MPHYG002: Research Computing with C++

• Coursework 2 - Conway's Game of Life •

Sebastiano Ferraris SN: 14108168 s.ferraris@ucl.ac.uk August 15, 2016

Introducing the Game

Conway's Game of Life is a deterministic evolutionary discrete event dynamical system played on an infinite 2-dimensional grid. At each position of the gird there is a cell, that can be in two states: *alive* 1, or *dead* 0. The state of cell C at time t, indicated with $\text{state}_t(C)$ evolves over a discrete time according to its degree. The degree of a cell, $\deg_t(C)$ is the number of alive cells into its eight neigbours, and the evolution model from the state t to the state $t+1$ happens according to the following rules, called update rule, applied simultaneously for all the cells in the grid for a given time-state:

1. $\text{state}_t(C) = 1$ and $\deg_t(C) < 2$ then $\text{state}_{t+1}(C) = 0$.
2. $\text{state}_t(C) = 1$ and $2 \leq \deg_t(C) \leq 3$ then $\text{state}_{t+1}(C) = 1$.
3. $\text{state}_t(C) = 1$ and $\deg_t(C) > 3$ then $\text{state}_{t+1}(C) = 0$.
4. $\text{state}_t(C) = 0$ and $\deg_t(C) = 3$ then $\text{state}_{t+1}(C) = 1$.

The initial state is called *seed*, and it completely determines the subsequent steps of the game.

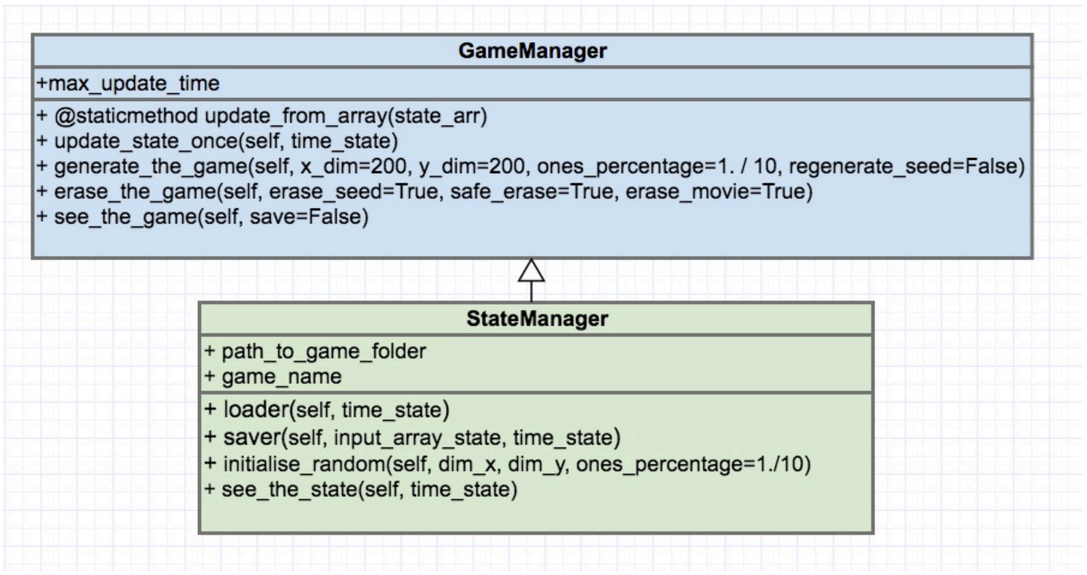
GameManager
+max_update_time
+ @staticmethod update_from_array(state_arr)
+ update_state_once(self, time_state)
+ generate_the_game(self, x_dim=200, y_dim=200, ones_percentage=1. / 10, regenerate_seed=False)
+ erase_the_game(self, erase_seed=True, safe_erase=True, erase_movie=True)
+ see_the_game(self, save=False)
StateManager
+ path_to_game_folder
+ game_name
+ loader(self, time_state)
+ saver(self, input_array_state, time_state)
+ initialise_random(self, dim_x, dim_y, ones_percentage=1./10)
+ see_the_state(self, time_state)

Figure 1: UML diagram representing the structure proposed for the StateManager and the GameManager classes implemented in python.

Part 1: Serial Solution

In the simple implementation here proposed, a *state* is represented by a *.txt* file that contains a binary matrix whose zeros represents the dead cells and ones represents the living cells. A *game*, in the perspective here proposed, is a sequence of *.txt* file where in the filename appears the name of the game, shared for each state, and an integer time parameter, starting from 0 for the seed. Two states with consecutive indexes are related to each others by the update rule. Given the need of plotting computing statistics for computational comparisons purposes and of obtaining an animation of a game (see the README.md on github repository), I opted for wrapping the C++ code inside python with boost::python, and to call the shared object (.so) generated with the C++ compiler as libraries inside python code.

Design and code structure

The object oriented structure is driven by the definition of state and game. To reduce computational costs and the length of the code, there is no class implemented for a state or a game, being structured .txt files. The classes are implemented for the operations between states, as well as to create random seeds, visualize states and games.

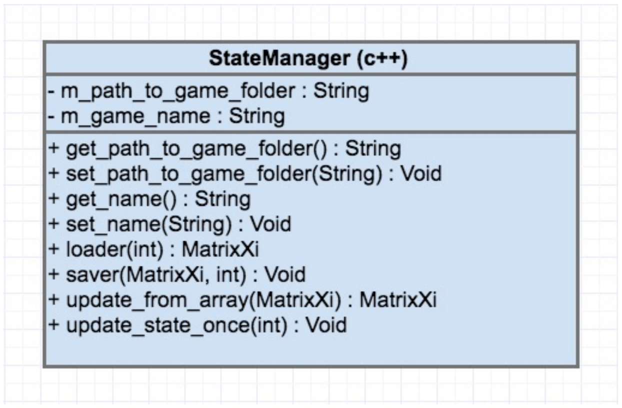
StateManager (c++)
- m_path_to_game_folder : String
- m_game_name : String
+ get_path_to_game_folder() : String
+ set_path_to_game_folder(String) : Void
+ get_name() : String
+ set_name(String) : Void
+ loader(int) : MatrixXi
+ saver(MatrixXi, int) : Void
+ update_from_array(MatrixXi) : MatrixXi
+ update_state_once(int) : Void

Figure 2: UML diagram representing the structure proposed for the StateManager and the GameManager classes implemented in C++.

The folder structure is organized as follows:

examples - contains some hello-world style examples and some code to create games of life, from random seeds and from seeds created ad hoc.

report contains this report and the figures produced with statistical methods.

src is divided into

core

- fdsa

stats Contains the methods that produced the statistical results.

test has the tests, performed with the library Nosetest. The C++ code is tested here as well, in its main functions.

Part 2: Parallel computation - OpenMP

qsub command here!

Part 2: Remote computation - OpenMP

Please see the link include in the README.md on the github repository for references.