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. Initially, pumas and hares are randomly distributed across the land cells. All these tasks are implemented in methods within the MapReader 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 below.

Spatial variation of hare density after 50 time steps.

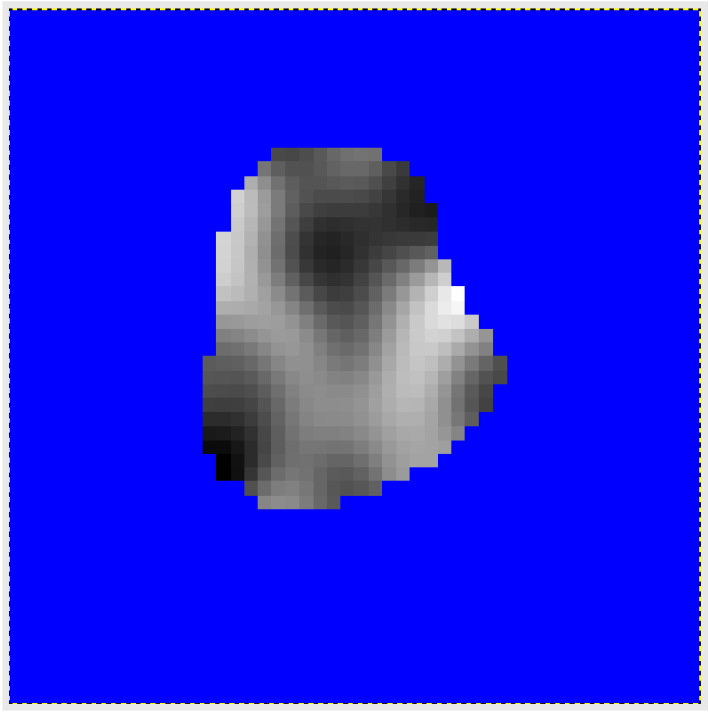

Figure 3.1: This figure shows one of the output files created during a simulation using a 100×100 grid and default initial conditions. Various shades of grey represent the density of hares after 50 time steps, with a time step set to 0.4.

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

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

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

```
1 @Before
2 public void setUp() {
3     ...
4     testAnimal = new Animal(numbAnimals);
5     testAnimal.setName("Puma");
6     ...
7 @Test
8 public void testAnimal() {
9     assertNotNull(testAnimal);
10    assertNotNull(testAnimal2);
11 }
12 ...
```

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 Performance Analysis

Because most optimisation in Java is done at runtime using Just-In-Time (JIT) compilation, there is not much to choose from in the way of compiler flags. The way in which a problem is laid out is always very important, however; our main aim in performance testing was to identify inefficiencies within the code itself, such as unnecessary loops and calls to methods.

Profiling of the code was done mostly with NetBeans and the unix *time* command. Parameters were varied to asses the effect of a number of factors, most notably overhead, output write time, and the effect of map size and complexity.

One of the main things which we identified very quickly as a bottleneck was the output . Fortunately, implementing a bufered writer is very easy in Java; this alone increased performance by a factor of 5-6 (see Figure 4.1).

In Figure 4.2 we attempt to quantify the overhead in our code. This is done by calculating the total time spent on each cell in a simulation, relative to the size of the simulation. With this metric, the code actually becomes 'slower' for smaller simulations. This is because a significant fraction of the simulation time is being spent on finding arrays of neighbours and other 'non essential' tasks. The graph shows that our code only begins to run as efficiently as possible once grids of $\sim$100x100 are used, above which the simulation time scales almost exactly as the number of cells (the flat part of Figure 4.2).

A 'perfect' algorithm would spend the same amount of time on each cell regardless of grid size, but our code expends some effort on simplifying the problem at the start of a run by creating the neighbours array. This allows it to be slightly faster on large grids, but adds overhead which becomes non-negligible for small simulations.

The main conclusion we reached in our testing is that output, espcially large amounts of plain text, is *expensive*. The performance of the program could be dramatically increased by having the output in a more efficient format. This can be shown more simply just by running the code with an

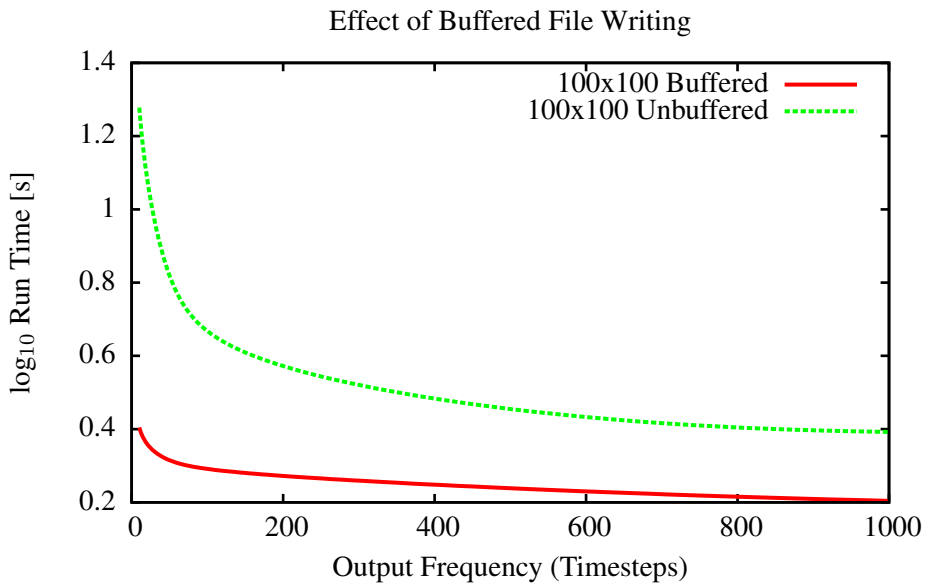

Figure 4.1: This graph shows the advantage of using a buffered file writer over repeated calls to the printf() method. The data was generated from a simulation using a 100x100 grid, but results will be similar for all grid sizes, since write time and computation time (neglecting overhead) both scale linearly with the number of cells.

output frequency of 20 timesteps compared with an output frequency of 2000 timesteps (a single output for a t=0..500, timestep=0.4 simulation). This give run times of $\sim$38.0 and $\sim$31.5 seconds respectively, even with buffered output. This output then accounts for $\sim$ 17% of the computation time of a simulation, which obviously not ideal.

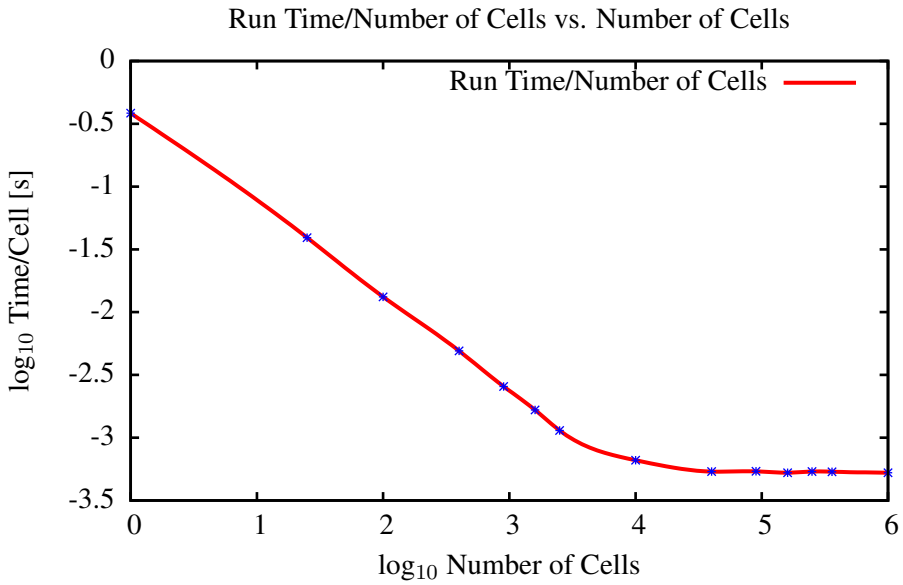

Figure 4.2: This is a graph of total time spent on each cell, which shows that overheads such as array initialization take up a significant fraction of computation time with grids smaller than $\sim$100 by 100.

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.

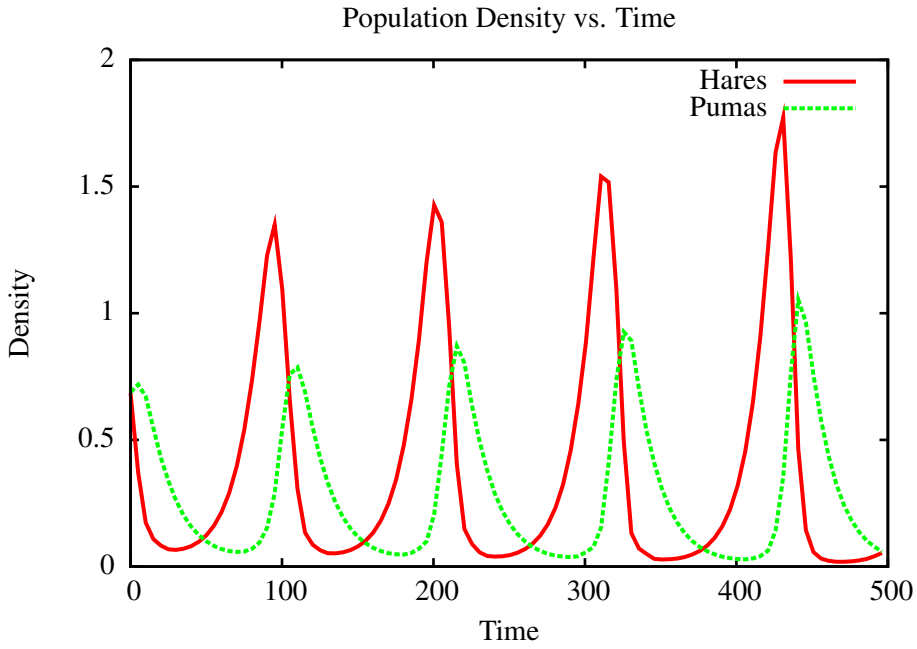

Figure 5.1: Population density vs. time for Hares and Pumas. This periodic behaviour is typical of predator-prey interactions.

Chapter 6

Group Evaluation