynamical system* (Elsevier, 2020).