



# Coursework for Programming Skills

Nikilesh Balakrishnan

Mark Florisson

Dante Gama Dessavre

He Li

Shun Liang

Sinan Shi

November 9, 2011

# Contents

# 1 Introduction

The object of this coursework is to implement a sequential version of a two-dimensional predator-prey model with spatial diffusion. The governing equations for this problem are:

$$\frac{\partial H}{\partial t} = rH - aHP + k \left( \frac{\partial^2 H}{\partial x^2} + \frac{\partial^2 H}{\partial y^2} \right) \quad (1)$$

where  $H$  is the density of hares (prey) and  $P$  the density of pumas (predators).  $r$  is the intrinsic rate of prey population increase,  $a$  the predation rate coefficient,  $b$  the reproduction rate of predators per one prey eaten and  $m$  the predator mortality rate.  $k$  and  $l$  are the diffusion rates for the two species.

Even though this is not a simple problem to analyze from a theoretical perspective, a lot can be learned from a simplification of it. If we assume that both initial densities of hares and pumas are uniform, then the diffusion term in equation (1) has no effect, so the problem is reduced to:

$$\frac{\partial H}{\partial t} = rH - aHP \quad (2)$$

$$\frac{\partial P}{\partial t} = bHP - mP \quad (3)$$

This are the Lotka-Volterra equations for prey-predator systems. After doing some mathematical analysis using the parameters provided in the handout, there are two stable points. The first is (0,0) which is a saddle point attracting vectors in the y axis and repelling them in the x axis. The second is (3,2) which is a neutrally stable point, which means that cycles are formed around this point with a counter clockwise direction. The vector field and a sample cycle is shown in Figure 1. All this analysis proved useful during testing of the code.

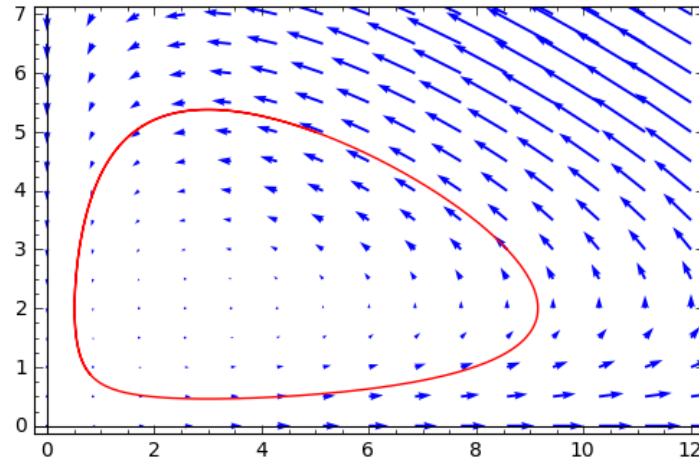


Figure 1: Vector field of equation 2

## 2 Planning and Task Division

The Systems Modeling Language standard was utilized for designing the software, since it is a modification of UML that can be much better adapted to non object oriented software.

The block diagram of the system can be seen in Figure 2. There are four main blocks or components: an input/output (I/O) library, the partial differential equation (PDE) computation kernel, the main simulator and a unit testing component.

The I/O library (Figure 3) consists of two main elements. Firstly there is the read function which accepts the matrixes that correspond to the island, hares and pumas. The second one is the write function that handles the creation of the `ppm` files that contain the output for visualization. It also contains two auxiliary components alongside an error handling element.

The PDE component (Figure 4) consists mainly of the compute function that implements the approximation formula for estimating the solution to the partial differential equations that model the problem.

The main component takes care of parsing the user input and printing the usage, but its main purpose is to utilize the I/O and PDE components in order to solve the problem. The internal block diagram can be seen in Figure 5. In there a flow diagram of how the main component can also be seen.

Finally, the figure 6 details de unit testing component, whose main element is similar to the one in the main program's component, but it has additional testing elements and is completely automated, so it doesn't need to parse the user input.

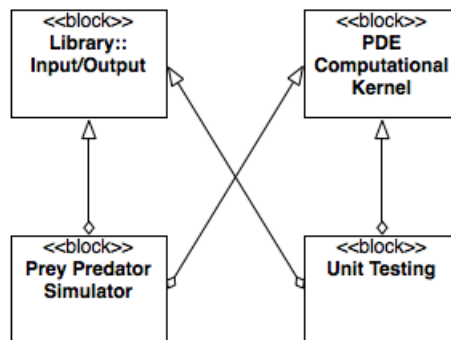


Figure 2: Vector field of equation 2

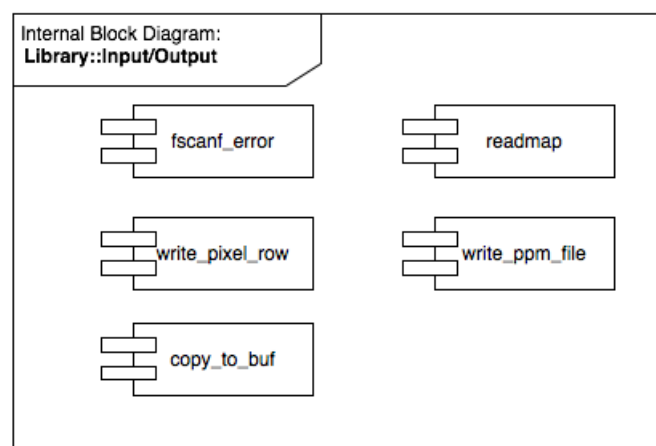


Figure 3: Vector field of equation 2

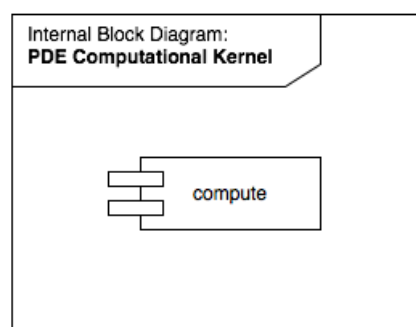


Figure 4: Vector field of equation 2

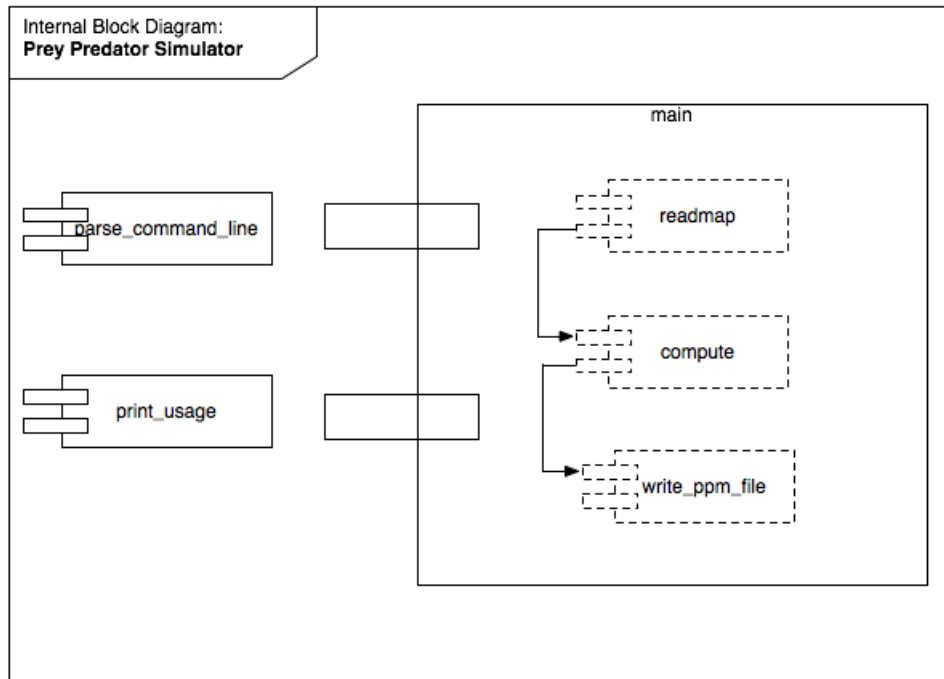


Figure 5: Vector field of equation 2

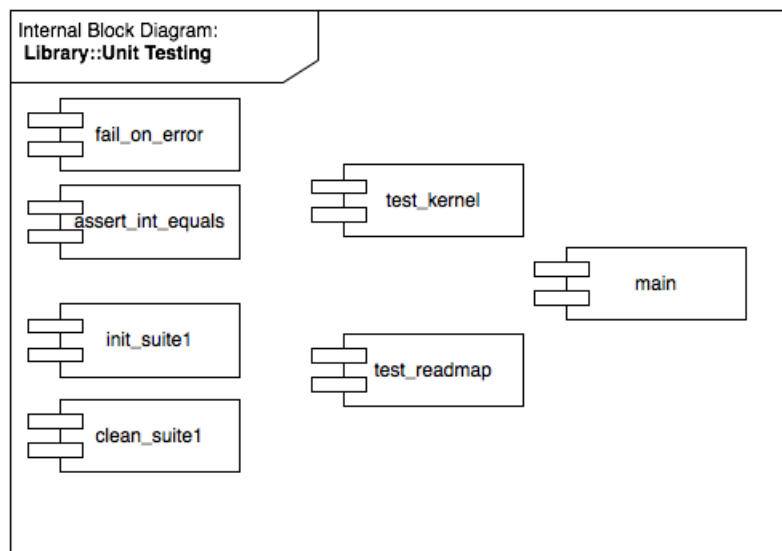


Figure 6: Vector field of equation 2

## 2.1 Task Division

Work on the components was divided as follows:

Input/Output	Mark and Nikilesh
Computational Kernel	Shun and He
Error Testing	Sinan and Dante

## 3 Implementation and documenting

The implementation was made in the C language, with an additional Matlab program to generate the test cases.

### 3.1 Data Structures

The map is modeled as a grid of land/water cells and implemented as two-dimensional *C* array. The population densities of predators and preys are modeled in the same way. The `Real` type defined in a macro represents the value in the density grids, so different floating point precision can be specified in compile time.

A struct, `EquationVariables`, was created as a wrapper for the parameters of the equations.

### 3.2 Computational Kernel

The computational kernel consists of the `compute` function. The parameters of the function are: the predator and prey density grids, the grid of land/water cells and a pointer to a `EquationVariables` struc.

Two  $2000 * 2000$  size global grids are used temporarily to store the value of new predator and prey densities. For each land cell in the density grid the values of the adjacent cells are read, which then are used for the calculation of the new values (along with the equation parameters). On the other side water cells are ignored. After the new density is computed the value is stored into the temporary density grid. Once all new densities are computed, the new values are assigned back to the original predator and prey density grids from the temporary grids.

The computation is performed until the time, given by the user, is reached.

### 3.3 Data Input and Output

The `readmap` function is used to read in a two dimensional array of integers. The first line specifies the length of the rows and columns, and any subsequent lines specify the values for the array. This is useful to read in files that specify the landscape.

### 3.4 Error Handling

The file `include/_errors.h` has the project-specific *errno*s. These can be returned from a function to indicate which error occurred. The caller can then act based on this value, or retrieve and print an error message associated with that `errno` (`puma_strerror`). If the error was caused by the OS, the functions return `PUMA_OSError`, for which message retrieval is delegated to the function `strerror`.

Also a simple logging utility was implemented that can write debug and error messages. The debug messages will be enabled when the code is compiled with the `-DDEBUG` flag.

## 4 Testing

### 4.1 Unit Testing

The testing framework CUnit was used for performing unit tests. The basic functionality of the framework was utilized, which provides a non-interactive output to `stdout`. The basic structure of CUnit is a hierarchy of test registry, suite, and test case, which are initialized before the test, and cleaned up afterwards.

#### 4.1.1 Input and Output Testing

Input and Output were tested in two different ways. The output was tested manually since it was easy to provide a data file and visually check if the result is the expected one.

The input was tested using the CUnit framework by reading the identity matrix and checking the resulting matrix in the program.



### 4.1.2 Computational Kernel Testing

## 4.2 Script Testing

Wrote perl script to compare C program results with results produce by the Matlab program.

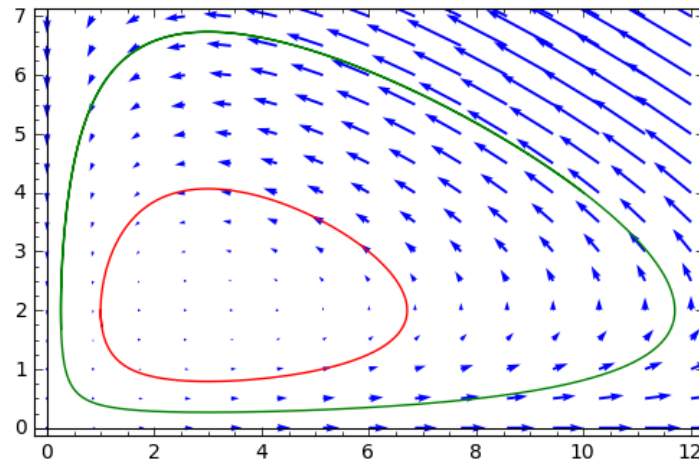


Figure 7: Vector field of equation 2

## 5 Deployment and Performance Analysis

## 6 Conclusions

:O :S :P :D

## References

- [1] Hoppensteadt, F., *Predator-prey model*, Scholarpedia, 1(10), 1563, (2006)
- [2] F.Bloggs. *1993 Latex Users do it in Environments* Int. Journal of Silly Findings. pp 23-29.