

GradAutomaton

P. Baillehache

March 29, 2020

Contents

1	Definitions	2
2	Interface	2
3	Code	16
3.1	gradautomaton.c	16
3.2	gradautomaton-inline.c	47
4	Makefile	64
5	Unit tests	64
6	Unit tests output	88

Introduction

GradAutomaton is a C library providing structures and functions to manipulate cellular automaton based on Grad structures.

It currently implements the following cellular automaton:

- GradAutomatonWolframOriginal: Cellular automaton described page 53 of "A new kind of science" by S. Wolfram
- GradAutomatonNeuraNet: Cellular Automaton on GradSquare and GradHexa where the automaton function is a NeuraNet

It uses the PBErr, Grad, NeuraNet, PBJson libraries.

1 Definitions

2 Interface

```
// ===== GRADAUTOMATON.H =====

#ifndef GRADAUTOMATON_H
#define GRADAUTOMATON_H

// ===== Include =====

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"
#include "grad.h"
#include "neuranet.h"

// ----- GrACell

// ===== Define =====

// ===== Data structure =====

typedef struct GrACell {

    // Index of the current status of the cell
    unsigned char curStatus;

    // Pointer toward the supporting GradCell
    GradCell* gradCell;

} GrACell;

typedef struct GrACellShort {

    // Parent GrACell
    GrACell gradAutomatonCell;

    // Double buffered status of the cell
    VecShort* status[2];

} GrACellShort;

typedef struct GrACellFloat {

    // Parent GrACell
    GrACell gradAutomatonCell;

    // Double buffered status of the cell
    VecFloat* status[2];

} GrACellFloat;

// ===== Functions declaration =====
```

```

// Create a new static GradAutomatonCell
GrACell GradAutomatonCellCreateStatic(
    GradCell* const gradCell);

// Create a new GrACellShort with a status vector of dimension 'dim'
// for the GradCell 'gradCell'
GrACellShort* GrACellCreateShort(
    const long dim,
    GradCell* const gradCell);

// Create a new GrACellFloat with a status vector of dimension 'dim'
// for the GradCell 'gradCell'
GrACellFloat* GrACellCreateFloat(
    const long dim,
    GradCell* const gradCell);

// Free the memory used by the GrACellShort 'that'
void _GrACellShortFree(GrACellShort** that);

// Free the memory used by the GrACellFloat 'that'
void _GrACellFloatFree(GrACellFloat** that);

// Switch the current status of the GrACell 'that'
#if BUILDMODE != 0
static inline
#endif
void _GrACellSwitchStatus(GrACell* const that);

// Return the current status of the GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
VecShort* _GrACellShortCurStatus(const GrACellShort* const that);

// Return the current status of the GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* _GrACellFloatCurStatus(const GrACellFloat* const that);

// Return the previous status of the GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
VecShort* _GrACellShortPrevStatus(const GrACellShort* const that);

// Return the previous status of the GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* _GrACellFloatPrevStatus(const GrACellFloat* const that);

// Return the 'iVal'-th value of the previous status of the
// GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
short _GrACellShortGetPrevStatus(
    const GrACellShort* const that,
    const unsigned long iVal);

// Return the 'iVal'-th value of the previous status of the

```

```

// GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
float _GrACellFloatGetPrevStatus(
    const GrACellFloat* const that,
    const unsigned long iVal);

// Set the 'iVal'-th value of the previous status of the
// GrACellShort 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void _GrACellShortSetPrevStatus(
    const GrACellShort* const that,
    const unsigned long iVal,
    const short val);

// Set the 'iVal'-th value of the previous status of the
// GrACellFloat 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void _GrACellFloatSetPrevStatus(
    const GrACellFloat* const that,
    const unsigned long iVal,
    const float val);

// Return the 'iVal'-th value of the current status of the
// GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
short _GrACellShortGetCurStatus(
    const GrACellShort* const that,
    const unsigned long iVal);

// Return the 'iVal'-th value of the current status of the
// GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
float _GrACellFloatGetCurStatus(
    const GrACellFloat* const that,
    const unsigned long iVal);

// Set the 'iVal'-th value of the current status of the
// GrACellShort 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void _GrACellShortSetCurStatus(
    const GrACellShort* const that,
    const unsigned long iVal,
    const short val);

// Set the 'iVal'-th value of the current status of the
// GrACellFloat 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void _GrACellFloatSetCurStatus(

```

```

    const GrACellFloat* const that,
        const unsigned long iVal,
        const float val);

// Return the GradCell of the GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
GradCell* _GrACellShortGradCell(const GrACellShort* const that);

// Return the GradCell of the GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
GradCell* _GrACellFloatGradCell(const GrACellFloat* const that);

// ===== Polymorphism =====

#define GrACellFree(G) _Generic(G, \
    GrACellShort*: _GrACellShortFree, \
    GrACellFloat*: _GrACellFloatFree, \
    default: PBErrInvalidPolymorphism)(G)

#define GrACellSwitchStatus(G) _Generic(G, \
    GrACell*: _GrACellSwitchStatus, \
    GrACellShort*: _GrACellSwitchStatus, \
    GrACellFloat*: _GrACellSwitchStatus, \
    default: PBErrInvalidPolymorphism)((GrACell*)(G))

#define GrACellCurStatus(G) _Generic(G, \
    GrACellShort*: _GrACellShortCurStatus, \
    const GrACellShort*: _GrACellShortCurStatus, \
    GrACellFloat*: _GrACellFloatCurStatus, \
    const GrACellFloat*: _GrACellFloatCurStatus, \
    default: PBErrInvalidPolymorphism)(G)

#define GrACellPrevStatus(G) _Generic(G, \
    GrACellShort*: _GrACellShortPrevStatus, \
    const GrACellShort*: _GrACellShortPrevStatus, \
    GrACellFloat*: _GrACellFloatPrevStatus, \
    const GrACellFloat*: _GrACellFloatPrevStatus, \
    default: PBErrInvalidPolymorphism)(G)

#define GrACellGetCurStatus(G, I) _Generic(G, \
    GrACellShort*: _GrACellShortGetCurStatus, \
    const GrACellShort*: _GrACellShortGetCurStatus, \
    GrACellFloat*: _GrACellFloatGetCurStatus, \
    const GrACellFloat*: _GrACellFloatGetCurStatus, \
    default: PBErrInvalidPolymorphism)(G, I)

#define GrACellGetPrevStatus(G, I) _Generic(G, \
    GrACellShort*: _GrACellShortGetPrevStatus, \
    const GrACellShort*: _GrACellShortGetPrevStatus, \
    GrACellFloat*: _GrACellFloatGetPrevStatus, \
    const GrACellFloat*: _GrACellFloatGetPrevStatus, \
    default: PBErrInvalidPolymorphism)(G, I)

#define GrACellSetCurStatus(G, I, V) _Generic(G, \
    GrACellShort*: _GrACellShortSetCurStatus, \
    GrACellFloat*: _GrACellFloatSetCurStatus, \
    default: PBErrInvalidPolymorphism)(G, I, V)

```

```

#define GrACellSetPrevStatus(G, I, V) _Generic(G, \
    GrACellShort*: _GrACellShortSetPrevStatus, \
    GrACellFloat*: _GrACellFloatSetPrevStatus, \
    default: PBErrInvalidPolymorphism)(G, I, V)

#define GrACellGradCell(G) _Generic(G, \
    GrACellShort*: _GrACellShortGradCell, \
    const GrACellShort*: _GrACellShortGradCell, \
    GrACellFloat*: _GrACellFloatGradCell, \
    const GrACellFloat*: _GrACellFloatGradCell, \
    default: PBErrInvalidPolymorphism)(G)

// ----- GrAFun

// ===== Define =====

// ===== Data structure =====

typedef enum GrAFunType {

    GrAFunTypeDummy,
    GrAFunTypeWolframOriginal,
    GrAFunTypeNeuraNet

} GrAFunType;

typedef struct GrAFun {

    // Type of GrAFun
    GrAFunType type;

} GrAFun;

// ===== Functions declaration =====

// Create a static GrAFun with type 'type'
GrAFun GrAFunCreateStatic(const GrAFunType type);

// Free the memory used by the GrAFun 'that'
void _GrAFunFreeStatic(GrAFun* that);

// Return the type of the GrAFun 'that'
#if BUILDMODE != 0
static inline
#endif
GrAFunType _GrAFunGetType(const GrAFun* const that);

// ----- GrAFunDummy

// ===== Define =====

// ===== Data structure =====

typedef struct GrAFunDummy {

    // GrAFun
    GrAFun grAFun;

} GrAFunDummy;

// ===== Functions declaration =====

```

```

// Create a new GrAFunDummy
GrAFunDummy* GrAFunCreateDummy(void);

// Free the memory used by the GrAFunDummy 'that'
void _GrAFunDummyFree(GrAFunDummy** that);

// ----- GrAFunWolframOriginal

// ===== Define =====

// ===== Data structure =====

typedef struct GrAFunWolframOriginal {

    // GrAFun
    GrAFun grAFun;

    // Rule, cf "A new kind of science" p.53
    unsigned char rule;

} GrAFunWolframOriginal;

// ===== Functions declaration =====

// Create a new GrAFunWolframOriginal
GrAFunWolframOriginal* GrAFunCreateWolframOriginal(
    const unsigned char rule);

// Free the memory used by the GrAFunWolframOriginal 'that'
void _GrAFunWolframOriginalFree(GrAFunWolframOriginal** that);

// Return the rule of the GrAFunWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned char GrAFunWolframOriginalGetRule(
    GrAFunWolframOriginal* const that);

// Apply the step function for the GrAFunWolframOriginal 'that'
// to the GrACellShort 'cell' in the GradSquare 'grad'
void _GrAFunWolframOriginalApply(
    GrAFunWolframOriginal* const that,
    GradSquare* const grad,
    GrACellShort* const cell);

// ----- GrAFunNeuraNet

// ===== Define =====

// ===== Data structure =====

typedef struct GrAFunNeuraNet {

    // GrAFun
    GrAFun grAFun;

    // NeuraNet applied to the cells
    NeuraNet* nn;

} GrAFunNeuraNet;

// ===== Functions declaration =====

```

```

// Create a new GrAFunNeuraNet
GrAFunNeuraNet* GrAFunCreateNeuraNet(
    const int nbIn,
    const int nbOut,
    const VecLong* const hiddenLayers);

// Free the memory used by the GrAFunNeuraNet 'that'
void _GrAFunNeuraNetFree(GrAFunNeuraNet** that);

// Return the NeuraNet of the GrAFunNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
NeuraNet* GrAFunNeuraNetNN(
    GrAFunNeuraNet* const that);

// Apply the step function for the GrAFunNeuraNet 'that'
// to the GrACellShort 'cell' in the GradSquare 'grad'
void _GrAFunNeuraNetApply(
    GrAFunNeuraNet* const that,
    Grad* const grad,
    GrACellFloat* const cell);

// ===== Polymorphism =====

#define GrAFunFree(G) _Generic(G, \
    GrAFun*: _GrAFunFreeStatic, \
    GrAFunDummy*: _GrAFunDummyFree, \
    GrAFunWolframOriginal*: _GrAFunWolframOriginalFree, \
    GrAFunNeuraNet*: _GrAFunNeuraNetFree, \
    default: PBErrInvalidPolymorphism)(G)

#define GrAFunGetType(G) _Generic(G, \
    GrAFun*: _GrAFunGetType, \
    const GrAFun*: _GrAFunGetType, \
    GrAFunDummy*: _GrAFunGetType, \
    const GrAFunDummy*: _GrAFunGetType, \
    GrAFunWolframOriginal*: _GrAFunGetType, \
    const GrAFunWolframOriginal*: _GrAFunGetType, \
    GrAFunNeuraNet*: _GrAFunGetType, \
    const GrAFunNeuraNet*: _GrAFunGetType, \
    default: PBErrInvalidPolymorphism)((const GrAFun*)(G))

#define GrAFunApply(F, G, C) _Generic(F, \
    GrAFunWolframOriginal*: _GrAFunWolframOriginalApply, \
    GrAFunNeuraNet*: _GrAFunNeuraNetApply, \
    default: PBErrInvalidPolymorphism)(F, G, C)

// ----- GradAutomaton

// ===== Define =====

// ===== Data structure =====

typedef enum GradAutomatonType {

    GradAutomatonTypeDummy,
    GradAutomatonTypeWolframOriginal,
    GradAutomatonTypeNeuraNet

} GradAutomatonType;

```



```

typedef struct GradAutomaton {

    // Type of the GradAutomaton
    GradAutomatonType type;

    // Dimension of the status vector of each cell
    long dimStatus;

    // Grad
    Grad* grad;

    // GrAFun
    GrAFun* fun;

} GradAutomaton;

// ===== Functions declaration =====

// Create a new static GradAutomaton
GradAutomaton GradAutomatonCreateStatic(
    const GradAutomatonType type,
    Grad* const grad,
    GrAFun* const fun,
    const long dimStatus);

// Return the Grad of the GradAutomaton 'that'
#if BUILDMODE != 0
static inline
#endif
Grad* _GradAutomatonGrad(const GradAutomaton* const that);

// Return the GrACellShort at position 'pos' for the
// GradAutomaton 'that'
#if BUILDMODE != 0
static inline
#endif
GrACell* _GradAutomatonCellPos(
    GradAutomaton* const that,
    const VecShort2D* const pos);

// Return the GrACellShort at index 'iCell' for the GradAutomaton 'that'
#if BUILDMODE != 0
static inline
#endif
GrACell* _GradAutomatonCellIndex(
    GradAutomaton* const that,
    const long iCell);

// Switch the status of all the cells of the GradAutomaton 'that'
void _GradAutomatonSwitchAllStatus(GradAutomaton* const that);

// Return the dimension of the status of the GradAutomaton 'that'
#if BUILDMODE != 0
static inline
#endif
long _GradAutomatonGetDimStatus(const GradAutomaton* const that);

// ----- GradAutomatonDummy

// ===== Define =====

```

```

// ===== Data structure =====

// GradSquare (2x2, no diag), GraFunDummy, GrACellShort dimension 1
typedef struct GradAutomatonDummy {

    // Parent GradAutomaton
    GradAutomaton gradAutomaton;

} GradAutomatonDummy;

// ===== Functions declaration =====

// Create a new GradAutomatonDummy
GradAutomatonDummy* GradAutomatonCreateDummy();

// Free the memory used by the GradAutomatonDummy 'that'
void GradAutomatonDummyFree(GradAutomatonDummy** that);

// Step the GradAutomatonDummy
void _GradAutomatonDummyStep(GradAutomatonDummy* const that);

// Return the Grad of the GradAutomatonDummy 'that'
#if BUILDMODE != 0
static inline
#endif
GradSquare* _GradAutomatonDummyGrad(
    const GradAutomatonDummy* const that);

// Return the GraFun of the GradAutomatonDummy 'that'
#if BUILDMODE != 0
static inline
#endif
GraFunDummy* _GradAutomatonDummyFun(
    const GradAutomatonDummy* const that);

// Return the GrACellShort at position 'pos' for the
// GradAutomatonDummy 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonDummyCellPos(
    GradAutomatonDummy* const that,
    const VecShort2D* const pos);

// Return the GrACellShort at index 'iCell' for the GradAutomatonDummy 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonDummyCellIndex(
    GradAutomatonDummy* const that,
    const long iCell);

// ----- GradAutomatonWorlframOriginal

// ===== Define =====

// ===== Data structure =====

// GradSquare (Nx1, no diag), GraFunWolframOriginal, GrACellShort dimension 1
typedef struct GradAutomatonWolframOriginal {

    // Parent GradAutomaton

```

```

    GradAutomaton gradAutomaton;

} GradAutomatonWolframOriginal;

// ===== Functions declaration =====

// Create a new GradAutomatonWolframOriginal
GradAutomatonWolframOriginal* GradAutomatonCreateWolframOriginal(
    const unsigned char rule,
    const long size);

// Free the memory used by the GradAutomatonWolframOriginal 'that'
void GradAutomatonWolframOriginalFree(
    GradAutomatonWolframOriginal** that);

// Step the GradAutomatonWolframOriginal
void _GradAutomatonWolframOriginalStep(
    GradAutomatonWolframOriginal* const that);

// JSON encoding of GradAutomatonWolframOriginal 'that'
JSONNode* _GradAutomatonWolframOriginalEncodeAsJSON(
    const GradAutomatonWolframOriginal* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool _GradAutomatonWolframOriginalDecodeAsJSON(
    GradAutomatonWolframOriginal** that,
    const JSONNode* const json);

// Return the Grad of the GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GradSquare* _GradAutomatonWolframOriginalGrad(
    const GradAutomatonWolframOriginal* const that);

// Return the GrAFun of the GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GrAFunWolframOriginal* _GradAutomatonWolframOriginalFun(
    const GradAutomatonWolframOriginal* const that);

// Return the GrACellShort at position 'pos' for the
// GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonWolframOriginalCellPos(
    GradAutomatonWolframOriginal* const that,
    const VecShort2D* const pos);

// Return the GrACellShort at index 'iCell' for the
// GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonWolframOriginalCellIndex(
    GradAutomatonWolframOriginal* const that,
    const long iCell);

// Print the GradAutomatonWolframOriginal 'that' on the FILE 'stream'
void _GradAutomatonWolframOriginalPrintln(

```

```

    GradAutomatonWolframOriginal* const that,
        FILE* stream);

// Save the GradAutomatonWolframOriginal 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if the GradAutomatonWolframOriginal could be saved,
// false else
bool _GradAutomatonWolframOriginalSave(
    const GradAutomatonWolframOriginal* const that,
        FILE* const stream,
        const bool compact);

// Load the GradAutomatonWolframOriginal 'that' from the stream 'stream'
// If 'that' is not null the memory is first freed
// Return true if the GradAutomatonWolframOriginal could be loaded,
// false else
bool _GradAutomatonWolframOriginalLoad(
    GradAutomatonWolframOriginal** that,
        FILE* const stream);

// ----- GradAutomatonNeuraNet

// ===== Define =====

// ===== Data structure =====

// GradSquare/GradHexa, GraFunNeuraNet, GrACellFloat
typedef struct GradAutomatonNeuraNet {

    // Parent GradAutomaton
    GradAutomaton gradAutomaton;

    // Number of hidden layers
    long nbHiddenLayers;

} GradAutomatonNeuraNet;

// ===== Functions declaration =====

// Create a new GradAutomatonNeuraNet with a GradSquare
GradAutomatonNeuraNet* GradAutomatonCreateNeuraNetSquare(
    const long dimStatus,
    const VecShort2D* const dimGrad,
    const bool diagLink,
    const long nbHiddenLayers);

// Create a new GradAutomatonNeuraNet with a GradHexa
GradAutomatonNeuraNet* GradAutomatonCreateNeuraNetHexa(
    const long dimStatus,
    const VecShort2D* const dimGrad,
    const GradHexaType gradType,
    const long nbHiddenLayers);

// Free the memory used by the GradAutomatonNeuraNet 'that'
void GradAutomatonNeuraNetFree(
    GradAutomatonNeuraNet** that);

// Step the GradAutomatonNeuraNet
void _GradAutomatonNeuraNetStep(GradAutomatonNeuraNet* const that);

// Return the Grad of the GradAutomatonNeuraNet 'that'

```

```

#if BUILDMODE != 0
static inline
#endif
Grad* _GradAutomatonNeuraNetGrad(
    const GradAutomatonNeuraNet* const that);

// Return the type of Grad of the GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
GradType GradAutomatonNeuraNetGetGradType(
    GradAutomatonNeuraNet* const that);

// Return the GrAFun of the GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
GrAFunNeuraNet* _GradAutomatonNeuraNetFun(
    const GradAutomatonNeuraNet* const that);

// Return the GrCellFloat at position 'pos' for the
// GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
GrCellFloat* _GradAutomatonNeuraNetCellPos(
    GradAutomatonNeuraNet* const that,
    const VecShort2D* const pos);

// Return the GrCellFloat at index 'iCell' for the
// GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
GrCellFloat* _GradAutomatonNeuraNetCellIndex(
    GradAutomatonNeuraNet* const that,
    const long iCell);

// JSON encoding of GradAutomatonNeuraNet 'that'
JSONNode* _GradAutomatonNeuraNetEncodeAsJSON(
    const GradAutomatonNeuraNet* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool _GradAutomatonNeuraNetDecodeAsJSON(
    GradAutomatonNeuraNet** that,
    const JSONNode* const json);

// Save the GradAutomatonNeuraNet 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if the GradAutomatonNeuraNet could be saved,
// false else
bool _GradAutomatonNeuraNetSave(
    const GradAutomatonNeuraNet* const that,
    FILE* const stream,
    const bool compact);

// Load the GradAutomatonWolframOriginal 'that' from the stream 'stream'
// If 'that' is not null the memory is first freed
// Return true if the GradAutomatonNeuraNet could be loaded,
// false else
bool _GradAutomatonNeuraNetLoad(

```

```

    GradAutomatonNeuraNet** that,
        FILE* const stream);

// Return the number of hidden layers of the GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
long GradAutomatonNeuraNetGetNbHiddenLayers(
    const GradAutomatonNeuraNet* const that);

// ===== Polymorphism =====

#define GradAutomatonSwitchAllStatus(G) _Generic(G, \
    GradAutomaton* : _GradAutomatonSwitchAllStatus, \
    GradAutomatonDummy* : _GradAutomatonSwitchAllStatus, \
    GradAutomatonWolframOriginal* : _GradAutomatonSwitchAllStatus, \
    GradAutomatonNeuraNet* : _GradAutomatonSwitchAllStatus, \
    default: PBErrInvalidPolymorphism)((GradAutomaton*)(G))

#define GradAutomatonStep(G) _Generic(G, \
    GradAutomatonDummy* : _GradAutomatonDummyStep, \
    GradAutomatonWolframOriginal* : _GradAutomatonWolframOriginalStep, \
    GradAutomatonNeuraNet* : _GradAutomatonNeuraNetStep, \
    default: PBErrInvalidPolymorphism)(G)

#define GradAutomatonGrad(G) _Generic(G, \
    GradAutomaton* : _GradAutomatonGrad, \
    const GradAutomaton* : _GradAutomatonGrad, \
    GradAutomatonDummy* : _GradAutomatonDummyGrad, \
    const GradAutomatonDummy* : _GradAutomatonDummyGrad, \
    GradAutomatonWolframOriginal* : _GradAutomatonWolframOriginalGrad, \
    const GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalGrad, \
    GradAutomatonNeuraNet* : _GradAutomatonNeuraNetGrad, \
    const GradAutomatonNeuraNet* : _GradAutomatonNeuraNetGrad, \
    default: PBErrInvalidPolymorphism)(G)

#define GradAutomatonFun(G) _Generic(G, \
    GradAutomatonDummy* : _GradAutomatonDummyFun, \
    const GradAutomatonDummy* : _GradAutomatonDummyFun, \
    GradAutomatonWolframOriginal* : _GradAutomatonWolframOriginalFun, \
    const GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalFun, \
    GradAutomatonNeuraNet* : _GradAutomatonNeuraNetFun, \
    const GradAutomatonNeuraNet* : _GradAutomatonNeuraNetFun, \
    default: PBErrInvalidPolymorphism)(G)

#define GradAutomatonCell(G, P) _Generic(G, \
    GradAutomaton* : _Generic(P, \
        VecShort2D* : _GradAutomatonCellPos, \
        const VecShort2D* : _GradAutomatonCellPos, \
        long : _GradAutomatonCellIndex, \
        const long : _GradAutomatonCellIndex, \
        default: PBErrInvalidPolymorphism), \
    GradAutomatonDummy* : _Generic(P, \
        VecShort2D* : _GradAutomatonDummyCellPos, \
        const VecShort2D* : _GradAutomatonDummyCellPos, \
        long : _GradAutomatonDummyCellIndex, \
        const long : _GradAutomatonDummyCellIndex, \
        default: PBErrInvalidPolymorphism), \
    GradAutomatonWolframOriginal* : _Generic(P, \
        VecShort2D* : _GradAutomatonWolframOriginalCellPos, \

```

```

    const VecShort2D*: _GradAutomatonWolframOriginalCellPos, \
    long: _GradAutomatonWolframOriginalCellIndex, \
    const long: _GradAutomatonWolframOriginalCellIndex, \
    default: PBErrInvalidPolymorphism), \
GradAutomatonNeuraNet* : _Generic(P, \
    VecShort2D*: _GradAutomatonNeuraNetCellPos, \
    const VecShort2D*: _GradAutomatonNeuraNetCellPos, \
    long: _GradAutomatonNeuraNetCellIndex, \
    const long: _GradAutomatonNeuraNetCellIndex, \
    default: PBErrInvalidPolymorphism), \
default: PBErrInvalidPolymorphism)(G, P)

#define GradAutomatonPrintln(G, S) _Generic(G, \
    GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalPrintln, \
    const GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalPrintln, \
    default: PBErrInvalidPolymorphism)(G, S)

#define GradAutomatonEncodeAsJSON(G) _Generic(G, \
    GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalEncodeAsJSON, \
    const GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalEncodeAsJSON, \
    GradAutomatonNeuraNet* : \
        _GradAutomatonNeuraNetEncodeAsJSON, \
    const GradAutomatonNeuraNet* : \
        _GradAutomatonNeuraNetEncodeAsJSON, \
    default: PBErrInvalidPolymorphism)(G)

#define GradAutomatonDecodeAsJSON(G, J) _Generic(G, \
    GradAutomatonWolframOriginal** : \
        _GradAutomatonWolframOriginalDecodeAsJSON, \
    GradAutomatonNeuraNet** : \
        _GradAutomatonNeuraNetDecodeAsJSON, \
    default: PBErrInvalidPolymorphism)(G, J)

#define GradAutomatonSave(G, S, C) _Generic(G, \
    GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalSave, \
    const GradAutomatonWolframOriginal* : \
        _GradAutomatonWolframOriginalSave, \
    GradAutomatonNeuraNet* : \
        _GradAutomatonNeuraNetSave, \
    const GradAutomatonNeuraNet* : \
        _GradAutomatonNeuraNetSave, \
    default: PBErrInvalidPolymorphism)(G, S, C)

#define GradAutomatonLoad(G, S) _Generic(G, \
    GradAutomatonWolframOriginal** : \
        _GradAutomatonWolframOriginalLoad, \
    GradAutomatonNeuraNet** : \
        _GradAutomatonNeuraNetLoad, \
    default: PBErrInvalidPolymorphism)(G, S)

#define GradAutomatonGetDimStatus(G) _Generic(G, \
    GradAutomaton* : \
        _GradAutomatonGetDimStatus, \
    const GradAutomaton* : \
        _GradAutomatonGetDimStatus, \
    GradAutomatonWolframOriginal* : \
        _GradAutomatonGetDimStatus, \

```

```

const GradAutomatonWolframOriginal* :\
    _GradAutomatonGetDimStatus, \
GradAutomatonNeuraNet* : \
    _GradAutomatonGetDimStatus, \
const GradAutomatonNeuraNet* :\
    _GradAutomatonGetDimStatus, \
default: PBErrInvalidPolymorphism)(((const GradAutomaton*)(G)))

// ===== static inline =====

#if BUILDMODE != 0
#include "gradautomaton-inline.c"
#endif

#endif

```

3 Code

3.1 gradautomaton.c

```

// ===== GRADAUTOMATON.C =====

// ===== Include =====

#include "gradautomaton.h"
#if BUILDMODE == 0
#include "gradautomaton-inline.c"
#endif

// ----- GrACell

// ===== Functions declaration =====

// ===== Functions implementation =====

// Create a new static GrACell
GrACell GradAutomatonCellCreateStatic(
    GradCell* const gradCell) {

    // Create the new GradAutomatonCell
    GrACell cell;

    // Set the properties
    cell.curStatus = 0;
    cell.gradCell = gradCell;

    // Return the new GradAutomatonCell
    return cell;
}

// Create a new GrACellShort with a status vector of dimension 'dim'
// for the GradCell 'gradCell'
GrACellShort* GrACellCreateShort(
    const long dim,
    GradCell* const gradCell) {

    // Allocate memory

```



```

GrACellShort* that =
    PBErrMalloc(
        GradAutomatonErr,
        sizeof(GrACellShort));

// Initialise properties
that->status[0] = VecShortCreate(dim);
that->status[1] = VecShortCreate(dim);
that->gradAutomatonCell = GradAutomatonCellCreateStatic(gradCell);

// Return the new GrACellShort
return that;
}

// Create a new GrACellFloat with a status vector of dimension 'dim'
// for the GradCell 'gradCell'
GrACellFloat* GrACellCreateFloat(
    const long dim,
    GradCell* const gradCell) {

    // Allocate memory
    GrACellFloat* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GrACellFloat));

    // Initialise properties
    that->status[0] = VecFloatCreate(dim);
    that->status[1] = VecFloatCreate(dim);
    that->gradAutomatonCell = GradAutomatonCellCreateStatic(gradCell);

    // Return the new GrACellFloat
    return that;
}

// Free the memory used by the GrACellShort 'that'
void _GrACellShortFree(GrACellShort** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

        // Do nothing
        return;
    }

    // Free memory
    VecFree(&((*that)->status[0]));
    VecFree(&((*that)->status[1]));
    free(*that);
    *that = NULL;
}

// Free the memory used by the GrACellFloat 'that'
void _GrACellFloatFree(GrACellFloat** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

```

```

    // Do nothing
    return;

}

// Free memory
VecFree(&((*that)->status[0]));
VecFree(&((*that)->status[1]));
free(*that);
*that = NULL;

}

// ----- GrAFun

// ===== Functions declaration =====

// ===== Functions implementation =====

// Create a static GrAFun with type 'type'
GrAFun GrAFunCreateStatic(const GrAFunType type) {

    // Declare the new GrAFun
    GrAFun that;

    // Set properties
    that.type = type;

    // Return the new GrAFun
    return that;

}

// Free the memory used by the GrAFun 'that'
void _GrAFunFreeStatic(GrAFun* that) {

    // If that is null
    if (that == NULL) {

        // Do nothing
        return;

    }

}

// ----- GrAFunDummy

// ===== Functions declaration =====

// ===== Functions implementation =====

// Create a new GrAFunDummy
GrAFunDummy* GrAFunCreateDummy(void) {

    // Declare the new GrAFun
    GrAFunDummy* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GrAFunDummy));

    // Set properties

```

```

    that->grAFun = GrAFunCreateStatic(GrAFunTypeDummy);

    // Return the new GrAFun
    return that;
}

// Free the memory used by the GrAFunDummy 'that'
void _GrAFunDummyFree(GrAFunDummy** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

        // Do nothing
        return;
    }

    // Free memory
    _GrAFunFreeStatic((GrAFun*)(*that));
    free(*that);
    *that = NULL;
}

// ----- GrAFunWolframOriginal
// ===== Functions declaration =====
// ===== Functions implementation =====

// Create a new GrAFunWolframOriginal
GrAFunWolframOriginal* GrAFunCreateWolframOriginal(
    const unsigned char rule) {

    // Declare the new GrAFun
    GrAFunWolframOriginal* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GrAFunWolframOriginal));

    // Set properties
    that->grAFun = GrAFunCreateStatic(GrAFunTypeWolframOriginal);
    that->rule = rule;

    // Return the new GrAFun
    return that;
}

// Free the memory used by the GrAFunWolframOriginal 'that'
void _GrAFunWolframOriginalFree(GrAFunWolframOriginal** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

        // Do nothing
        return;
    }

    // Free memory

```

```

_GrAFunFreeStatic((GrAFun*)(*that));
free(*that);
*that = NULL;

}

// Apply the step function for the GrAFunWolframOriginal 'that'
// to the GrACellShort 'cell' in the GradSquare 'grad'
void _GrAFunWolframOriginalApply(
    GrAFunWolframOriginal* const that,
    GradSquare* const grad,
    GrACellShort* const cell) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (grad == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'grad' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (cell == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'cell' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Declare a variable to memorize the current status of the
    // cell and its neighbour
    short status[3] = {0, 0, 0};

    // Get the current status of the left cell
    long leftLink =
        GradCellGetLink(
            GrACellGradCell(cell),
            GradSquareDirW);
    if (leftLink != -1) {

        GradCell* leftNeighbour =
            GradCellNeighbour(
                grad,
                GrACellGradCell(cell),
                GradSquareDirW);
    }

```

```

    GrACellShort* leftCell =
        (GrACellShort*)GradCellData(leftNeighbour);
    status[0] =
        VecGet(
            GrACellCurStatus(leftCell),
            0);

}

// Get the current status of the cell
status[1] =
    VecGet(
        GrACellCurStatus(cell),
        0);

// Get the current status of the right cell
long rightLink =
    GradCellGetLink(
        GrACellGradCell(cell),
        GradSquareDirE);
if (rightLink != -1) {

    GradCell* rightNeighbour =
        GradCellNeighbour(
            grad,
            GrACellGradCell(cell),
            GradSquareDirE);
    GrACellShort* rightCell =
        (GrACellShort*)GradCellData(rightNeighbour);
    status[2] =
        VecGet(
            GrACellCurStatus(rightCell),
            0);

}

// Get the corresponding mask in the rule
unsigned char mask =
    powi(
        2,
        ((status[0] * 2) + status[1]) * 2 + status[2]);

// Get the new status of the cell
short newStatus = 0;
if (GrAFunWolframOriginalGetRule(that) & mask) {

    newStatus = 1;

}

// Update the previous status with the new status
// (it will be switch later)
GrACellSetPrevStatus(
    cell,
    0,
    newStatus);

}

// ----- GrAFunNeuraNet
// ===== Functions declaration =====

```

```

// ===== Functions implementation =====

// Create a new GrAFunNeuraNet
GrAFunNeuraNet* GrAFunCreateNeuraNet(
    const int nbIn,
    const int nbOut,
    const VecLong* const hiddenLayers) {

    // Declare the new GrAFun
    GrAFunNeuraNet* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GrAFunNeuraNet));

    // Set properties
    that->grAFun = GrAFunCreateStatic(GrAFunTypeNeuraNet);
    that->nn =
        NeuraNetCreateFullyConnected(
            nbIn,
            nbOut,
            hiddenLayers);

    // Return the new GrAFunNeuraNet
    return that;
}

// Free the memory used by the GrAFunNeuraNet 'that'
void _GrAFunNeuraNetFree(GrAFunNeuraNet** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

        // Do nothing
        return;

    }

    // Free memory
    NeuraNetFree(&((*that)->nn));
    _GrAFunFreeStatic((GrAFun*)(*that));
    free(*that);
    *that = NULL;
}

// Apply the step function for the GrAFunNeuraNet 'that'
// to the GrACellShort 'cell' in the GradSquare 'grad'
void _GrAFunNeuraNetApply(
    GrAFunNeuraNet* const that,
    Grad* const grad,
    GrACellFloat* const cell) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }
#endif
}

```

```

}

if (grad == NULL) {

    GradAutomatonErr->_type = PBErrTypeNullPointer;
    sprintf(
        GradAutomatonErr->_msg,
        "'grad' is null");
    PBErrCatch(GradAutomatonErr);

}

if (cell == NULL) {

    GradAutomatonErr->_type = PBErrTypeNullPointer;
    sprintf(
        GradAutomatonErr->_msg,
        "'cell' is null");
    PBErrCatch(GradAutomatonErr);

}

#endif

// Get the number of links of the cell
int nbLinks = GradCellGetNbLink(GrACellGradCell(cell));

// Get the dimension of the input vector for the NeuraNet
long dimInput = (nbLinks + 1) * VecGetDim(GrACellCurStatus(cell));

// Declare a variable to memorize the input of the NeuraNet
VecFloat* input = VecFloatCreate(dimInput);

// Declare a variable to memorize the output of the NeuraNet
VecFloat* output = VecFloatCreate(VecGetDim(GrACellCurStatus(cell)));

// Set the current status of the cell in the input vector
for (
    long iDim = VecGetDim(output);
    iDim--;) {

    float val =
        GrACellGetCurStatus(
            cell,
            iDim);

    VecSet(
        input,
        iDim,
        val);

}

// Loop on the links toward neighbour cells
for (
    long iLink = nbLinks;
    iLink--;) {

    // Get the link
    long link =
        GradCellGetLink(

```

```

        GrACellGradCell(cell),
        iLink);

// If the link is active
if (link != -1) {

    // Get the neighbour cell and its status
    GradCell* neighbour =
        GradCellNeighbour(
            grad,
            GrACellGradCell(cell),
            iLink);
    GrACellFloat* neighbourCell =
        (GrACellFloat*)GradCellData(neighbour);

    // Set the current status of the neighbour cell in the
    // input vector
    for (
        long iDim = VecGetDim(output);
        iDim--;) {

        float val =
            GrACellGetCurStatus(
                neighbourCell,
                iDim);

        VecSet(
            input,
            (link + 1) * VecGetDim(output) + iDim,
            val);

    }

}

// Apply the NeuraNet
NNEval(
    GrAFunNeuraNetNN(that),
    input,
    output);

// Update the previous status with the output of the NeuraNet
// (it will be switch later)
for (
    long iDim = VecGetDim(output);
    iDim--;) {

    float val =
        VecGet(
            output,
            iDim);

    GrACellSetPrevStatus(
        cell,
        iDim,
        val);

}

// Free memory

```



```

    VecFree(&input);
    VecFree(&output);
}

// ----- GradAutomaton

// Create a new static GradAutomaton
GradAutomaton GradAutomatonCreateStatic(
    const GradAutomatonType type,
    Grad* const grad,
    GrAFun* const fun,
    const long dimStatus) {

#ifdef BUILDMODE == 0
    if (grad == NULL) {

        GradAutomatonErr->_type = PErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'grad' is null");
        PErrCatch(GradAutomatonErr);

    }

    if (fun == NULL) {

        GradAutomatonErr->_type = PErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'fun' is null");
        PErrCatch(GradAutomatonErr);

    }

#endif

    // Declare the new GradAutomaton
    GradAutomaton that;

    // Set the properties
    that.type = type;
    that.grad = grad;
    that.fun = fun;
    that.dimStatus = dimStatus;

    // Return the new GradAutomaton
    return that;
}

// Switch the status of all the cells of the GradAutomaton 'that'
void _GradAutomatonSwitchAllStatus(GradAutomaton* const that) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PErrCatch(GradAutomatonErr);

    }

#endif
}

```

```

    }

#endif

    // Get the number of cells in the grad
    long nbCell = GradGetArea(GradAutomatonGrad(that));

    // Loop on the cell
    for (
        long iCell = nbCell;
        iCell--;) {

        // Get the cell
        GrACell* cell =
            GradAutomatonCell(
                that,
                iCell);

        // Switch the status of the cell
        GrACellSwitchStatus(cell);

    }
}

// ----- GradAutomatonDummy

// Create a new GradAutomatonDummy
GradAutomatonDummy* GradAutomatonCreateDummy() {

    // Allocate memory for the new GradAutomatonDummy
    GradAutomatonDummy* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GradAutomatonDummy));

    // Create the associated Grad and GrAFun
    bool diagLink = false;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(
        &dim,
        0,
        2);
    VecSet(
        &dim,
        1,
        2);
    Grad* grad =
        (Grad*)GradSquareCreate(
            &dim,
            diagLink);
    GrAFun* fun = (GrAFun*)GrAFunCreateDummy();

    // Initialize the properties
    long dimStatus = 1;
    that->gradAutomaton =
        GradAutomatonCreateStatic(
            GradAutomatonTypeDummy,
            grad,
            fun,
            dimStatus);
}

```

```

// Add a GrACell to each cell of the Grad
VecShort2D pos = VecShortCreateStatic2D();
bool flag = true;
do {

    GradCell* cell =
        GradCellAt(
            grad,
            &pos);

    GrACellShort* cellStatus =
        GrACellCreateShort(
            dimStatus,
            cell);

    GradCellSetData(
        cell,
        cellStatus);

    flag =
        VecStep(
            &pos,
            &dim);

} while(flag);

// Return the new GradAutomatonDummy
return that;

}

// Free the memory used by the GradAutomatonDummy 'that'
void GradAutomatonDummyFree(GradAutomatonDummy** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

        // Do nothing
        return;

    }

    // Free the GrACell attached to the cells of the Grad
    VecShort2D pos = VecShortCreateStatic2D();
    bool flag = true;
    do {

        GradCell* cell =
            GradCellAt(
                GradAutomatonGrad(*that),
                &pos);

        GrACellShort* cellStatus = GradCellData(cell);

        GrACellFree(&cellStatus);

        flag =
            VecStep(
                &pos,
                GradDim(GradAutomatonGrad(*that)));

    } while(flag);
}

```

```

    } while(flag);

    // Free memory
    GradSquareFree((GradSquare**) &((*that)->gradAutomaton.grad));
    _GrAFunDummyFree((GrAFunDummy**) &((*that)->gradAutomaton.fun));
    free(*that);
    *that = NULL;
}

// Step the GradAutomatonDummyStep
void _GradAutomatonDummyStep(GradAutomatonDummy* const that) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    (void)that;
}

// ----- GradAutomatonWolframOriginal

// Create a new GradAutomatonWolframOriginal
GradAutomatonWolframOriginal* GradAutomatonCreateWolframOriginal(
    const unsigned char rule,
    const long size) {

    // Allocate memory for the new GradAutomatonWolframOriginal
    GradAutomatonWolframOriginal* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GradAutomatonWolframOriginal));

    // Create the associated Grad and GrAFun
    bool diagLink = false;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(
        &dim,
        0,
        size);
    VecSet(
        &dim,
        1,
        1);
    Grad* grad =
        (Grad*)GradSquareCreate(
            &dim,
            diagLink);
    GrAFun* fun = (GrAFun*)GrAFunCreateWolframOriginal(rule);

    // Initialize the properties
    long dimStatus = 1;

```

```

that->gradAutomaton =
    GradAutomatonCreateStatic(
        GradAutomatonTypeWolframOriginal,
        grad,
        fun,
        dimStatus);

// Get the index of the cell in the center of the Grad
long iCellCenter = size / 2;

// Add a GrACell to each cell of the Grad
for (
    long iCell = size;
    iCell--;) {

    GradCell* cell =
        GradCellAt(
            grad,
            iCell);

    GrACellShort* cellStatus =
        GrACellCreateShort(
            dimStatus,
            cell);

    // If it's the cell in the center of the Grad
    if (iCell == iCellCenter) {

        // Initialise the cell value to 1
        long iStatus = 0;
        short val = 1;
        GrACellSetPrevStatus(
            cellStatus,
            iStatus,
            val);
        GrACellSetCurStatus(
            cellStatus,
            iStatus,
            val);

    }

    GradCellSetData(
        cell,
        cellStatus);

};

// Return the new GradAutomatonWolframOriginal
return that;

}

// Free the memory used by the GradAutomatonWolframOriginal 'that'
void GradAutomatonWolframOriginalFree(
    GradAutomatonWolframOriginal** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

        // Do nothing
        return;
    }
}

```

```

}

// Get the number of cells in the grad
long nbCell = GradGetArea(GradAutomatonGrad(*that));

// Free the GrACell attached to the cells of the Grad
for (
    long iCell = nbCell;
    iCell--;) {

    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(*that),
            iCell);

    GrACellShort* cellStatus = GradCellData(cell);

    GrACellFree(&cellStatus);
}

// Free memory
GradSquareFree((GradSquare**) &((*that)->gradAutomaton.grad));
_GrAFunWolframOriginalFree(
    (GrAFunWolframOriginal**) &((*that)->gradAutomaton.fun));
free(*that);
*that = NULL;
}

// Step the GradAutomatonWolframOriginalStep
void _GradAutomatonWolframOriginalStep(
    GradAutomatonWolframOriginal* const that) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }
#endif

// Get the number of cells in the grad
long nbCell = GradGetArea(GradAutomatonGrad(that));

// Loop on the cell
for (
    long iCell = nbCell;
    iCell--;) {

    // Get the cell
    GrACellShort* cell =
        GradAutomatonCell(
            that,
            iCell);

```

```

        // Apply the step function to the cell
        GrAFunApply(
            GradAutomatonFun(that),
            GradAutomatonGrad(that),
            cell);
    }

    // Switch all the cells
    GradAutomatonSwitchAllStatus(that);
}

// Print the GradAutomatonWolframOriginal 'that' on the FILE 'stream'
void _GradAutomatonWolframOriginalPrintln(
    GradAutomatonWolframOriginal* const that,
    FILE* stream) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

    if (stream == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'stream' is null");
        PBErrCatch(GradAutomatonErr);
    }
}

#endif

    // Get the number of cells in the grad
    long nbCell = GradGetArea(GradAutomatonGrad(that));

    fprintf(
        stream,
        "[");

    // Loop on the cell
    for (
        long iCell = 0;
        iCell < nbCell;
        ++iCell) {

        // Get the cell
        GrACellShort* cell =
            GradAutomatonCell(
                that,
                iCell);

        // Get the current status of the cell
        short status =

```

```

        VecGet(
            GrACellCurStatus(cell),
            0);

    // Print the status
    if (status == 0) {

        fprintf(
            stream,
            " ");

    } else {

        fprintf(
            stream,
            "*");

    }

}

fprintf(
    stream,
    "]\n");
}

// JSON encoding of GradAutomatonWolframOriginal 'that'
JSONNode* _GradAutomatonWolframOriginalEncodeAsJSON(
    const GradAutomatonWolframOriginal* const that) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Create the JSON structure
    JSONNode* json = JSONCreate();

    // Declare a buffer to convert value into string
    char val[100];

    // Encode the rule
    unsigned char rule =
        GrAFunWolframOriginalGetRule(GradAutomatonFun(that));
    sprintf(
        val,
        "%d",
        rule);
    JSONAddProp(
        json,
        "rule",
        val);

```



```

// Encode the size
const VecShort2D* dim = GradDim(GradAutomatonGrad(that));
long size =
    VecGet(
        dim,
        0);
sprintf(
    val,
    "%ld",
    size);
JSONAddProp(
    json,
    "size",
    val);

// Return the created JSON
return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool _GradAutomatonWolframOriginalDecodeAsJSON(
    GradAutomatonWolframOriginal** that,
    const JSONNode* const json) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (json == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'json' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

// If 'that' is already allocated
if (*that != NULL) {

    // Free memory
    GradAutomatonWolframOriginalFree(that);

}

// Decode the rule
JSONNode* prop =
    JSONProperty(
        json,

```

```

        "rule");
    if (prop == NULL) {

        return false;

    }

    unsigned char rule = atoi(JSONLblVal(prop));

    // Decode the size
    prop =
        JSONProperty(
            json,
            "size");
    if (prop == NULL) {

        return false;

    }

    long size = atol(JSONLblVal(prop));

    // Create the GradAutomatonWolframOriginal
    *that =
        GradAutomatonCreateWolframOriginal(
            rule,
            size);

    // Return the success code
    return true;

}

// Save the GradAutomatonWolframOriginal 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if the GradAutomatonWolframOriginal could be saved,
// false else
bool _GradAutomatonWolframOriginalSave(
    const GradAutomatonWolframOriginal* const that,
        FILE* const stream,
        const bool compact) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (stream == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'stream' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif
}

```

```

    }

#endif

    // Get the JSON encoding
    JSONNode* json = GradAutomatonEncodeAsJSON(that);

    // Save the JSON
    bool ret =
        JSONSave(
            json,
            stream,
            compact);

    // Free memory
    JSONFree(&json);

    // Return success code
    return ret;
}

// Load the GradAutomatonWolframOriginal 'that' from the stream 'stream'
// If 'that' is not null the memory is first freed
// Return true if the GradAutomatonWolframOriginal could be loaded,
// false else
bool _GradAutomatonWolframOriginalLoad(
    GradAutomatonWolframOriginal** that,
    FILE* const stream) {

    #if BUILDMODE == 0

        if (that == NULL) {

            GradAutomatonErr->_type = PBErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'that' is null");
            PBErrCatch(GradAutomatonErr);

        }

        if (stream == NULL) {

            GradAutomatonErr->_type = PBErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'stream' is null");
            PBErrCatch(GradAutomatonErr);

        }

    #endif

    // Declare a json to load the encoded data
    JSONNode* json = JSONCreate();

    // Load the whole encoded data
    bool ret =
        JSONLoad(
            json,

```

```

        stream);

    if (ret == true) {

        // Decode the data from the JSON
        ret =
            GradAutomatonDecodeAsJSON(
                that,
                json);

    }

    // Free the memory used by the JSON
    JSONFree(&json);

    // Return the success code
    return ret;

}

// ----- GradAutomatonNeuraNet

// Create a new GradAutomatonNeuraNet with a GradSquare
GradAutomatonNeuraNet* GradAutomatonCreateNeuraNetSquare(
    const long dimStatus,
    const VecShort2D* const dimGrad,
    const bool diagLink,
    const long nbHiddenLayers) {

    // Allocate memory for the new GradAutomatonNeuraNet
    GradAutomatonNeuraNet* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GradAutomatonNeuraNet));

    // Create the associated Grad and GrAFun
    Grad* grad =
        (Grad*)GradSquareCreate(
            dimGrad,
            diagLink);
    int nbIn = 0;
    if (diagLink == true) {

        nbIn = dimStatus * 9;

    } else {

        nbIn = dimStatus * 5;

    }

    int nbOut = dimStatus;
    VecLong* hiddenLayers = VecLongCreate(nbHiddenLayers);
    for (
        int iLayer = nbHiddenLayers;
        iLayer--;) {

        VecSet(
            hiddenLayers,
            iLayer,
            nbIn);
    }
}

```

```

}

GrAFun* fun =
    (GrAFun*)GrAFunCreateNeuraNet(
        nbIn,
        nbOut,
        hiddenLayers);

// Initialize the properties
that->nbHiddenLayers = nbHiddenLayers;
that->gradAutomaton =
    GradAutomatonCreateStatic(
        GradAutomatonTypeNeuraNet,
        grad,
        fun,
        dimStatus);

// Add a GrACell to each cell of the Grad
long area = GradGetArea(GradAutomatonGrad(that));
for (
    long iCell = area;
    iCell--;) {

    GradCell* cell =
        GradCellAt(
            grad,
            iCell);

    GrACellFloat* cellStatus =
        GrACellCreateFloat(
            dimStatus,
            cell);

    GradCellSetData(
        cell,
        cellStatus);

}

// Return the new GradAutomatonNeuraNet
return that;

}

// Create a new GradAutomatonNeuraNet with a GradHexa
GradAutomatonNeuraNet* GradAutomatonCreateNeuraNetHexa(
    const long dimStatus,
    const VecShort2D* const dimGrad,
    const GradHexaType gradType,
    const long nbHiddenLayers) {

    // Allocate memory for the new GradAutomatonNeuraNet
    GradAutomatonNeuraNet* that =
        PBErrMalloc(
            GradAutomatonErr,
            sizeof(GradAutomatonNeuraNet));

    // Create the associated Grad and GrAFun
    Grad* grad = NULL;
    switch (gradType) {

        case GradHexaTypeEvenQ:

```

```

        grad = (Grad*)GradHexaCreateEvenQ(
            dimGrad);
        break;
    case GradHexaTypeEvenR:
        grad = (Grad*)GradHexaCreateEvenR(
            dimGrad);
        break;
    case GradHexaTypeOddQ:
        grad = (Grad*)GradHexaCreateOddQ(
            dimGrad);
        break;
    case GradHexaTypeOddR:
        grad = (Grad*)GradHexaCreateOddR(
            dimGrad);
        break;
    default:
        break;
}

int nbIn = dimStatus * 6;
int nbOut = dimStatus;
VecLong* hiddenLayers = VecLongCreate(nbHiddenLayers);
for (
    int iLayer = nbHiddenLayers;
    iLayer--;) {

    VecSet(
        hiddenLayers,
        iLayer,
        nbIn);
}

GrAFun* fun =
    (GrAFun*)GrAFunCreateNeuraNet(
        nbIn,
        nbOut,
        hiddenLayers);

// Initialize the properties
that->nbHiddenLayers = nbHiddenLayers;
that->gradAutomaton =
    GradAutomatonCreateStatic(
        GradAutomatonTypeNeuraNet,
        grad,
        fun,
        dimStatus);

// Add a GrACell to each cell of the Grad
long area = GradGetArea(GradAutomatonGrad(that));
for (
    long iCell = area;
    iCell--;) {

    GradCell* cell =
        GradCellAt(
            grad,
            iCell);

    GrACellFloat* cellStatus =
        GrACellCreateFloat(

```

```

        dimStatus,
        cell);

    GradCellSetData(
        cell,
        cellStatus);
}

// Return the new GradAutomatonNeuraNet
return that;
}

// Free the memory used by the GradAutomatonNeuraNet 'that'
void GradAutomatonNeuraNetFree(
    GradAutomatonNeuraNet** that) {

    // If that is null
    if (that == NULL || *that == NULL) {

        // Do nothing
        return;
    }

    // Free the GrACell attached to the cells of the Grad
    long area = GradGetArea(GradAutomatonGrad(*that));
    for (
        long iCell = area;
        iCell--;) {

        GradCell* cell =
            GradCellAt(
                GradAutomatonGrad(*that),
                iCell);

        GradCellFloat* cellStatus = GradCellData(cell);

        GrACellFree(&cellStatus);
    }

    // Free memory
    GradSquareFree((GradSquare**)&((*that)->gradAutomaton.grad));
    free(*that);
    *that = NULL;
}

// Step the GradAutomatonNeuraNetStep
void _GradAutomatonNeuraNetStep(GradAutomatonNeuraNet* const that) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }
#endif
}

```

```

    }

#endif

    // Get the number of cells in the grad
    long nbCell = GradGetArea(GradAutomatonGrad(that));

    // Loop on the cell
    for (
        long iCell = nbCell;
        iCell--;) {

        // Get the cell
        GrACellFloat* cell =
            GradAutomatonCell(
                that,
                iCell);

        // Apply the step function to the cell
        GrAFunApply(
            GradAutomatonFun(that),
            GradAutomatonGrad(that),
            cell);

    }

    // Switch all the cells
    GradAutomatonSwitchAllStatus(that);

}

// JSON encoding of GradAutomatonNeuraNet 'that'
JSONNode* _GradAutomatonNeuraNetEncodeAsJSON(
    const GradAutomatonNeuraNet* const that) {

#if BUILDMODE == 0

    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Create the JSON structure
    JSONNode* json = JSONCreate();

    // Declare a buffer to convert value into string
    char val[100];

    // Encode the type of the Grad
    GradType typeGrad = GradGetType(GradAutomatonGrad(that));
    sprintf(
        val,
        "%d",
        typeGrad);
    JSONAddProp(

```



```

    json,
    "typeGrad",
    val);

// Encode the dimensions of the Grad
const VecShort2D* dimGrad = GradDim(GradAutomatonGrad(that));
JSONNode* dimGradJSON = VecEncodeAsJSON((VecShort*)dimGrad);
JSONAddProp(
    json,
    "dimGrad",
    dimGradJSON);

// Encode the dimensions of the status
long dimStatus = GradAutomatonGetDimStatus(that);
sprintf(
    val,
    "%ld",
    dimStatus);
JSONAddProp(
    json,
    "dimStatus",
    val);

// Encode the number of hidden layers
long nbHiddenLayers = GradAutomatonNeuraNetGetNbHiddenLayers(that);
sprintf(
    val,
    "%ld",
    nbHiddenLayers);
JSONAddProp(
    json,
    "nbHiddenLayers",
    val);

// If the associated grad is of type hexa
if (typeGrad == GradTypeHexa) {

    // Encode the type of GradHexa
    GradHexaType typeGradHexa =
        GradHexaGetType((GradHexa*)GradAutomatonGrad(that));
    sprintf(
        val,
        "%d",
        typeGradHexa);
    JSONAddProp(
        json,
        "typeGradHexa",
        val);

// Else, if the associated grad is of type hexa
} else if (typeGrad == GradTypeSquare) {

    // Encode the diagonal link flag
    bool diagLink =
        GradSquareHasDiagonalLink((GradSquare*)GradAutomatonGrad(that));
    sprintf(
        val,
        "%d",
        diagLink);
    JSONAddProp(
        json,
        "diagLink",

```

```

        val);

    }

    // Encode the NeuraNet
    const NeuraNet* nn =
        GrAFunNeuraNetNN((GrAFunNeuraNet*)(GradAutomatonFun(that)));
    JSONNode* nnJSON = NNEncodeAsJSON(nn);
    JSONAddProp(
        json,
        "nn",
        nnJSON);

    // Return the created JSON
    return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool _GradAutomatonNeuraNetDecodeAsJSON(
    GradAutomatonNeuraNet** that,
    const JSONNode* const json) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (json == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'json' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // If 'that' is already allocated
    if (*that != NULL) {

        // Free memory
        GradAutomatonNeuraNetFree(that);

    }

    // Decode the type of grad
    JSONNode* prop =
        JSONProperty(
            json,
            "typeGrad");
    if (prop == NULL) {

```

```

        return false;
    }

    GradType typeGrad = atoi(JSONLblVal(prop));

    // Decode the dimension of the status
    prop =
        JSONProperty(
            json,
            "dimStatus");
    if (prop == NULL) {

        return false;
    }

    long dimStatus = atol(JSONLblVal(prop));

    // Decode the dimensions of the Grad
    prop =
        JSONProperty(
            json,
            "dimGrad");
    if (prop == NULL) {

        return false;
    }

    VecShort2D* dimGrad = NULL;
    bool ret =
        VecDecodeAsJSON(
            (VecShort**)(&dimGrad),
            prop);
    if (ret == false) {

        return false;
    }

    // Decode the number of hidden layers
    prop =
        JSONProperty(
            json,
            "nbHiddenLayers");
    if (prop == NULL) {

        return false;
    }

    long nbHiddenLayers = atol(JSONLblVal(prop));

    // If the associated grad is of type hexa
    if (typeGrad == GradTypeHexa) {

        // Decode the type of grad hexa
        prop =
            JSONProperty(
                json,
                "typeGradHexa");
    }

```

```

if (prop == NULL) {

    return false;

}

GradHexaType typeGradHexa = atoi(JSONLblVal(prop));

// Create the GradAutomatonNeuraNet
*that =
    GradAutomatonCreateNeuraNetHexa(
        dimStatus,
        dimGrad,
        typeGradHexa,
        nbHiddenLayers);

// Else, if the associated grad is of type square
} else if (typeGrad == GradTypeSquare) {

    // Decode the diagonal link flag
    prop =
        JSONProperty(
            json,
            "diagLink");
    if (prop == NULL) {

        return false;

    }

    bool diagLink = atoi(JSONLblVal(prop));

    // Create the GradAutomatonNeuraNet
    *that =
        GradAutomatonCreateNeuraNetSquare(
            dimStatus,
            dimGrad,
            diagLink,
            nbHiddenLayers);

} else {

    return false;

}

// Load the NeuraNet
prop =
    JSONProperty(
        json,
        "nn");
if (prop == NULL) {

    return false;

}

ret =
    NNDecodeAsJSON(
        &(((GrAFunNeuraNet*)GradAutomatonFun(*that))->nn),
        prop);
if (ret == false) {

```

```

        return false;
    }

    // Free memory
    VecFree((VecShort**>(&dimGrad));

    // Return the success code
    return true;
}

// Save the GradAutomatonNeuraNet 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if the GradAutomatonNeuraNet could be saved,
// false else
bool _GradAutomatonNeuraNetSave(
    const GradAutomatonNeuraNet* const that,
    FILE* const stream,
    const bool compact) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (stream == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'stream' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Get the JSON encoding
    JSONNode* json = GradAutomatonEncodeAsJSON(that);

    // Save the JSON
    bool ret =
        JSONSave(
            json,
            stream,
            compact);

    // Free memory
    JSONFree(&json);

    // Return success code
    return ret;
}

```

```

}

// Load the GradAutomatonWolframOriginal 'that' from the stream 'stream'
// If 'that' is not null the memory is first freed
// Return true if the GradAutomatonNeuraNet could be loaded,
// false else
bool _GradAutomatonNeuraNetLoad(
    GradAutomatonNeuraNet** that,
    FILE* const stream) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (stream == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'stream' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Declare a json to load the encoded data
    JSONNode* json = JSONCreate();

    // Load the whole encoded data
    bool ret =
        JSONLoad(
            json,
            stream);

    if (ret == true) {

        // Decode the data from the JSON
        ret =
            GradAutomatonDecodeAsJSON(
                that,
                json);

    }

    // Free the memory used by the JSON
    JSONFree(&json);

    // Return the success code
    return ret;

}

```

3.2 gradautomaton-inline.c

```
// ===== GRADAUTOMATON_INLINE.C =====

// ----- GrACell

// ===== Functions implementation =====

// Switch the current status of the GrACell 'that'
#if BUILDMODE != 0
static inline
#endif
void _GrACellSwitchStatus(GrACell* const that) {

    #if BUILDMODE == 0
        if (that == NULL) {

            GradAutomatonErr->_type = PErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'that' is null");
            PErrCatch(GradAutomatonErr);

        }

    #endif

    that->curStatus = 1 - that->curStatus;

}

// Return the current status of the GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
VecShort* _GrACellShortCurStatus(const GrACellShort* const that) {

    #if BUILDMODE == 0
        if (that == NULL) {

            GradAutomatonErr->_type = PErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'that' is null");
            PErrCatch(GradAutomatonErr);

        }

    #endif

    return that->status[that->gradAutomatonCell.curStatus];

}

// Return the current status of the GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* _GrACellFloatCurStatus(const GrACellFloat* const that) {

    #if BUILDMODE == 0
        if (that == NULL) {
```

```

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

#endif

    return that->status[that->gradAutomatonCell.curStatus];
}

// Return the previous status of the GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
VecShort* _GrACellShortPrevStatus(const GrACellShort* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

#endif

    return that->status[1 - that->gradAutomatonCell.curStatus];
}

// Return the previous status of the GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* _GrACellFloatPrevStatus(const GrACellFloat* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

#endif

    return that->status[1 - that->gradAutomatonCell.curStatus];
}

// Return the 'iVal'-th value of the previous status of the

```



```

// GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
short _GrACellShortGetPrevStatus(
    const GrACellShort* const that,
    const unsigned long iVal) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    return VecGet(
        GrACellPrevStatus(that),
        iVal);

}

// Return the 'iVal'-th value of the previous status of the
// GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
float _GrACellFloatGetPrevStatus(
    const GrACellFloat* const that,
    const unsigned long iVal) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    return VecGet(
        GrACellPrevStatus(that),
        iVal);

}

// Set the 'iVal'-th value of the previous status of the
// GrACellShort 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void _GrACellShortSetPrevStatus(
    const GrACellShort* const that,

```

```

        const unsigned long iVal,
            const short val) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    VecSet(
        GrACellPrevStatus(that),
        iVal,
        val);

}

// Set the 'iVal'-th value of the previous status of the
// GrACellFloat 'that' to 'val'
#ifdef BUILDMODE != 0
static inline
#endif
void _GrACellFloatSetPrevStatus(
    const GrACellFloat* const that,
    const unsigned long iVal,
    const float val) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    VecSet(
        GrACellPrevStatus(that),
        iVal,
        val);

}

// Return the 'iVal'-th value of the current status of the
// GrACellShort 'that'
#ifdef BUILDMODE != 0
static inline
#endif
short _GrACellShortGetCurStatus(
    const GrACellShort* const that,
    const unsigned long iVal) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    return VecGet(
        GrACellCurStatus(that),
        iVal);

}

// Return the 'iVal'-th value of the current status of the
// GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
float _GrACellFloatGetCurStatus(
    const GrACellFloat* const that,
    const unsigned long iVal) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    return VecGet(
        GrACellCurStatus(that),
        iVal);

}

// Set the 'iVal'-th value of the current status of the
// GrACellShort 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void _GrACellShortSetCurStatus(
    const GrACellShort* const that,
    const unsigned long iVal,
    const short val) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(

```

```

        GradAutomatonErr->_msg,
        "'that' is null");
    PBErrCatch(GradAutomatonErr);

}

#endif

VecSet(
    GrACellCurStatus(that),
    iVal,
    val);

}

// Set the 'iVal'-th value of the current status of the
// GrACellFloat 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void _GrACellFloatSetCurStatus(
    const GrACellFloat* const that,
    const unsigned long iVal,
    const float val) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    VecSet(
        GrACellCurStatus(that),
        iVal,
        val);

}

// Return the GradCell of the GrACellShort 'that'
#if BUILDMODE != 0
static inline
#endif
GradCell* _GrACellShortGradCell(const GrACellShort* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

}

```

```

#endif

    return that->gradAutomatonCell.gradCell;

}

// Return the GradCell of the GrACellFloat 'that'
#if BUILDMODE != 0
static inline
#endif
GradCell* _GrACellFloatGradCell(const GrACellFloat* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    return that->gradAutomatonCell.gradCell;

}

// ----- GrAFun

// ===== Functions implementation =====

// Return the type of the GrAFun 'that'
#if BUILDMODE != 0
static inline
#endif
GrAFunType _GrAFunGetType(const GrAFun* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    return that->type;

}

// ----- GrAFunWolframOriginal

// ===== Functions implementation =====

// Return the rule of the GrAFunWolframOriginal 'that'
#if BUILDMODE != 0

```

```

static inline
#endif
unsigned char GrAFunWolframOriginalGetRule(
    GrAFunWolframOriginal* const that) {

    #if BUILDMODE == 0
        if (that == NULL) {

            GradAutomatonErr->_type = PBErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'that' is null");
            PBErrCatch(GradAutomatonErr);

        }

    #endif

    return that->rule;

}

// ----- GrAFunNeuraNet
// ===== Functions implementation =====

// Return the NeuraNet of the GrAFunNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
NeuraNet* GrAFunNeuraNetNN(
    GrAFunNeuraNet* const that) {

    #if BUILDMODE == 0
        if (that == NULL) {

            GradAutomatonErr->_type = PBErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'that' is null");
            PBErrCatch(GradAutomatonErr);

        }

    #endif

    return that->nn;

}

// ----- GradAutomaton
// ===== Functions implementation =====

// Return the Grad of the GradAutomaton 'that'
#if BUILDMODE != 0
static inline
#endif
Grad* _GradAutomatonGrad(const GradAutomaton* const that) {

    #if BUILDMODE == 0
        if (that == NULL) {

```

```

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

#endif

    // Return the Grad
    return that->grad;
}

// Return the GrACellShort at position 'pos' for the
// GradAutomaton 'that'
#if BUILDMODE != 0
static inline
#endif
GrACell* _GradAutomatonCellPos(
    GradAutomaton* const that,
    const VecShort2D* const pos) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

    if (pos == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'pos' is null");
        PBErrCatch(GradAutomatonErr);
    }

#endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            pos);

    // Return the GrACellShort associated to the cell
    return (GrACell*)GradCellData(cell);
}

// Return the GrACellShort at index 'iCell' for the GradAutomaton 'that'
#if BUILDMODE != 0
static inline

```

```

#endif
GrACell* _GradAutomatonCellIndex(
    GradAutomaton* const that,
    const long iCell) {

    #if BUILDMODE == 0
        if (that == NULL) {

            GradAutomatonErr->_type = PBErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'that' is null");
            PBErrCatch(GradAutomatonErr);

        }

    #endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            iCell);

    // Return the GrACellShort associated to the cell
    return (GrACell*)GradCellData(cell);

}

// Return the dimension of the status of the GradAutomaton 'that'
#if BUILDMODE != 0
static inline
#endif
long _GradAutomatonGetDimStatus(const GradAutomaton* const that) {

    #if BUILDMODE == 0
        if (that == NULL) {

            GradAutomatonErr->_type = PBErrTypeNullPointer;
            sprintf(
                GradAutomatonErr->_msg,
                "'that' is null");
            PBErrCatch(GradAutomatonErr);

        }

    #endif

    // Return the dimension of the status
    return that->dimStatus;

}

// ----- GradAutomatonDummy

// ===== Functions implementation =====

// Return the Grad of the GradAutomatonDummy 'that'
#if BUILDMODE != 0
static inline
#endif
GradSquare* _GradAutomatonDummyGrad(

```



```

    const GradAutomatonDummy* const that) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Return the Grad
    return (GradSquare*)((GradAutomaton*)that)->grad;

}

// Return the GrAFun of the GradAutomatonDummy 'that'
#ifdef BUILDMODE != 0
static inline
#endif
GrAFunDummy* _GradAutomatonDummyFun(
    const GradAutomatonDummy* const that) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Return the GrAFun
    return (GrAFunDummy*)((GradAutomaton*)that)->fun;

}

// Return the GrACellShort at position 'pos' for the
// GradAutomatonDummy 'that'
#ifdef BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonDummyCellPos(
    GradAutomatonDummy* const that,
    const VecShort2D* const pos) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

```

```

    }

    if (pos == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'pos' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            pos);

    // Return the GrACellShort associated to the cell
    return (GrACellShort*)GradCellData(cell);

}

// Return the GrACellShort at index 'iCell' for the
// GradAutomatonDummy 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonDummyCellIndex(
    GradAutomatonDummy* const that,
    const long iCell) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            iCell);

    // Return the GrACellShort associated to the cell
    return (GrACellShort*)GradCellData(cell);

}

// ----- GradAutomatonWolframOriginal
// ===== Functions implementation =====

```

```

// Return the Grad of the GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GradSquare* _GradAutomatonWolframOriginalGrad(
    const GradAutomatonWolframOriginal* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Return the Grad
    return (GradSquare*)((GradAutomaton*)that)->grad;
}

// Return the GrAFun of the GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GrAFunWolframOriginal* _GradAutomatonWolframOriginalFun(
    const GradAutomatonWolframOriginal* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Return the GrAFun
    return (GrAFunWolframOriginal*)((GradAutomaton*)that)->fun;
}

// Return the GrACellShort at position 'pos' for the
// GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonWolframOriginalCellPos(
    GradAutomatonWolframOriginal* const that,
    const VecShort2D* const pos) {

#if BUILDMODE == 0
    if (that == NULL) {

```

```

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

    if (pos == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'pos' is null");
        PBErrCatch(GradAutomatonErr);
    }

#endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            pos);

    // Return the GrACellShort associated to the cell
    return (GrACellShort*)GradCellData(cell);
}

// Return the GrACellShort at index 'iCell' for the
// GradAutomatonWolframOriginal 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellShort* _GradAutomatonWolframOriginalCellIndex(
    GradAutomatonWolframOriginal* const that,
    const long iCell) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);
    }

#endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            iCell);

    // Return the GrACellShort associated to the cell
    return (GrACellShort*)GradCellData(cell);
}

```

```

}

// ----- GradAutomatonNeuraNet

// ===== Functions implementation =====

// Return the Grad of the GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
Grad* _GradAutomatonNeuraNetGrad(
    const GradAutomatonNeuraNet* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PErrCatch(GradAutomatonErr);

    }

#endif

    // Return the Grad
    return ((GradAutomaton*)that)->grad;

}

// Return the type of Grad of the GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
GradType GradAutomatonNeuraNetGetGradType(
    GradAutomatonNeuraNet* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PErrCatch(GradAutomatonErr);

    }

#endif

    // Return the type of the Grad
    return GradGetType(((GradAutomaton*)that)->grad);

}

// Return the GraFun of the GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
GraFunNeuraNet* _GradAutomatonNeuraNetFun(

```

```

    const GradAutomatonNeuraNet* const that) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Return the GrAFun
    return (GrAFunNeuraNet*)((((GradAutomaton*)that)->fun);

}

// Return the GrACellFloat at position 'pos' for the
// GradAutomatonNeuraNet 'that'
#ifdef BUILDMODE != 0
static inline
#endif
GrACellFloat* _GradAutomatonNeuraNetCellPos(
    GradAutomatonNeuraNet* const that,
    const VecShort2D* const pos) {

#ifdef BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

    if (pos == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'pos' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            pos);

    // Return the GrACellFloat associated to the cell
    return (GrACellFloat*)GradCellData(cell);

}

```

```

// Return the GrACellFloat at index 'iCell' for the
// GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
GrACellFloat* _GradAutomatonNeuraNetCellIndex(
    GradAutomatonNeuraNet* const that,
    const long iCell) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Get the GradCell at the requested position
    GradCell* cell =
        GradCellAt(
            GradAutomatonGrad(that),
            iCell);

    // Return the GrACellFloat associated to the cell
    return (GrACellFloat*)GradCellData(cell);

}

// Return the number of hidden layers of the GradAutomatonNeuraNet 'that'
#if BUILDMODE != 0
static inline
#endif
long GradAutomatonNeuraNetGetNbHiddenLayers(
    const GradAutomatonNeuraNet* const that) {

#if BUILDMODE == 0
    if (that == NULL) {

        GradAutomatonErr->_type = PBErrTypeNullPointer;
        sprintf(
            GradAutomatonErr->_msg,
            "'that' is null");
        PBErrCatch(GradAutomatonErr);

    }

#endif

    // Return the number of hidden layers
    return that->nbHiddenLayers;

}

```

4 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=0

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=gradautomaton
$(repo)_EXENAME: \
$(repo)_EXENAME.o \
$(repo)_EXE_DEP \
$(repo)_DEP
$(COMPILER) 'echo "$(repo)_EXE_DEP" "$(repo)_EXENAME.o" | tr ' ' '\n' | sort -u' $(LINK_ARG) $(repo)_LINK_ARG

$(repo)_EXENAME.o: \
$(repo)_DIR/$(repo)_EXENAME.c \
$(repo)_INC_H_EXE \
$(repo)_EXE_DEP
$(COMPILER) $(BUILD_ARG) $(repo)_BUILD_ARG 'echo "$(repo)_INC_DIR" | tr ' ' '\n' | sort -u' -c $(repo)_DIR/
```

5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "gradautomaton.h"

#define RANDOMSEED 0

void UnitTestGrACellCreateFree(void) {

    int dim = 2;
    GradCell gradCell;
    GrACellShort* cellShort =
        GrACellCreateShort(
            dim,
            &gradCell);
    if (
        cellShort == NULL ||
        VecGetDim(cellShort->status[0]) != dim ||
        VecGetDim(cellShort->status[1]) != dim ||
```



```

    cellShort->gradAutomatonCell.curStatus != 0 ||
    cellShort->gradAutomatonCell.gradCell != &gradCell) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellCreateShort failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFree(&cellShort);
if (cellShort != NULL) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortFree failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFloat* cellFloat =
    GrACellCreateFloat(
        dim,
        &gradCell);
if (
    cellFloat == NULL ||
    VecGetDim(cellFloat->status[0]) != dim ||
    VecGetDim(cellFloat->status[1]) != dim ||
    cellFloat->gradAutomatonCell.curStatus != 0 ||
    cellFloat->gradAutomatonCell.gradCell != &gradCell) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellCreateFloat failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFree(&cellFloat);
if (cellFloat != NULL) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatFree failed");
    PBErrCatch(GradAutomatonErr);

}

printf("UnitTestGrACellCreateFree OK\n");

}

void UnitTestGrACellSwitchStatus(void) {

    int dim = 2;
    GrACellShort* cellShort =
        GrACellCreateShort(
            dim,

```

```

        NULL);
GrACellSwitchStatus(cellShort);
if (cellShort->gradAutomatonCell.curStatus != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortSwitchStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellSwitchStatus(cellShort);
if (cellShort->gradAutomatonCell.curStatus != 0) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortSwitchStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFree(&cellShort);

GrACellFloat* cellFloat =
    GrACellCreateFloat(
        dim,
        NULL);
GrACellSwitchStatus(cellFloat);
if (cellFloat->gradAutomatonCell.curStatus != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatSwitchStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellSwitchStatus(cellFloat);
if (cellFloat->gradAutomatonCell.curStatus != 0) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatSwitchStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFree(&cellFloat);

printf("UnitTestGrACellSwitchStatus OK\n");

}

void UnitTestGrACellCurPrevStatus(void) {

    int dim = 2;
    GrACellShort* cellShort =
        GrACellCreateShort(

```

```

        dim,
        NULL);
if (cellShort->status[0] != GrACellCurStatus(cellShort)) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

if (cellShort->status[1] != GrACellPrevStatus(cellShort)) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFree(&cellShort);

GrACellFloat* cellFloat =
    GrACellCreateFloat(
        dim,
        NULL);
if (cellFloat->status[0] != GrACellCurStatus(cellFloat)) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

if (cellFloat->status[1] != GrACellPrevStatus(cellFloat)) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFree(&cellFloat);

printf("UnitTestGrACellCurPrevStatus OK\n");

}

void UnitTestGrACellGetSet(void) {

    int dim = 1;
    GradCell gradCell;
    GrACellShort* cellShort =
        GrACellCreateShort(
            dim,
            &gradCell);

```

```

GrACellSetCurStatus(
    cellShort,
    0,
    1);
short curStatusS =
    VecGet(
        GrACellCurStatus(cellShort),
        0);
if (curStatusS != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortSetCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

curStatusS =
    GrACellGetCurStatus(
        cellShort,
        0);
if (curStatusS != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortGetCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellSetPrevStatus(
    cellShort,
    0,
    1);
short prevStatusS =
    VecGet(
        GrACellPrevStatus(cellShort),
        0);
if (prevStatusS != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortSetPrevStatus failed");
    PBErrCatch(GradAutomatonErr);

}

prevStatusS =
    GrACellGetPrevStatus(
        cellShort,
        0);
if (prevStatusS != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortGetPrevStatus failed");
    PBErrCatch(GradAutomatonErr);

}

```

```

}

if (GrACellGradCell(cellShort) != &gradCell) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellShortGradCell failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellFree(&cellShort);

GrACellFloat* cellFloat =
    GrACellCreateFloat(
        dim,
        &gradCell);
GrACellSetCurStatus(
    cellFloat,
    0,
    1);
float curStatusF =
    VecGet(
        GrACellCurStatus(cellFloat),
        0);
if (curStatusF != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatSetCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

curStatusF =
    GrACellGetCurStatus(
        cellFloat,
        0);
if (curStatusF != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatGetCurStatus failed");
    PBErrCatch(GradAutomatonErr);

}

GrACellSetPrevStatus(
    cellFloat,
    0,
    1);
float prevStatusF =
    VecGet(
        GrACellPrevStatus(cellFloat),
        0);
if (prevStatusF != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(

```

```

        GradAutomatonErr->_msg,
        "GrACellFloatSetPrevStatus failed");
    PBErrCatch(GradAutomatonErr);
}

prevStatusF =
    GrACellGetPrevStatus(
        cellFloat,
        0);
if (prevStatusF != 1) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatGetPrevStatus failed");
    PBErrCatch(GradAutomatonErr);
}

if (GrACellGradCell(cellFloat) != &gradCell) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrACellFloatGradCell failed");
    PBErrCatch(GradAutomatonErr);
}

GrACellFree(&cellFloat);

printf("UnitTestGrACellCurGetSet OK\n");
}

void UnitTestGrACell(void) {

    UnitTestGrACellCreateFree();
    UnitTestGrACellSwitchStatus();
    UnitTestGrACellCurPrevStatus();
    UnitTestGrACellGetSet();
    printf("UnitTestGrACell OK\n");
}

void UnitTestGrAFunDummyCreateFree(void) {

    GrAFunDummy* fun = GrAFunCreateDummy();
    if (
        fun == NULL ||
        fun->grAFun.type != GrAFunTypeDummy) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GrAFunCreateDummy failed");
        PBErrCatch(GradAutomatonErr);
    }

    GrAFunFree(&fun);
}

```

```

if (fun != NULL) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrAFunFree failed");
    PBErrCatch(GradAutomatonErr);

}

printf("UnitTestGrAFunDummyCreateFree OK\n");
}

void UnitTestGrAFunDummyGetType(void) {

    GrAFunDummy* fun = GrAFunCreateDummy();
    if (GrAFunGetType(fun) != GrAFunTypeDummy) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GrAFunDummyGetType failed");
        PBErrCatch(GradAutomatonErr);

    }

    GrAFunFree(&fun);

    printf("UnitTestGrAFunDummyGetType OK\n");
}

void UnitTestGrAFunDummy(void) {

    UnitTestGrAFunDummyCreateFree();
    UnitTestGrAFunDummyGetType();
    printf("UnitTestGrAFunDummy OK\n");
}

void UnitTestGrAFunWolframOriginalCreateFree(void) {

    unsigned char rule = 42;
    GrAFunWolframOriginal* fun = GrAFunCreateWolframOriginal(rule);
    if (
        fun == NULL ||
        fun->grAFun.type != GrAFunTypeWolframOriginal ||
        fun->rule != rule) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GrAFunCreateWolframOriginal failed");
        PBErrCatch(GradAutomatonErr);

    }

    GrAFunFree(&fun);
    if (fun != NULL) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;

```

```

        sprintf(
            GradAutomatonErr->_msg,
            "GrAFunFree failed");
        PBErrCatch(GradAutomatonErr);
    }

    printf("UnitTestGrAFunWolframOriginalCreateFree OK\n");
}

void UnitTestGrAFunWolframOriginalGetType(void) {
    unsigned char rule = 42;
    GrAFunWolframOriginal* fun = GrAFunCreateWolframOriginal(rule);
    if (GrAFunGetType(fun) != GrAFunTypeWolframOriginal) {
        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GrAFunWolframOriginalGetType failed");
        PBErrCatch(GradAutomatonErr);
    }

    GrAFunFree(&fun);

    printf("UnitTestGrAFunWolframOriginalGetType OK\n");
}

void UnitTestGrAFunWolframOriginalGetRule(void) {
    unsigned char rule = 42;
    GrAFunWolframOriginal* fun = GrAFunCreateWolframOriginal(rule);
    if (GrAFunWolframOriginalGetRule(fun) != rule) {
        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GrAFunWolframOriginalGetRule failed");
        PBErrCatch(GradAutomatonErr);
    }

    GrAFunFree(&fun);

    printf("UnitTestGrAFunWolframOriginalGetRule OK\n");
}

void UnitTestGrAFunWolframOriginal(void) {
    UnitTestGrAFunWolframOriginalCreateFree();
    UnitTestGrAFunWolframOriginalGetType();
    UnitTestGrAFunWolframOriginalGetRule();
    printf("UnitTestGrAFunWolframOriginal OK\n");
}

void UnitTestGrAFunNeuraNetCreateFree(void) {

```



```

int nbIn = 1;
int nbOut = 1;
VecLong* hiddenLayers = VecLongCreate(1);
VecSet(
    hiddenLayers,
    0,
    1);
GrAFunNeuraNet* fun =
    GrAFunCreateNeuraNet(
        nbIn,
        nbOut,
        hiddenLayers);
if (
    fun == NULL ||
    fun->grAFun.type != GrAFunTypeNeuraNet ||
    NNGetNbInput(fun->nn) != nbIn ||
    NNGetNbOutput(fun->nn) != nbOut) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrAFunCreateNeuraNet failed");
    PBErrCatch(GradAutomatonErr);

}

GrAFunFree(&fun);
if (fun != NULL) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GrAFunFree failed");
    PBErrCatch(GradAutomatonErr);

}

VecFree(&hiddenLayers);

printf("UnitTestGrAFunNeuraNetCreateFree OK\n");

}

void UnitTestGrAFunNeuraNetGetType(void) {

    int nbIn = 1;
    int nbOut = 1;
    VecLong* hiddenLayers = VecLongCreate(1);
    VecSet(
        hiddenLayers,
        0,
        1);
    GrAFunNeuraNet* fun =
        GrAFunCreateNeuraNet(
            nbIn,
            nbOut,
            hiddenLayers);
    if (GrAFunGetType(fun) != GrAFunTypeNeuraNet) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,

```

```

        "GrAFunNeuraNetGetType failed");
        PBErrCatch(GradAutomatonErr);

    }

    GrAFunFree(&fun);
    VecFree(&hiddenLayers);

    printf("UnitTestGrAFunNeuraNetGetType OK\n");
}

void UnitTestGrAFunNeuraNetNN(void) {

    int nbIn = 1;
    int nbOut = 1;
    VecLong* hiddenLayers = VecLongCreate(1);
    VecSet(
        hiddenLayers,
        0,
        1);
    GrAFunNeuraNet* fun =
        GrAFunCreateNeuraNet(
            nbIn,
            nbOut,
            hiddenLayers);
    if (GrAFunNeuraNetNN(fun) != fun->nn) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GrAFunNeuraNetNN failed");
        PBErrCatch(GradAutomatonErr);

    }

    GrAFunFree(&fun);
    VecFree(&hiddenLayers);

    printf("UnitTestGrAFunNeuraNetNN OK\n");
}

void UnitTestGrAFunNeuraNet(void) {

    UnitTestGrAFunNeuraNetCreateFree();
    UnitTestGrAFunNeuraNetGetType();
    UnitTestGrAFunNeuraNetNN();
    printf("UnitTestGrAFunNeuraNet OK\n");
}

void UnitTestGrAFun(void) {

    UnitTestGrAFunDummy();
    UnitTestGrAFunWolframOriginal();
    UnitTestGrAFunNeuraNet();
    printf("UnitTestGrAFun OK\n");
}

void UnitTestGradAutomatonDummyCreateFree(void) {

```

```

GradAutomatonDummy* ga = GradAutomatonCreateDummy();
if (
    ga == NULL ||
    ga->gradAutomaton.grad == NULL ||
    ga->gradAutomaton.fun == NULL ||
    ga->gradAutomaton.type != GradAutomatonTypeDummy) {

    GradAutomatonErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonCreateDummy failed");
    PErrCatch(GradAutomatonErr);

}

GradAutomatonDummyFree(&ga);
if (ga != NULL) {

    GradAutomatonErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonDummyFree failed");
    PErrCatch(GradAutomatonErr);

}

printf("UnitTestGradAutomatonDummyCreateFree OK\n");

}

void UnitTestGradAutomatonDummyGet(void) {

    GradAutomatonDummy* ga = GradAutomatonCreateDummy();
    if (GradAutomatonGrad(ga) != (GradSquare*)(ga->gradAutomaton.grad)) {

        GradAutomatonErr->_type = PErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonDummyGrad failed");
        PErrCatch(GradAutomatonErr);

    }

    if (GradAutomatonFun(ga) != (GrAFunDummy*)(ga->gradAutomaton.fun)) {

        GradAutomatonErr->_type = PErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonDummyFun failed");
        PErrCatch(GradAutomatonErr);

    }

    for (
        long i = 0;
        i < 4;
        ++i) {

        void* cellA =
            GradAutomatonCell(
                ga,

```

```

        i);
void* cellB =
    GradCellAt(
        ga->gradAutomaton.grad,
        i);
if (cellA != GradCellData(cellB)) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonDummyCellIndex failed");
    PBErrCatch(GradAutomatonErr);

}

}

VecShort2D dim = VecShortCreateStatic2D(2);
VecSet(
    &dim,
    0,
    2);
VecSet(
    &dim,
    1,
    2);
VecShort2D pos = VecShortCreateStatic2D(2);
bool flag = true;
do {

    void* cellA =
        GradAutomatonCell(
            ga,
            &pos);
    void* cellB =
        GradCellAt(
            ga->gradAutomaton.grad,
            &pos);
    if (cellA != GradCellData(cellB)) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonDummyCellPos failed");
        PBErrCatch(GradAutomatonErr);

    }

    flag =
        VecStep(
            &pos,
            &dim);

} while(flag);

GradAutomatonDummyFree(&ga);

printf("UnitTestGradAutomatonDummyGet OK\n");

}

void UnitTestGradAutomatonDummyStep(void) {

```

```

    GradAutomatonDummy* ga = GradAutomatonCreateDummy();

    GradAutomatonStep(ga);

    GradAutomatonDummyFree(&ga);

    printf("UnitTestGradAutomatonDummyStep OK\n");
}

void UnitTestGradAutomatonDummy(void) {

    UnitTestGradAutomatonDummyCreateFree();
    UnitTestGradAutomatonDummyGet();
    UnitTestGradAutomatonDummyStep();
    printf("UnitTestGradAutomatonDummy OK\n");
}

void UnitTestGradAutomatonWolframOriginalCreateFree(void) {

    unsigned char rule = 42;
    long size = 20;
    GradAutomatonWolframOriginal* ga =
        GradAutomatonCreateWolframOriginal(
            rule,
            size);
    if (
        ga == NULL ||
        ga->gradAutomaton.grad == NULL ||
        ga->gradAutomaton.fun == NULL ||
        ga->gradAutomaton.type != GradAutomatonTypeWolframOriginal ||
        ((GraFunWolframOriginal*)(ga->gradAutomaton.fun))->rule != rule ||
        ga->gradAutomaton.grad->_dim._val[0] != size ||
        ga->gradAutomaton.grad->_dim._val[1] != 1) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonCreateWolframOriginal failed");
        PBErrCatch(GradAutomatonErr);
    }

    GradAutomatonWolframOriginalFree(&ga);
    if (ga != NULL) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonWolframOriginalFree failed");
        PBErrCatch(GradAutomatonErr);
    }

    printf("UnitTestGradAutomatonWolframOriginalCreateFree OK\n");
}

void UnitTestGradAutomatonWolframOriginalGet(void) {

```

```

unsigned char rule = 42;
long size = 20;
GradAutomatonWolframOriginal* ga =
    GradAutomatonCreateWolframOriginal(
        rule,
        size);
if (GradAutomatonGrad(ga) != (GradSquare*)(ga->gradAutomaton.grad)) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonWolframOriginalGrad failed");
    PBErrCatch(GradAutomatonErr);

}

if ((void*)GradAutomatonFun(ga) != ga->gradAutomaton.fun) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonWolframOriginalFun failed");
    PBErrCatch(GradAutomatonErr);

}

for (
    long i = 0;
    i < 4;
    ++i) {

    void* cellA =
        GradAutomatonCell(
            ga,
            i);
    void* cellB =
        GradCellAt(
            ga->gradAutomaton.grad,
            i);
    if (cellA != GradCellData(cellB)) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonWolframOriginalCellIndex failed");
        PBErrCatch(GradAutomatonErr);

    }

}

VecShort2D dim = VecShortCreateStatic2D(2);
VecSet(
    &dim,
    0,
    size);
VecSet(
    &dim,
    1,
    1);
VecShort2D pos = VecShortCreateStatic2D(2);
bool flag = true;

```

```

do {

    void* cellA =
        GradAutomatonCell(
            ga,
            &pos);
    void* cellB =
        GradCellAt(
            ga->gradAutomaton.grad,
            &pos);
    if (cellA != GradCellData(cellB)) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonWolframOriginalCellPos failed");
        PBErrCatch(GradAutomatonErr);

    }

    flag =
        VecStep(
            &pos,
            &dim);

} while(flag);

GradAutomatonWolframOriginalFree(&ga);

printf("UnitTestGradAutomatonWolframOriginalGet OK\n");

}

void UnitTestGradAutomatonWolframOriginalStepPrintln(void) {

    unsigned char rule = 30;
    long size = 100;
    GradAutomatonWolframOriginal* ga =
        GradAutomatonCreateWolframOriginal(
            rule,
            size);

    GradAutomatonPrintln(
        ga,
        stdout);

    for (
        long iStep = 0;
        iStep < size;
        ++iStep) {

        GradAutomatonStep(ga);

        GradAutomatonPrintln(
            ga,
            stdout);

    }

    GradAutomatonWolframOriginalFree(&ga);

    printf("UnitTestGradAutomatonWolframOriginalStepPrintln OK\n");
}

```

```

}

void UnitTestGradAutomatonWolframOriginalLoadSave(void) {

    unsigned char rule = 30;
    long size = 100;
    GradAutomatonWolframOriginal* ga =
        GradAutomatonCreateWolframOriginal(
            rule,
            size);

    FILE* fp =
        fopen(
            "./unitTestGradAutomatonWolframOriginalSave.json",
            "w");
    bool compact = false;
    bool ret =
        GradAutomatonSave(
            ga,
            fp,
            compact);
    if (ret == false) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonWolframOriginalSave failed");
        PBErrCatch(GradAutomatonErr);

    }

    GradAutomatonWolframOriginalFree(&ga);
    fclose(fp);
    fp =
        fopen(
            "./unitTestGradAutomatonWolframOriginalSave.json",
            "r");

    ret =
        GradAutomatonLoad(
            &ga,
            fp);

    if (
        ret == false ||
        GrAFunWolframOriginalGetRule(GradAutomatonFun(ga)) != rule) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonWolframOriginalLoad failed");
        PBErrCatch(GradAutomatonErr);

    }

    const VecShort2D* dim = GradDim(GradAutomatonGrad(ga));
    long sizeLoaded =
        VecGet(
            dim,
            0);
    if (sizeLoaded != size) {

```



```

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonWolframOriginalLoad failed");
    PBErrCatch(GradAutomatonErr);
}

GradAutomatonWolframOriginalFree(&ga);
fclose(fp);

printf("UnitTestGradAutomatonWolframOriginalLoadSave OK\n");
}

void UnitTestGradAutomatonWolframOriginal(void) {

    UnitTestGradAutomatonWolframOriginalCreateFree();
    UnitTestGradAutomatonWolframOriginalGet();
    UnitTestGradAutomatonWolframOriginalStepPrintln();
    UnitTestGradAutomatonWolframOriginalLoadSave();
    printf("UnitTestGradAutomatonWolframOriginal OK\n");
}

void UnitTestGradAutomatonNeuraNetCreateFree(void) {

    long dimStatus = 3;
    VecShort2D dimGrad = VecShortCreateStatic2D();
    VecSet(
        &dimGrad,
        0,
        2);
    VecSet(
        &dimGrad,
        1,
        2);
    bool diagLink = true;
    long nbHiddenLayers = 1;
    GradAutomatonNeuraNet* ga =
        GradAutomatonCreateNeuraNetSquare(
            dimStatus,
            &dimGrad,
            diagLink,
            nbHiddenLayers);
    if (
        ga == NULL ||
        ga->gradAutomaton.grad == NULL ||
        ga->gradAutomaton.fun == NULL ||
        ga->gradAutomaton.type != GradAutomatonTypeNeuraNet ||
        ga->gradAutomaton.grad->_type != GradTypeSquare ||
        ga->gradAutomaton.grad->_dim._val[0] != 2 ||
        ga->gradAutomaton.grad->_dim._val[1] != 2) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonCreateNeuraNetSquare failed");
        PBErrCatch(GradAutomatonErr);
    }
}

```

```

GradAutomatonNeuraNetFree(&ga);
if (ga != NULL) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonNeuraNetFree failed");
    PBErrCatch(GradAutomatonErr);

}

printf("UnitTestGradAutomatonNeuraNetCreateFree OK\n");
}

void UnitTestGradAutomatonNeuraNetGet(void) {

    long dimStatus = 3;
    VecShort2D dimGrad = VecShortCreateStatic2D();
    VecSet(
        &dimGrad,
        0,
        2);
    VecSet(
        &dimGrad,
        1,
        2);
    bool diagLink = true;
    long nbHiddenLayers = 1;
    GradAutomatonNeuraNet* ga =
        GradAutomatonCreateNeuraNetSquare(
            dimStatus,
            &dimGrad,
            diagLink,
            nbHiddenLayers);
    if (GradAutomatonGrad(ga) != ga->gradAutomaton.grad) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetGrad failed");
        PBErrCatch(GradAutomatonErr);

    }

    if (GradAutomatonNeuraNetGetGradType(ga) != GradTypeSquare) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetGradType failed");
        PBErrCatch(GradAutomatonErr);

    }

    if ((void*)GradAutomatonFun(ga) != ga->gradAutomaton.fun) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetFun failed");
    }
}

```

```

        PBErrCatch(GradAutomatonErr);
    }

    for (
        long i = 0;
        i < 4;
        ++i) {

        void* cellA =
            GradAutomatonCell(
                ga,
                i);
        void* cellB =
            GradCellAt(
                ga->gradAutomaton.grad,
                i);
        if (cellA != GradCellData(cellB)) {

            GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
            sprintf(
                GradAutomatonErr->_msg,
                "GradAutomatonNeuraNetCellIndex failed");
            PBErrCatch(GradAutomatonErr);

        }

    }

    VecShort2D pos = VecShortCreateStatic2D(2);
    bool flag = true;
    do {

        void* cellA =
            GradAutomatonCell(
                ga,
                &pos);
        void* cellB =
            GradCellAt(
                ga->gradAutomaton.grad,
                &pos);
        if (cellA != GradCellData(cellB)) {

            GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
            sprintf(
                GradAutomatonErr->_msg,
                "GradAutomatonNeuraNetCellPos failed");
            PBErrCatch(GradAutomatonErr);

        }

        flag =
            VecStep(
                &pos,
                &dimGrad);

    } while(flag);

    GradAutomatonNeuraNetFree(&ga);

    printf("UnitTestGradAutomatonNeuraNetGet OK\n");

```

```

}

void UnitTestGradAutomatonNeuraNetStep(void) {

    long dimStatus = 3;
    VecShort2D dimGrad = VecShortCreateStatic2D();
    VecSet(
        &dimGrad,
        0,
        2);
    VecSet(
        &dimGrad,
        1,
        2);
    bool diagLink = true;
    long nbHiddenLayers = 1;
    GradAutomatonNeuraNet* ga =
        GradAutomatonCreateNeuraNetSquare(
            dimStatus,
            &dimGrad,
            diagLink,
            nbHiddenLayers);

    for (
        long iStep = 0;
        iStep < 2;
        ++iStep) {

        GradAutomatonStep(ga);

    }

    GradAutomatonNeuraNetFree(&ga);

    printf("UnitTestGradAutomatonNeuraNetStep OK\n");

}

void UnitTestGradAutomatonNeuraNetSquareLoadSave(void) {

    long dimStatus = 3;
    VecShort2D dimGrad = VecShortCreateStatic2D();
    VecSet(
        &dimGrad,
        0,
        2);
    VecSet(
        &dimGrad,
        1,
        2);
    bool diagLink = false;
    long nbHiddenLayers = 1;
    GradAutomatonNeuraNet* ga =
        GradAutomatonCreateNeuraNetSquare(
            dimStatus,
            &dimGrad,
            diagLink,
            nbHiddenLayers);

    FILE* fp =
        fopen(
            "./unitTestGradAutomatonNeuraNetSquareSave.json",

```

```

        "w");
    bool compact = false;
    bool ret =
        GradAutomatonSave(
            ga,
            fp,
            compact);
    if (ret == false) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetSave failed");
        PBErrCatch(GradAutomatonErr);

    }

    GradAutomatonNeuraNetFree(&ga);
    fclose(fp);
    fp =
        fopen(
            "./unitTestGradAutomatonNeuraNetSquareSave.json",
            "r");

    ret =
        GradAutomatonLoad(
            &ga,
            fp);

    if (
        ret == false) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetLoad failed");
        PBErrCatch(GradAutomatonErr);

    }

    const VecShort2D* dim = GradDim(GradAutomatonGrad(ga));
    bool sameSize =
        VecIsEqual(
            &dimGrad,
            dim);
    if (sameSize == false) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetLoad failed");
        PBErrCatch(GradAutomatonErr);

    }

    GradAutomatonNeuraNetFree(&ga);
    fclose(fp);

    printf("UnitTestGradAutomatonNeuraNetSquareLoadSave OK\n");
}

```

```

void UnitTestGradAutomatonNeuraNetHexaLoadSave(void) {

    long dimStatus = 3;
    VecShort2D dimGrad = VecShortCreateStatic2D();
    VecSet(
        &dimGrad,
        0,
        2);
    VecSet(
        &dimGrad,
        1,
        2);
    long nbHiddenLayers = 1;
    GradHexaType hexaType = GradHexaTypeOddQ;
    GradAutomatonNeuraNet* ga =
        GradAutomatonCreateNeuraNetHexa(
            dimStatus,
            &dimGrad,
            hexaType,
            nbHiddenLayers);

    FILE* fp =
        fopen(
            "./unitTestGradAutomatonNeuraNetHexaSave.json",
            "w");
    bool compact = false;
    bool ret =
        GradAutomatonSave(
            ga,
            fp,
            compact);
    if (ret == false) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetSave failed");
        PBErrCatch(GradAutomatonErr);

    }

    GradAutomatonNeuraNetFree(&ga);
    fclose(fp);
    fp =
        fopen(
            "./unitTestGradAutomatonNeuraNetHexaSave.json",
            "r");

    ret =
        GradAutomatonLoad(
            &ga,
            fp);

    if (
        ret == false) {

        GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            GradAutomatonErr->_msg,
            "GradAutomatonNeuraNetLoad failed");
        PBErrCatch(GradAutomatonErr);

    }

}

```

```

}

const VecShort2D* dim = GradDim(GradAutomatonGrad(ga));
bool sameSize =
    VecIsEqual(
        &dimGrad,
        dim);
if (sameSize == false) {

    GradAutomatonErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        GradAutomatonErr->_msg,
        "GradAutomatonNeuraNetLoad failed");
    PBErrCatch(GradAutomatonErr);

}

GradAutomatonNeuraNetFree(&ga);
fclose(fp);

printf("UnitTestGradAutomatonNeuraNetHexaLoadSave OK\n");

}

void UnitTestGradAutomatonNeuraNet(void) {

    UnitTestGradAutomatonNeuraNetCreateFree();
    UnitTestGradAutomatonNeuraNetGet();
    UnitTestGradAutomatonNeuraNetStep();
    UnitTestGradAutomatonNeuraNetSquareLoadSave();
    UnitTestGradAutomatonNeuraNetHexaLoadSave();
    printf("UnitTestGradAutomatonNeuraNet OK\n");

}

void UnitTestGradAutomaton(void) {

    UnitTestGradAutomatonDummy();
    UnitTestGradAutomatonWolframOriginal();
    UnitTestGradAutomatonNeuraNet();
    printf("UnitTestGradAutomaton OK\n");

}

void UnitTestAll(void) {

    UnitTestGrACell();
    UnitTestGrAFun();
    UnitTestGradAutomaton();
    printf("UnitTestAll OK\n");

}

int main(void) {

    UnitTestAll();

    // Return success code
    return 0;

}

```

6 Unit tests output

unitTestRef.txt:

```

UnitTestGrACellCreateFree OK
UnitTestGrACellSwitchStatus OK
UnitTestGrACellCurPrevStatus OK
UnitTestGrACellCurGetSet OK
UnitTestGrACell OK
UnitTestGrAFunDummyCreateFree OK
UnitTestGrAFunDummyGetType OK
UnitTestGrAFunDummy OK
UnitTestGrAFunWolframOriginalCreateFree OK
UnitTestGrAFunWolframOriginalGetType OK
UnitTestGrAFunWolframOriginalGetRule OK
UnitTestGrAFunWolframOriginal OK
UnitTestGrAFunNeuraNetCreateFree OK
UnitTestGrAFunNeuraNetGetType OK
UnitTestGrAFunNeuraNetNN OK
UnitTestGrAFunNeuraNet OK
UnitTestGrAFun OK
UnitTestGradAutomatonDummyCreateFree OK
UnitTestGradAutomatonDummyGet OK
UnitTestGradAutomatonDummyStep OK
UnitTestGradAutomatonDummy OK
UnitTestGradAutomatonWolframOriginalCreateFree OK
UnitTestGradAutomatonWolframOriginalGet OK

```

[illegible]

89


```
"typeGradHexa": "2",  
"nn": {  
    "_nbInputVal": "18",  
    "_nbOutputVal": "3",  
    "_nbMaxHidVal": "18",  
    "_nbMaxBases": "378",  
    "_nbMaxLinks": "378",  
    "_bases": {  
        "_dim": "1134",  
        "_val": ["0.000000", "0.000000", "0.000000", "0.000000", "0.000000", "0.000000", "0.000000", "0.000000", "0.000000", "0.  
    },  
    "_links": {  
        "_dim": "1134",  
        "_val": ["0", "0", "18", "1", "0", "19", "2", "0", "20", "3", "0", "21", "4", "0", "22", "5", "0", "23", "6", "0", "24", "7", "0", "25"  
    }  
}  
}
```