University of Edinburgh

Programming Skills

Development Coursework

# Programming Skills Report

*Authors:*
Jack Frankland

Denitsa Bankova

November 3, 2017

**Abstract**

This report forms part of the documentation for the coursework. It accompanies a reference manual which provides information on the class structures and methods, as well as the comments in the source code itself, which provide the implementation details. We have also provided a ReadMe file which can be viewed here `https://github.com/FranklandJack/2DPredPreyModel/blob/master/README.md` and has some overlap with this report. This document is designed to contain more general information about the code, how to run it and any important design decisions.

# 1 Introduction

In what follows the basic information about the project implementation is given and instructions are provided for building, running and testing the code. Finally, key design decisions are discussed and justified.

# 2 The Program

This program simulates the evolution of a system of predator(pumas) and prey(hares) densities across a 2D landscape which includes land and water. Neither the predators nor prey are considered able to swim so in wet areas density is always zero. The evolution of the system is governed by differential equations provided in the coursework development pdf:

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

and

$$\frac{\partial P}{\partial t} = bHP - mP + l\nabla^2(P) \tag{2}$$

where:

- H is the density of hares (prey).
- P is the density of pumas (predators).
- r is the birth rate of hares.

- a is the predation rate at which pumas eat hares.
- b is the birth rate of pumas per one hare eaten.
- m is the pumas mortality rate.
- k is the diffusion rate for hares.
- l is the diffusion rate for pumas.

# 3 Prerequisites

The source code in this project makes use of C++ move semantics, as such it needs to be compiled under the C++11 standard (or higher) hence the compiler on the user's machine must be C++11 compliant.

The unit test framework used is cppunit. To download, build and install please see the ReadMe file or `https://github.com/softwaresaved/build_and_test_examples/blob/master/cpp/README.md`.

# 4 General Information

## 4.1 Programming Language

The project source code and tests are written in C++. As mentioned above, due to the fact that it uses move semantics, the source code must be compiled under the C++11 (or later) standard, which is when move semantics were introduced.

## 4.2 Version Control

The version control system used for this project was Git with a master repository hosted on Github. Team members worked from the command line, using the git commands to pull, add, commit changes and push to the repository hosted on Github. The Github repository can be found here: `https://github.com/FranklandJack/2DPredPreyModel`.

### 4.3 Debuggers

The GNU GDB debugger was used for debugging from the command line and memory leak checks were performed with Valgrind.

### 4.4 Build Tools

A makefile was used with the make command for automated building. The main functionality of the makefile compiles and links the source code, but it also includes commands to test the code, to generate input files for the code from pnm files using the tool provided in the assignment, and various utility commands such as cleaning up all auto-generated files. There is a second configuration makefile which is included in the primary makefile to take care of any automated building for the converter tool that was provided in the assignment.

### 4.5 Test Tools

Unit tests for the program were written using the CppUnit test framework. For continuous integration Travis CI was used, meaning that each time a change was committed to the repository hosted on Github, the code was automatically built and tested by Travis CI using our unit tests. The Travis CI repository can be found here: `https://travis-ci.org/FranklandJack/2DPredPreyModel`

### 4.6 Documentation

Doxygen was used as a documentation tool for the source code. A doxyfile is present in the main directory which specifies preferences for the documentation. Using the doxygen commands it is then possible to generate a reference manual(in latex) and on-line documentation browser(in HTML) from the special comments in the source code header files.

## 5 Usage Instructions

### 5.1 Building

To build the executable run:

```
$ make
```

from the main directory. This will generate the object files and place them in the main directory, as well as linking them to create an executable called predprey.

## 5.2 Running

To run the code run:

```
$ ./predprey YOURINPUTLANDSCAPE.dat
```

where the first command line argument YOURINPUTLANDSCAPE.dat is the name of the .dat file containing the landscape you wish to simulate.

## 5.3 Testing

To build and run the unit tests on the code:

```
$ make test
```

Alternatively see `https://travis-ci.org/FranklandJack/2DPredPreyModel` for the build status.

## 5.4 Generating Documentation

To generate the documentation run:

```
$ doxygen Doxyfile
```

The generated files will then be outputted to the documentation directory, which in turn contains two directories; latex and html. To generate the reference manual, move into the latex directory

```
$ mv documentation/latex
```

and run:

```
$ make
```

which will generate a refman.pdf file that contains the class documentation. To open the on-line code browser move to the html directory:

```
$ mv documentation/html
```

and open the index.html file with a web browser such as Chrome. (Note: for ease of access the reference manual and on-line code browser files have been copied into the main directory, since they will not need regenerating after submission.)

### 5.5 Misc.

To generate .dat files from any pnm files, first place the pnm files in the landscape directory then run:

```
$ make dats
```

This will generate .dat files for all .pnm files in the landscape folder and place them in the main directory, ready to be run with the program.

To clean up all auto-generated files (this includes any output files in the output directory from the program after it is run, object files and executables, as well as .dat files in the main directory) run:

```
$ make clean
```

For a full list of make commands and the function run:

```
$ make help
```

## 6 Summary of Key Design Decisions

### 6.1 Directory Hierarchy

To keep the project organised into its component parts we have the following hierarchy:

- 2DPredPreyModel - contains the entire project.
  - documentation - contains auto-generated and written documentation.
  - landscapes - where the user should store their input landscapes as .pnm files.
  - output - where the output of the simulation will be sent to.
  - pnm2datConvertor - where the convertor tool for converting pnm files to the input format is stored.

- src - where all the source code for running the simulation is stored.
- test - where all the code for running tests is stored.

## 6.2 Classes

We chose to implement the 2D landscape using two classes; a Cell and a Grid. The Grid corresponds to the discretisation of the landscape and can be thought of as a 2D array, where each entry corresponds to a square in the landscape. The Cell class then corresponds to a square which obviously has three properties, predator density, prey density and whether it is wet or dry.

### 6.2.1 Cell

Within the Cell class it was obvious that the densities should each be modelled with floating point values, since they are not necessarily integers. We choose to create an enumeration to represent the state of the cell being wet or dry, rather than using 0 and 1 or false and true. This is because it simplifies the source code enormously, since whenever we set or check a state it is clear what we are doing as we either either have Cell::Wet or Cell:Dry in the code. It also saves having to remember which is which of 0 and 1 is dry and which is wet.

### 6.2.2 Grid

Within the Grid class it is obvious that the values that represent the width and height of the landscape (the number of columns and rows) should be chosen to be integers. We choose to implement the 2D array of cells as a 1D array within the class. This is because each Cell is in the Grid is dynamically allocated at runtime according to the user input. Dynamically allocating a 2D array of Cells is computationally expensive due to padding memory in each array entry. We overloaded the () operator to take two arguments and used a clever indexing system so that the Cells in the Grid can be accessed through the operator as if they are stored in a 2D array. We choose to implement the indexing system to match that in the coursework development pdf, that is Grid(i,j) corresponds to the ith column and the jth row, so they are x-y coordinate, this is not the same as matrix notation. They column and row numbers are also indexed from 1 to total number columns and 1 to total number rows which matches the convention in the pdf.