Predator Prey Model

Names Here

November 2011

Abstract

Contents

Chapter 1

Introduction

In this task we worked as a group to implement a 2D, sequential predator-prey algorithm with spatial diffusion in Java. We tried to design the program in such a way as to take advantage of Java's object-orientated, modular nature. Due to the large size of our group we also decided to develop a GUI for the program, although the code was designed in such a way as to be usable from the command line as well. ——is it?——

There is always some compromise between readability and performance in coding, and we tried to keep this balance in mind when designing our code. We made best use of class structure whilst still keeping our implementation of the given algorithm as efficient as possible.

The predator prey algorithm implemented in this project crudely models the interaction between population densities of different animals, specifically Hares and Pumas. Each population has a self-interaction coefficient (birth for Hares, death for Pumas) and a coefficient describing how it interacts with other species' populations.

These populations exist on a 'world' consisting of land and water, and are also able to diffuse across land squares with a rate determined by a diffusion coefficient. Thus, each population has N+1 coefficients which determine it's behaviour, where N is the number of animals.

Chapter 2

Model

$H_{ij}^{new} = H_{ij}^{old} + \Delta t(C_1 H_{ij}^{old} + C_2 H_{ij}^{old} P_{ij}^{old} + l(H_{i+1j}^{old} + H_{i-1j}^{old} + H_{ij+1}^{old} + H_{ij-1}^{old} - H_{ij}^{old}))$

for k=1:NumberOfAnimals
if (k=ThisAnimal) then
$N_k^{new} = N_k^{old} + \Delta t C_k(k) N_k^{old}$
else
$N_k^{new} = N_k^{old} + \Delta t C_k(k) N_k^{old} N_{ThisAnimal}^{old}$
end if
end

where

$$C_{Hare} = \begin{pmatrix} H & PH \end{pmatrix}$$

Chapter 3

Design & Implementation

3.1 Code

3.1.1 Structure

3.1.2 Algorithm

3.1.3 Input/Output

Computers cannot deal with continuous data; therefore, when modelling any kind of a system, discrete approximations must be used instead. This means that to see, for example, how a particular system evolves in time one needs to represent it by a sequence of events occurring one after another and separated by discrete time steps.
In our model the "world", or landscape, is represented by a rectangular grid of alterable dimensions $N \times N$, where N can be as large as 2000. We designed the code in a way such that it can read in an ASCII file that contains a set of numbers, those being either 0 or 1 and representing water and land, respectively, parses them and stores in a two-dimensional array. This means that the code can be used to model landscapes of different sizes with various distributions of land and water.
The boundary conditions are assigned to the model by setting the values for density of both, pumas and hares, to 0 at the edges of the grid.
The initial values for the densities of pumas and hares are chosen randomly from a range of numbers between ******dMin and dMax ****** and distributed across the land cells. All these tasks are implemented in methods within the InOut class.
The initial values for parameters that appear in the partial differential equations governing the evolution of the system with time can be set up by the user, who will be able to interact with the program by means of Graphical User Interface (GUI).

Specifically, it is allowed for the user to initialise the values for the birth, death and diffusion rates of both types of animals as well as the time step (dt), which controls how often the system is updated with new data. The user can also control the frequency of creating the output files, by choosing a value for the parameter T in the GUI window. The output gets created every T time steps. We have also included an optional "range" feature which allows the user to set a range of initial values for all the differential coefficients and an increment. In this case the code will run several times creating separate outputs for every simulation.
Alternatively, the user can execute the code directly from within the terminal using the command line ******list of arguments or a file??*******. In the next section we talk about the GUI in more detail.

The methods responsible for creating the output are included in the Output class. Depending on the user's preference, the code will create one output directory (for single initial values) or a number of such directories (for the range of initial values), i.e. one for each run. In both cases the user will be able to view the distribution of both populations across the landscape every T time steps. The form of the output is a number of *.ppm files which can be viewed as images, or "maps", similar to these showed in the ****FIGURE******.

The number of the *.ppm files created depends on the initial parameters specified by the user, dt and T. To maximise the level of output clarity we decided to make separate landscape "maps" for each type of animal, here pumas and hares. In each "map" the pixels corresponding to water cells are shown in blue, while the land cells are white. The variation in density of pumas/hares across the land cells is represented by different shades of grey, with black colour corresponding to the maximum value.
For each type of animal, every T time steps, the code also calculates the mean population density and writes it along with the corresponding time into a file.

3.1.4 GUI

3.2 Tools

3.2.1 SVN

3.2.2 Makefile

3.2.3 Unit Testing

Every class of the project form a functional unit according to the design elaborated at the beginning of the
has been tested using the Java JUnit framework
The libraries necessary to the tests are thus accessed from the test cases using Java import statements and extending the TestCase superclass in the class declaration:

Listing 3.1: Test case headers

```
1 import org.junit.Before;
2 import org.junit.Test;
3 import junit.framework.TestCase;
4
5 public class TestAnimal extends TestCase {
6 ...
```

Then the test cases, for each class, follow the normal directives in use for Unit testing with JUnit. The most relevant methods were submitted to thorough tests, where their properties, return values and correct functioning were evaluated by comparing the actual results they provide with the expected ones. These procedures were implemented by test methods in each test case using the format required by the JUnit standards.

Every test case included a setup() method that builds the initial object and its different parameters. Directives like @Before and @Test for every method declaration were thus inserted in the code were appropriate. The functioning of a method is then assessed comparing the expected results with the actual ones with JUnit testing methods such as AssertEquals(), AssertNotNull(), etc.

Listing 3.2: Use of JUnit directives in test cases

```
1 @Before
2 public void setUp() {
3     ...
4     testAnimal = new Animal(numbAnimals);
5     testAnimal.setName("Puma");
```

```
6       testAnimal.setDiffCo(diffCoIn);
7       ...
8   @Test
9   public void testAnimal() {
10      assertNotNull(testAnimal);
11      assertNotNull(testAnimal2);
12  }
13  ...
```

Chapter 4

Performance Analysis

4.1 Testing

Tests were preformed in order to profile the code's performance and scalability. The unix *time* command was used to time the code runs, with 'user' and 'system' time being summed to find the total computation time.

The relationship between total computation time per cell and the number of cells was measured in order to quantify the overhead needed by the code (see Figure 4.1). We found that, below a grid size of $\sim$100 by 100, overhead became important, whereas at larger grid sizes the computation time scaled linearly with the total number of cells. Any scaling worse than linear (such as total computation time being proportional to the NumberOfCells$^{1.2}$ would be a huge (and unnecessary) inefficiency, especially with larger grids.

Some overhead is inevitable in any code, but the amount of effort put into simplifying a problem (such as creating arrays of neighbours) should depend on the expected size of the problem. Our code could probably have had less overhead, allowing it to run faster with smaller grid sizes, but this would likely led to it having a worse (linear) scaling with cell number.

4.2 Analysis

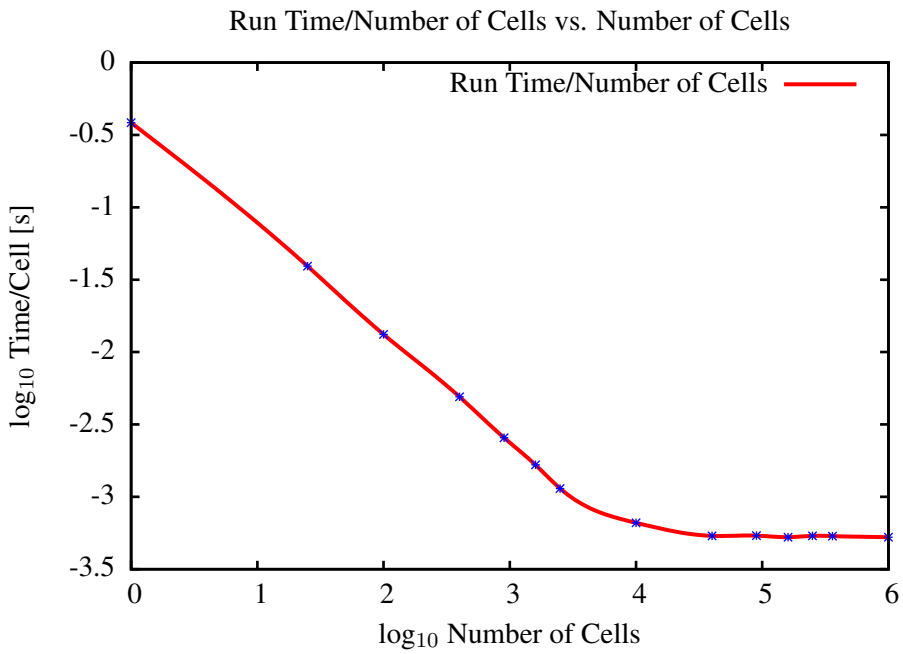

Figure 4.1: This is a graph of total time spent on each cell, which shows that overheads such as array initialisation and output (i.e. things not directly involved in the simulation) take up a significant fraction of computation time with grids smaller than $\sim$100 by 100.

Chapter 5

Conclusions

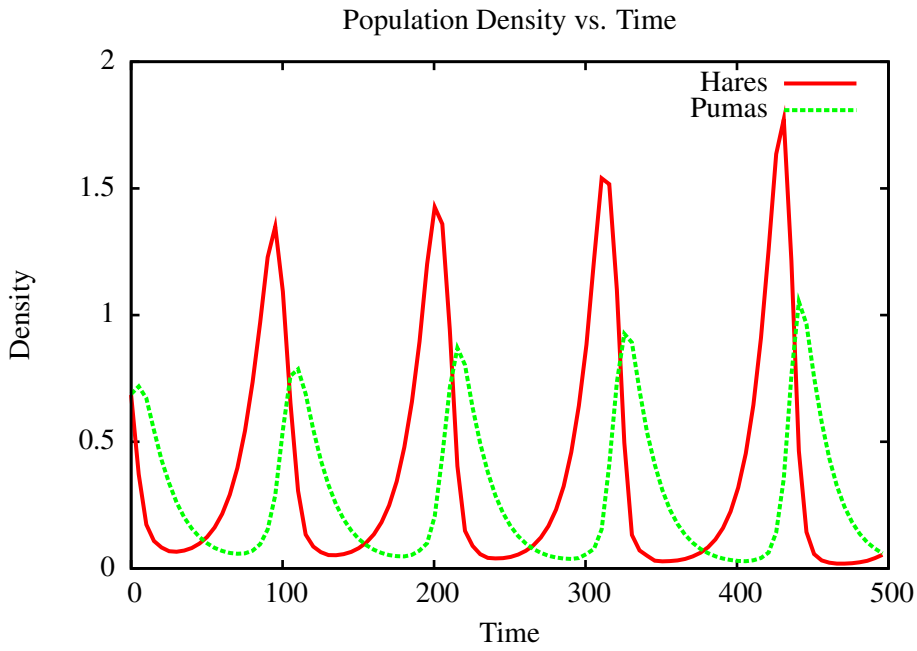

Figure 5.1: Population density vs. time for Hares and Pumas. This periodic behaviour is typical of predator-prey interactions.

Chapter 6

Group Evaluation