Predator Prey Model

Names Here

November 2011

When writing this report, a lot of emphasis was put on interactivity, so, a reader would benefit from reading an electronic version (PDF) rather than a paper one

Contents

Chapter 1

Introduction

The problem was as follows:

- An island represented by a rectangular domain, where every grid point is either land or water
- Density of prey (hares) and predators (pumas) are changing with time depending on parameters r,a,b m, k, l.[1]
- differential equation governing changes in population density where H is prey population density, P is predator population density, Δt is discreet timestep size:

  $$H_{ij}^{new} = H_{ij}^{old} + \Delta t(rH_{ij}^{old} + aH_{ij}^{old}P_{ij}^{old} + k((H_{i+1j}^{old} + H_{i-1j}^{old} + H_{ij+1}^{old} + H_{ij-1}^{old}) - N_{ij}H_{ij}^{old})$$

  $$P_{ij}^{new} = P_{ij}^{old} + \Delta t(bH_{ij}^{old}P_{ij}^{old} - mP_{ij}^{old} + l((P_{i+1j}^{old} + P_{i-1j}^{old} + P_{ij+1}^{old} + P_{ij-1}^{old}) - N_{ij}P_{ij}^{old})$$

- Aim to simulate a grid up to 2000x2000 points updating populations for a certain time.

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.

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.

[1] Where r is prey birth rate, a is predation rate, b is predator birth rate, m is predator death rate, k is prey diffusion rate, l is predator diffusion rate

Chapter 2

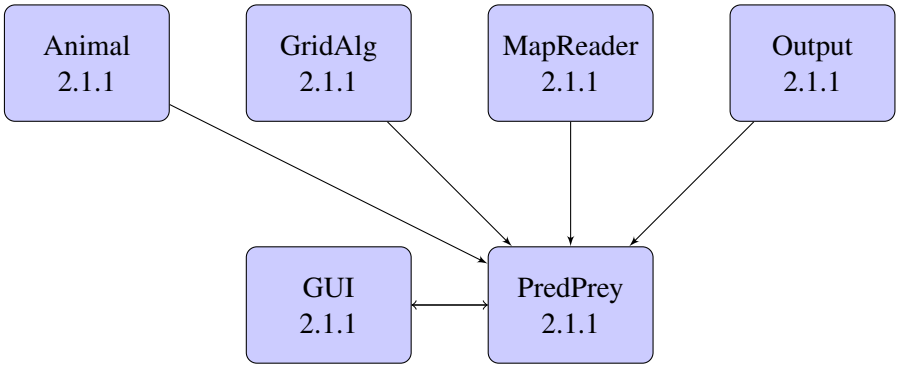
Design & Implementation

2.1 Code

2.1.1 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.

Classes Flow Chart



Animal

The Animal class holds the information about the animal; it's density distribution and how it is updated relating to other animals. It is called in the PredPrey class where it' values are initialised from those taken from the GUI or input file.

GridAlg

The Grid Algebra class governs methods for updating the animal densities across the grid. The constructor takes the array of animals passed after creation in the PredPrey class and the 2 dimensional

```
double diffusionRate
// The coeficients controling the effect of other animals on the animal
double diffCo[]
```

To illustrate more clearly what values diffusionRate and diffCo[] take, they are shown for the simple hare puma case that was used for the entirety of testing.

Listing 2.2: Behavioural coefficients for the hare instance of animal.

```
diffusionRate = k
diffCo[0] = r   // How hares effects the hare
diffCo[1] = -a  // How pumas effects the hare
```

Listing 2.3: Behavioural coefficients for the puma instance of animal.

```
diffusionRate = l
diffCo[0] = b    // How hares effect the puma
diffCo[1] = -m   // How pumas effect the puma
```

The governing class (PredPrey) creates an array of Animals, passing each their diffusion rate, and the array diffCo dictating how the of the other animals effect them. Armed with these tools, the Animal objects are able to iterate the equations shown in the model themselves. As long as the constants governing the effect of each animal on others are known, the size of the array of Animals can be increased to as many as desired. At each time step, the code loops over the animals, invoking the calcNextDensity() method. Part of this method is shown in listing 3.4.

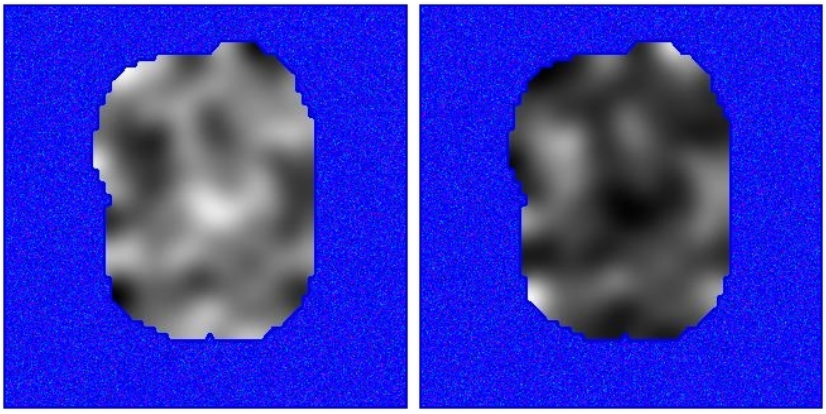
Listing 2.4: The key part of the computational kernel within animal - called when the density for a single cell is to be updated to the next time step.

```
// Loop over the total number of animals/behavioral coefficients.
for (int k = 0; k < animals.length; k++) {

  /*
  Add the contribution to the new density depending on the weather
  the animal is this or not.
  */
  if (animals[k] == this) {
    newDensity += dt*getDiffCo()[k]*oldDensity;
  }
  else {
    newDensity += dt*getDiffCo()[k]*animals[k].getDensity(y, x)*oldDensity;
  }
}

// Add the contribution based on the animals diffusion rate
...
...
...
```

2.2 Input/output

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 PDE's 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 a 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.



(a) $t = 360$ (b) $t = 416$

Figure 2.1: This figure shows examples of the output files created during a simulation using a 500×500 grid and default initial conditions. Various shades of grey represent the density of pumas after 900 (a) and 1040 (b) timesteps, with timestep size 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.

2.2.1 GUI

GUIs are not used in HPC as they use memory and slow down computations and are generally not necessary as all the interactions with HPC machines is done through a command line. However, our group has chosen to code in Java, and GUI is one of Java's strengths as it should look the same on every machine. So it was decided that we are not taking advantages of our chosen language unless we create a GUI. GUI created is responsible for the input parameters of the simulation. Initially, it would just take the values and run the simulation. However, in later) versions, *range* option was added to be able to run multiple simulations changing one of the parameters. The frame is divided into three parts (all on separate frames): input, input2, button. Input holds labels and text fields for parameter inputs. Input2 holds second text field and radio buttons to chose which parameter to range over, is not visible unless range is selected. Button holds start button, choice between range and preset values and parameter values that are general like input file, steps before output. GUI is still very simplistic and could be enhanced more but was not due to time constraints.

2.3 Tools

2.3.1 IDE

IDE stands for 'Integrated Development Environment', which is widely used in modern program development, especially in object-oriented projects. Because Java is a completely object-oriented programming language, IDEs became an indispensable tool during this project. Most of the group members 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 in the following sections.

Compared to Eclipse, Netbeans is a new IDE for Java developed by Sun, which provides more powerful debugging and performance scanning tools. The performance analysis section was held on Netbeans IDE by Netbeans Profiler.

2.3.2 SVN

We chose to host the SVN repository using a Google Code project. The home page can be found at http://code.google.com/p/ps-predator-prey/. 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 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 r29[1]. This feature is compatible with all text files, and so also proved useful for the group editing of latex files.

2.3.3 Ant File

To automate the building of the software, we have coded an Ant file using the XML language. The Ant file contains the appropriate sections that ensures compilation uses the correct files in the right packages. In order to maintain portability, we included a lib/ directory that contains libraries necessary to the Ant program, so it needs not search the local file system. The scripts also produces a .jar file containing the whole project for easy transportation.

2.3.4 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.

[1] http://code.google.com/p/ps-predator-prey/source/diff?spec=svn29&r=29&format=side&path=/code/Output.java

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 2.5: Test case headers

```
import org.junit.Before;
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 2.6: Use of JUnit directives in test cases

```
@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 3

Results

3.1 Output Analysis

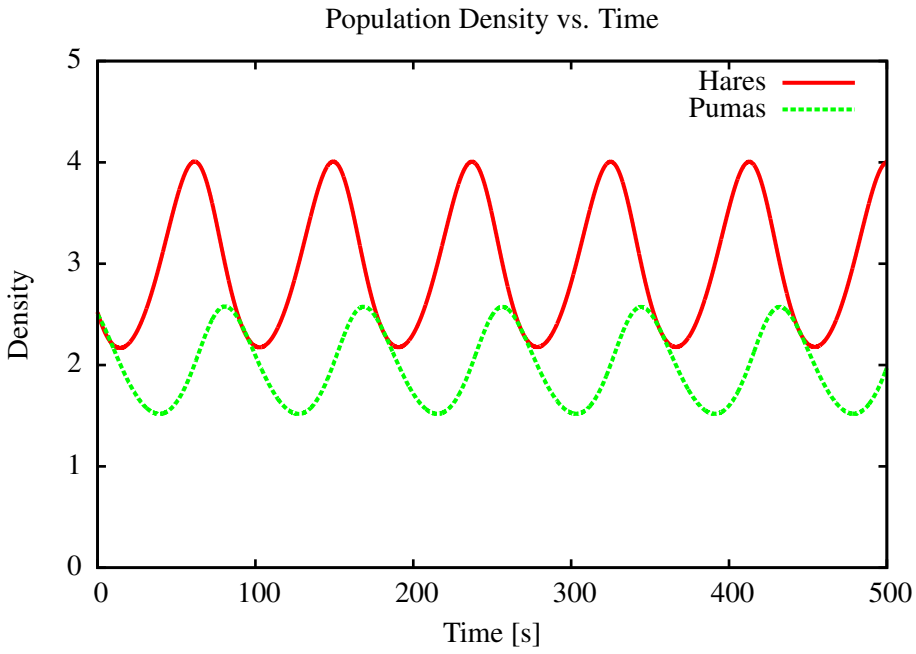


Figure 3.1: Population density vs. time for Hares and Pumas. This periodic behaviour is typical of predator-prey interactions.

3.2 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, so our main aim in performance testing was to identify inefficien-

cies 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 3.2).

The performance of the program could be increased even further by having the output in a more efficient format. This can be shown more simply by running the code with an output frequency of 50 timesteps compared with an output frequency of 5000 timesteps (a single output for a t=0..500, timestep=0.4 simulation). This give run times of $\sim$21.5 and $\sim$17.0 seconds respectively, even with buffered output. This output then accounts for $\sim$ 17% of the computation time of a simulation, which is obviously not ideal.

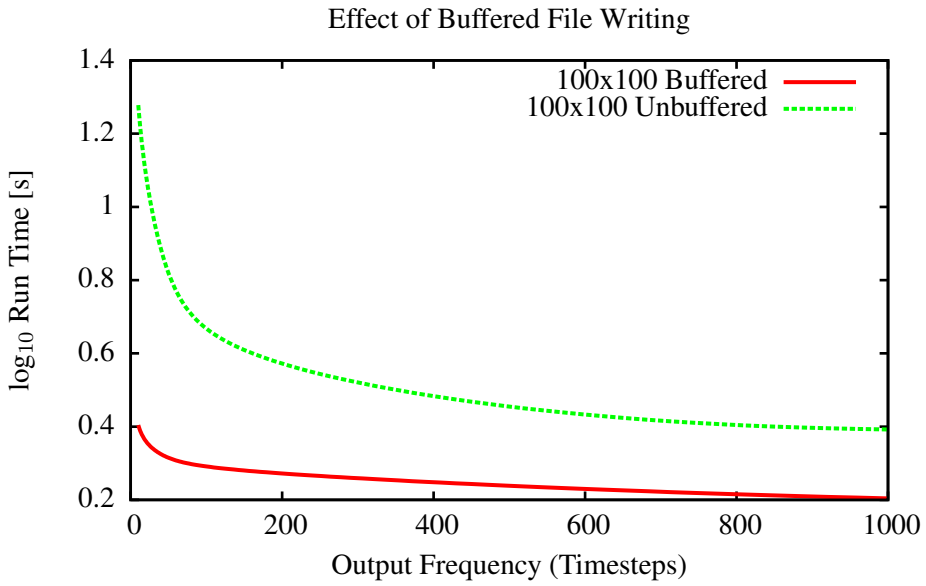


Figure 3.2: This graph shows the advantage of using a buffered file writer over repeated calls to the printf() method. The data was generated from a simulation using a 100x100 grid, but results will be similar for all grid sizes, since write time and computation time (neglecting overhead) both scale linearly with the number of cells.

In Figure 3.3 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 3.3).

It can be seen from Figure 3.3 that, for large simulations, the total time per cell per timestep is constant. The value on the graph is roughly -3.27, although this is total time per cell. Thus, $10^{-3.27}/1200$, where 1200 is the number of timesteps, gives us the time per cell per timestep: $\sim$450 nanoseconds.

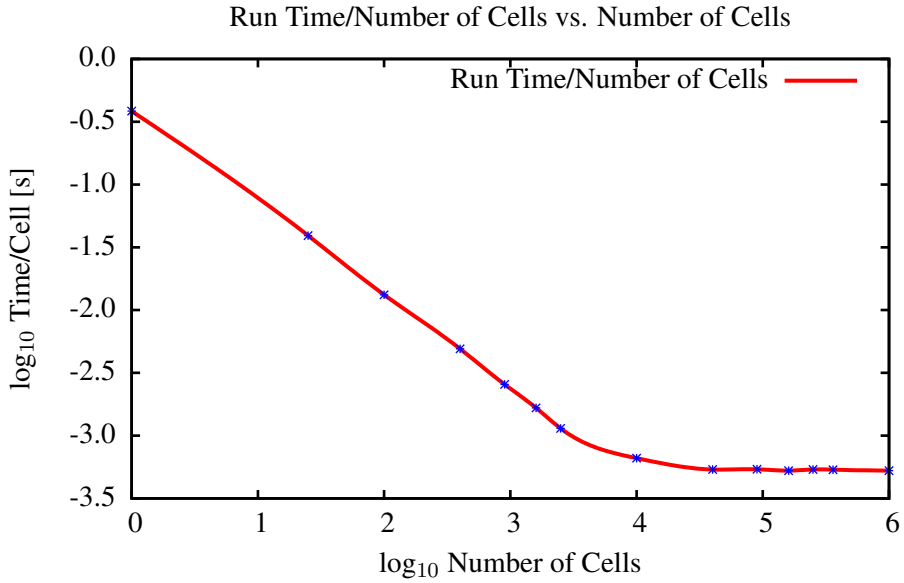


Figure 3.3: This is a graph of total time spent on each cell, which shows that overheads such as array initialization take up a significant fraction of computation time with grids smaller than $\sim$100 by 100.

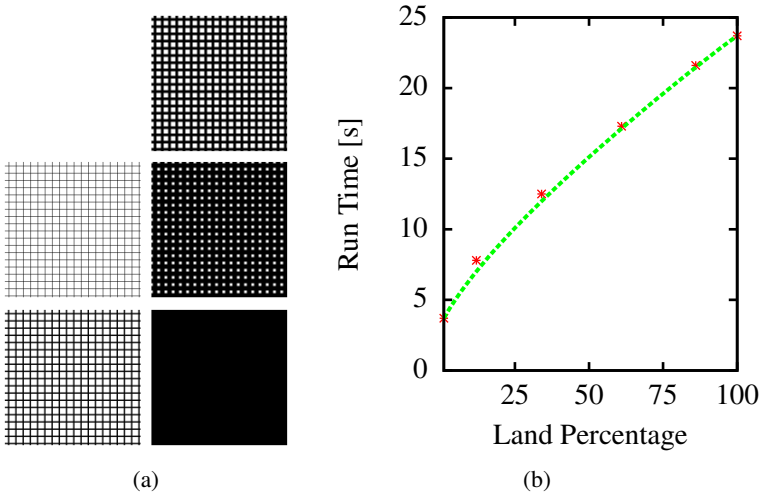


Figure 3.4: This shows the effect of fill percentage on run time. The six panels in a) are the input maps used, all of which were 300x300. As expected, the computation time increases fairly linearly with the amount of land, since the bulk of the main algorithm is only run for land cells.

Chapter 4

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.