Coursework 2 - Programming Skills

Anthony Bourached, James Clark, Ishita Mathur, Zhuowei Si

General Information

Programming Language
We decided to use C as the programming language because some members of the team were familiar and others wanted to learn more about the programming language.

Compiler
From the beginning, we decided it was important to use open source tools for the development of this project. It made sense to use the GNU compiler, gcc, as it is the leading open source compiler for C programs.

Build Environment
In keeping with using open source tools, we used GNU Make for our automated build environment. The team was familiar with this tool as we had used it previously in class.

Version Control
As the team may be working away from the CP Lab machines, we decided a private code repository on Bitbucket was the ideal choice. This also provided us with a web interface to easily see any changes that have happened. Bitbucket allows Git or Mercurial for the type of repositiory. We chose Git because we have more experience with it.
A log of commits has been included in the `git.log` file.

Libraries
During the development, we used existing libraries to make the development easier. Most of the libraries used are standard with the GNU compiler. The following list shows some of the libraries used:

• ranlux	for random number generation	[included in source code, Lüscher 2015]
• argp.h	for argument parsing	[on CPLab machines]
• OpenMP	for timing	[on CPLab machines]
• CUnit	for unit testing	[on CPLab machines]

Debugging Tools
Three major debugging tools were used during the development of this project:

- Valgrind was used to check for memory leaks and other memory issues.
- ddd, the GUI frontend to gdb, was used to check for any subtle errors during runtime, the program has to be compiled with the `-g` flag for debugging.
- Cppcheck was used to spot any other errors in the code while developing.

How to Build

To compile the program, simply run:

```
make clean && make
```

To compile the unit tests, the command is:

```
make clean && make test
```

How to Run

The program can be run, after compilation, by typing:

```
./population [-N] [-S] [-t T] [-v] [-c config.file] Landscape.dat
```

The landscape.dat file is **required** and is an ASCII file that describes the land and water squares in the landscape. There are some sample landscapes in the `landscapes` directory.

Example:

```
./population landscapes/islands.dat
```

The optional runtime options are enclosed in `[ ]`. The following table shows what these arguments mean:

Short Option	Long Option	Meaning	Notes
`-N`	`--no_output_files`	No files will be written to the disk	
`-S`	`--silent`	No output will be written to `stdout` during the simulation	Output before and after the simulation will still be written to `stdout`
`-t T`	`--interval=T`	Manually specifying a time step to write an output to disk	Overrides the timestep given by the verbose option
`-v`	`--verbose`	Turns on verbose output	
`-c config.file`	`--config=CONFIG_FILE`	Used for supplying a configuration file	
`-?`	`--help`	Display a short help message	
	`--usage`	Display all the options	

How to Run Tests

The tests can easily be run after compiling them by running:

```
./population-test
```

Output

Depending on the runtime flags, at every 25, 10 or T steps, an image of the landscape is written to the `outputs` directory.

Configuration File

The configuration file is a text file that takes a list of real numbers such as:

```
hare birth rate
hare death rate
puma birth rate
puma death rate
hare diffusion rate
puma diffusion rate
time step
```

All of the parameters **must** be included. An example configuration file is:

```
0.08
0.04
0.02
0.06
0.2
0.2
0.4
```

References

- *Ranlux Random Number Generator*, Martin Lüscher, 19-Feb-2015 http://lusher.web.cern.ch/luscher/ranlux/, Accessed 19-Oct-2015