Coursework for Programming Skills

Nikilesh Balakrishnan
Mark Florisson
Dante Gama Dessavre
He Li
Shun Liang
Sinan Shi

November 11, 2011

Contents

1 Introduction

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

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

$$\frac{\partial P}{\partial t} = bHP - mP + l\left(\frac{\partial^2 P}{\partial^2 x} + \frac{\partial^2 P}{\partial^2 y}\right) \tag{2}$$

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 equations (1) and (2) has no effect, so the problem is reduced to:

$$\frac{\partial H}{\partial t} = rH - aHP \tag{3}$$

$$\frac{\partial P}{\partial t} = bHP - mP \tag{4}$$

These 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.

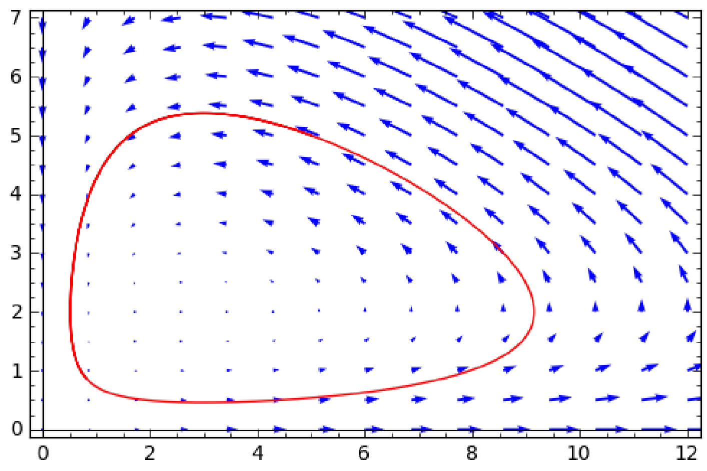


Figure 1: Vector field of equation 2 and example of a solution

2 Planning and Task Division

The Systems Modeling Language standard was utilized for designing the software, since it is a modification of UML that is more suitable for non object oriented software.

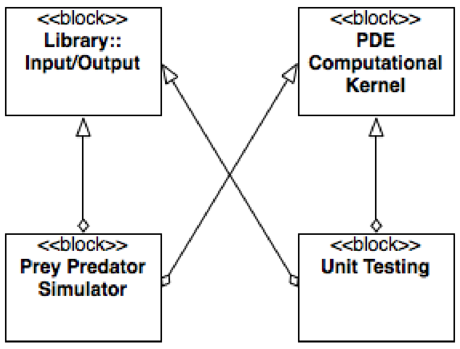
The block diagram of the system can be seen in figure 2. There are four main blocks or components: an input/output (I/O) library, the partial differential equation (PDE) computation kernel, the main simulator and a unit testing component.

The I/O library (figure 3) consists of two main elements. Firstly there is the read function which accepts the matrices 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 primary 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, which contains a flow diagram of how the main component was designed.

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



<<block>> Library:: Input/Output
<<block>> PDE Computational Kernel
<<block>> Prey Predator Simulator
<<block>> Unit Testing

Figure 2: System Block Design

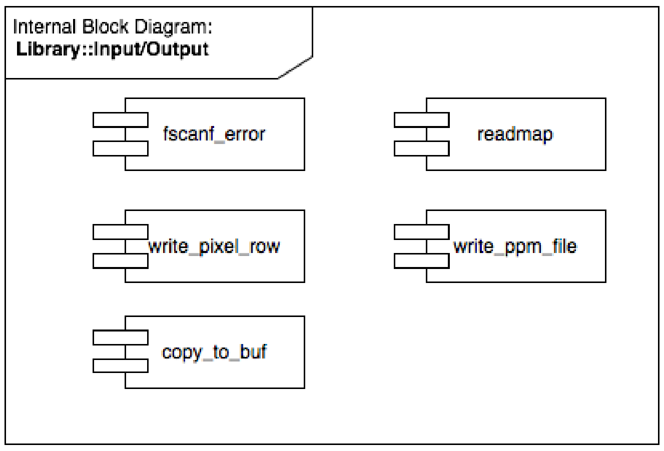


Figure 3: I/O Internal Block Design

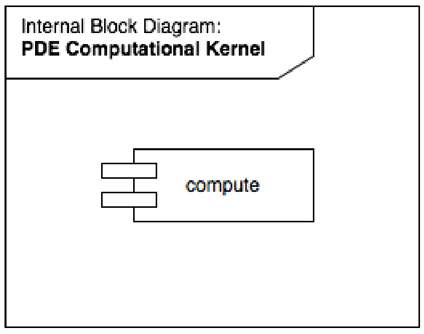


Figure 4: PDE Computational Kernel Internal Block Design

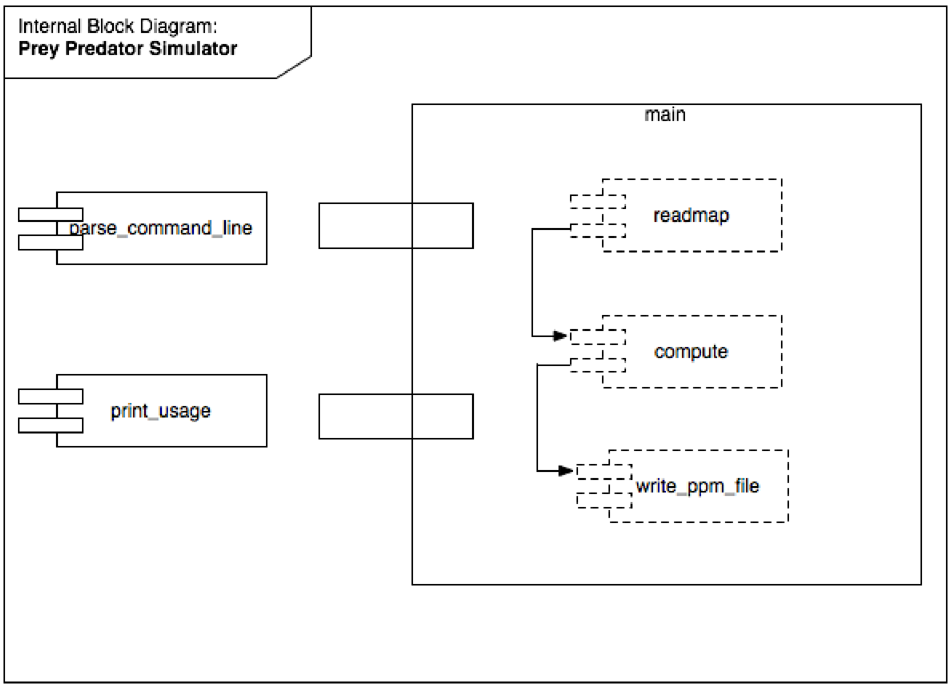


Figure 5: Prey Predator Simulator Internal Block Design

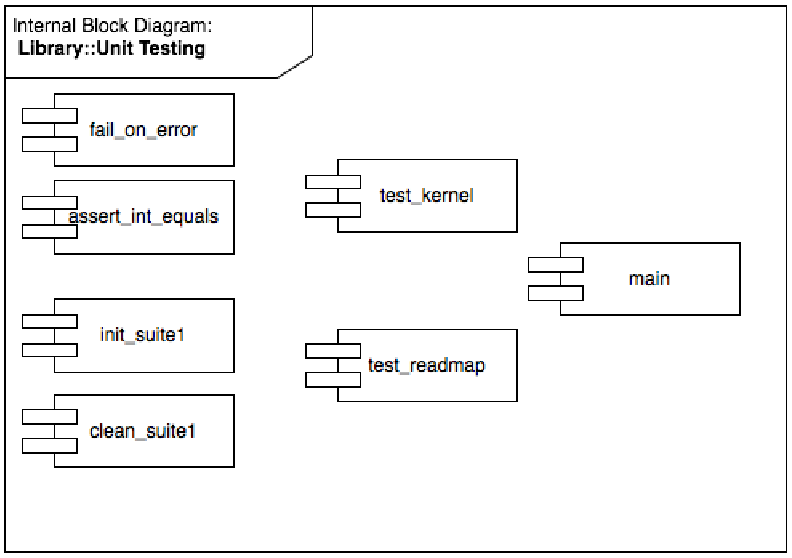


Figure 6: Unit Testing Internal Block Design

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

Work on performance analysis and deployment was mainly shared between all the members of the team.

3 Implementation and documenting

The implementation was made in the C language, with an additional Matlab program to generate the test cases. We used a distributed revision control software called Git, with a central repository located in `https://github.com/markflorisson88/puma`.

3.1 Data Structures

The map was 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 as 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 (PDE) computational kernel consists primarily 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` struct that contains the parameters of the equations.

Two 2002 $\times$ 2002 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 columns and rows, and the subsequent lines specify the values for the elements in 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 *errnos*. 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 registries, suites, and test cases, 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

For testing the computational kernel two techniques were used. The first one was CUnit, utilizing results from a simple MATLAB program as test cases. Only the values of the hare density were checked for simplicity, and the fact that a correct answer there pretty much guarantees a correct answer in the pumas density. A method, called `CU_ASSERT_DOUBLE_EQUAL`, was added to verify the solutions, since the native CUnit functions are limited with regards to identifying the errors (they just print that there is an error, but not what error or where). The tolerance for the tests was 1×10^{-5}.

There are a couple notable aspects about performing this tests, and after doing the mathematical analysis of the equations, the results started to make sense.

First, the results of the MATLAB code were verified thanks to the analysis presented in the introduction of the report.

When time steps of size 0.4 were used, the software never passed the the test cases. The results when using that value can be seen in figure 7. But when the time step was changed to 0.004 (which in principle should produce more accurate results), the test were passed satisfactorily.

4.2 Script Testing

In order to verify the behaviour of the results in the program, particularly important given the failed test cases when using 0.4 as step size, a perl script to compare the C program results with results produce by the Matlab program at various time steps of the algorithm.

A graph of the results of the MATLAB algorithm can be seen in figure 7 (once again using uniform starting densities and the handout parameters). Contrary to the results predicted by the analysis, the results diverged, and if a big final time parameter was used, the results wen to infinity (throwing a `NaN` result). A similar behaviour was observed in the C program.

On the other hand two results of the C program, using time steps of 0.004, can be seen in figure 8 (in the figure the Vector Field of the theoretical analysis was also added). Here the result was the stable cycle that was expected, and it was the same result for both MATLAB and C programs.

Also there was an introduction to this kind of problems in a recent lecture Parallel Numerical Algorithms, that talked about how this algorithms can fail due to the choice of time step. That seems to be the case in this result, the use of time steps of 0.4 produces a divergent result that is due to error in the algorithm and not an actual approximation to the solution. Both the C and MATLAB codes produced wrong results, and the fact that the wrong results they produced were different actually told us nothing about the correctness or the program. It was when using an adequate time step that both programs had a result similar to what we had expected, and since they coincided when dealing with these correct result one can conclude that the program passes the tests that it should once adequate parameters are utilized.

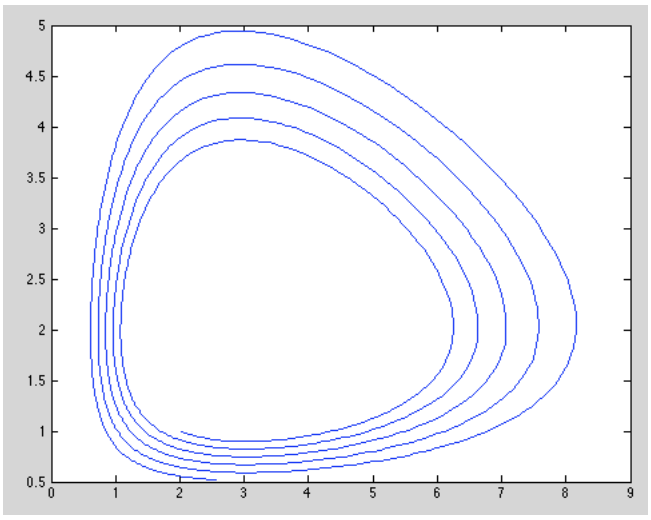


Figure 7: Example of divergent wrong approximation

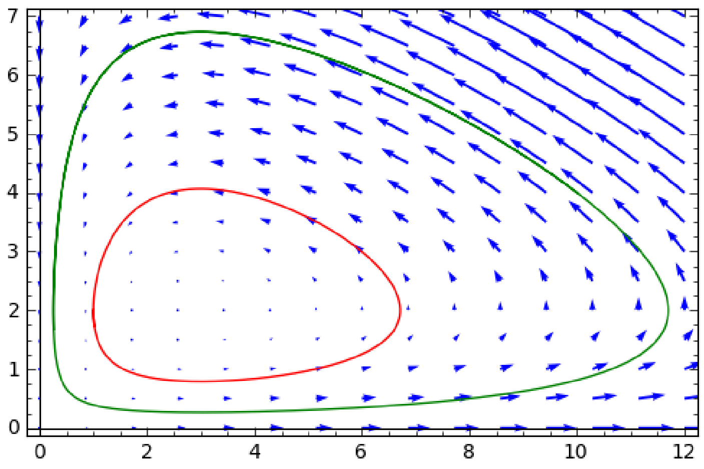


Figure 8: Correct approximations

4.3 Stress Testing

Stress testing was the final stage of the testing phase of development. It was performed using extreme and incorrect parameters. The stress test cases included:

1. Bad coefficient values (0 or negative) in command line arguments;
2. Bad time step value (negative or greater than 1) in command line arguments;
3. Bad write out interval (negative or greater than the number of iterations) in command line arguments;
4. Input file which is larger than the allowable size (2000×2000);
5. Input file dataset with the max allowable size (2000×2000);
6. Input file contains header only and no map cells;
7. Input file contains nothing;
8. Input file contains less number of cells than the number specified in its header;
9. Input file contains more number of cells than the number specified in its header;

For the stress test cases 1 to 8, the program exits outputting a relevant error message. For case 9, the program runs with the subset of the map in the input file with the size specified in the file header.

5 Deployment and Performance Analysis

5.1 Performance testing

The first step of the performance testing was utilizing different compiler optimization flags, which are shown in the next table:

gcc optimization flags:	`-O1, -O2, -O3`
Running environment:	Quad Core Intel i7 Computer

Figure 9 shows the running time of different optimization options and landscape sizes, utilizing single precision. The most notable aspect is the difference between the code compiled with no flags with the one compiled with the `-O1` flag. The optimizations provided by this level made the performance behaviour with respect to matrix size to be a lot more linear than the unoptimized code (which shows a near exponential growth), which wasn't a result that was expected beforehand. On the other hand, the similar behaviour between the `-O1, -O2` and `-O3` levels was to be expected given the simplicity of the code, and the fact that the data alignment optimizations provided by the `-O2` flags don't help much since the data was already nicely aligned to start with.

Figure 10 shows that when double precision was used, there were no significant difference in either the behaviour nor the execution time when compared to single precision. What this shows is that the code was executed on the CPU which is equally capable of double precision floating point operations and single precision ones and that the compiler optimizations had the same effects in both cases.

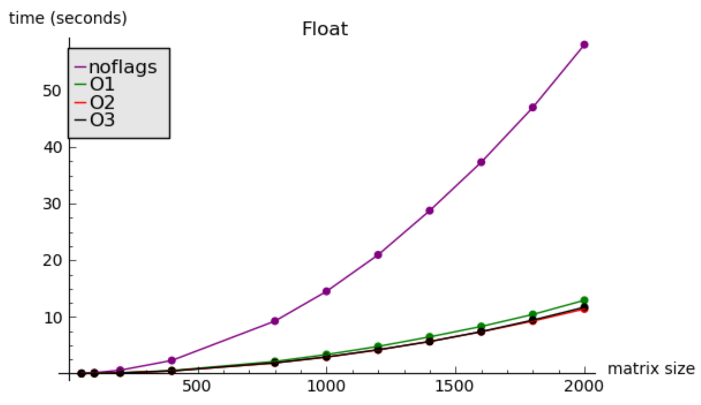


Figure 9: Performance of the program using single precision

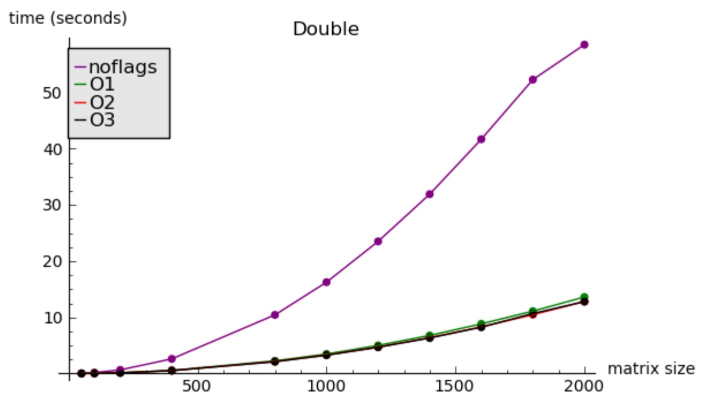


Figure 10: Performance of the program using double precision

The execution times for different landscape masks can be seen in figure 11. There seems to be a clear correlation between the amount of land (represented by 1s) and the execution time, which is perfectly logical. Nonetheless there was no evidence that the structure of the islands influenced execution time, but more experimentation is needed to determine this (which was not possible to do given the time constraints of the project).

5.2 Profiling

GNU GProf was utilized for profiling the software. The compute function (part of the PDE Kernel) took the majority of execution time. However, the running time of readmap function rose when increasing matrix size. When the input was a 50×50

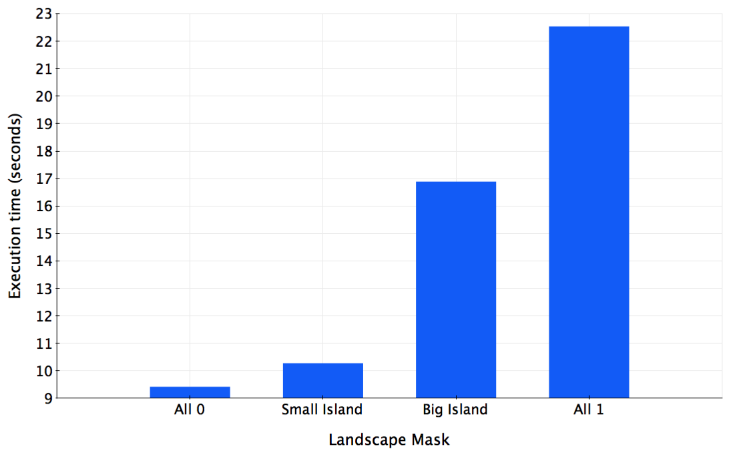


Figure 11: Performance of the program using different landscape masks

matrix, the compute function took 99.99% percent of CPU time for the process. When the input was a 2000×2000 x matrix, the percentage of time spent on the compute function was much less than the time spent for the small matrix.

This behaviour was actually expected beforehand, but it still tells us that all the optimization efforts must be used in the compute function and disk access should be avoided as much as possible, since this is (by far) the main source of overhead when landscape size is augmented.

6 Conclusions

The main objective of this work was to use good software development practices and standards, and see the importance that all stages of the process have, along with producing the best implementation given the time constraints of the project, which was accomplished successfully.

Working in a team has lots of challenges. Employing good software development processes can improve the quality and efficiency of the project deliverables. We experienced this firsthand while working on this project. By taking a modularized approach to tackle the various tasks in the project, the amount of work was divided as equally as possible. Periodic checkpoints and meetings were essential in order to complete the deliverables on time.

The coding standards used in the project ensured easy readability and quick detection of bugs within the code, along with easy integration of different components. Code reviews and running stress tests played an important part in making the program robust. The use of a testing framework made the task of unit testing quick and reliable, and also allowed making improvements and optimizations to the code base without breaking the functionality. Also the use of a good revision control system such as Git made the maintenance of the source code simple. Also automating repetitive tasks in the project using scripts improved the overall productivity of the project.

One unexpected aspect was the particular importance of automated error testing since it allowed us to realize some aspects of the behaviour that would have otherwise been impossible (at least given the time frame and other projects and course works), like the role of the time step in the convergence algorithm.

The use of performance analysis tools was very helpful for finding the bottlenecks within the code. A huge performance improvement was achieved by avoiding expensive operations such as copying segments of memory repeatedly in a loop. Use of tools like gdb was also quite useful in debugging segmentation faults.

By working on this project, every member in the team was able to use the various tools and techniques taught as part of the Programming Skills course, given that some of the main stages of development were shared among all the members.

References

[1] Hoppensteadt, F., *Predator-prey model*, Scholarpedia, 1(10), 1563, (2006)

[2] `'http://www.sysml.org/specs/'`