AutoCell

Generated by Doxygen 1.8.13

Contents

Chapter 1

Main Page

To generate the Documentation, go in Documentation directory and run `make`.

It will generate html doc (in `output/html/index.html`) and latex doc (pdf output directely in Documentation directory (`docPdf.pdf`).

Chapter 2

Presentation

What is AutoCell

The purpose of this project is to create a Cellular Automate Simulator.

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

Class Documentation

5.1 Cell Class Reference

Contains the state, the next state and the neighbours.

```
#include <cell.h>
```

Public Member Functions

- Cell (unsigned int state=0)

  Constructs a cell with the state given. State 0 is dead state.
- void setState (unsigned int state)

  Set temporary state.
- void validState ()

  Validate temporary state.
- unsigned int getState () const

  Access current cell state.
- bool addNeighbour (const Cell ∗neighbour)

  Add a new neighbour to the Cell.
- QVector< const Cell ∗ > getNeighbours () const

  Access neighbours list.

Private Attributes

- unsigned int m_state

  Current state.
- unsigned int m_nextState

  Temporary state, before validation.
- QVector< const Cell ∗ > m_neighbours

  Cell's neighbours.

5.1.1 Detailed Description

Contains the state, the next state and the neighbours.

Definition at line 10 of file cell.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Cell()

```
Cell::Cell (
             unsigned int state = 0 )
```

Constructs a cell with the state given. State 0 is dead state.

Parameters

state	Cell state, dead state by default

Definition at line 8 of file cell.cpp.

5.1.3 Member Function Documentation

5.1.3.1 addNeighbour()

```
bool Cell::addNeighbour (
             const Cell * neighbour )
```

Add a new neighbour to the Cell.

Parameters

neighbour	New neighbour

Returns

False if the neighbour already exists

Definition at line 52 of file cell.cpp.

References m_neighbours.

5.1.3.2 getNeighbours()

```
QVector< const Cell * > Cell::getNeighbours ( ) const
```

Access neighbours list.

Definition at line 63 of file cell.cpp.

References m_neighbours.

5.1.3.3 getState()

```
unsigned int Cell::getState ( ) const
```

Access current cell state.

Definition at line 41 of file cell.cpp.

References m_state.

5.1.3.4 setState()

```
void Cell::setState (
             unsigned int state )
```

Set temporary state.

To change current cell state, use setState(unsigned int state) then validState().

Parameters

state	New state

Definition at line 22 of file cell.cpp.

References m_nextState.

5.1.3.5 validState()

```
void Cell::validState ( )
```

Validate temporary state.

To change current cell state, use setState(unsigned int state) then validState().

Definition at line 33 of file cell.cpp.

References m_nextState, and m_state.

5.1.4 Member Data Documentation

5.1.4.1 m_neighbours

```
QVector<const Cell*> Cell::m_neighbours [private]
```

Cell's neighbours.

Definition at line 26 of file cell.h.

Referenced by addNeighbour(), and getNeighbours().

5.1.4.2 m_nextState

```
unsigned int Cell::m_nextState [private]
```

Temporary state, before validation.

Definition at line 24 of file cell.h.

Referenced by setState(), and validState().

5.1.4.3 m_state

```
unsigned int Cell::m_state [private]
```

Current state.

Definition at line 23 of file cell.h.

Referenced by getState(), and validState().

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

- cell.h
- cell.cpp

5.2 CellHandler Class Reference

Cell container and cell generator.

```
#include <cellhandler.h>
```

Classes

- class iterator

  Implementation of iterator design pattern.

Public Types

- enum generationTypes { empty, random, symetric }

  Type of random generation.

Public Member Functions

- CellHandler (const QString filename)

  Construct all the cells from the json file given.
- CellHandler (const QVector< unsigned int > dimensions, generationTypes type=empty, unsigned int state↩ Max=1, unsigned int density=50)

  Construct a CellHandler of the given dimension.
- virtual ∼CellHandler ()

  Destroys all cells in the CellHandler.
- Cell ∗ getCell (const QVector< unsigned int > position) const

  Access the cell to the given position.
- void nextStates ()

  Valid the state of all cells.
- bool save (QString filename)

  Save the CellHandler current configuration in the file given.
- void generate (generationTypes type, unsigned int stateMax=1, unsigned short density=50)

  Replace Cell values by random values (symetric or not)
- void print (std::ostream &stream)

  Print in the given stream the CellHandler.
- iterator begin ()

  Give the iterator which corresponds to the current CellHandler.
- bool end ()

  End condition of the iterator.

Private Member Functions

- bool load (const QJsonObject &json)

  Load the config file in the CellHandler.
- void foundNeighbours ()

  Set the neighbours of each cells.
- void positionIncrement (QVector< unsigned int > &pos, unsigned int value=1) const

  Increment the QVector given by the value choosen.
- QVector< QVector< unsigned int > > ∗ getListNeighboursPositionsRecursive (const QVector< unsigned int > position, unsigned int dimension, QVector< unsigned int > lastAdd) const

  Recursive function which browse the position possibilities tree.
- QVector< QVector< unsigned int > > & getListNeighboursPositions (const QVector< unsigned int > position) const

  Prepare the call of the recursive version of itself.

Private Attributes

- QVector< unsigned int > m_dimensions

  Vector of x dimensions.
- QMap< QVector< unsigned int >, Cell ∗> m_cells

  Map of cells, with a x dimensions vector as key.

5.2.1 Detailed Description

Cell container and cell generator.

Generate cells from a json file.

Definition at line 18 of file cellhandler.h.

5.2.2 Member Enumeration Documentation

5.2.2.1 generationTypes

`enum CellHandler::generationTypes`

Type of random generation.

Enumerator

empty	Only empty cells.
random	Random cells.
symetric	Random cells but with vertical symetry (on the 1st dimension component)

Definition at line 63 of file cellhandler.h.

5.2.3 Constructor & Destructor Documentation

5.2.3.1 CellHandler() [1/2]

```
CellHandler::CellHandler (
              const QString filename )
```

Construct all the cells from the json file given.

The size of "cells" array must be the product of all dimensions (60 in the following example). Typical Json file:

```
{
"dimensions":"3x4x5",
"cells":[0,1,4,4,2,5,3,4,2,4,
         4,2,5,0,0,0,0,0,0,0,
         2,4,1,1,1,1,1,2,1,1,
         0,0,0,0,0,0,2,2,2,2,
         3,4,5,1,2,0,9,0,0,0,
         1,2,0,0,0,0,1,2,3,2]
}
```

Parameters

filename	Json file which contains the description of all the cells

Exceptions

QString	Unreadable file
QString	Empty file
QString	Not valid file

Definition at line 26 of file cellhandler.cpp.

References foundNeighbours(), and load().

5.2.3.2 CellHandler() [2/2]

```
CellHandler::CellHandler (
            const QVector< unsigned int > dimensions,
            generationTypes type = empty,
            unsigned int stateMax = 1,
            unsigned int density = 50 )
```

Construct a CellHandler of the given dimension.

If generationTypes is given, the CellHandler won't be empty.

Parameters

dimensions	Dimensions of the CellHandler
type	Generation type, empty by default
stateMax	Generate states between 0 and stateMax
density	Average (%) of non-zeros

Definition at line 66 of file cellhandler.cpp.

References empty, generate(), m_cells, m_dimensions, and positionIncrement().

5.2.3.3 ∼CellHandler()

```
CellHandler::∼CellHandler ( )  [virtual]
```

Destroys all cells in the CellHandler.

Definition at line 97 of file cellhandler.cpp.

References m_cells.

5.2.4 Member Function Documentation

5.2.4.1 begin()

```
CellHandler::iterator CellHandler::begin ( )
```

Give the iterator which corresponds to the current CellHandler.

Definition at line 271 of file cellhandler.cpp.

Referenced by save().

5.2.4.2 end()

```
bool CellHandler::end ( )
```

End condition of the iterator.

See iterator::operator!=(bool finished) for further information.

Definition at line 281 of file cellhandler.cpp.

Referenced by save().

5.2.4.3 foundNeighbours()

```
void CellHandler::foundNeighbours ( )  [private]
```

Set the neighbours of each cells.

Careful, this is in O(n∗3^d), with n the number of cells and d the number of dimensions

Definition at line 376 of file cellhandler.cpp.

References getListNeighboursPositions(), m_cells, m_dimensions, and positionIncrement().

Referenced by CellHandler().

5.2.4.4 generate()

```
void CellHandler::generate (
            CellHandler::generationTypes type,
            unsigned int stateMax = 1,
            unsigned short density = 50 )
```

Replace Cell values by random values (symetric or not)

Parameters

type	Type of random generation
stateMax	Generate states between 0 and stateMax
density	Average (%) of non-zeros

Definition at line 174 of file cellhandler.cpp.

References m_cells, m_dimensions, positionIncrement(), random, and symetric.

Referenced by CellHandler().

5.2.4.5 getCell()

```
Cell * CellHandler::getCell (
            const QVector< unsigned int > position ) const
```

Access the cell to the given position.

Definition at line 108 of file cellhandler.cpp.

References m_cells.

5.2.4.6 getListNeighboursPositions()

```
QVector< QVector< unsigned int > > & CellHandler::getListNeighboursPositions (
            const QVector< unsigned int > position ) const  [private]
```

Prepare the call of the recursive version of itself.

Parameters

position	Position of the central cell (x1,x2,x3,..,xn)

Returns

List of positions

Definition at line 437 of file cellhandler.cpp.

References getListNeighboursPositionsRecursive().

Referenced by foundNeighbours().

5.2.4.7 getListNeighboursPositionsRecursive()

```
QVector< QVector< unsigned int > > * CellHandler::getListNeighboursPositionsRecursive (
              const QVector< unsigned int > position,
              unsigned int dimension,
              QVector< unsigned int > lastAdd ) const  [private]
```

Recursive function which browse the position possibilities tree.

Careful, the complexity is in O(3^dimension)
Piece of the tree:

```
            x_d -1
          /
x_(d-1)-1/_ x_d
         \
          \
            x_d +1

            x_d -1
          /
x_(d-1)  /_ x_d
         \
          \
            x_d +1

            x_d -1
          /
x_(d-1)+1/ x_d
         \
          \
            x_d +1
```

The path in the tree to reach the leaf give the position

Parameters

position	Position of the cell
dimension	Current working dimension (number of the digit). Dimension = 2 <=> working on x2 coordinates on (x1, x2, x3, ..., xn) vector
lastAdd	Last position added. Like the father node of the new tree

Returns

List of position

Definition at line 479 of file cellhandler.cpp.

References m_dimensions.

Referenced by getListNeighboursPositions().

5.2.4.8 load()

```
bool CellHandler::load (
              const QJsonObject & json )  [private]
```

Load the config file in the CellHandler.

Exemple of a way to print cell states :

```
QVector<unsigned int> position;
for (unsigned short i = 0; i < m_dimensions.size(); i++)
{
     position.push_back(0);
}
for (unsigned int j = 0; j < m_cells.size(); j++)
{
     std::cout << m_cells.value(position)->getState() << " ";
     position.replace(0, position.at(0)+1);
     for (unsigned short i = 0; i < m_dimensions.size(); i++)
     {
         if (position.at(i) >= m_dimensions.at(i))
         {
             position.replace(i, 0);
             std::cout << std::endl;
             if (i + 1 != m_dimensions.size())
                 position.replace(i+1, position.at(i+1)+1);
         }

     }
}
```

Parameters

json	Json Object which contains the grid configuration

Returns

False if the Json Object is not correct

Definition at line 317 of file cellhandler.cpp.

References m_cells, m_dimensions, and positionIncrement().

Referenced by CellHandler().

5.2.4.9 nextStates()

```
void CellHandler::nextStates ( )
```

Valid the state of all cells.

Definition at line 117 of file cellhandler.cpp.

References m_cells.

5.2.4.10 positionIncrement()

```
void CellHandler::positionIncrement (
             QVector< unsigned int > & pos,
             unsigned int value = 1 ) const  [private]
```

Increment the QVector given by the value choosen.

Careful, when the position reach the maximum, it goes to zero without leaving the function

Parameters

pos	Position to increment
value	Value to add, 1 by default

Definition at line 407 of file cellhandler.cpp.

References m_dimensions.

Referenced by CellHandler(), foundNeighbours(), generate(), and load().

5.2.4.11 print()

```
void CellHandler::print (
             std::ostream & stream )
```

Print in the given stream the CellHandler.

Parameters

stream	Stream to print into

Definition at line 243 of file cellhandler.cpp.

References m_cells, and m_dimensions.

Referenced by main().

5.2.4.12 save()

```
bool CellHandler::save (
             QString filename )
```

Save the CellHandler current configuration in the file given.

Parameters

filename	Path to the file

Returns

False if there was a problem

Exceptions

QString	Impossible to open the file

Definition at line 133 of file cellhandler.cpp.

References begin(), end(), and m_dimensions.

5.2.5 Member Data Documentation

5.2.5.1 m_cells

```
QMap<QVector<unsigned int>, Cell* > CellHandler::m_cells  [private]
```

Map of cells, with a x dimensions vector as key.

Definition at line 91 of file cellhandler.h.

Referenced by CellHandler(), foundNeighbours(), generate(), getCell(), load(), nextStates(), CellHandler::iterator::operator*(), CellHandler::iterator::operator->(), print(), and ∼CellHandler().

5.2.5.2 m_dimensions

```
QVector<unsigned int> CellHandler::m_dimensions  [private]
```

Vector of x dimensions.

Definition at line 90 of file cellhandler.h.

Referenced by CellHandler(), foundNeighbours(), generate(), getListNeighboursPositionsRecursive(), CellHandler::iterator::iterator(), load(), CellHandler::iterator::operator++(), positionIncrement(), print(), and save().

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

- cellhandler.h
- cellhandler.cpp

5.3 CellHandler::iterator Class Reference

Implementation of iterator design pattern.

```
#include <cellhandler.h>
```

Public Member Functions

- iterator (const CellHandler *handler)

  Construct an initial iterator to browse the CellHandler.
- iterator & operator++ ()

  Increment the current position and handle dimension changes.
- Cell * operator-> () const

  Get the current cell.
- Cell * operator* () const
- bool operator!= (bool finished) const
- unsigned int changedDimension () const

  Return the number of dimensions we change.

Private Attributes

- const CellHandler * m_handler

  CellHandler to go through.
- QVector< unsigned int > m_position

  Current position of the iterator.
- bool m_finished = false

  If we reach the last position.
- QVector< unsigned int > m_zero

  Nul vector of the good dimension (depend of m_handler)
- unsigned int m_changedDimension

  Save the number of dimension change.

Friends

- class CellHandler

5.3.1 Detailed Description

Implementation of iterator design pattern.

Example of use:

```
CellHandler handler("file.atc");
for (CellHandler::iterator it = handler.begin(); it != handler.end(); ++it)
{
     for (unsigned int i = 0; i < it.changedDimension(); i++)
         std::cout << std::endl;
     std::cout << it->getState() << " ";
}
```

This code will print each cell states and go to a new line when there is a change of dimension. So if there is 3 dimensions, there will be a empty line between 2D groups.

Definition at line 37 of file cellhandler.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 iterator()

```
CellHandler::iterator::iterator (
               const CellHandler * handler )
```

Construct an initial iterator to browse the CellHandler.

Parameters

handler	CellHandler to browse

Definition at line 519 of file cellhandler.cpp.

References CellHandler::m_dimensions, m_position, and m_zero.

5.3.3 Member Function Documentation

5.3.3.1 changedDimension()

```
unsigned int CellHandler::iterator::changedDimension ( ) const
```

Return the number of dimensions we change.

For example, if we were at the (3,4,4) cell, and we incremented the position, we are now at (4,0,0), and changed↩ Dimension return 2 (because of the 2 zeros).

Definition at line 580 of file cellhandler.cpp.

References m_changedDimension.

Referenced by operator!=().

5.3.3.2 operator"!=()

```
bool CellHandler::iterator::operator!= (
             bool finished ) const  [inline]
```

Definition at line 47 of file cellhandler.h.

References changedDimension(), and m_finished.

5.3.3.3 operator∗()

```
Cell * CellHandler::iterator::operator* ( ) const
```

Definition at line 569 of file cellhandler.cpp.

References CellHandler::m_cells, m_handler, and m_position.

5.3.3.4 operator++()

```
CellHandler::iterator & CellHandler::iterator::operator++ ( )
```

Increment the current position and handle dimension changes.

Definition at line 533 of file cellhandler.cpp.

References m_changedDimension, CellHandler::m_dimensions, m_finished, m_handler, m_position, and m_zero.

5.3.3.5 operator->()

```
Cell * CellHandler::iterator::operator-> ( ) const
```

Get the current cell.

Definition at line 561 of file cellhandler.cpp.

References CellHandler::m_cells, m_handler, and m_position.

5.3.4 Friends And Related Function Documentation

5.3.4.1 CellHandler

```
friend class CellHandler [friend]
```

Definition at line 39 of file cellhandler.h.

5.3.5 Member Data Documentation

5.3.5.1 m_changedDimension

```
unsigned int CellHandler::iterator::m_changedDimension [private]
```

Save the number of dimension change.

Definition at line 57 of file cellhandler.h.

Referenced by changedDimension(), and operator++().

5.3.5.2 m_finished

```
bool CellHandler::iterator::m_finished = false  [private]
```

If we reach the last position.

Definition at line 55 of file cellhandler.h.

Referenced by operator!=(), and operator++().

5.3.5.3 m_handler

```
const CellHandler∗ CellHandler::iterator::m_handler  [private]
```

CellHandler to go through.

Definition at line 53 of file cellhandler.h.

Referenced by operator∗(), operator++(), and operator->().

5.3.5.4 m_position

```
QVector<unsigned int> CellHandler::iterator::m_position  [private]
```

Current position of the iterator.

Definition at line 54 of file cellhandler.h.

Referenced by iterator(), operator∗(), operator++(), and operator->().

5.3.5.5 m_zero

```
QVector<unsigned int> CellHandler::iterator::m_zero  [private]
```

Nul vector of the good dimension (depend of m_handler)

Definition at line 56 of file cellhandler.h.

Referenced by iterator(), and operator++().

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

- cellhandler.h
- cellhandler.cpp

Chapter 6

File Documentation

6.1 cell.cpp File Reference

```
#include "cell.h"
```

6.2 cell.cpp

```
#include "cell.h"

Cell::Cell(unsigned int state):
    m_state(state), m_nextState(state)
{

}

void Cell::setState(unsigned int state)
{
    m_nextState = state;
}

void Cell::validState()
{
    m_state = m_nextState;
}

unsigned int Cell::getState() const
{
    return m_state;
}

bool Cell::addNeighbour(const Cell* neighbour)
{
    if (m_neighbours.count(neighbour))
        return false;
    m_neighbours.push_back(neighbour);
    return true;
}

QVector<const Cell*> Cell::getNeighbours() const
{
    return m_neighbours;
}
```

6.3 cell.h File Reference

```
#include <QVector>
#include <QDebug>
```

Classes

- class Cell

  Contains the state, the next state and the neighbours.

6.4 cell.h

```
#ifndef CELL_H
#define CELL_H

#include <QVector>
#include <QDebug>

class Cell
{
public:
    Cell(unsigned int state = 0);

    void setState(unsigned int state);
    void validState();
    unsigned int getState() const;

    bool addNeighbour(const Cell* neighbour);
    QVector<const Cell*> getNeighbours() const;

private:
    unsigned int m_state;
    unsigned int m_nextState;

    QVector<const Cell*> m_neighbours;
};

#endif // CELL_H
```

6.5 cellhandler.cpp File Reference

```
#include <iostream>
#include "cellhandler.h"
```

6.6 cellhandler.cpp

```
#include <iostream>
#include "cellhandler.h"

CellHandler::CellHandler(const QString filename)
{
    QFile loadFile(filename);
    if (!loadFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qWarning("Couldn’t open given file.");
        throw QString(QObject::tr("Couldn’t open given file"));
    }

    QJsonParseError parseErr;
    QJsonDocument loadDoc(QJsonDocument::fromJson(loadFile.readAll(), &parseErr));



    if (loadDoc.isNull() || loadDoc.isEmpty()) {
        qWarning() << "Could not read data : ";
        qWarning() << parseErr.errorString();
    }

    // Loadding of the json file
    if (!load(loadDoc.object()))
    {
        qWarning("File not valid");
        throw QString(QObject::tr("File not valid"));
    }

    foundNeighbours();


}

CellHandler::CellHandler(const QVector<unsigned int> dimensions, generationTypes type, unsigned int stateMax, unsigned int density)
{
    m_dimensions = dimensions;
    QVector<unsigned int> position;
    unsigned int size = 1;

    // Set position vector to 0

    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        position.push_back(0);
        size *= m_dimensions.at(i);
    }


    // Creation of cells
    for (unsigned int j = 0; j < size; j++)
    {
        m_cells.insert(position, new Cell(0));

        positionIncrement(position);
    }

    if (type != empty)
        generate(type, stateMax, density);

}

CellHandler::~CellHandler()
{
    for (QMap<QVector<unsigned int>, Cell* >::iterator it = m_cells.begin(); it != m_cells.end(); ++it)
    {
        delete it.value();
    }
}

Cell *CellHandler::getCell(const QVector<unsigned int> position) const
{
    return m_cells.value(position);
}

void CellHandler::nextStates()
{
    for (QMap<QVector<unsigned int>, Cell* >::iterator it = m_cells.begin(); it != m_cells.end(); ++it)
    {
        it.value()->validState();
    }
}

bool CellHandler::save(QString filename)
{
    QFile saveFile(filename);
    if (!saveFile.open(QIODevice::WriteOnly)) {
        qWarning("Couldn’t create or open given file.");
        throw QString(QObject::tr("Couldn’t create or open given file"));
    }

    QJsonObject json;
    QString stringDimension;
    // Creation of the dimension string
    for (unsigned int i = 0; i < m_dimensions.size(); i++)
    {
        if (i != 0)
            stringDimension.push_back("x");
        stringDimension.push_back(QString::number(m_dimensions.at(i)));
    }
    json["dimensions"] = QJsonValue(stringDimension);

    QJsonArray cells;
    for (CellHandler::iterator it = begin(); it != end(); ++it)
    {
        cells.append(QJsonValue((int)it->getState()));
    }
    json["cells"] = cells;


    QJsonDocument saveDoc(json);
    saveFile.write(saveDoc.toJson());

    saveFile.close();
    return true;
}

void CellHandler::generate(CellHandler::generationTypes type, unsigned int stateMax, unsigned short density)
{
    if (type == random)
    {
        QVector<unsigned int> position;
        for (unsigned short i = 0; i < m_dimensions.size(); i++)
        {
            position.push_back(0);
        }
        QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
        for (unsigned int j = 0; j < m_cells.size(); j++)
        {
            unsigned int state = 0;
            // 0 have (1-density)% of chance of being generate
            if (generator.generateDouble()*100.0 < density)
                 state = (float)generator.generateDouble()*(stateMax+1);
            if (state > stateMax)
                state = stateMax;
            m_cells.value(position)->setState(state);
            m_cells.value(position)->validState();

            positionIncrement(position);
        }
    }
    else if (type == symetric)
    {
        QVector<unsigned int> position;
        for (unsigned short i = 0; i < m_dimensions.size(); i++)
        {
            position.push_back(0);
        }

        QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
        QVector<unsigned int> savedStates;
        for (unsigned int j = 0; j < m_cells.size(); j++)
        {
            if (j % m_dimensions.at(0) == 0)
                savedStates.clear();
            if (j % m_dimensions.at(0) < (m_dimensions.at(0)+1) / 2)
            {
                unsigned int state = 0;
                // 0 have (1-density)% of chance of being generate
                if (generator.generateDouble()*100.0 < density)
                    state = (float)generator.generateDouble()*(stateMax+1);
                if (state > stateMax)
                    state = stateMax;
                savedStates.push_back(state);
                m_cells.value(position)->setState(state);
                m_cells.value(position)->validState();
            }
            else
            {
                unsigned int i = savedStates.size() - (j % m_dimensions.at(0) - (m_dimensions.at(0)-1)/2 + (m_dimensions.at(0) % 2 == 0 ? 0 : 1));
                m_cells.value(position)->setState(savedStates.at(i));
                m_cells.value(position)->validState();
            }
            positionIncrement(position);


        }

    }
}

void CellHandler::print(std::ostream &stream)
{
    QVector<unsigned int> position;
    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        position.push_back(0);
    }
    for (unsigned int j = 0; j < m_cells.size(); j++)
    {
        stream << m_cells.value(position)->getState() << " ";
        position.replace(0, position.at(0)+1);
        for (unsigned short i = 0; i < m_dimensions.size(); i++)
        {
            if (position.at(i) >= m_dimensions.at(i))
            {
                position.replace(i, 0);
                stream << std::endl;
                if (i + 1 != m_dimensions.size())
                    position.replace(i+1, position.at(i+1)+1);
            }
        }
    }

}

CellHandler::iterator CellHandler::begin()
{
    return iterator(this);
}

bool CellHandler::end()
{
    return true;
}

bool CellHandler::load(const QJsonObject &json)
{
    if (!json.contains("dimensions") || !json["dimensions"].isString())
        return false;

    // RegExp to validate dimensions field format : "10x10"
    QRegExpValidator dimensionValidator(QRegExp("([0-9]*x?)*"));
    QString stringDimensions = json["dimensions"].toString();
    int pos= 0;
    if (dimensionValidator.validate(stringDimensions, pos) != QRegExpValidator::Acceptable)
        return false;

    // Split of dimensions field : "10x10" => "10", "10"
    QRegExp rx("x");
    QStringList list = json["dimensions"].toString().split(rx, QString::SkipEmptyParts);

    unsigned int product = 1;
    // Dimensions construction
    for (unsigned int i = 0; i < list.size(); i++)
    {
        product = product * list.at(i).toInt();
        m_dimensions.push_back(list.at(i).toInt());
    }
    if (!json.contains("cells") || !json["cells"].isArray())
        return false;

    QJsonArray cells = json["cells"].toArray();
    if (cells.size() != product)
        return false;

    QVector<unsigned int> position;
    // Set position vector to 0
    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        position.push_back(0);
    }

    // Creation of cells
    for (unsigned int j = 0; j < cells.size(); j++)
    {
        if (!cells.at(j).isDouble())
            return false;
        if (cells.at(j).toDouble() < 0)
            return false;
        m_cells.insert(position, new Cell(cells.at(j).toDouble()));

        positionIncrement(position);
    }

    return true;

}

void CellHandler::foundNeighbours()
{
    QVector<unsigned int> currentPosition;
    // Set position vector to 0
    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        currentPosition.push_back(0);
    }
    // Modification of all the cells
    for (unsigned int j = 0; j < m_cells.size(); j++)
    {
        // Get the list of the neighbours positions
        // This function is recursive
        QVector<QVector<unsigned int> > listPosition(getListNeighboursPositions(currentPosition));

        // Adding neighbours
        for (unsigned int i = 0; i < listPosition.size(); i++)
            m_cells.value(currentPosition)->addNeighbour(m_cells.value(listPosition.at(i)));

        positionIncrement(currentPosition);
    }
}

void CellHandler::positionIncrement(QVector<unsigned int> &pos, unsigned int value) const
{
    pos.replace(0, pos.at(0) + value); // adding the value to the first digit

    // Carry management
    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        if (pos.at(i) >= m_dimensions.at(i) && pos.at(i) < m_dimensions.at(i)*2)
        {
            pos.replace(i, 0);
            if (i + 1 != m_dimensions.size())
                pos.replace(i+1, pos.at(i+1)+1);
        }
        else if (pos.at(i) >= m_dimensions.at(i))
        {
            pos.replace(i, pos.at(i) - m_dimensions.at(i));
            if (i + 1 != m_dimensions.size())
                pos.replace(i+1, pos.at(i+1)+1);
            i--;
        }

    }
}

QVector<QVector<unsigned int> >& CellHandler::getListNeighboursPositions(const QVector<unsigned int> position) const
{
    QVector<QVector<unsigned int> > *list = getListNeighboursPositionsRecursive(position, position.size(), position);
    // We remove the position of the cell
    list->removeAll(position);
    return *list;
}

QVector<QVector<unsigned int> >* CellHandler::getListNeighboursPositionsRecursive(const QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const
{
    if (dimension == 0) // Stop condition
    {
        QVector<QVector<unsigned int> > *list = new QVector<QVector<unsigned int> >;
        return list;
    }
    QVector<QVector<unsigned int> > *listPositions = new QVector<QVector<unsigned int> >;

    QVector<unsigned int> modifiedPosition(lastAdd);

    // "x_d - 1" tree
    if (modifiedPosition.at(dimension-1) != 0) // Avoid "negative" position
        modifiedPosition.replace(dimension-1, position.at(dimension-1) - 1);
    listPositions->append(*getListNeighboursPositionsRecursive(position, dimension -1, modifiedPosition));
    if (!listPositions->count(modifiedPosition))
        listPositions->push_back(modifiedPosition);

    // "x_d" tree
    modifiedPosition.replace(dimension-1, position.at(dimension-1));
    listPositions->append(*getListNeighboursPositionsRecursive(position, dimension -1, modifiedPosition));
    if (!listPositions->count(modifiedPosition))
        listPositions->push_back(modifiedPosition);

    // "x_d + 1" tree
    if (modifiedPosition.at(dimension -1) + 1 < m_dimensions.at(dimension-1)) // Avoid position out of the cell space
        modifiedPosition.replace(dimension-1, position.at(dimension-1) +1);
    listPositions->append(*getListNeighboursPositionsRecursive(position, dimension -1, modifiedPosition));
    if (!listPositions->count(modifiedPosition))
        listPositions->push_back(modifiedPosition);

    return listPositions;

}
```

```

CellHandler::iterator::iterator(const CellHandler *handler):
        m_handler(handler), m_changedDimension(0)
{
    // Initialisation of m_position
    for (unsigned short i = 0; i < handler->m_dimensions.size(); i++)
    {
        m_position.push_back(0);
    }
    m_zero = m_position;
}

CellHandler::iterator &CellHandler::iterator::operator++()
{
    m_position.replace(0, m_position.at(0) + 1); // adding the value to the first digit

    m_changedDimension = 0;
    // Carry management
    for (unsigned short i = 0; i < m_handler->m_dimensions.size(); i++)
    {
        if (m_position.at(i) >= m_handler->m_dimensions.at(i))
        {
            m_position.replace(i, 0);
            m_changedDimension++;
            if (i + 1 != m_handler->m_dimensions.size())
                m_position.replace(i+1, m_position.at(i+1)+1);
        }

    }
    // If we return to zero, we have finished
    if (m_position == m_zero)
        m_finished = true;

    return *this;

}

Cell *CellHandler::iterator::operator->() const
{
    return m_handler->m_cells.value(m_position);
}

Cell *CellHandler::iterator::operator*() const
{
    return m_handler->m_cells.value(m_position);
}

unsigned int CellHandler::iterator::changedDimension() const
{
    return m_changedDimension;
}

```

6.7 cellhandler.h File Reference

```
#include <QString>
#include <QFile>
#include <QJsonDocument>
#include <QtWidgets>
#include <QMap>
#include <QRegExpValidator>
#include "cell.h"
```

Classes

- class CellHandler

  Cell container and cell generator.
- class CellHandler::iterator

  Implementation of iterator design pattern.

6.8 cellhandler.h

```
#ifndef CELLHANDLER_H
#define CELLHANDLER_H

#include <QString>
#include <QFile>
#include <QJsonDocument>
#include <QtWidgets>
#include <QMap>
#include <QRegExpValidator>

#include "cell.h"

class CellHandler
{
public:
    class iterator
    {
        friend class CellHandler;
    public:
        iterator(const CellHandler* handler);

        iterator& operator++();
        Cell* operator->() const;
        Cell* operator*() const;

        bool operator!=(bool finished) const { return (m_finished != finished); }
        unsigned int changedDimension() const;



    private:
        const CellHandler *m_handler;
        QVector<unsigned int> m_position;
        bool m_finished = false;
        QVector<unsigned int> m_zero;
        unsigned int m_changedDimension;
    };

    enum generationTypes {
        empty,
        random,
        symetric
    };

    CellHandler(const QString filename);
    CellHandler(const QVector<unsigned int> dimensions,
generationTypes type = empty, unsigned int stateMax = 1, unsigned int density = 50);
    virtual ~CellHandler();

    Cell* getCell(const QVector<unsigned int> position) const;
    void nextStates();

    bool save(QString filename);
    void generate(generationTypes type, unsigned int stateMax = 1, unsigned short
density = 50);
    void print(std::ostream &stream);

    iterator begin();
    bool end();

private:
    bool load(const QJsonObject &json);
    void foundNeighbours();
    void positionIncrement(QVector<unsigned int> &pos, unsigned int value = 1) const;
    QVector<QVector<unsigned int> > *getListNeighboursPositionsRecursive
(const QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const;
    QVector<QVector<unsigned int> > &getListNeighboursPositions(const
QVector<unsigned int> position) const;

    QVector<unsigned int> m_dimensions;
    QMap<QVector<unsigned int>, Cell* > m_cells;
};

#endif // CELLHANDLER_H
```

6.9 homepage.md File Reference

6.10 homepage.md

```
\page Presentation
# What is AutoCell
The purpose of this project is to create a Cellular Automate Simulator.
```

6.11 main.cpp File Reference

```
#include <QApplication>
#include <QDebug>
#include <iostream>
#include "cell.h"
#include "cellhandler.h"
```

Functions

- int main (int argc, char ∗argv[ ])

6.11.1 Function Documentation

6.11.1.1 main()

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

Definition at line 7 of file main.cpp.

References CellHandler::print(), and CellHandler::symetric.

6.12 main.cpp

```
00001 #include <QApplication>
00002 #include <QDebug>
00003 #include <iostream>
00004 #include "cell.h"
00005 #include "cellhandler.h"
00006 
00007 int main(int argc, char * argv[])
00008 {
00009     QApplication app(argc, argv);
00010     CellHandler handler(QVector<unsigned int>{30,30},
      CellHandler::symetric, 5);
00011     handler.print(std::cout);
00012     handler.save("testSave.atc");
00013     return 0;
00014 }
```

6.13 README.md File Reference

6.14 README.md

```
\mainpage

To generate the Documentation, go in Documentation directory and run ‘make‘.

It will generate html doc (in ‘output/html/index.html‘) and latex doc (pdf output directely in Documentation directory (‘docPdf.pdf‘).
```

Index