GameOfLife

1.0

Generated by Doxygen 1.10.0

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 functions Namespace Reference

Functions

- void initialize_random (Grid *grid, GameParams *params)

  Initialize the board with random data.
- void initialize_from_file (Grid *grid, GameParams *params, std::string file)

  Initialize the board from a file.
- void iteration_one_board (Board *board, GameParams *params, Array1D *store_row, Array1D *store_col)

  Update the board for a given number of steps.
- int find_opt_divisor (int n)

  Find the largest divisor d of a number n that is smaller than sqrt(n)
- bool test_grid_parameters (int board_size, int d1, int d2)

  Test if the grid parameters allow for a suitable Cartesian grid communicator.

5.1.1 Function Documentation

5.1.1.1 find_opt_divisor()

```
int functions::find_opt_divisor (
            int n )
```

Find the largest divisor d of a number n that is smaller than sqrt(n)

Parameters

n	The number to find the divisor of

Returns

The largest divisor of n that is smaller than sqrt(n)

5.1.1.2 initialize_from_file()

```
void functions::initialize_from_file (
            Grid * grid,
            GameParams * params,
            std::string file )
```

Initialize the board from a file.

Parameters

grid	The grid to be initialized
params	The parameters for the game
file	The file to read the data from

5.1.1.3 initialize_random()

```
void functions::initialize_random (
            Grid * grid,
            GameParams * params )
```

Initialize the board with random data.

Parameters

grid	The grid to be initialized
params	The parameters for the game

5.1.1.4 iteration_one_board()

```
void functions::iteration_one_board (
            Board * board,
            GameParams * params,
            Array1D * store_row,
            Array1D * store_col )
```

Update the board for a given number of steps.

Parameters

board	The board to be updated
params	The parameters for the game, including the number of evolve steps
store_row	An array to store ghost rows
store_col	An array to store ghost columns

5.1.1.5 test_grid_parameters()

```
bool functions::test_grid_parameters (
```

```
int board_size,
int d1,
int d2 )
```

Test if the grid parameters allow for a suitable Cartesian grid communicator.

Parameters

board_size	The size of the board
d1	The first divisor
d2	The second divisor (with d2 >= d1)

Returns

True if the parameters are suitable, false otherwise

Chapter 6

Class Documentation

6.1 Array1D Class Reference

A class for 1D arrays.

`#include <Array1D.hpp>`

Public Member Functions

- Array1D (int size)
  Constructor.
- ~Array1D ()
  Destructor.
- int & operator() (int i)
  Overload the () operator to access the data.
- void overwrite (Array1D arr, int shift=0)
- void copy_into (Array1D *arr)
- Array1D sub_arr (int i_low, int i_upp)
- void display ()
  Display the data of the array.

Public Attributes

- int size
  Size of the array.
- int * data
  Pointer to the data.

6.1.1 Detailed Description

A class for 1D arrays.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 Array1D()

```
Array1D::Array1D (
            int size )  [inline]
```

Constructor.

6.1.2.2 ~Array1D()

```
Array1D::~Array1D ( )  [inline]
```

Destructor.

6.1.3 Member Function Documentation

6.1.3.1 copy_into()

```
void Array1D::copy_into (
            Array1D * arr )  [inline]
```

Copy the data of the array into another array.

Parameters

arr	The array from which the data is to be copied. Accessed by reference.

6.1.3.2 display()

```
void Array1D::display ( )  [inline]
```

Display the data of the array.

6.1.3.3 operator()()

```
int & Array1D::operator() (
            int i )  [inline]
```

Overload the () operator to access the data.

6.1.3.4 overwrite()

```
void Array1D::overwrite (
            Array1D arr,
            int shift = 0 )  [inline]
```

Overwrite the data of the array with the data of another array

Parameters

arr	The array to be copied into the current array
shift	The shift with which the array to be copied is loaded in the current array. If non-zero, arr needs to be smaller than the current array.

6.1.3.5 sub_arr()

```
Array1D Array1D::sub_arr (
            int i_low,
            int i_upp )  [inline]
```

Create a subarray of the current array.

Parameters

i_low	The lower index of the subarray
i_upp	The upper index of the subarray

6.1.4 Member Data Documentation

6.1.4.1 data

```
int* Array1D::data
```

Pointer to the data.

6.1.4.2 size

```
int Array1D::size
```

Size of the array.

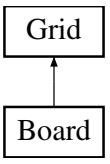
The documentation for this class was generated from the following file:

- src/lib/Array1D.hpp

6.2 Board Class Reference

```
#include <Board.hpp>
```

Inheritance diagram for Board:



Public Member Functions

- Board (int N_row, int N_col)

  Constructor.
- void init_from_motherboard (Grid *motherboard, int row_low, int col_left)

  Initialize the board with values from (a subgrid of) the motherboard.
- void set_bottom_ghost_row (Array1D *target)

  Set the bottom ghost row based on an input array.
- void set_upper_ghost_row (Array1D *target)

  Set the upper ghost row based on an input array.
- void set_left_ghost_col (Array1D *target)

  Set the left ghost column based on an input array.
- void set_right_ghost_col (Array1D *target)

  Set the right ghost column based on an input array.
- void store_neighbour_row (Array1D *store, int n_row)

  Store the neighbour counts of a row in an array.
- void store_upper_ghost_neighbour_row (Array1D *store)

  Store the neighbour counts of the upper ghost row in an array.
- void store_bottom_ghost_neighbour_row (Array1D *store)

  Store the neighbour counts of the bottom ghost row in an array.
- void ghost_display ()

  Display the board, including the ghost rows and columns.
- void update_board ()

  Update the board based on the rules of the game of life.

Public Member Functions inherited from Grid

- Grid (int N_row, int N_col, int N_nb_crit=3)

  Constructor.
- ~Grid ()

  Destructor.
- int & operator() (int i, int j)

  Overload the () operator to access the data.
- void store_row (Array1D *store, int n_row, int shift=0)
- void store_col (Array1D *store, int n_col)
- Array1D sub_row (int n_row, int i_low, int i_upp)
- Array1D sub_col (int n_col, int i_low, int i_upp)
- void display ()

  Display the data of the grid.
- Array1D periodic_row (int n_row)
- void save (std::string file)
- void store_data (int *arr)
- void read_data (int *arr)
- void overwrite_sub_board (int *arr, int row_low, int row_upp, int col_low, int col_upp)

Public Attributes

- Array1D bottom_ghost_row

  The ghost row at the bottom of the grid, including the corners.
- Array1D upper_ghost_row

  The ghost row at the top of the grid, including the corners.
- Array1D left_ghost_col

  The ghost column on the left side of the grid.
- Array1D right_ghost_col

  The ghost column on the right side of the grid.
- Array1D temp1

  Storage arrays to hold the horizontal neighbours counts in a row.
- Array1D temp2
- Array1D temp3

Public Attributes inherited from Grid

- int N_row

  Number of rows in the grid.
- int N_col

  Number of columns in the grid.
- int * data

  Pointer to the data.
- int N_nb_crit

  Number of critical neighbours used in the game rules.
- int size

  Number of rows times the number of columns.

6.2.1 Detailed Description

A class inheriting from Grid, that adds the functionality to update the board.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 Board()

```
Board::Board (
            int N_row,
            int N_col )  [inline]
```

Constructor.

Parameters

N_row	The number of rows in the grid
N_col	The number of columns in the grid

6.2.3 Member Function Documentation

6.2.3.1 ghost_display()

```
void Board::ghost_display ( )  [inline]
```

Display the board, including the ghost rows and columns.

6.2.3.2 init_from_motherboard()

```
void Board::init_from_motherboard (
            Grid * motherboard,
            int row_low,
            int col_left )  [inline]
```

Initialize the board with values from (a subgrid of) the motherboard.

Parameters

motherboard	The motherboard grid to copy values from
row_low	The lowest row index to copy from the motherboard
col_left	The leftmost column index to copy from the motherboard

6.2.3.3 set_bottom_ghost_row()

```
void Board::set_bottom_ghost_row (
            Array1D * target )  [inline]
```

Set the bottom ghost row based on an input array.

Parameters

target	The array to copy values from

6.2.3.4 set_left_ghost_col()

```
void Board::set_left_ghost_col (
            Array1D * target )  [inline]
```

Set the left ghost column based on an input array.

Parameters

target	The array to copy values from

6.2.3.5 set_right_ghost_col()

```
void Board::set_right_ghost_col (
            Array1D * target )  [inline]
```

Set the right ghost column based on an input array.

Parameters

target	The array to copy values from

6.2.3.6 set_upper_ghost_row()

```
void Board::set_upper_ghost_row (
            Array1D * target )  [inline]
```

Set the upper ghost row based on an input array.

Parameters

target	The array to copy values from

6.2.3.7 store_bottom_ghost_neighbour_row()

```
void Board::store_bottom_ghost_neighbour_row (
            Array1D * store )  [inline]
```

Store the neighbour counts of the bottom ghost row in an array.

Parameters

store	The array to store the neighbour counts in

6.2.3.8 store_neighbour_row()

```
void Board::store_neighbour_row (
            Array1D * store,
            int n_row )  [inline]
```

Store the neighbour counts of a row in an array.

Parameters

store	The array to store the neighbour counts in
n_row	The row index to store the neighbour counts of

6.2.3.9 store_upper_ghost_neighbour_row()

```
void Board::store_upper_ghost_neighbour_row (
            Array1D * store )  [inline]
```

Store the neighbour counts of the upper ghost row in an array.

Parameters

store	The array to store the neighbour counts in

6.2.3.10 update_board()

```
void Board::update_board ( )  [inline]
```

Update the board based on the rules of the game of life.

6.2.4 Member Data Documentation

6.2.4.1 bottom_ghost_row

```
Array1D Board::bottom_ghost_row
```

The ghost row at the bottom of the grid, including the corners.

6.2.4.2 left_ghost_col

```
Array1D Board::left_ghost_col
```

The ghost column on the left side of the grid.

6.2.4.3 right_ghost_col

```
Array1D Board::right_ghost_col
```

The ghost column on the right side of the grid.

6.2.4.4 temp1

```
Array1D Board::temp1
```

Storage arrays to hold the horizontal neighbours counts in a row.

6.2.4.5 temp2

```
Array1D Board::temp2
```

6.2.4.6 temp3

```
Array1D Board::temp3
```

6.2.4.7 upper_ghost_row

```
Array1D Board::upper_ghost_row
```

The ghost row at the top of the grid, including the corners.

The documentation for this class was generated from the following file:

- src/lib/Board.hpp

6.3 GameParams Class Reference

A class that stores the parameters for the Game of Life.

```
#include <GameParams.hpp>
```

Public Member Functions

- GameParams ()
  Default constructor.
- void readParams (const std::string &filename)
- void display () const
  Function that displays the parameters.

Public Attributes

- int board_size {10}
  The size of the board.
- int N_critical {3}
  The number of critical neighbours for a cell to survive.
- int save_interval {1}
  The interval at which the board is saved.
- int evolve_steps {20}
  The number of steps over which the board is evolved.
- int random_data {1}
- int num_threads {1}
  The number of OMP threads to use.
- double prob_live {0.5}
- std::string board_file {"examples/"}
  The path to the initialization file, in case random_data is 0.
- std::string output_path {"examples/"}
  The path where to store the output files.

6.3.1 Detailed Description

A class that stores the parameters for the Game of Life.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 GameParams()

```
GameParams::GameParams ( )  [inline]
```

Default constructor.

6.3.3 Member Function Documentation

6.3.3.1 display()

```
void GameParams::display ( ) const  [inline]
```

Function that displays the parameters.

6.3.3.2 readParams()

```
void GameParams::readParams (
            const std::string & filename )  [inline]
```

Function that reads the parameters from a text file

Parameters

filename	path to params file, parsed through command line

6.3.4 Member Data Documentation

6.3.4.1 board_file

```
std::string GameParams::board_file {"examples/"}
```

The path to the initialization file, in case random_data is 0.

6.3.4.2 board_size

```
int GameParams::board_size {10}
```

The size of the board.

6.3.4.3 evolve_steps

```
int GameParams::evolve_steps {20}
```

The number of steps over which the board is evolved.

6.3.4.4 N_critical

```
int GameParams::N_critical {3}
```

The number of critical neighbours for a cell to survive.

6.3.4.5 num_threads

```
int GameParams::num_threads {1}
```

The number of OMP threads to use.

6.3.4.6 output_path

```
std::string GameParams::output_path {"examples/"}
```

The path where to store the output files.

6.3.4.7 prob_live

```
double GameParams::prob_live {0.5}
```

The probability that a cell is alive at the start, parameter in a Binomial distribution

6.3.4.8 random_data

```
int GameParams::random_data {1}
```

Whether to initialize the board with random data or from a file. 1: random, 0: file (board_file)

6.3.4.9 save_interval

```
int GameParams::save_interval {1}
```

The interval at which the board is saved.

The documentation for this class was generated from the following file:

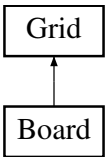
- src/lib/GameParams.hpp

6.4 Grid Class Reference

A class for a 2D grid that contains the entire board for the Game of Life.

`#include <Grid.hpp>`

Inheritance diagram for Grid:



Public Member Functions

- Grid (int N_row, int N_col, int N_nb_crit=3)
  Constructor.
- ∼Grid ()
  Destructor.
- int & operator() (int i, int j)
  Overload the () operator to access the data.
- void store_row (Array1D ∗store, int n_row, int shift=0)
- void store_col (Array1D ∗store, int n_col)
- Array1D sub_row (int n_row, int i_low, int i_upp)
- Array1D sub_col (int n_col, int i_low, int i_upp)
- void display ()
  Display the data of the grid.
- Array1D periodic_row (int n_row)
- void save (std::string file)
- void store_data (int ∗arr)
- void read_data (int ∗arr)
- void overwrite_sub_board (int ∗arr, int row_low, int row_upp, int col_low, int col_upp)

Public Attributes

- int N_row
  Number of rows in the grid.
- int N_col
  Number of columns in the grid.
- int ∗ data
  Pointer to the data.
- int N_nb_crit
  Number of critical neighbours used in the game rules.
- int size
  Number of rows times the number of columns.

6.4.1 Detailed Description

A class for a 2D grid that contains the entire board for the Game of Life.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 Grid()

```
Grid::Grid (
            int N_row,
            int N_col,
            int N_nb_crit = 3 )  [inline]
```

Constructor.

6.4.2.2 ∼Grid()

```
Grid::∼Grid ( )  [inline]
```

Destructor.

6.4.3 Member Function Documentation

6.4.3.1 display()

```
void Grid::display ( )  [inline]
```

Display the data of the grid.

6.4.3.2 operator()()

```
int & Grid::operator() (
            int i,
            int j )  [inline]
```

Overload the () operator to access the data.

6.4.3.3 overwrite_sub_board()

```
void Grid::overwrite_sub_board (
            int * arr,
            int row_low,
            int row_upp,
            int col_low,
            int col_upp )  [inline]
```

Overwrite a subgrid of the grid with the data in an array

Parameters

arr	The array from which the data is to be copied
row_low	The index of the lower row of the subgrid
row_upp	The index of the upper row of the subgrid
col_low	The index of the lower column of the subgrid
col_upp	The index of the upper column of the subgrid

6.4.3.4 periodic_row()

```
Array1D Grid::periodic_row (
            int n_row )  [inline]
```

Return a row, with one cell added to the left and right, for periodic boundary conditions

Parameters

n_row	The index of the row to be returned with the additional cells

6.4.3.5 read_data()

```
void Grid::read_data (
            int * arr )  [inline]
```

Read the data of the grid from an array

Parameters

arr	The array from which the data is to be read

6.4.3.6 save()

```
void Grid::save (
            std::string file )  [inline]
```

Save the data of the grid to a file

Parameters

file	The name of the file to which the data is to be saved

6.4.3.7 store_col()

```
void Grid::store_col (
            Array1D * store,
            int n_col )  [inline]
```

Store a column of the grid in an Array1D object

Parameters

store	The Array1D object in which the column is to be stored
n_col	The index of the column to be stored

6.4.3.8 store_data()

```
void Grid::store_data (
            int * arr )  [inline]
```

Store the data of the grid in an array

Parameters

arr	The array in which the data is to be stored

6.4.3.9 store_row()

```
void Grid::store_row (
            Array1D * store,
            int n_row,
            int shift = 0 )  [inline]
```

Store a row of the grid in an Array1D object

Parameters

store	The Array1D object in which the row is to be stored
n_row	The index of the row to be stored
shift	The shift with which the row is loaded in the Array1D object

6.4.3.10 sub_col()

```
Array1D Grid::sub_col (
            int n_col,
            int i_low,
            int i_upp )  [inline]
```

Return a subarray of a given column

Parameters

n_col	The index of the column from which the subarray is to be taken
i_low	The lower index of the subarray
i_upp	The upper index of the subarray

6.4.3.11 sub_row()

```
Array1D Grid::sub_row (
            int n_row,
            int i_low,
            int i_upp )  [inline]
```

Return a subarray of a given row

Parameters

n_row	The index of the row from which the subarray is to be taken
i_low	The lower index of the subarray
i_upp	The upper index of the subarray

6.4.4 Member Data Documentation

6.4.4.1 data

```
int* Grid::data
```

Pointer to the data.

6.4.4.2 N_col

```
int Grid::N_col
```

Number of columns in the grid.

6.4.4.3 N_nb_crit

```
int Grid::N_nb_crit
```

Number of critical neighbours used in the game rules.

6.4.4.4 N_row

```
int Grid::N_row
```

Number of rows in the grid.

6.4.4.5 size

```
int Grid::size
```

Number of rows times the number of columns.

The documentation for this class was generated from the following file:

- src/lib/Grid.hpp

Chapter 7

File Documentation

7.1 src/lib/Array1D.hpp File Reference

```
#include <iostream>
```

Classes

- class Array1D

  A class for 1D arrays.

7.2 Array1D.hpp

Go to the documentation of this file.

```
00001 #include <iostream>
00002
00003 #ifndef ARRAY1D_HPP
00004 #define ARRAY1D_HPP
00005
00007 class Array1D {
00008  public:
00010   int size;
00012   int* data;
00013
00015   Array1D(int size) {
00016     this->size = size;
00017     this->data = new int[size];
00018   }
00019
00021   ~Array1D() { delete[] this->data; }
00022
00024   int& operator()(int i) { return this->data[i]; }
00025
00031   void overwrite(Array1D arr, int shift = 0) {
00032     for (int i = 0; i < arr.size; ++i) {
00033       data[i + shift] = arr(i);
00034     }
00035   }
00036
00040   void copy_into(Array1D* arr) {
00041     for (int i = 0; i < size; ++i) {
00042       data[i] = (*arr)(i);
00043     }
00044   }
00045
00049   Array1D sub_arr(int i_low, int i_upp) {
00050     int len;
00051     if (i_low > i_upp) {
```

```
00052       len = size + i_upp - i_low;
00053     } else {
00054       len = i_upp - i_low;
00055     }
00056     Array1D sub(len);
00057     for (int i = 0; i < len; ++i) {
00058       sub(i) = data[(i_low + i) % size];
00059     }
00060     return sub;
00061   }
00062
00064   void display() {
00065     for (int i = 0; i < size; ++i) {
00066       std::cout « data[i] « " ";
00067     }
00068     std::cout « std::endl;
00069   }
00070 };
00071
00072 #endif
```

7.3 src/lib/Board.hpp File Reference

```
#include <omp.h>
#include <cassert>
#include <fstream>
#include <iostream>
#include "Array1D.hpp"
#include "Grid.hpp"
```

Classes

- class Board

Macros

- #define BOARD_HPP

7.3.1 Macro Definition Documentation

7.3.1.1 BOARD_HPP

```
#define BOARD_HPP
```

7.4 Board.hpp

Go to the documentation of this file.

```
00001 #include <omp.h>
00002
00003 #include <cassert>
00004 #include <fstream>
00005 #include <iostream>
00006
00007 #include "Array1D.hpp"
00008 #include "Grid.hpp"
00009
00010 #ifndef BOARD_HPP
```

```
00011 #define BOARD_HPP
00012
00015 class Board : public Grid {
00016  public:
00018   Array1D bottom_ghost_row;
00020   Array1D upper_ghost_row;
00022   Array1D left_ghost_col;
00024   Array1D right_ghost_col;
00025
00027   Array1D temp1, temp2, temp3;
00028
00030
00034   Board(int N_row, int N_col)
00035       : Grid(N_row, N_col),
00036         bottom_ghost_row(N_col + 2),
00037         upper_ghost_row(N_col + 2),
00038         left_ghost_col(N_row),
00039         right_ghost_col(N_row),
00040         temp1(N_col),
00041         temp2(N_col),
00042         temp3(N_col) {
00043     // Check if the grid is large enough to be sensible in the update procedure.
00044     assert(N_row > 2 && N_col > 2);
00045   }
00046
00048
00053   void init_from_motherboard(Grid* motherboard, int row_low, int col_left) {
00054     N_nb_crit = (*motherboard).N_nb_crit;
00055 #pragma omp parallel for collapse(2)
00056     for (int i = 0; i < N_row; ++i) {
00057       for (int j = 0; j < N_col; ++j) {
00058         data[i * N_col + j] = (*motherboard)(row_low + i, col_left + j);
00059       }
00060     }
00061   }
00062
00064
00067   void set_bottom_ghost_row(Array1D* target) {
00068     assert(target->size == N_col + 2);
00069 #pragma omp parallel for
00070     for (int i = 0; i < N_col + 2; ++i) {
00071       bottom_ghost_row(i) = (*target)(i);
00072     }
00073   }
00074
00076
00079   void set_upper_ghost_row(Array1D* target) {
00080     assert(target->size == N_col + 2);
00081 #pragma omp parallel for
00082     for (int i = 0; i < N_col + 2; ++i) {
00083       upper_ghost_row(i) = (*target)(i);
00084     }
00085   }
00086
00088
00091   void set_left_ghost_col(Array1D* target) {
00092     assert(target->size == N_row);
00093 #pragma omp parallel for
00094     for (int i = 0; i < N_row; ++i) {
00095       left_ghost_col(i) = (*target)(i);
00096     }
00097   }
00098
00100
00103   void set_right_ghost_col(Array1D* target) {
00104     assert(target->size == N_row);
00105 #pragma omp parallel for
00106     for (int i = 0; i < N_row; ++i) {
00107       right_ghost_col(i) = (*target)(i);
00108     }
00109   }
00111
00115   void store_neighbour_row(Array1D* store, int n_row) {
00116     (*store)(0) = left_ghost_col(n_row) + data[n_row * N_col + 0] +
00117                   data[n_row * N_col + 1];
00118 #pragma omp parallel for
00119     for (int i = 1; i < N_col - 1; ++i) {
00120       (*store)(i) = data[n_row * N_col + i - 1] + data[n_row * N_col + i] +
00121                     data[n_row * N_col + i + 1];
00122     }
00123     (*store)(N_col - 1) = data[n_row * N_col + N_col - 2] +
00124                           data[n_row * N_col + N_col - 1] +
00125                           right_ghost_col(n_row);
00126   }
00127
00129
00132   void store_upper_ghost_neighbour_row(Array1D* store) {
```

```
00133 #pragma omp parallel for
00134     for (int i = 0; i < N_col; ++i) {
00135       (*store)(i) =
00136           upper_ghost_row(i) + upper_ghost_row(i + 1) + upper_ghost_row(i + 2);
00137     }
00138   }
00139
00141
00144   void store_bottom_ghost_neighbour_row(Array1D* store) {
00145 #pragma omp parallel for
00146     for (int i = 0; i < N_col; ++i) {
00147       (*store)(i) = bottom_ghost_row(i) + bottom_ghost_row(i + 1) +
00148                     bottom_ghost_row(i + 2);
00149     }
00150   }
00151
00153   void ghost_display() {
00154     upper_ghost_row.display();
00155     for (int i = 0; i < N_row; ++i) {
00156       std::cout « left_ghost_col(i) « " ";
00157       for (int j = 0; j < N_col; ++j) {
00158         std::cout « data[i * N_col + j] « " ";
00159       }
00160       std::cout « right_ghost_col(i) « std::endl;
00161     }
00162     bottom_ghost_row.display();
00163   }
00164
00166   void update_board() {
00167     // Storage
00168     int N_nb{0};
00169     int val{0};
00170
00171     // Start with the top row, which requires the neighbours of the upper ghost
00172     // row
00173     store_upper_ghost_neighbour_row(&temp1);
00174     store_neighbour_row(&temp2, 0);
00175     store_neighbour_row(&temp3, 1);
00176 #pragma omp parallel for
00177     for (int j = 0; j < N_col; ++j) {
00178       val = data[j];
00179       N_nb = temp1(j) + temp2(j) + temp3(j) - val;
00180       data[j] = (1 - val) * (N_nb == N_nb_crit) +
00181                 val * (N_nb == N_nb_crit || N_nb == N_nb_crit - 1);
00182     }
00183
00184     // Then the middle rows
00185     for (int i = 1; i < N_row - 1; ++i) {
00186       temp1.copy_into(&temp2);
00187       temp2.copy_into(&temp3);
00188       store_neighbour_row(&temp3, i + 1);
00189 #pragma omp parallel for
00190       for (int j = 0; j < N_col; ++j) {
00191         val = data[i * N_col + j];
00192         N_nb = temp1(j) + temp2(j) + temp3(j) - val;
00193         data[i * N_col + j] =
00194             (1 - val) * (N_nb == N_nb_crit) +
00195             val * (N_nb == N_nb_crit || N_nb == N_nb_crit - 1);
00196       }
00197     }
00198
00199     // Finally the bottom row, which requires the neighbours of the bottom ghost
00200     // row
00201     temp1.copy_into(&temp2);
00202     temp2.copy_into(&temp3);
00203     store_bottom_ghost_neighbour_row(&temp3);
00204 #pragma omp parallel for
00205     for (int j = 0; j < N_col; ++j) {
00206       val = data[(N_row - 1) * N_col + j];
00207       N_nb = temp1(j) + temp2(j) + temp3(j) - val;
00208       data[(N_row - 1) * N_col + j] =
00209           (1 - val) * (N_nb == N_nb_crit) +
00210           val * (N_nb == N_nb_crit || N_nb == N_nb_crit - 1);
00211     }
00212   }
00213 };
00214
00215 #endif
```

7.5 src/lib/Functions.cpp File Reference

```
#include "Functions.hpp"
#include <omp.h>
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include "Array1D.hpp"
#include "Board.hpp"
#include "GameParams.hpp"
#include "Grid.hpp"
```

7.6 src/lib/Functions.hpp File Reference

```
#include "Board.hpp"
#include "GameParams.hpp"
```

Namespaces

- namespace functions

Functions

- void functions::initialize_random (Grid *grid, GameParams *params)

  Initialize the board with random data.
- void functions::initialize_from_file (Grid *grid, GameParams *params, std::string file)

  Initialize the board from a file.
- void functions::iteration_one_board (Board *board, GameParams *params, Array1D *store_row, Array1D *store_col)

  Update the board for a given number of steps.
- int functions::find_opt_divisor (int n)

  Find the largest divisor d of a number n that is smaller than sqrt(n)
- bool functions::test_grid_parameters (int board_size, int d1, int d2)

  Test if the grid parameters allow for a suitable Cartesian grid communicator.

7.7 Functions.hpp

Go to the documentation of this file.

```
00001 #ifndef FUNCTIONS_HPP
00002 #define FUNCTIONS_HPP
00003
00004 #include "Board.hpp"
00005 #include "GameParams.hpp"
00006
00007 namespace functions {
00008
00009 void initialize_random(Grid* grid, GameParams* params);
00010
00011 void initialize_from_file(Grid* grid, GameParams* params, std::string file);
```

```
00012
00013 void iteration_one_board(Board* board, GameParams* params, Array1D* store_row,
00014                         Array1D* store_col);
00015
00016 int find_opt_divisor(int n);
00017
00018 bool test_grid_parameters(int board_size, int d1, int d2);
00019
00020 }  // namespace functions
00021
00022 #endif
```

7.8 src/lib/GameParams.hpp File Reference

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
```

Classes

- class GameParams

  A class that stores the parameters for the Game of Life.

7.9 GameParams.hpp

Go to the documentation of this file.

```
00001 #ifndef GAMEPARAMS_HPP
00002 #define GAMEPARAMS_HPP
00003
00004 #include <fstream>
00005 #include <iostream>
00006 #include <sstream>
00007 #include <string>
00008
00010 class GameParams {
00011  public:
00013   int board_size{10};
00015   int N_critical{3};
00017   int save_interval{1};
00019   int evolve_steps{20};
00022   int random_data{1};
00024   int num_threads{1};
00027   double prob_live{0.5};
00029   std::string board_file{"examples/"};
00031   std::string output_path{"examples/"};
00032
00034   GameParams() {}
00035
00038   void readParams(const std::string& filename) {
00039     std::ifstream inputFile(filename);  // Open the text file for reading
00040
00041     if (!inputFile) {  // Check if the file was opened successfully
00042       std::cerr « "Unable to open file " « filename « std::endl;
00043       return;
00044     }
00045
00046     // Read parameters from the file and set member variables
00047     std::string line;
00048     while (std::getline(inputFile, line)) {
00049       if (line.empty() || line[0] == '#' || line.substr(0, 2) == "//") {
00050         continue;
00051       }
00052
00053       std::istringstream iss(line);
00054       std::string paramName, equalsSign, paramValue;
00055
```

```
00056       // Parse the line into parameter name, '=', and parameter value
00057       if (iss » paramName » equalsSign » paramValue && equalsSign == "=") {
00058         // Set member variables based on parameter name
00059         if (paramName == "board_size") {
00060           std::istringstream(paramValue) » board_size;
00061         } else if (paramName == "N_critical") {
00062           std::istringstream(paramValue) » N_critical;
00063         } else if (paramName == "save_interval") {
00064           std::istringstream(paramValue) » save_interval;
00065         } else if (paramName == "num_evolve_steps") {
00066           std::istringstream(paramValue) » evolve_steps;
00067         } else if (paramName == "random_data") {
00068           std::istringstream(paramValue) » random_data;
00069         } else if (paramName == "prob_live") {
00070           std::istringstream(paramValue) » prob_live;
00071         } else if (paramName == "board_file") {
00072           std::istringstream(paramValue) » board_file;
00073         } else if (paramName == "output_path") {
00074           std::istringstream(paramValue) » output_path;
00075         } else if (paramName == "num_threads") {
00076           std::istringstream(paramValue) » num_threads;
00077         }
00078       }
00079     }
00080
00081     // Close the file
00082     inputFile.close();
00083   }
00084
00086   void display() const {
00087     std::cout « "board size: " « board_size « std::endl;
00088     std::cout « "N_critical: " « N_critical « std::endl;
00089     std::cout « "save interval: " « save_interval « std::endl;
00090     std::cout « "evolve steps: " « evolve_steps « std::endl;
00091     std::cout « "num omp threads: " « num_threads « std::endl;
00092     std::cout « "probability to live: " « prob_live « std::endl;
00093     if (random_data) {
00094       std::cout « "initialization: random" « std::endl;
00095     } else {
00096       std::cout « "initialization: " « board_file « std::endl;
00097     }
00098   }
00099 };
00100
00101 #endif
```

7.10 src/lib/Grid.hpp File Reference

```
#include <omp.h>
#include <fstream>
#include <iostream>
#include "Array1D.hpp"
```

Classes

- class Grid

  A class for a 2D grid that contains the entire board for the Game of Life.

7.11 Grid.hpp

Go to the documentation of this file.

```
00001 #include <omp.h>
00002
00003 #include <fstream>
00004 #include <iostream>
00005
00006 #include "Array1D.hpp"
```

```
00007
00008 #ifndef GRID_HPP
00009 #define GRID_HPP
00010
00012 class Grid {
00013  public:
00015   int N_row;
00017   int N_col;
00019   int* data;
00021   int N_nb_crit;
00023   int size;
00024
00026   Grid(int N_row, int N_col, int N_nb_crit = 3) {
00027     this->N_row = N_row;
00028     this->N_col = N_col;
00029     this->N_nb_crit = N_nb_crit;
00030     this->data = new int[N_row * N_col];
00031     size = N_row * N_col;
00032   }
00034   ~Grid() { delete[] this->data; }
00035
00037   int& operator()(int i, int j) { return this->data[i * N_col + j]; }
00038
00043
00044   void store_row(Array1D* store, int n_row, int shift = 0) {
00045 #pragma omp parallel for
00046     for (int i = 0; i < N_col; ++i) {
00047       (*store)(i + shift) = data[n_row * N_col + i];
00048     }
00049   }
00050
00054   void store_col(Array1D* store, int n_col) {
00055 #pragma omp parallel for
00056     for (int i = 0; i < N_row; ++i) {
00057       (*store)(i) = data[i * N_col + n_col];
00058     }
00059   }
00060
00065   Array1D sub_row(int n_row, int i_low, int i_upp) {
00066     Array1D temp(N_col);
00067     store_row(&temp, n_row);
00068     return temp.sub_arr(i_low, i_upp);
00069   }
00070
00076   Array1D sub_col(int n_col, int i_low, int i_upp) {
00077     Array1D temp(N_row);
00078     store_col(&temp, n_col);
00079     return temp.sub_arr(i_low, i_upp);
00080   }
00081
00083   void display() {
00084     for (int i = 0; i < N_row; ++i) {
00085       for (int j = 0; j < N_col; ++j) {
00086         std::cout « data[i * N_col + j] « " ";
00087       }
00088       std::cout « std::endl;
00089     }
00090   }
00091
00095   Array1D periodic_row(int n_row) {
00096     Array1D temp(N_col + 2);
00097     temp(0) = data[n_row * N_col + N_col - 1];
00098     store_row(&temp, n_row, 1);
00099     temp(N_col + 1) = data[n_row * N_col];
00100     return temp;
00101   }
00102
00105   void save(std::string file) {
00106     std::ofstream outputFile(file);
00107
00108     if (!outputFile.is_open()) {
00109       std::cerr « "Error opening file for writing!" « std::endl;
00110     }
00111
00112     for (int i = 0; i < N_row; ++i) {
00113       for (int j = 0; j < N_col - 1; ++j) {
00114         outputFile « data[i * N_col + j] « " ";
00115       }
00116       outputFile « data[i * N_col + N_col - 1];
00117       outputFile « std::endl;
00118     }
00119
00120     outputFile.close();
00121   }
00122
00125   void store_data(int* arr) {
00126 #pragma omp parallel for
```

```
00127     for (int i = 0; i < size; i++) {
00128       arr[i] = data[i];
00129     }
00130   }
00131
00134   void read_data(int* arr) {
00135 #pragma omp parallel for
00136     for (int i = 0; i < size; i++) {
00137       data[i] = arr[i];
00138     }
00139   }
00140
00147   void overwrite_sub_board(int* arr, int row_low, int row_upp, int col_low,
00148                            int col_upp) {
00149     int n_rows = row_upp - row_low;
00150     int n_cols = col_upp - col_low;
00151 #pragma omp parallel for collapse(2)
00152     for (int i = 0; i < n_rows; i++) {
00153       for (int j = 0; j < n_cols; j++) {
00154         data[(row_low + i) * N_col + col_low + j] = arr[i * n_cols + j];
00155       }
00156     }
00157   }
00158 };
00159
00160 #endif
```

7.12 src/main_parallel.cpp File Reference

```
#include <mpi.h>
#include <omp.h>
#include <cassert>
#include <iostream>
#include "lib/Array1D.hpp"
#include "lib/Board.hpp"
#include "lib/Functions.hpp"
#include "lib/GameParams.hpp"
#include "lib/Grid.hpp"
```

Functions

- int main (int argc, char ∗argv[ ])

7.12.1 Function Documentation

7.12.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

7.13 src/main_simple.cpp File Reference

```
#include <iostream>
#include "lib/Array1D.hpp"
#include "lib/Board.hpp"
#include "lib/Functions.hpp"
#include "lib/GameParams.hpp"
#include "lib/Grid.hpp"
```

Functions

- int main (int argc, char ∗argv[ ])

7.13.1 Function Documentation

7.13.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Index