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 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 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 in 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 its 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

The language our group decided to code the predator-prey model in was java. Using Java gave us easy access to numerous tools such as Javadoc and Swing for creating GUIs, and the main advantage for most of us was the familiarity of working with object oriented code design. Object orientation provided us with the opportunity to design the central workings of the code with extensive extendibility, leaving just the user interface specifically designed for the problem at hand. The main code was split into Input, Output, GUI, Animal and Grid Algorithm classes classes which covers the different requirements of the program. A main class, PredPrey, governs the creation of occurrences of all of the other classes in the order required. When running to program there is an option to run with or without a GUI, this is implemented by either passing the input file names in the command line when running the program or not. If the arguments are passed, the input parameters are read from the file, if they are not then the user can input them in themselves. The parameters required by the user are the coefficients that govern the central update equation, the time step, diffusion rates of Pumas and Hares and the differential interaction coefficients between the animals, all described in ***section***. The input class is responsible for reading the file which holds where the land and water are. It reads the information from the file into a two dimensional array and from there calculates the number of nearest neighbours each cell has and holds that in a new two dimensional array. Water cells are held as -1 to differentiate between land cells with no neighbours and water. This array of neighbours is all that is required by the rest of the program. The Animal class holds it's name and all the information required about each animal and their interactions with others. Upon creation an object of the animal class is passed the number of animals in the simulation and the size of the map; initialising the arrays to hold the animals density distribution and interaction coefficients to the correct size. The values are then passed from the GUI/input file using setter methods, to set the differential coefficients, diffusion coefficients and time step for the animal. The Animal classes main function is that it holds the method which is called to update the densities in a cell on the grid. This class has been designed so that a program can be extended from this one to take any number of animals interacting to first order with each other. The Grid Algebra class governs methods for updating the animal densities across the grid. The constructor takes an array of animals passed from the main class and the 2 dimensional array of neighbours passed from the input class. Upon creation the GridAlg class initialises the densities of each animal in each grid cell to a uniformly distributed random number between 0 and 1, if different density distributions were required for each animal it would be easy to edit the code to accommodate this. A couple of different methods of updating the grid are held in the class, either updating in parallel, for all animals (all of the old densities are replaced by new one's at the same time) or updating the animals one by one. Other update methods which would be interesting to investigate would be updating the cells one by one and updating random cells and or animals. In this experiment only the fully parallel update is used for investigation. The output class takes the density distribution of the animals every ***time steps, calculates and outputs the average density as well as an image of the density distribution. All that is passed to methods in this class from the program is the density array of an animal and a file name for saving as output.

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 Input 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. In the next section we talk about the GUI in more detail.
Alternatively, the user can execute the code directly from within the terminal by specifying two arguments. The first argument required is the name of the file containing initial values for all the parameters, and the second is the name of the input ASCII file that will be used to create the grid.

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

Most of the group members (DID WE ALL USE ECLIPSE?) used the Eclipse IDE to develop the program. Eclipse provides many useful features that aid the development process including an extensive debugger, realtime error checking, comprehensive autocompleting and support for JUnit testing. In addition to Eclipse a number of other tools were utilised, details of which can be found below.

3.2.1 SVN

We chose to host the SVN repository using a Google Code project. The home page can be found [here]. Doing so allowed us to make use of the many excellent tools that are available for managing such projects; the most obvious example being the ability to quickly and easily browse all of the revisions from [r1 to current] via the web interface. The fact that the code was hosted externally on a 24/h server meant that it could be accessed from anywhere at any time. There was also no issue with compatibility: Some of the group members opted to use the SVN Eclipse plug that interfaces seamlessly with the Google Code while others chose to use the more traditional approach of checking code in and out via the command line. Another invaluable feature was the ability to compare side by side any and all changes between revisions. See for example [the changes made to Output.java in revision 29]. This feature is compatible with all text files, and so also proved useful for the group editing of latex files.

3.2.2 Ant File

3.2.3 Unit Testing

Every class of the project form a functional unit according to the design elaborated at the beginning of the project. In order to ensure the software's good functioning and correct behaviour, all classes have been tested using the Java JUnit framework with test cases to tackle programming errors and code bugs.

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

```
import org.junit.Before;
```

```java
import org.junit.Test;
import junit.framework.TestCase;

public class TestAnimal extends TestCase {
...
```

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 includes 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

```java
@Before
public void setUp() {
    ...
    testAnimal = new Animal(numbAnimals);
    testAnimal.setName("Puma");
    ...
@Test
public void testAnimal() {
    assertNotNull(testAnimal);
    assertNotNull(testAnimal2);
}
...
```

Where possible and where the tests were failing, we corrected and bettered the code until all tests passed correctly. We have employed several different methods and tools to debug the code, mainly with Eclipse IDE's debugger and JDB (Java Debugger) at the command line, that enabled us to verify the variables in use in classes and methods are holding the correct desired values during runtime. We have in this manner corrected many mistakes and wrong operations until the software was performing as expected. We also have used certain programming techniques such as Test Driven Development, unfortunately in just a few classes, although the procedure would deserve more time and practice to achieve a reasonable degree of maturity and efficiency to prove a powerful tool.

Chapter 4

Performance Analysis

4.1 Testing

Tests were performed 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

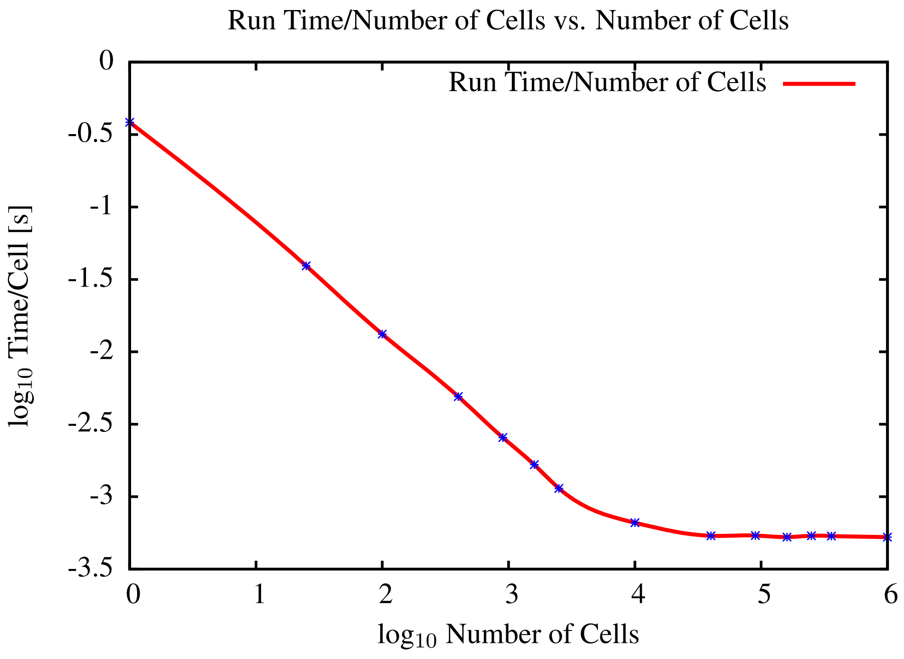

Figure 4.1: This is a graph of total time spent on each cell, which shows that overheads such as array initialisation and output (i.e. things not directly involved in the simulation) take up a significant fraction of computation time with grids smaller than $\sim$100 by 100.

Chapter 5

Conclusions

The Google Code SNV combined with the ability to compile and run Java code on any platform provided a universal code base that could be accessed from anywhere at any time.

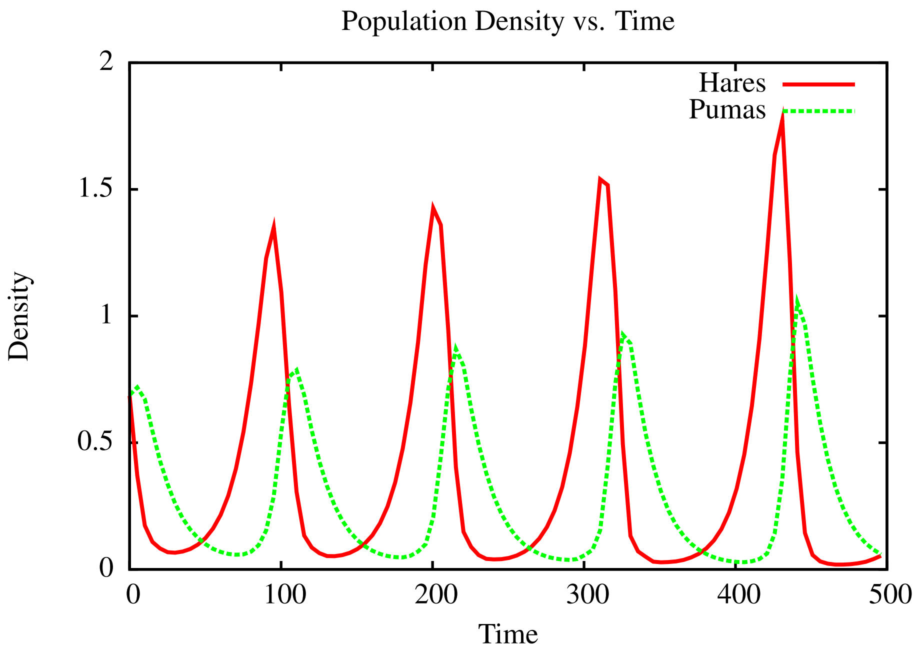

Figure 5.1: Population density vs. time for Hares and Pumas. This periodic behaviour is typical of predator-prey interactions.

Chapter 6

Group Evaluation