

# Grad

P. Baillehache

February 8, 2020

## Contents

<b>1</b>	<b>Definitions</b>	<b>2</b>
<b>2</b>	<b>Interface</b>	<b>2</b>
<b>3</b>	<b>Code</b>	<b>13</b>
3.1	grad.c . . . . .	13
3.2	grad-inline.c . . . . .	32
<b>4</b>	<b>Makefile</b>	<b>41</b>
<b>5</b>	<b>Unit tests</b>	<b>42</b>
<b>6</b>	<b>Unit tests output</b>	<b>55</b>

## Introduction

Grad is a C library providing structures and functions to manipulate square and hexagonal 2D grids.

The Grad represents internally the grid as a graph where the links between a cell and its neighbour can be deleted or recreated to define connection between cells of the grid. Links are automatically generated according to the type of Grad. Cells can also be marked as blocked to temporarily be ignored by the two main functions of the Grad: flooding and search for the shortest path between two cells.

The flooding can be done from several sources simultaneously, and be constrained by the number of steps and/or the distance from the source.

The flooding occurs in order consistent with the distance between cells. The distance is set up by default and can be modified by the user.

The search path is done using the A\* algorithm with a look up table for the evaluation of distance. The Grad provides an automatically generated look up table, but the user can also use its own. The automatically generated look up table ensures the fastest and optimal search if there was no modification (link edited, cell blocked) to the Grad between the table generation and the search.

Hexagonal Grad supports the 4 types of possible alignements (line/column, even/odd).

Content of the cells of the Grad can be extended via a void pointer toward a user defined structure.

It uses the PBErr, PBMath and GSet library.

## 1 Definitions

## 2 Interface

```
// ===== GRAD.H =====

#ifndef GRAD_H
#define GRAD_H

// https://www.redblobgames.com/grids/hexagons/

// ===== 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"

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

#define GRAD_NBMAXLINK 8
#define GRAD_DEFAULTLINKVAL 1.0

#define GradPosToIndex(Grad_, Pos) \
    (VecGet(Pos, 1) * VecGet(GradDim(Grad_), 0) + VecGet(Pos, 0))
```

```

// ----- GradCell

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

typedef struct GradCell {
    // Index in the grad
    int _id;
    // Position in the grad
    VecShort2D _pos;
    // Index in the grad of the neighbour cells
    // -1 means no link
    int _links[GRAD_NBMAXLINK];
    // Nb of links
    int _nbLink;
    // Links' value
    float _linksVal[GRAD_NBMAXLINK];
    // Flood value (_flood == -1: not flooded, _flood >= 0: flooded by
    // the _flood-th source
    int _flood;
    // Flag to mark blocked cell in lookuptable, flood and getPath
    // if this flag equals true the cell is not explored
    bool _flagBlocked;
    // User data
    void* _data;
} GradCell;

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

// Create a new GradCell with index 'id', position 'pos' and
// 'nbLink' links
GradCell* GradCellCreate(const int id, const int nbLink,
    const VecShort2D* const pos);

// Free the memory used by the GradCell 'that'
void GradCellFree(GradCell** that);

// Create a new static GradCell with index 'id', position 'pos'
// and 'nbLink' links
GradCell GradCellCreateStatic(const int id, const int nbLink,
    const VecShort2D* const pos);

// Get the user data of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
void* GradCellData(const GradCell* const that);

// Set the user data of the GradCell 'that' to 'data'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetData(GradCell* const that, void* const data);

// Get the position of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* GradCellPos(const GradCell* const that);

// Get the index of the GradCell 'that'
#if BUILDMODE != 0
static inline

```

```

#endif
int GradCellGetId(const GradCell* const that);

// Get the index of 'iLink'-th link of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
int GradCellGetLink(const GradCell* const that, const int iLink);

// Set the index of 'iLink'-th link of the GradCell 'that' to 'iCell'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetLink(GradCell* const that, const int iLink,
    const int iCell);

// Get the number of links of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
int GradCellGetNbLink(const GradCell* const that);

// Get the value of 'iLink'-th link of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
float GradCellLinkVal(const GradCell* const that, const int iLink);

// Set the value of 'iLink'-th link of the GradCell 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetLinkVal(GradCell* const that, const int iLink,
    const float val);

// Get the flood value of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
int GradCellGetFlood(const GradCell* const that);

// Set the flood value of the GradCell 'that' to 'iSource'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetFlood(GradCell* const that, const int iSource);

// Get the flag blocked of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
bool GradCellIsBlocked(const GradCell* const that);

// Set the flag blocked of the GradCell 'that' to 'flag'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetBlocked(GradCell* const that, const bool flag);

// ----- Grad
// ===== Data structure =====

```

```

typedef enum GradType {
    GradTypeSquare, GradTypeHexa
} GradType;

typedef enum GradHexaType {
    GradHexaTypeOddR, GradHexaTypeEvenR, GradHexaTypeOddQ, GradHexaTypeEvenQ
} GradHexaType;

typedef struct Grad {
    // Type
    GradType _type;
    // Cells
    GradCell* _cells;
    // Dimensions
    VecShort2D _dim;
} Grad;

// Direction for x toward east and y toward south
typedef enum GradSquareDir {
    GradSquareDirN, GradSquareDirE, GradSquareDirS, GradSquareDirW,
    GradSquareDirNW, GradSquareDirNE, GradSquareDirSE, GradSquareDirSW
} GradSquareDir;

typedef struct GradSquare {
    // Grad
    Grad _grad;
    // Flag to memorize if diagonal links are allowed
    bool _diagLink;
} GradSquare;

// Direction for x toward east and y toward south
typedef enum GradHexaOddDir {
    GradHexaOddDirNW, GradHexaOddDirNE, GradHexaOddDirE,
    GradHexaOddDirSE, GradHexaOddDirSW, GradHexaOddDirW
} GradHexaOddDir;

typedef enum GradHexaEvenDir {
    GradHexaEvenDirN, GradHexaEvenDirNE, GradHexaEvenDirSE,
    GradHexaEvenDirS, GradHexaEvenDirSW, GradHexaEvenDirNW
} GradHexaEvenDir;

typedef struct GradHexa {
    // Grad
    Grad _grad;
    // Type
    GradHexaType _type;
} GradHexa;

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

// Create a new GradSquare of dimensions 'dim' and diagonal links
// allowed if 'diagLink' equals true
GradSquare* GradSquareCreate(const VecShort2D* const dim,
    const bool diagLink);

// Free the memory used by the GradSquare 'that'
void GradSquareFree(GradSquare** that);

// Create a new GradHexa of dimensions 'dim' and orientation odd-r
GradHexa* GradHexaCreateOddR(const VecShort2D* const dim);

```

```

// Create a new GradHexa of dimensions 'dim' and orientation even-r
GradHexa* GradHexaCreateEvenR(const VecShort2D* const dim);

// Create a new GradHexa of dimensions 'dim' and orientation odd-q
GradHexa* GradHexaCreateOddQ(const VecShort2D* const dim);

// Create a new GradHexa of dimensions 'dim' and orientation even-q
GradHexa* GradHexaCreateEvenQ(const VecShort2D* const dim);

// Free the memory used by the GradHexa 'that'
void GradHexaFree(GradHexa** that);

// Get the GradCell at index 'iCell' in the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
GradCell* _GradCellAtIndex(const Grad* const that, const int iCell);

// Get the GradCell at position 'pos' int the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
GradCell* _GradCellAtPos(const Grad* const that,
    const VecShort2D* const pos);

// Get the GradType of the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
GradType _GradGetType(const Grad* const that);

// Get the GradHexaType of the GradHexa 'that'
#if BUILDMODE != 0
static inline
#endif
GradHexaType GradHexaGetType(const GradHexa* const that);

// Return true if the GradSquare 'that' has diagonal link
// Return false else
#if BUILDMODE != 0
static inline
#endif
bool GradSquareHasDiagonalLink(const GradSquare* const that);

// Get the number of cells (area) of the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
int _GradGetArea(const Grad* const that);

// Get the look up table for distance between each pair of cell of the
// Grad 'that'
// Return a MatFloat where first index is the 'from' cell's index
// and second index is the 'to' cell index
// Distances in the matrix are equal to the sum of the value of links
// between cells
// Negative distance means there is no path for the pair of cell
MatFloat* _GradGetLookupTableMinDist(const Grad* const that);

// Get the path from cell at index 'from' to cell at index 'to' in
// the Grad 'that' using the A* algorithm and the look up table 'lookup'
// for distance estimation between cells

```

```

// Return a VecShort of position (index) ordered from 'from' to 'to'
// Return NULL if there is no path
VecShort* _GradGetPath(const Grad* const that, const int from,
    const int to, const MatFloat* const lookup);

// Flood the Grad 'that' from positions (index) 'sources' up to a
// maximum distance in link's value from the source 'distMax' or
// maximum distance in nb of cell from the source 'stepMax'
// If 'distMax' and/or 'stepMax' are/is negative(s) their is no limit
// on the maximum distance/maximum number of steps
// The flood occurs in order consistent with the links' value
// interpreted as distance
// The result is stored in the _flood property of the GradCell
// _flood == -1: not flooded, _flood >= 0: flooded by the _flood-th
// source
// Conflicting cells (several sources arriving at the same step to the
// cell) are left undecided (_flood==-1)
void _GradFlood(Grad* const that, const VecShort* const sources,
    const float distMax, const int stepMax);

// Get the number of flooded cells from 'iSource'-th source in the Grad
// 'that'
int _GradGetFloodArea(const Grad* const that, const int iSource);

// Get the dimensions of the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* _GradDim(const Grad* const that);

// Check if the position 'pos' is inside the GradSquare 'that'
#if BUILDMODE != 0
static inline
#endif
bool _GradIsPosInside(const Grad* const that,
    const VecShort2D* const pos);

// Clone the GradSquare 'that'
// The user data are not cloned but shared between the original and
// its clone
GradSquare* GradSquareClone(const GradSquare* const that);

// Clone the GradHexa 'that'
// The user data are not cloned but shared between the original and
// its clone
GradHexa* GradHexaClone(const GradHexa* const that);

// Return true if the Grad 'that' is same as the Grad 'tho'
// Return false else
bool _GradIsSameAs(const Grad* const that, const Grad* const tho);

// Set the flag blocked of all cells in the Grad 'that' to false
#if BUILDMODE != 0
static inline
#endif
void _GradResetFlagBlocked(Grad* const that);

// Remove the link from cell 'fromCell' to cell 'toCell' in the
// Grad 'that'
// If 'symmetric' equals true the symetric link is removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveLinkIndex(Grad* const that, const int fromCell,

```

```

    const int toCell, const bool symmetric);

// Remove the link from cell at position 'fromCell' to cell at
// position 'toCell' in the Grad 'that'
// If 'symmetric' equals true the symmetric link is removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
static inline
#endif
void _GradRemoveLinkPos(Grad* const that,
    const VecShort2D* const fromCell, const VecShort2D* const toCell,
    const bool symmetric);

// Remove the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If 'symmetric' equals true the symmetric link is removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveDirIndex(Grad* const that, const int fromCell,
    const int dir, const bool symmetric);

// Remove the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If 'symmetric' equals true the symmetric link is removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
static inline
#endif
void _GradRemoveDirPos(Grad* const that,
    const VecShort2D* const fromCell, const int dir, const bool symmetric);

// Remove all the links from cell 'fromCell' in the Grad 'that'
// If 'symmetric' equals true the symmetric links are removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveAllLinkIndex(Grad* const that, const int fromCell,
    const bool symmetric);

// Remove all the links from cell at position 'fromCell' in
// the Grad 'that'
// If 'symmetric' equals true the symmetric links are removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
static inline
#endif
void _GradRemoveAllLinkPos(Grad* const that,
    const VecShort2D* const fromCell, const bool symmetric);

// Add the link from cell 'fromCell' to cell 'toCell' in the
// Grad 'that'
// If the cells are not neighbours do nothing
// If 'symmetric' equals true the symmetric link is added too
void _GradAddLinkIndex(Grad* const that, const int fromCell,
    const int toCell, const bool symmetric);

// Add the link from cell at position 'fromCell' to cell at
// position 'toCell' in the Grad 'that'
// If the cells are not neighbours do nothing
// If 'symmetric' equals true the symmetric link is added too
#if BUILDMODE != 0
static inline
#endif
void _GradAddLinkPos(Grad* const that, const VecShort2D* const fromCell,
    const VecShort2D* const toCell, const bool symmetric);

```



```

// Add the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If the cells are not neighbours do nothing
// If 'symmetric' equals true the symetric link is added too
void _GradAddDirIndex(Grad* const that, const int fromCell, const int dir,
    const bool symmetric);

// Add the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If the cells are not neighbours do nothing
// If 'symmetric' equals true the symetric link is added too
#if BUILDMODE != 0
static inline
#endif
void _GradAddDirPos(Grad* const that, const VecShort2D* const fromCell,
    const int dir, const bool symmetric);

// Add all the links from cell 'fromCell' in the Grad 'that'
// If 'symmetric' equals true the symetric links are removed too
void _GradAddAllLinkIndex(Grad* const that, const int fromCell,
    const bool symmetric);

// Add all the links from cell at position 'fromCell' in
// the Grad 'that'
// If 'symmetric' equals true the symetric links are removed too
#if BUILDMODE != 0
static inline
#endif
void _GradAddAllLinkPos(Grad* const that,
    const VecShort2D* const fromCell, const bool symmetric);

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

#define GradCellAt(Grad_, Pos) _Generic(Grad_, \
    Grad*: _Generic(Pos, \
        short: _GradCellAtIndex, \
        int: _GradCellAtIndex, \
        VecShort2D*: _GradCellAtPos, \
        default: PBErrInvalidPolymorphism), \
    GradSquare*: _Generic(Pos, \
        short: _GradCellAtIndex, \
        int: _GradCellAtIndex, \
        VecShort2D*: _GradCellAtPos, \
        default: PBErrInvalidPolymorphism), \
    GradHexa*: _Generic(Pos, \
        short: _GradCellAtIndex, \
        int: _GradCellAtIndex, \
        VecShort2D*: _GradCellAtPos, \
        default: PBErrInvalidPolymorphism), \
    const Grad*: _Generic(Pos, \
        short: _GradCellAtIndex, \
        int: _GradCellAtIndex, \
        VecShort2D*: _GradCellAtPos, \
        default: PBErrInvalidPolymorphism), \
    const GradSquare*: _Generic(Pos, \
        short: _GradCellAtIndex, \
        int: _GradCellAtIndex, \
        VecShort2D*: _GradCellAtPos, \
        default: PBErrInvalidPolymorphism), \
    const GradHexa*: _Generic(Pos, \
        short: _GradCellAtIndex, \

```

```

        int: _GradCellAtIndex, \
        VecShort2D*: _GradCellAtPos, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)((const Grad*)(Grad_), Pos)

#define GradGetArea(Grad_) _Generic(Grad_, \
    Grad*: _GradGetArea, \
    GradSquare*: _GradGetArea, \
    GradHexa*: _GradGetArea, \
    const Grad*: _GradGetArea, \
    const GradSquare*: _GradGetArea, \
    const GradHexa*: _GradGetArea, \
    default: PBErrInvalidPolymorphism)((const Grad*)(Grad_))

#define GradGetLookupTableMinDist(Grad_) _Generic(Grad_, \
    Grad*: _GradGetLookupTableMinDist, \
    GradSquare*: _GradGetLookupTableMinDist, \
    GradHexa*: _GradGetLookupTableMinDist, \
    const Grad*: _GradGetLookupTableMinDist, \
    const GradSquare*: _GradGetLookupTableMinDist, \
    const GradHexa*: _GradGetLookupTableMinDist, \
    default: PBErrInvalidPolymorphism)((const Grad*)(Grad_))

#define GradGetPath(Grad_, From, To, Dist) _Generic(Grad_, \
    Grad*: _GradGetPath, \
    GradSquare*: _GradGetPath, \
    GradHexa*: _GradGetPath, \
    const Grad*: _GradGetPath, \
    const GradSquare*: _GradGetPath, \
    const GradHexa*: _GradGetPath, \
    default: PBErrInvalidPolymorphism)((const Grad*)(Grad_), From, To, Dist)

#define GradFlood(Grad_, Sources, DistMax, StepMax) _Generic(Grad_, \
    Grad*: _GradFlood, \
    GradSquare*: _GradFlood, \
    GradHexa*: _GradFlood, \
    default: PBErrInvalidPolymorphism)((Grad*)(Grad_), Sources, \
        DistMax, StepMax)

#define GradGetFloodArea(Grad_, Source) _Generic(Grad_, \
    Grad*: _GradGetFloodArea, \
    GradSquare*: _GradGetFloodArea, \
    GradHexa*: _GradGetFloodArea, \
    const Grad*: _GradGetFloodArea, \
    const GradSquare*: _GradGetFloodArea, \
    const GradHexa*: _GradGetFloodArea, \
    default: PBErrInvalidPolymorphism)((const Grad*)(Grad_), Source)

#define GradDim(Grad_) _Generic(Grad_, \
    Grad*: _GradDim, \
    GradSquare*: _GradDim, \
    GradHexa*: _GradDim, \
    const Grad*: _GradDim, \
    const GradSquare*: _GradDim, \
    const GradHexa*: _GradDim, \
    default: PBErrInvalidPolymorphism)((const Grad*)(Grad_))

#define GradIsPosInside(Grad_, Pos) _Generic(Grad_, \
    Grad*: _GradIsPosInside, \
    GradSquare*: _GradIsPosInside, \
    GradHexa*: _GradIsPosInside, \
    const Grad*: _GradIsPosInside, \

```

```

const GradSquare*: _GradIsPosInside, \
const GradHexa*: _GradIsPosInside, \
default: PBErriInvalidPolymorphism)((const Grad*)(Grad_), Pos)

#define GradClone(Grad_) _Generic(Grad_, \
    GradSquare*: GradSquareClone, \
    GradHexa*: GradHexaClone, \
    const GradSquare*: GradSquareClone, \
    const GradHexa*: GradHexaClone, \
    default: PBErriInvalidPolymorphism)(Grad_)

#define GradIsSameAs(GradA, GradB) _Generic(GradA, \
    GradSquare*: _Generic(GradB, \
        GradSquare*: _GradIsSameAs, \
        GradHexa*: _GradIsSameAs, \
        const GradSquare*: _GradIsSameAs, \
        const GradHexa*: _GradIsSameAs, \
        default: PBErriInvalidPolymorphism), \
    GradHexa*: _Generic(GradB, \
        GradSquare*: _GradIsSameAs, \
        GradHexa*: _GradIsSameAs, \
        const GradSquare*: _GradIsSameAs, \
        const GradHexa*: _GradIsSameAs, \
        default: PBErriInvalidPolymorphism), \
    const GradSquare*: _Generic(GradB, \
        GradSquare*: _GradIsSameAs, \
        GradHexa*: _GradIsSameAs, \
        const GradSquare*: _GradIsSameAs, \
        const GradHexa*: _GradIsSameAs, \
        default: PBErriInvalidPolymorphism), \
    const GradHexa*: _Generic(GradB, \
        GradSquare*: _GradIsSameAs, \
        GradHexa*: _GradIsSameAs, \
        const GradSquare*: _GradIsSameAs, \
        const GradHexa*: _GradIsSameAs, \
        default: PBErriInvalidPolymorphism), \
    default: PBErriInvalidPolymorphism)((const Grad*)(GradA), \
        (const Grad*)(GradB))

#define GradResetFlagBlocked(Grad_) _Generic(Grad_, \
    Grad*: _GradResetFlagBlocked, \
    GradSquare*: _GradResetFlagBlocked, \
    GradHexa*: _GradResetFlagBlocked, \
    default: PBErriInvalidPolymorphism)((Grad*)(Grad_))

#define GradFree(Grad_) _Generic(Grad_, \
    GradSquare*: GradSquareFree, \
    GradHexa*: GradHexaFree, \
    default: PBErriInvalidPolymorphism)(Grad_)

#define GradGetType(Grad_) _Generic(Grad_, \
    Grad*: _GradGetType, \
    GradSquare*: _GradGetType, \
    GradHexa*: _GradGetType, \
    const Grad*: _GradGetType, \
    const GradSquare*: _GradGetType, \
    const GradHexa*: _GradGetType, \
    default: PBErriInvalidPolymorphism)((const Grad*)(Grad_))

#define GradRemoveLinkTo(Grad_, From, To, Sym) _Generic(Grad_, \
    Grad*: _Generic (From, \
        int: _GradRemoveLinkIndex, \

```

```

    VecShort2D*: _GradRemoveLinkPos, \
    default: PBErInvalidPolymorphism), \
GradSquare*: _Generic (From, \
    int: _GradRemoveLinkIndex, \
    VecShort2D*: _GradRemoveLinkPos, \
    default: PBErInvalidPolymorphism), \
GradHexa*: _Generic (From, \
    int: _GradRemoveLinkIndex, \
    VecShort2D*: _GradRemoveLinkPos, \
    default: PBErInvalidPolymorphism), \
default: PBErInvalidPolymorphism) ((Grad*)(Grad_), From, To, Sym)

#define GradRemoveLinkToward(Grad_, From, Toward, Sym) _Generic(Grad_, \
    Grad*: _Generic (From, \
        int: _GradRemoveDirIndex, \
        VecShort2D*: _GradRemoveDirPos, \
        default: PBErInvalidPolymorphism), \
    GradSquare*: _Generic (From, \
        int: _GradRemoveDirIndex, \
        VecShort2D*: _GradRemoveDirPos, \
        default: PBErInvalidPolymorphism), \
    GradHexa*: _Generic (From, \
        int: _GradRemoveDirIndex, \
        VecShort2D*: _GradRemoveDirPos, \
        default: PBErInvalidPolymorphism), \
    default: PBErInvalidPolymorphism) ((Grad*)(Grad_), From, Toward, Sym)

#define GradRemoveAllLink(Grad_, From, Sym) _Generic(Grad_, \
    Grad*: _Generic (From, \
        int: _GradRemoveAllLinkIndex, \
        VecShort2D*: _GradRemoveAllLinkPos, \
        default: PBErInvalidPolymorphism), \
    GradSquare*: _Generic (From, \
        int: _GradRemoveAllLinkIndex, \
        VecShort2D*: _GradRemoveAllLinkPos, \
        default: PBErInvalidPolymorphism), \
    GradHexa*: _Generic (From, \
        int: _GradRemoveAllLinkIndex, \
        VecShort2D*: _GradRemoveAllLinkPos, \
        default: PBErInvalidPolymorphism), \
    default: PBErInvalidPolymorphism) ((Grad*)(Grad_), From, Sym)

#define GradAddLinkTo(Grad_, From, To, Sym) _Generic(Grad_, \
    Grad*: _Generic (From, \
        int: _GradAddLinkIndex, \
        VecShort2D*: _GradAddLinkPos, \
        default: PBErInvalidPolymorphism), \
    GradSquare*: _Generic (From, \
        int: _GradAddLinkIndex, \
        VecShort2D*: _GradAddLinkPos, \
        default: PBErInvalidPolymorphism), \
    GradHexa*: _Generic (From, \
        int: _GradAddLinkIndex, \
        VecShort2D*: _GradAddLinkPos, \
        default: PBErInvalidPolymorphism), \
    default: PBErInvalidPolymorphism) ((Grad*)(Grad_), From, To, Sym)

#define GradAddLinkToward(Grad_, From, Toward, Sym) _Generic(Grad_, \
    Grad*: _Generic (From, \
        int: _GradAddDirIndex, \
        VecShort2D*: _GradAddDirPos, \
        default: PBErInvalidPolymorphism), \

```

```

GradSquare*: _Generic (From, \
    int: _GradAddDirIndex, \
    VecShort2D*: _GradAddDirPos, \
    default: PBErrInvalidPolymorphism), \
GradHexa*: _Generic (From, \
    int: _GradAddDirIndex, \
    VecShort2D*: _GradAddDirPos, \
    default: PBErrInvalidPolymorphism), \
default: PBErrInvalidPolymorphism) ((Grad*)(Grad_), From, Toward, Sym)

#define GradAddAllLink(Grad_, From, Sym) _Generic(Grad_, \
    Grad*: _Generic (From, \
        int: _GradAddAllLinkIndex, \
        VecShort2D*: _GradAddAllLinkPos, \
        default: PBErrInvalidPolymorphism), \
    GradSquare*: _Generic (From, \
        int: _GradAddAllLinkIndex, \
        VecShort2D*: _GradAddAllLinkPos, \
        default: PBErrInvalidPolymorphism), \
    GradHexa*: _Generic (From, \
        int: _GradAddAllLinkIndex, \
        VecShort2D*: _GradAddAllLinkPos, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism) ((Grad*)(Grad_), From, Sym)

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

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

#endif

```

## 3 Code

### 3.1 grad.c

```

// ===== GRAD.C =====

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

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

// ----- GradCell

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

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

// Create a new GradCell with index 'id', position 'pos' and
// 'nbLink' links
GradCell* GradCellCreate(const int id, const int nbLink,
    const VecShort2D* const pos) {
#if BUILDMODE == 0

```

```

if (id < 0) {
    GradErr->_type = PBErrTypeInvalidArg;
    sprintf(GradErr->_msg, "'id' is invalid (%d>=0)", id);
    PBErrCatch(GradErr);
}
if (nbLink < 0 || nbLink > GRAD_NBMAXLINK) {
    GradErr->_type = PBErrTypeInvalidArg;
    sprintf(GradErr->_msg, "'nbLink' is invalid (0<=%d<=%d)",
        nbLink, GRAD_NBMAXLINK);
    PBErrCatch(GradErr);
}
if (pos == NULL) {
    GradErr->_type = PBErrTypeNullPointer;
    sprintf(GradErr->_msg, "'pos' is null");
    PBErrCatch(GradErr);
}
if (VecGet(pos, 0) < 0 || VecGet(pos, 1) < 0) {
    GradErr->_type = PBErrTypeInvalidArg;
    sprintf(GradErr->_msg, "'pos' is invalid ((0,0)<=(%d,%d))",
        VecGet(pos, 0), VecGet(pos, 1));
    PBErrCatch(GradErr);
}
#endif
// Allocate memory
GradCell* that = PBErrMalloc(GradErr, sizeof(GradCell));
// Set properties
*that = GradCellCreateStatic(id, nbLink, pos);
// Return the new GradCell
return that;
}

// Free the memory used by the GradCell 'that'
void GradCellFree(GradCell** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        // Nothing to do
        return;
    // Free memory
    free(*that);
    *that = NULL;
}

// Create a new static GradCell with index 'id', position 'pos'
// and 'nbLink' links
GradCell GradCellCreateStatic(const int id, const int nbLink,
    const VecShort2D* const pos) {
    #if BUILDMODE == 0
        if (id < 0) {
            GradErr->_type = PBErrTypeInvalidArg;
            sprintf(GradErr->_msg, "'id' is invalid (%d>=0)", id);
            PBErrCatch(GradErr);
        }
        if (nbLink < 0 || nbLink > GRAD_NBMAXLINK) {
            GradErr->_type = PBErrTypeInvalidArg;
            sprintf(GradErr->_msg, "'nbLink' is invalid (0<=%d<=%d)",
                nbLink, GRAD_NBMAXLINK);
            PBErrCatch(GradErr);
        }
        if (pos == NULL) {
            GradErr->_type = PBErrTypeNullPointer;
            sprintf(GradErr->_msg, "'pos' is null");
            PBErrCatch(GradErr);
        }
    #endif
}

```

```

    }
    if (VecGet(pos, 0) < 0 || VecGet(pos, 1) < 0) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'pos' is invalid ((0,0)<=(%d,%d))",
            VecGet(pos, 0), VecGet(pos, 1));
        PBErrCatch(GradErr);
    }
#endif
    // Declare the new GradCell
    GradCell that;
    // Set properties
    that._id = id;
    that._data = NULL;
    that._pos = *pos;
    for (int iLink = GRAD_NBMAXLINK; iLink--;) {
        that._links[iLink] = -1;
        that._linksVal[iLink] = 1.0;
    }
    that._nbLink = nbLink;
    that._flood = -1;
    that._flagBlocked = false;
    // Return the new GradCell
    return that;
}

// ----- Grad

int GradDeltaSquare[16] =
    {0,-1, 1,0, 0,1, -1,0, -1,-1, 1,-1, 1,1, -1,1};
int GradDeltaAEvenQ[12] = {0,-1, 1,-1, 1,0, 0,1, -1,0, -1,-1};
int GradDeltaBEvenQ[12] = {0,-1, 1,0, 1,1, 0,1, -1,1, -1,0};
int GradDeltaAEvenR[12] = {-1,-1, 0,-1, 1,0, 0,1, -1,1, -1,0};
int GradDeltaBEvenR[12] = {0,-1, 1,-1, 1,0, 1,1, 0,1, -1,0};
int GradDeltaAOddQ[12] = {0,-1, 1,0, 1,1, 0,1, -1,1, -1,0};
int GradDeltaBOddQ[12] = {0,-1, 1,-1, 1,0, 0,1, -1,0, -1,-1};
int GradDeltaAOddR[12] = {0,-1, 1,-1, 1,0, 1,1, 0,1, -1,0};
int GradDeltaBOddR[12] = {-1,-1, 0,-1, 1,0, 0,1, -1,1, -1,0};

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

// Create a new static Grad with dimensions 'dim' and type 'type' with
// cells of 'nbLink' sides
Grad GradCreateStatic(const VecShort2D* const dim, const GradType type,
    const int nbLink);

// Free memory used by the properties of the Grad 'that'
void GradFreeStatic(Grad* const that);

// Create a new GradHexa with dimensions 'dim' and type 'type'
GradHexa* GradHexaCreate(const VecShort2D* const dim,
    const GradHexaType type);

// Get the appropriate deltas of positions according to the type of the
// Grad 'that' and the position 'pos'
int* _GradGetDelta(const Grad* const that, const VecShort2D* const pos);

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

#define GradGetDelta(Grad_, Pos) _Generic(Grad_, \
    Grad*: _GradGetDelta, \
    GradSquare*: _GradGetDelta, \
    GradHexa*: _GradGetDelta, \

```

```

        default: PBErrInvalidPolymorphism)((Grad*)(Grad_), Pos)

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

// Create a new static Grad with dimensions 'dim' and type 'type' with
// cells of 'nbLink' sides
Grad GradCreateStatic(const VecShort2D* const dim, const GradType type,
    const int nbLink) {
    #if BUILDMODE == 0
        if (dim == NULL) {
            GradErr->_type = PBErrTypeNullPointer;
            sprintf(GradErr->_msg, "'dim' is null");
            PBErrCatch(GradErr);
        }
        if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
            GradErr->_type = PBErrTypeInvalidArg;
            sprintf(GradErr->_msg, "'dim' is invalid ((0,0)<(%d,%d))",
                VecGet(dim, 0), VecGet(dim, 1));
            PBErrCatch(GradErr);
        }
        if (nbLink < 0) {
            GradErr->_type = PBErrTypeInvalidArg;
            sprintf(GradErr->_msg, "'nbLink' is invalid (0<=%d)", nbLink);
            PBErrCatch(GradErr);
        }
    #endif
    // Declare the new Grad
    Grad that;
    // Set properties
    that._type = type;
    that._dim = *dim;
    int area = GradGetArea(&that);
    that._cells = PBErrMalloc(GradErr, sizeof(GradCell) * area);
    VecShort2D pos = VecShortCreateStatic2D();
    int iCell = 0;
    // Loop on cells
    do {
        // Initialise the cell
        that._cells[iCell] = GradCellCreateStatic(iCell, nbLink, &pos);
        ++iCell;
    } while (VecPStep(&pos, dim));
    // Return the new Grad
    return that;
}

// Free memory used by the properties of the Grad 'that'
void GradFreeStatic(Grad* const that) {
    // Check arguments
    if (that == NULL)
        // Nothing to do
        return;
    // Free memory
    free(that->_cells);
}

// Get the appropriate deltas of positions according to the type of the
// Grad 'that' and the position 'pos'
int* _GradGetDelta(const Grad* const that, const VecShort2D* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GradErr->_type = PBErrTypeNullPointer;
            sprintf(GradErr->_msg, "'that' is null");
        }
    #endif
}

```



```

    PBErCatch(GradErr);
}
if (pos == NULL) {
    GradErr->_type = PBErTypeNullPointer;
    sprintf(GradErr->_msg, "'pos' is null");
    PBErCatch(GradErr);
}
#endif
if (GradGetType(that) == GradTypeSquare) {
    return GradDeltaSquare;
} else if (GradGetType(that) == GradTypeHexa) {
    if (GradHexaGetType((GradHexa*)that) == GradHexaTypeEvenQ) {
        if ((VecGet(pos, 0) % 2) != 0)
            return GradDeltaAEvenQ;
        else
            return GradDeltaBEvenQ;
    } else if (GradHexaGetType((GradHexa*)that) == GradHexaTypeEvenR) {
        if ((VecGet(pos, 1) % 2) != 0)
            return GradDeltaAEvenR;
        else
            return GradDeltaBEvenR;
    } else if (GradHexaGetType((GradHexa*)that) == GradHexaTypeOddQ) {
        if ((VecGet(pos, 0) % 2) != 0)
            return GradDeltaAOddQ;
        else
            return GradDeltaBOddQ;
    } else if (GradHexaGetType((GradHexa*)that) == GradHexaTypeOddR) {
        if ((VecGet(pos, 1) % 2) != 0)
            return GradDeltaAOddR;
        else
            return GradDeltaBOddR;
    }
}
return NULL;
}

// Create a new GradSquare of dimensions 'dim' and diagonal links
// allowed if 'diagLink' equals true
GradSquare* GradSquareCreate(const VecShort2D* const dim,
    const bool diagLink) {
    #if BUILDMODE == 0
        if (dim == NULL) {
            GradErr->_type = PBErTypeNullPointer;
            sprintf(GradErr->_msg, "'dim' is null");
            PBErCatch(GradErr);
        }
        if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
            GradErr->_type = PBErTypeInvalidArg;
            sprintf(GradErr->_msg, "'dim' is invalid ((0,0)<(%d,%d))",
                VecGet(dim, 0), VecGet(dim, 1));
            PBErCatch(GradErr);
        }
    #endif
    // Allocate memory
    GradSquare *that = PBErMalloc(GradErr, sizeof(GradSquare));
    // Set properties
    int nbLink = (diagLink ? 8 : 4);
    that->_grad = GradCreateStatic(dim, GradTypeSquare, nbLink);
    that->_diagLink = diagLink;
    // Loop on cells to initialise the links
    VecShort2D pos = VecShortCreateStatic2D();
    int* delta = GradGetDelta(that, &pos);

```

```

VecShort2D p = VecShortCreateStatic2D();
do {
    // Initialise the links of the cell
    for (int iLink = nbLink; iLink--;) {
        VecSet(&p, 0, VecGet(&pos, 0) + delta[iLink * 2]);
        VecSet(&p, 1, VecGet(&pos, 1) + delta[iLink * 2 + 1]);
        if (GradIsPosInside(that, &p))
            GradCellSetLink(GradCellAt(that, &pos), iLink,
                GradCellGetId(GradCellAt(that, &p)));
        if (iLink >= 4)
            GradCellSetLinkVal(GradCellAt(that, &pos), iLink,
                PBMATH_SQRTTWO);
    }
} while (VecStep(&pos, dim));
// Return the new Grad
return that;
}

// Free the memory used by the GradSquare 'that'
void GradSquareFree(GradSquare** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        // Nothing to do
        return;
    // Free memory
    GradFreeStatic(&((*that)->_grad));
    free(*that);
    *that = NULL;
}

// Create a new GradHexa with dimensions 'dim' and type 'type'
GradHexa* GradHexaCreate(const VecShort2D* const dim,
    const GradHexaType type) {
#ifdef BUILDMODE == 0
    if (dim == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'dim' is null");
        PBErrCatch(GradErr);
    }
    if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'dim' is invalid ((0,0)<(%d,%d))",
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(GradErr);
    }
}
#endif
    // Allocate memory
    GradHexa* that = PBErrMalloc(GradErr, sizeof(GradHexa));
    // Set properties
    that->_grad = GradCreateStatic(dim, GradTypeHexa, 6);
    that->_type = type;
    // Return the new Grad
    return that;
}

// Create a new GradHexa of dimensions 'dim' and orientation odd-r
GradHexa* GradHexaCreateOddR(const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (dim == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'dim' is null");
        PBErrCatch(GradErr);
    }

```

```

    }
    if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'dim' is invalid ((0,0)<(%d,%d))",
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(GradErr);
    }
#endif
    // Create the GradHexa
    GradHexa* that = GradHexaCreate(dim, GradHexaTypeOddR);
    // Loop on cells to initialise the links
    VecShort2D pos = VecShortCreateStatic2D();
    VecShort2D p = VecShortCreateStatic2D();
    int* delta = NULL;
    do {
        // Initialise the links of the cell
        delta = GradGetDelta(that, &pos);
        for (int iLink = 6; iLink--;) {
            VecSet(&p, 0, VecGet(&pos, 0) + delta[iLink * 2]);
            VecSet(&p, 1, VecGet(&pos, 1) + delta[iLink * 2 + 1]);
            if (GradIsPosInside(that, &p))
                GradCellSetLink(GradCellAt(that, &pos), iLink,
                    GradCellGetId(GradCellAt(that, &p)));
        }
    } while (VecStep(&pos, dim));
    // Return the new Grad
    return that;
}

// Create a new GradHexa of dimensions 'dim' and orientation even-r
GradHexa* GradHexaCreateEvenR(const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (dim == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'dim' is null");
        PBErrCatch(GradErr);
    }
    if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'dim' is invalid ((0,0)<(%d,%d))",
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(GradErr);
    }
#endif
    // Create the GradHexa
    GradHexa* that = GradHexaCreate(dim, GradHexaTypeEvenR);
    // Loop on cells to initialise the links
    VecShort2D pos = VecShortCreateStatic2D();
    VecShort2D p = VecShortCreateStatic2D();
    int* delta = NULL;
    do {
        // Initialise the links of the cell
        delta = GradGetDelta(that, &pos);
        for (int iLink = 6; iLink--;) {
            VecSet(&p, 0, VecGet(&pos, 0) + delta[iLink * 2]);
            VecSet(&p, 1, VecGet(&pos, 1) + delta[iLink * 2 + 1]);
            if (GradIsPosInside(that, &p))
                GradCellSetLink(GradCellAt(that, &pos), iLink,
                    GradCellGetId(GradCellAt(that, &p)));
        }
    } while (VecStep(&pos, dim));
    // Return the new Grad

```

```

    return that;
}

// Create a new GradHexa of dimensions 'dim' and orientation odd-q
GradHexa* GradHexaCreateOddQ(const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (dim == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'dim' is null");
        PBErrCatch(GradErr);
    }
    if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'dim' is invalid ((0,0)<(%d,%d))",
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(GradErr);
    }
#endif
    // Create the GradHexa
    GradHexa* that = GradHexaCreate(dim, GradHexaTypeOddQ);
    // Loop on cells to initialise the links
    VecShort2D pos = VecShortCreateStatic2D();
    VecShort2D p = VecShortCreateStatic2D();
    int* delta = NULL;
    do {
        // Initialise the links of the cell
        delta = GradGetDelta(that, &pos);
        for (int iLink = 6; iLink--;) {
            VecSet(&p, 0, VecGet(&pos, 0) + delta[iLink * 2]);
            VecSet(&p, 1, VecGet(&pos, 1) + delta[iLink * 2 + 1]);
            if (GradIsPosInside(that, &p))
                GradCellSetLink(GradCellAt(that, &pos), iLink,
                    GradCellGetId(GradCellAt(that, &p)));
        }
    } while (VecStep(&pos, dim));
    // Return the new Grad
    return that;
}

// Create a new GradHexa of dimensions 'dim' and orientation even-q
GradHexa* GradHexaCreateEvenQ(const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (dim == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'dim' is null");
        PBErrCatch(GradErr);
    }
    if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'dim' is invalid ((0,0)<(%d,%d))",
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(GradErr);
    }
#endif
    // Create the GradHexa
    GradHexa* that = GradHexaCreate(dim, GradHexaTypeEvenQ);
    // Loop on cells to initialise the links
    VecShort2D pos = VecShortCreateStatic2D();
    VecShort2D p = VecShortCreateStatic2D();
    int* delta = NULL;
    do {
        // Initialise the links of the cell

```

```

    delta = GradGetDelta(that, &pos);
    for (int iLink = 6; iLink--;) {
        VecSet(&p, 0, VecGet(&pos, 0) + delta[iLink * 2]);
        VecSet(&p, 1, VecGet(&pos, 1) + delta[iLink * 2 + 1]);
        if (GradIsPosInside(that, &p))
            GradCellSetLink(GradCellAt(that, &pos), iLink,
                GradCellGetId(GradCellAt(that, &p)));
    }
} while (VecStep(&pos, dim));
// Return the new Grad
return that;
}

// Free the memory used by the GradHexa 'that'
void GradHexaFree(GradHexa** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        // Nothing to do
        return;
    // Free memory
    GradFreeStatic(&((*that)->_grad));
    free(*that);
    *that = NULL;
}

// Get the GradCell at position 'pos' int the GradHexa 'that'
GradCell* _GradHexaCellAtPos(GradHexa* const that, VecShort2D* pos) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (pos == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'pos' is null");
        PBErrCatch(GradErr);
    }
    if (VecGet(pos, 0) < 0 || VecGet(pos, 1) < 0 ||
        VecGet(pos, 0) >= VecGet(GradDim(that), 0) ||
        VecGet(pos, 1) >= VecGet(GradDim(that), 1)) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'pos' is invalid ((0,0)<=(%d,%d)<(%d,%d))",
            VecGet(pos, 0), VecGet(pos, 1),
            VecGet(GradDim(that), 0), VecGet(GradDim(that), 1));
        PBErrCatch(GradErr);
    }
}
#endif
(void)that; (void)pos;
// Return the result
return NULL;
}

// Get the look up table for distance between each pair of cell of the
// Grad 'that'
// Return a MatFloat where first index is the 'from' cell's index
// and second index is the 'to' cell index
// Distances in the matrix are equal to the sum of the value of links
// between cells
// Negative distance means there is no path for the pair of cell
MatFloat* _GradGetLookupTableMinDist(const Grad* const that) {
#ifdef BUILDMODE == 0

```

```

if (that == NULL) {
    GradErr->_type = PBErrTypeNullPointer;
    sprintf(GradErr->_msg, "'that' is null");
    PBErrCatch(GradErr);
}
#endif
// Get the area of the grad
int area = GradGetArea(that);
// Create the result matrix
VecShort2D dim = VecShortCreateStatic2D();
VecSet(&dim, 0, area); VecSet(&dim, 1, area);
MatFloat* table = MatFloatCreate(&dim);
// Initialise the table
for (int iCell = area * area; iCell--;)
    table->_val[iCell] = -1.0;
VecShort2D pair = VecShortCreateStatic2D();
for (int iCell = area; iCell--;) {
    VecSet(&pair, 0, iCell);
    GradCell* cellFrom = GradCellAt(that, iCell);
    if (!GradCellIsBlocked(cellFrom)) {
        for (int iLink = GradCellGetNbLink(cellFrom); iLink--;) {
            int link = GradCellGetLink(cellFrom, iLink);
            if (link != -1 &&
                !GradCellIsBlocked(GradCellAt(that, link))) {
                VecSet(&pair, 1, GradCellGetLink(cellFrom, iLink));
                MatSet(table, &pair, GradCellLinkVal(cellFrom, iLink));
            }
        }
    }
}
}
// Loop until there is no more modification or we reach area steps
int nbStep = 0;
bool flagModif;
VecShort2D pairA = VecShortCreateStatic2D();
VecShort2D pairB = VecShortCreateStatic2D();
do {
    // Reset the flag for modification
    flagModif = false;
    // For each pair of cell
    VecSetNull(&pair);
    do {
        // If it's not a pair on the diagonal
        if (VecGet(&pair, 0) != VecGet(&pair, 1)) {
            // Search the minimum dist for this pair via another cell
            float min = -1.0;
            VecSet(&pairA, 0, VecGet(&pair, 0));
            VecSet(&pairB, 1, VecGet(&pair, 1));
            for (int k = area; k--;) {
                // If the other cell is different than the one in the
                // current pair
                if (k != VecGet(&pair, 0) && k != VecGet(&pair, 1)) {
                    VecSet(&pairA, 1, k);
                    VecSet(&pairB, 0, k);
                    // If the path through this other cell exists
                    if (MatGet(table, &pairA) >= 0.0 &&
                        MatGet(table, &pairB) >= 0.0) {
                        float d = MatGet(table, &pairA) + MatGet(table, &pairB);
                        if (min < 0.0 || min > d)
                            min = d;
                    }
                }
            }
        }
    }
}
}

```

```

        // If there was a path via another cell and this path is
        // shorter than the current one or there is no current one
        if (min >= 0.0 &&
            (MatGet(table, &pair) < 0.0 || MatGet(table, &pair) > min)) {
            // Update the min distance
            MatSet(table, &pair, min);
        }
    }
} while(VecStep(&pair, &dim));
// Increment the number of steps
++nbStep;
} while (nbStep < area && flagModif);
// Return the result
return table;
}

// Get the path from cell at index 'from' to cell at index 'to' in
// the Grad 'that' using the A* algorithm and the look up table 'lookup'
// for distance estimation between cells
// Return a VecShort of position (index) ordered from 'from' to 'to'
// Return NULL if there is no path
VecShort* _GradGetPath(const Grad* const that, const int from,
    const int to, const MatFloat* const lookup) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (lookup == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'lookup' is null");
        PBErrCatch(GradErr);
    }
}
#endif
// Declare a vector to consult the lookup table
VecShort2D iLookUp = VecShortCreateStatic2D();
// Get the estimated distance of the path
VecSet(&iLookUp, 0, from); VecSet(&iLookUp, 1, to);
float dist = MatGet(lookup, &iLookUp);
// Get the starting cell
GradCell* cell = GradCellAt(that, from);
// Declare a GSet of GradCell for computation
GSet openList = GSetCreateStatic();
// Declare a GSet to memorize the path
GSet path = GSetCreateStatic();
// Init the GSet with the starting cell
GSetPush(&openList, cell);
// Get the area of the grad
int area = GradGetArea(that);
// Declare arrays for computation
float* f = PBErrMalloc(GradErr, sizeof(float) * area);
float* g = PBErrMalloc(GradErr, sizeof(float) * area);
float* h = PBErrMalloc(GradErr, sizeof(float) * area);
bool* flagOpen = PBErrMalloc(GradErr, sizeof(bool) * area);
bool* flagClose = PBErrMalloc(GradErr, sizeof(bool) * area);
int* prev = PBErrMalloc(GradErr, sizeof(int) * area);
// Init the arrays
for (int i = area; i--;) {
    f[i] = dist;
    g[i] = 0.0;
    h[i] = dist;
}

```

```

    flagOpen[i] = false;
    flagClose[i] = false;
    prev[i] = -1;
}
flagOpen[from] = true;
// Loop until we have elements in the openList
while (GSetNbElem(&openList) > 0) {
    cell = GSetPop(&openList);
    int iCell = GradCellGetId(cell);
    if (iCell == to) {
        while (iCell != -1) {
            GSetPush(&path, cell);
            iCell = prev[GradCellGetId(cell)];
            if (iCell != -1)
                cell = GradCellAt(that, iCell);
        }
        GSetFlush(&openList);
    } else {
        flagClose[iCell] = true;
        float curDist = g[iCell];
        for (int iDir = GradCellGetNbLink(cell); iDir--;) {
            int ncell = GradCellGetLink(cell, iDir);
            if (ncell != -1) {
                if (flagClose[ncell] == false) {
                    GradCell* nextCell = GradCellAt(that, ncell);
                    if (flagOpen[ncell] == false) {
                        if (!GradCellIsBlocked(nextCell)) {
                            VecSet(&iLookUp, 0, iCell);
                            VecSet(&iLookUp, 1, ncell);
                            g[ncell] = curDist + MatGet(lookup, &iLookUp);
                            VecSet(&iLookUp, 0, ncell);
                            VecSet(&iLookUp, 1, to);
                            dist = MatGet(lookup, &iLookUp);
                            h[ncell] = dist;
                            f[ncell] = dist + g[ncell];
                            GSetAddSort(&openList, nextCell, f[ncell]);
                            flagOpen[ncell] = true;
                            prev[ncell] = iCell;
                        }
                    } else {
                        VecSet(&iLookUp, 0, iCell);
                        VecSet(&iLookUp, 1, ncell);
                        float ng = curDist + MatGet(lookup, &iLookUp);
                        if (ng < g[ncell]) {
                            GSetRemoveAll(&openList, nextCell);
                            g[ncell] = ng;
                            f[ncell] = g[ncell] + h[ncell];
                            prev[ncell] = iCell;
                            GSetAddSort(&openList, nextCell, f[ncell]);
                        }
                    }
                }
            }
        }
    }
}
// Free memory
free(f);
free(g);
free(h);
free(flagOpen);
free(flagClose);

```



```

    free(prev);
    GSetFlush(&openList);
    // Return the result
    if (GSetNbElem(&path) == 0) {
        return NULL;
    } else {
        VecShort* res = VecShortCreate(GSetNbElem(&path));
        int i = 0;
        while(GSetNbElem(&path) > 0) {
            cell = GSetPop(&path);
            VecSet(res, i, GradCellGetId(cell));
            ++i;
        }
        return res;
    }
}

// Structure used for flooding
typedef struct GradFloodPod {
    GradCell* _cell;
    long _src;
    int _nbStep;
} GradFloodPod;

// Flood the Grad 'that' from positions (index) 'sources' up to a
// maximum distance in link's value from the source 'distMax' or
// maximum distance in nb of cell from the source 'stepMax'
// If 'distMax' and/or 'stepMax' are/is negative(s) their is no limit
// on the maximum distance/maximum number of steps
// The flood occurs in order consistent with the links' value
// interpreted as distance
// The result is stored in the _flood property of the GradCell
// _flood == -1: not flooded, _flood >= 0: flooded by the _flood-th
// source
// Conflicting cells (several sources arriving at the same step to the
// cell) are left undecided (_flood==1)
void _GradFlood(Grad* const that, const VecShort* const sources,
    const float distMax, const int stepMax) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (sources == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'sources' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Reset all the flood value to -1
    for (int iCell = GradGetArea(that); iCell--;)
        GradCellSetFlood(GradCellAt(that, iCell), -1);
    // Get the nb of sources
    long nbSrc = VecGetDim(sources);
    // Declare a set of GradFloodPod
    GSet set = GSetCreateStatic();
    // For each sources
    for (long iSource = nbSrc; iSource--;) {
        // Add the first cell in the set
        GradFloodPod* pod = PBErrMalloc(GradErr, sizeof(GradFloodPod));
        pod->_src = iSource;
    }
}

```

```

    pod->_cell = GradCellAt(that, VecGet(sources, iSource));
    pod->_nbStep = 0;
    GSetAddSort(&set, pod, 0.0);
}
// Loop until the set is empty (ie every cell has
// been flooded
while (GSetNbElem(&set) > 0) {
    // Get the distance up to this cell
    float dist = GSetElement(&set, 0)->_sortVal;
    // Pop the cell
    GradFloodPod* pod = GSetPop(&set);
    // If the cell is inside the limit in nb of steps
    if (stepMax < 0 || pod->_nbStep <= stepMax) {
        // Declare a variable to manage conflict
        bool flagConflict = false;
        //if the set is not empty
        if (GSetNbElem(&set) > 0) {
            // Check references to this cell from other sources and
            // eliminate n-uples
            GSetIterForward iter = GSetIterForwardCreateStatic(&set);
            bool skipStep;
            do {
                skipStep = false;
                GradFloodPod* podCheck = GSetIterGet(&iter);
                if (podCheck->_cell == pod->_cell) {
                    float d = GSetIterGetElem(&iter)->_sortVal;
                    if (podCheck->_src != pod->_src && ISEQUALF(d, dist))
                        flagConflict = true;
                    free(podCheck);
                    skipStep = GSetIterRemoveElem(&iter);
                }
            } while (skipStep || GSetIterStep(&iter));
        }
        // If there was no conflict
        if (!flagConflict) {
            // Set the flood value of sources
            GradCellSetFlood(pod->_cell, pod->_src);
            // Loop on direction from this cell
            for (int iLink = GradCellGetNbLink(pod->_cell); iLink--;) {
                int toCell = GradCellGetLink(pod->_cell, iLink);
                // If there is a cell in this direction
                if (toCell != -1) {
                    // Get the distance to this cell from the source
                    float d = dist + GradCellLinkVal(pod->_cell, iLink);
                    // If it's within the max distance
                    if (distMax < 0.0 || d <= distMax) {
                        GradCell* cellTo = GradCellAt(that, toCell);
                        // If it's not yet flooded, not blocked, and not
                        // conflicting
                        if (GradCellGetFlood(cellTo) == -1 &&
                            !GradCellIsBlocked(cellTo)) {
                            // Add a new pod to the GSet 'setOut'
                            GradFloodPod* npod =
                                PBErrMalloc(GradErr, sizeof(GradFloodPod));
                            npod->_src = pod->_src;
                            npod->_nbStep = pod->_nbStep + 1;
                            npod->_cell = cellTo;
                            GSetAddSort(&set, npod, d);
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
}
// Free memory used by the GradFloodPod
free(pod);
}
// Free memory
while (GSetNbElem(&set) > 0) {
    GradFloodPod* pod = GSetPop(&set);
    free(pod);
}
}

// Get the number of flooded cells from 'iSource'-th source in the Grad
// 'that'
int _GradGetFloodArea(const Grad* const that, const int iSource) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Declare a variable to memorize the result
    int nb = 0;
    // Loop on cells
    for (int iCell = GradGetArea(that); iCell--;) {
        // If the flood value of the cell is the searched value
        if (GradCellGetFlood(GradCellAt(that, iCell)) == iSource)
            // Increment the result
            ++nb;
    }
    // Return the result
    return nb;
}

// Clone the GradSquare 'that'
// The user data are not cloned but shared between the original and
// its clone
GradSquare* GradSquareClone(const GradSquare* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Allocate memory
    GradSquare* clone = PBErrMalloc(GradErr, sizeof(GradSquare));
    // Copy the GradSquare
    *clone = *that;
    // Clone the GradCell
    clone->_grad._cells = PBErrMalloc(GradErr,
        sizeof(GradCell) * GradGetArea(that));
    memcpy(clone->_grad._cells, that->_grad._cells,
        sizeof(GradCell) * GradGetArea(that));
    // Return the clone
    return clone;
}

// Clone the GradHexa 'that'
// The user data are not cloned but shared between the original and
// its clone

```

```

GradHexa* GradHexaClone(const GradHexa* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Allocate memory
    GradHexa* clone = PBErrMalloc(GradErr, sizeof(GradHexa));
    // Copy the GradHexa
    *clone = *that;
    // Clone the GradCell
    clone->_grad._cells = PBErrMalloc(GradErr,
        sizeof(GradCell) * GradGetArea(that));
    memcpy(clone->_grad._cells, that->_grad._cells,
        sizeof(GradCell) * GradGetArea(that));
    // Return the clone
    return clone;
}

// Return true if the Grad 'that' is same as the Grad 'tho'
// Return false else
bool _GradIsSameAs(const Grad* const that, const Grad* const tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (tho == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'tho' is null");
        PBErrCatch(GradErr);
    }
#endif
    if (that->_type != tho->_type ||
        !VecIsEqual(&(that->_dim), &(tho->_dim)) ||
        (that->_type == GradTypeSquare &&
            ((GradSquare*)that)->_diagLink != ((GradSquare*)tho)->_diagLink) ||
        (that->_type == GradTypeHexa &&
            ((GradHexa*)that)->_type != ((GradHexa*)tho)->_type)) {
        return false;
    } else {
        for (int iCell = GradGetArea(that); iCell--;) {
            GradCell* cellA = GradCellAt(that, iCell);
            GradCell* cellB = GradCellAt(tho, iCell);
            if (cellA->_data != cellB->_data ||
                cellA->_id != cellB->_id ||
                !VecIsEqual(&(cellA->_pos), &(cellB->_pos)) ||
                cellA->_nbLink != cellB->_nbLink ||
                cellA->_flood != cellB->_flood ||
                cellA->_flagBlocked != cellB->_flagBlocked ||
                memcmp(cellA->_links, cellB->_links,
                    sizeof(int) * GRAD_NBMAXLINK) != 0 ||
                memcmp(cellA->_linksVal, cellB->_linksVal,
                    sizeof(float) * GRAD_NBMAXLINK) != 0) {
                return false;
            }
        }
        return true;
    }
}

```

```

}

// Remove the link from cell 'fromCell' to cell 'toCell' in the
// Grad 'that'
// If 'symmetric' equals true the symmetric link is removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveLinkIndex(Grad* const that, const int fromCell,
    const int toCell, const bool symmetric) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Loop on links of the 'fromCell'
    GradCell* cell = GradCellAt(that, fromCell);
    for (int iLink = GradCellGetNbLink(cell); iLink--;) {
        // If it's the link toward the 'toCell'
        if (GradCellGetLink(cell, iLink) == toCell) {
            // Remove it
            GradCellSetLink(cell, iLink, -1);
            // If we have to remove the symmetric link
            if (symmetric)
                _GradRemoveLinkIndex(that, toCell, fromCell, false);
            // Skip the end of the loop
            break;
        }
    }
}

// Remove the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If 'symmetric' equals true the symmetric link is removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveDirIndex(Grad* const that, const int fromCell,
    const int dir, const bool symmetric) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Get the cell
    GradCell* cell = GradCellAt(that, fromCell);
    // Get the neighbour cell
    int toCell = GradCellGetLink(cell, dir);
    // If there is a link in this direction
    if (toCell != -1) {
        // If we have to remove the symmetric link
        if (symmetric)
            _GradRemoveLinkIndex(that, toCell, fromCell, false);
        // Remove the link
        GradCellSetLink(cell, dir, -1);
    }
}

// Remove all the links from cell 'fromCell' in the Grad 'that'
// If 'symmetric' equals true the symmetric links are removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveAllLinkIndex(Grad* const that, const int fromCell,

```

```

    const bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Loop on links of the 'fromCell'
    GradCell* cell = GradCellAt(that, fromCell);
    for (int iLink = GradCellGetNbLink(cell); iLink--;) {
        // Memorize the link
        int toCell = GradCellGetLink(cell, iLink);
        // Remove the link
        GradCellSetLink(cell, iLink, -1);
        // If we have to remove the symmetric link and it exists
        if (symmetric && toCell != -1)
            _GradRemoveLinkIndex(that, toCell, fromCell, false);
    }
}

// Add the link from cell 'fromCell' to cell 'toCell' in the
// Grad 'that'
// If the cells are not neighbours do nothing
// If 'symmetric' equals true the symetric link is added too
void _GradAddLinkIndex(Grad* const that, int fromCell, int toCell,
    bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Declare pointer to the cells
    GradCell* cell = GradCellAt(that, fromCell);
    GradCell* cellTo = GradCellAt(that, toCell);
    // Declare variable for computation
    VecShort2D p = VecShortCreateStatic2D();
    // Get the table of delta position given the type of Grad
    int* delta = GradGetDelta(that, GradCellPos(cell));
    // Loop on links of the 'fromCell'
    for (int iLink = GradCellGetNbLink(cell); iLink--;) {
        // Get the position in this direction
        for (int i = 2; i--;)
            VecSet(&p, i, VecGet(GradCellPos(cell), i) + delta[2 * iLink + i]);
        // If it's the link toward the 'toCell'
        if (VecIsEqual(&p, GradCellPos(cellTo))) {
            // Add it
            GradCellSetLink(cell, iLink, toCell);
            // If we have to add the symmetric link
            if (symmetric)
                _GradAddLinkIndex(that, toCell, fromCell, false);
            // Skip the end of the loop
            break;
        }
    }
}

// Add the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If the cells are not neighbours do nothing

```

```

// If 'symmetric' equals true the symetric link is added too
void _GradAddDirIndex(Grad* const that, int const fromCell,
    const int dir, const bool symmetric) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GradErr->_type = PBErrTypeNullPointer;
            sprintf(GradErr->_msg, "'that' is null");
            PBErrCatch(GradErr);
        }
    #endif
    // Get the cell
    GradCell* cell = GradCellAt(that, fromCell);
    // Get the delta pos
    int* delta = GradGetDelta(that, GradCellPos(cell));
    // Get the neighbour cell pos
    VecShort2D p = VecShortCreateStatic2D();
    for (int i = 2; i--;)
        VecSet(&p, i, VecGet(GradCellPos(cell), i) + delta[2 * dir + i]);
    // If the neighbour cell exists
    if (GradIsPosInside(that, &p)) {
        // Get the neighbour cell
        GradCell* cellTo = GradCellAt(that, &p);
        int toCell = GradCellGetId(cellTo);
        // Set the link
        GradCellSetLink(cell, dir, toCell);
        // If we have to add the symmetric link
        if (symmetric)
            GradAddLinkTo(that, toCell, fromCell, false);
    }
}

// Add all the links from cell 'fromCell' in the Grad 'that'
// If 'symmetric' equals true the symetric links are removed too
void _GradAddAllLinkIndex(Grad* const that, const int fromCell,
    const bool symmetric) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GradErr->_type = PBErrTypeNullPointer;
            sprintf(GradErr->_msg, "'that' is null");
            PBErrCatch(GradErr);
        }
    #endif
    // Declare pointer to the cell
    GradCell* cell = GradCellAt(that, fromCell);
    // Declare variable for computation
    VecShort2D p = VecShortCreateStatic2D();
    // Get the table of delta position given the type of Grad
    int* delta = GradGetDelta(that, GradCellPos(cell));
    // Loop on links of the 'fromCell'
    for (int iLink = GradCellGetNbLink(cell); iLink--;) {
        // Get the position in this direction
        for (int i = 2; i--;)
            VecSet(&p, i, VecGet(GradCellPos(cell), i) + delta[2 * iLink + i]);
        // If the position is inside the Grad
        if (GradIsPosInside(that, &p)) {
            GradCell* cellTo = GradCellAt(that, &p);
            int toCell = GradCellGetId(cellTo);
            // Add the link
            GradCellSetLink(cell, iLink, toCell);
            // If we have to add the symmetric link
            if (symmetric)
                _GradAddLinkIndex(that, toCell, fromCell, false);
        }
    }
}

```

```

    }
}
}

```

## 3.2 grad-inline.c

```

// ===== GRAD-static inline.C =====

// ----- GradCell

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

// Get the user data of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
void* GradCellData(const GradCell* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_data;
}

// Set the user data of the GradCell 'that' to 'data'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetData(GradCell* const that, void* const data) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    that->_data = data;
}

// Get the position of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* GradCellPos(const GradCell* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return &(that->_pos);
}

// Get the index of the GradCell 'that'

```



```

#if BUILDMODE != 0
static inline
#endif
int GradCellGetId(const GradCell* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_id;
}

// Get the index of 'iLink'-th link of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
int GradCellGetLink(const GradCell* const that, const int iLink) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (iLink < 0 || iLink >= that->_nbLink) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'iLink' is invalid (0<=%d<%d)",
            iLink, that->_nbLink);
        PBErrCatch(GradErr);
    }
#endif
    return that->_links[iLink];
}

// Set the index of 'iLink'-th link of the GradCell 'that' to 'iCell'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetLink(GradCell* const that, const int iLink,
    const int iCell) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (iLink < 0 || iLink >= that->_nbLink) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'iLink' is invalid (0<=%d<%d)",
            iLink, that->_nbLink);
        PBErrCatch(GradErr);
    }
    if (iCell < -1) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'iCell' is invalid (-1<=%d)", iCell);
        PBErrCatch(GradErr);
    }
#endif
    that->_links[iLink] = iCell;
}

```

```

// Get the number of links of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
int GradCellGetNbLink(const GradCell* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_nbLink;
}

// Get the value of 'iLink'-th link of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif
float GradCellLinkVal(const GradCell* const that, const int iLink) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (iLink < 0 || iLink >= that->_nbLink) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'iLink' is invalid (0<=%d<=%d)",
            iLink, that->_nbLink);
        PBErrCatch(GradErr);
    }
#endif
    return that->_linksVal[iLink];
}

// Set the value of 'iLink'-th link of the GradCell 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void GradCellSetLinkVal(GradCell* const that, const int iLink,
    const float val) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (iLink < 0 || iLink >= that->_nbLink) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'iLink' is invalid (0<=%d<=%d)",
            iLink, that->_nbLink);
        PBErrCatch(GradErr);
    }
#endif
    that->_linksVal[iLink] = val;
}

// Get the flood value of the GradCell 'that'
#if BUILDMODE != 0
static inline
#endif

```

```

int GradCellGetFlood(const GradCell* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_flood;
}

// Set the flood value of the GradCell 'that' to 'iSource'
#ifdef BUILDMODE != 0
static inline
#endif
void GradCellSetFlood(GradCell* const that, const int iSource) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    that->_flood = iSource;
}

// Get the flag blocked of the GradCell 'that'
#ifdef BUILDMODE != 0
static inline
#endif
bool GradCellIsBlocked(const GradCell* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_flagBlocked;
}

// Set the flag blocked of the GradCell 'that' to 'flag'
#ifdef BUILDMODE != 0
static inline
#endif
void GradCellSetBlocked(GradCell* const that, const bool flag) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    that->_flagBlocked = flag;
}

// ----- Grad

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

// Get the GradCell at index 'iCell' in the Grad 'that'
#ifdef BUILDMODE != 0

```

```

static inline
#endif
GradCell* _GradCellAtIndex(const Grad* const that, const int iCell) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (iCell < 0 || iCell >= GradGetArea(that)) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'iCell' is invalid (0<=%d<=%d)",
            iCell, GradGetArea(that));
        PBErrCatch(GradErr);
    }
#endif
    return that->_cells + iCell;
}

// Get the GradCell at position 'pos' int the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
GradCell* _GradCellAtPos(const Grad* const that,
    const VecShort2D* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (VecGet(pos, 0) < 0 || VecGet(pos, 1) < 0 ||
        VecGet(pos, 0) >= VecGet(GradDim(that), 0) ||
        VecGet(pos, 1) >= VecGet(GradDim(that), 1)) {
        GradErr->_type = PBErrTypeInvalidArg;
        sprintf(GradErr->_msg, "'pos' is invalid ((0,0)<=(%d,%d)<(%d,%d))",
            VecGet(pos, 0), VecGet(pos, 1),
            VecGet(GradDim(that), 0), VecGet(GradDim(that), 1));
        PBErrCatch(GradErr);
    }
#endif
    return that->_cells + GradPosToIndex(that, pos);
}

// Get the GradType of the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
GradType _GradGetType(const Grad* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_type;
}

// Get the GradHexaType of the GradHexa 'that'
#if BUILDMODE != 0
static inline

```

```

#endif
GradHexaType GradHexaGetType(const GradHexa* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_type;
}

// Get the number of cells (area) of the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
int _GradGetArea(const Grad* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return VecGet(GradDim(that), 0) * VecGet(GradDim(that), 1);
}

// Get the dimensions of the Grad 'that'
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* _GradDim(const Grad* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return &(that->_dim);
}

// Check if the position 'pos' is inside the GradSquare 'that'
#if BUILDMODE != 0
static inline
#endif
bool _GradIsPosInside(const Grad* const that,
    const VecShort2D* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    if (VecGet(pos, 0) < 0 || VecGet(pos, 1) < 0 ||
        VecGet(pos, 0) >= VecGet(GradDim(that), 0) ||
        VecGet(pos, 1) >= VecGet(GradDim(that), 1)) {
        return false;
    } else {
        return true;
    }
}

```

```

}

// Set the flag blocked of all cells in the Grad 'that' to false
#if BUILDMODE != 0
static inline
#endif
void _GradResetFlagBlocked(Grad* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    for (int iCell = GradGetArea(that); iCell--;)
        GradCellSetBlocked(GradCellAt(that, iCell), false);
}

// Return true if the GradSquare 'that' has diagonal link
// Return false else
#if BUILDMODE != 0
static inline
#endif
bool GradSquareHasDiagonalLink(const GradSquare* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
#endif
    return that->_diagLink;
}

// Remove the link from cell at position 'fromCell' to cell at
// position 'toCell' in the Grad 'that'
// If 'symmetric' equals true the symmetric link is removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
static inline
#endif
void _GradRemoveLinkPos(Grad* const that,
    const VecShort2D* const fromCell, const VecShort2D* const toCell,
    const bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (fromCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'fromCell' is null");
        PBErrCatch(GradErr);
    }
    if (toCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'toCell' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Get the index of 'fromCell' and 'toCell'

```

```

    int from = GradPosToIndex(that, fromCell);
    int to = GradPosToIndex(that, toCell);
    // Remove the link
    _GradRemoveLinkIndex(that, from, to, symmetric);
}

// Remove the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If 'symmetric' equals true the symmetric link is removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
static inline
#endif
void _GradRemoveDirPos(Grad* const that,
    const VecShort2D* const fromCell, const int dir, const bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (fromCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'fromCell' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Get the index of 'fromCell'
    int from = GradPosToIndex(that, fromCell);
    // Remove the link
    _GradRemoveDirIndex(that, from, dir, symmetric);
}

// Remove all the links from cell at position 'fromCell' in
// the Grad 'that'
// If 'symmetric' equals true the symmetric links are removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
static inline
#endif
void _GradRemoveAllLinkPos(Grad* const that,
    const VecShort2D* const fromCell, const bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (fromCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'fromCell' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Get the index of 'fromCell'
    int from = GradPosToIndex(that, fromCell);
    // Remove the link
    _GradRemoveAllLinkIndex(that, from, symmetric);
}

// Add the link from cell at position 'fromCell' to cell at
// position 'toCell' in the Grad 'that'

```

```

// If the cells are not neighbours do nothing
// If 'symmetric' equals true the symetric link is added too
#if BUILDMODE != 0
static inline
#endif
void _GradAddLinkPos(Grad* const that, const VecShort2D* const fromCell,
    const VecShort2D* const toCell, const bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (fromCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'fromCell' is null");
        PBErrCatch(GradErr);
    }
    if (toCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'toCell' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Get the index of 'fromCell' and 'toCell'
    int from = GradPosToIndex(that, fromCell);
    int to = GradPosToIndex(that, toCell);
    // Remove the link
    _GradAddLinkIndex(that, from, to, symmetric);
}

// Add the link from cell at position 'fromCell' toward direction
// 'dir' in the Grad 'that'
// If the cells are not neighbours do nothing
// If 'symmetric' equals true the symetric link is added too
#if BUILDMODE != 0
static inline
#endif
void _GradAddDirPos(Grad* const that, const VecShort2D* const fromCell,
    const int dir, const bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (fromCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'fromCell' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Get the index of 'fromCell' and 'toCell'
    int from = GradPosToIndex(that, fromCell);
    // Remove the link
    _GradAddDirIndex(that, from, dir, symmetric);
}

// Add all the links from cell at position 'fromCell' in
// the Grad 'that'
// If 'symmetric' equals true the symetric links are removed too
#if BUILDMODE != 0

```



```

static inline
#endif
void _GradAddAllLinkPos(Grad* const that,
    const VecShort2D* const fromCell, const bool symmetric) {
#if BUILDMODE == 0
    if (that == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'that' is null");
        PBErrCatch(GradErr);
    }
    if (fromCell == NULL) {
        GradErr->_type = PBErrTypeNullPointer;
        sprintf(GradErr->_msg, "'fromCell' is null");
        PBErrCatch(GradErr);
    }
#endif
    // Get the index of 'fromCell' and 'toCell'
    int from = GradPosToIndex(that, fromCell);
    // Remove the link
    _GradAddAllLinkIndex(that, from, symmetric);
}

```

## 4 Makefile

```

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

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=grad
$($(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 "grad.h"

#define RANDOMSEED 0

void UnitTestGradCellCreateFree() {
    VecShort2D pos = VecShortCreateStatic2D();
    VecSet(&pos, 0, 3.0); VecSet(&pos, 1, 4.0);
    GradCell* cell = GradCellCreate(1, 2, &pos);
    if (cell == NULL ||
        cell->_id != 1 ||
        cell->_nbLink != 2 ||
        cell->_data != NULL ||
        cell->_flood != -1 ||
        cell->_flagBlocked != false ||
        VecIsEqual(&(cell->_pos), &pos) == false) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellCreate failed");
        PBErrCatch(GradErr);
    }
    for (int iLink = GRAD_NBMAXLINK; iLink--;) {
        if (cell->_links[iLink] != -1 ||
            ISEQUALF(cell->_linksVal[iLink], 1.0) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradCellCreate failed");
            PBErrCatch(GradErr);
        }
    }
    GradCellFree(&cell);
    if (cell != NULL) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellFree failed");
        PBErrCatch(GradErr);
    }
    printf("UnitTestGradCellCreateFree OK\n");
}

void UnitTestGradCellGetSet() {
    VecShort2D pos = VecShortCreateStatic2D();
    VecSet(&pos, 0, 3.0); VecSet(&pos, 1, 4.0);
    GradCell* cell = GradCellCreate(1, 2, &pos);
    if (GradCellGetId(cell) != cell->_id) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellGetId failed");
        PBErrCatch(GradErr);
    }
    if (GradCellGetFlood(cell) != cell->_flood) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellGetFlood failed");
        PBErrCatch(GradErr);
    }
    if (GradCellGetNbLink(cell) != cell->_nbLink) {
```

```

        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellGetFlood failed");
        PBErrCatch(GradErr);
    }
    if (GradCellIsBlocked(cell) != cell->_flagBlocked) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellIsBlocked failed");
        PBErrCatch(GradErr);
    }
    if (GradCellData(cell) != cell->_data) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellData failed");
        PBErrCatch(GradErr);
    }
    if (GradCellGetLink(cell, 1) != cell->_links[1]) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellLink failed");
        PBErrCatch(GradErr);
    }
    if (GradCellLinkVal(cell, 1) != cell->_linksVal[1]) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellLinkVal failed");
        PBErrCatch(GradErr);
    }
    int val;
    GradCellSetData(cell, &val);
    if (GradCellData(cell) != &val) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellSetData failed");
        PBErrCatch(GradErr);
    }
    GradCellSetLink(cell, 1, 2);
    if (GradCellGetLink(cell, 1) != 2) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellSetLink failed");
        PBErrCatch(GradErr);
    }
    GradCellSetLinkVal(cell, 1, 2.0);
    if (ISEQUALF(GradCellLinkVal(cell, 1), 2.0) == false) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellSetLinkVal failed");
        PBErrCatch(GradErr);
    }
    GradCellSetBlocked(cell, true);
    if (GradCellIsBlocked(cell) == false) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellSetBlocked failed");
        PBErrCatch(GradErr);
    }
    GradCellSetFlood(cell, 1);
    if (GradCellGetFlood(cell) != 1) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradCellSetFlood failed");
        PBErrCatch(GradErr);
    }
    GradCellFree(&cell);
    printf("UnitTestGradCellGetSet OK\n");
}

void UnitTestGradCell() {
    UnitTestGradCellCreateFree();
    UnitTestGradCellGetSet();
}

```

```

    printf("UnitTestGradCell OK\n");
}

void UnitTestGradCreateFree() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    GradSquare* gradSquare = GradSquareCreate(&dim, diagLink);
    if (gradSquare == NULL ||
        VecIsEqual(&(gradSquare->_grad._dim), &dim) == false ||
        gradSquare->_diagLink != diagLink ||
        gradSquare->_grad._type != GradTypeSquare ||
        gradSquare->_grad._cells == NULL) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradSquareCreate failed");
        PBErrCatch(GradErr);
    }
    int iCell = 0;
    VecShort2D pos = VecShortCreateStatic2D();
    do {
        if (VecIsEqual(GradCellPos(gradSquare->_grad._cells + iCell),
            &pos) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradSquareCreate failed");
            PBErrCatch(GradErr);
        }
        ++iCell;
    } while (VecPStep(&pos, &dim));
    int checkA[48] = {
        -1, 1, 2, -1, -1, -1, 3, -1,
        -1, -1, 3, 0, -1, -1, -1, 2,
        0, 3, 4, -1, -1, 1, 5, -1,
        1, -1, 5, 2, 0, -1, -1, 4,
        2, 5, -1, -1, -1, 3, -1, -1,
        3, -1, -1, 4, 2, -1, -1, -1
    };
    int iCheck = 0;
    for (int iCell = 0; iCell < 6; ++iCell) {
        if (gradSquare->_grad._cells[iCell]._nbLink != 8) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradSquareCreate failed");
            PBErrCatch(GradErr);
        }
        for (int iLink = 0; iLink < 8; ++iLink) {
            if (gradSquare->_grad._cells[iCell]._links[iLink] !=
                checkA[iCheck] ||
                (iLink < 4 &&
                 ISEQUALF(gradSquare->_grad._cells[iCell]._linksVal[iLink],
                     1.0) == false) ||
                (iLink >= 4 &&
                 ISEQUALF(gradSquare->_grad._cells[iCell]._linksVal[iLink],
                     PBMATH_SQRTTWO) == false)) {
                GradErr->_type = PBErrTypeUnitTestFailed;
                sprintf(GradErr->_msg, "GradSquareCreate failed");
                PBErrCatch(GradErr);
            }
            ++iCheck;
        }
    }
    GradSquareFree(&gradSquare);
    diagLink = false;
    gradSquare = GradSquareCreate(&dim, diagLink);
}

```

```

if (gradSquare == NULL) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradSquareCreate failed");
    PBErrCatch(GradErr);
}
int checkB[24] = {
    -1, 1, 2, -1,
    -1, -1, 3, 0,
    0, 3, 4, -1,
    1, -1, 5, 2,
    2, 5, -1, -1,
    3, -1, -1, 4
};
iCheck = 0;
for (int iCell = 0; iCell < 6; ++iCell) {
    if (gradSquare->_grad._cells[iCell]._nbLink != 4) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradSquareCreate failed");
        PBErrCatch(GradErr);
    }
    for (int iLink = 0; iLink < 4; ++iLink) {
        if (gradSquare->_grad._cells[iCell]._links[iLink] !=
            checkB[iCheck] ||
            ISEQUALF(gradSquare->_grad._cells[iCell]._linksVal[iLink],
                1.0) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradSquareCreate failed");
            PBErrCatch(GradErr);
        }
        ++iCheck;
    }
}
GradSquareFree(&gradSquare);
if (gradSquare != NULL) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradSquareFree failed");
    PBErrCatch(GradErr);
}
GradHexa* gradHexa = GradHexaCreateOddR(&dim);
if (gradHexa == NULL ||
    VecIsEqual(&(gradHexa->_grad._dim), &dim) == false ||
    gradHexa->_grad._type != GradTypeHexa ||
    gradHexa->_type != GradHexaTypeOddR ||
    gradHexa->_grad._cells == NULL) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradHexaCreateOddR failed");
    PBErrCatch(GradErr);
}
int checkC[36] = {
    -1, -1, 1, 2, -1, -1,
    -1, -1, -1, 3, 2, 0,
    0, 1, 3, 5, 4, -1,
    1, -1, -1, -1, 5, 2,
    -1, 2, 5, -1, -1, -1,
    2, 3, -1, -1, -1, 4
};
iCheck = 0;
for (int iCell = 0; iCell < 6; ++iCell) {
    if (gradHexa->_grad._cells[iCell]._nbLink != 6) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradHexaCreateOddR failed");
        PBErrCatch(GradErr);
    }
}

```

```

    }
    for (int iLink = 0; iLink < 6; ++iLink) {
        if (gradHexa->_grad._cells[iCell]._links[iLink] !=
            checkC[iCheck] ||
            ISEQUALF(gradHexa->_grad._cells[iCell]._linksVal[iLink],
                1.0) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradHexaCreateOddR failed");
            PBErrCatch(GradErr);
        }
        ++iCheck;
    }
}
}
GradHexaFree(&gradHexa);
gradHexa = GradHexaCreateEvenR(&dim);
if (gradHexa == NULL ||
    VecIsEqual(&(gradHexa->_grad._dim), &dim) == false ||
    gradHexa->_grad._type != GradTypeHexa ||
    gradHexa->_type != GradHexaTypeEvenR ||
    gradHexa->_grad._cells == NULL) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradHexaCreateEvenR failed");
    PBErrCatch(GradErr);
}
int checkD[36] = {
    -1, -1, 1, 3, 2, -1,
    -1, -1, -1, -1, 3, 0,
    -1, 0, 3, 4, -1, -1,
    0, 1, -1, 5, 4, 2,
    2, 3, 5, -1, -1, -1,
    3, -1, -1, -1, -1, 4
};
iCheck = 0;
for (int iCell = 0; iCell < 6; ++iCell) {
    if (gradHexa->_grad._cells[iCell]._nbLink != 6) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradHexaCreateEvenR failed");
        PBErrCatch(GradErr);
    }
    for (int iLink = 0; iLink < 6; ++iLink) {
        if (gradHexa->_grad._cells[iCell]._links[iLink] !=
            checkD[iCheck] ||
            ISEQUALF(gradHexa->_grad._cells[iCell]._linksVal[iLink],
                1.0) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradHexaCreateEvenR failed");
            PBErrCatch(GradErr);
        }
        ++iCheck;
    }
}
}
GradHexaFree(&gradHexa);
gradHexa = GradHexaCreateOddQ(&dim);
if (gradHexa == NULL ||
    VecIsEqual(&(gradHexa->_grad._dim), &dim) == false ||
    gradHexa->_grad._type != GradTypeHexa ||
    gradHexa->_type != GradHexaTypeOddQ ||
    gradHexa->_grad._cells == NULL) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradHexaCreateOddQ failed");
    PBErrCatch(GradErr);
}
}

```

```

int checkE[36] = {
    -1, -1, 1, 2, -1, -1,
    -1, -1, -1, 3, 2, 0,
    0, 1, 3, 4, -1, -1,
    1, -1, -1, 5, 4, 2,
    2, 3, 5, -1, -1, -1,
    3, -1, -1, -1, -1, 4
};

iCheck = 0;
for (int iCell = 0; iCell < 6; ++iCell) {
    if (gradHexa->_grad._cells[iCell]._nbLink != 6) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradHexaCreateOddQfailed");
        PBErrCatch(GradErr);
    }
    for (int iLink = 0; iLink < 6; ++iLink) {
        if (gradHexa->_grad._cells[iCell]._links[iLink] !=
            checkE[iCheck] ||
            ISEQUALF(gradHexa->_grad._cells[iCell]._linksVal[iLink],
                1.0) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradHexaCreateOddQ failed");
            PBErrCatch(GradErr);
        }
        ++iCheck;
    }
}
GradHexaFree(&gradHexa);
gradHexa = GradHexaCreateEvenQ(&dim);
if (gradHexa == NULL ||
    VecIsEqual(&(gradHexa->_grad._dim), &dim) == false ||
    gradHexa->_grad._type != GradTypeHexa ||
    gradHexa->_type != GradHexaTypeEvenQ ||
    gradHexa->_grad._cells == NULL) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradHexaCreateEvenQ failed");
    PBErrCatch(GradErr);
}
int checkF[36] = {
    -1, 1, 3, 2, -1, -1,
    -1, -1, -1, 3, 0, -1,
    0, 3, 5, 4, -1, -1,
    1, -1, -1, 5, 2, 0,
    2, 5, -1, -1, -1, -1,
    3, -1, -1, -1, 4, 2
};

iCheck = 0;
for (int iCell = 0; iCell < 6; ++iCell) {
    if (gradHexa->_grad._cells[iCell]._nbLink != 6) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradHexaCreateEvenQfailed");
        PBErrCatch(GradErr);
    }
    for (int iLink = 0; iLink < 6; ++iLink) {
        if (gradHexa->_grad._cells[iCell]._links[iLink] !=
            checkF[iCheck] ||
            ISEQUALF(gradHexa->_grad._cells[iCell]._linksVal[iLink],
                1.0) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradHexaCreateEvenQ failed");
            PBErrCatch(GradErr);
        }
    }
}

```

```

        ++iCheck;
    }
}
GradHexaFree(&gradHexa);
if (gradSquare != NULL) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradHexaFree failed");
    PBErrCatch(GradErr);
}
printf("UnitTestGradCreateFree OK\n");
}

void UnitTestGradCloneIsSame() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    GradSquare* gradSquare = GradSquareCreate(&dim, diagLink);
    GradSquare* cloneSquare = GradSquareClone(gradSquare);
    if (cloneSquare == NULL) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradSquareClone failed");
        PBErrCatch(GradErr);
    }
    if (GradIsSameAs(gradSquare, cloneSquare) == false) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradIsSameAs failed");
        PBErrCatch(GradErr);
    }
    GradFree(&gradSquare);
    GradFree(&cloneSquare);
    GradHexa* gradHexa = GradHexaCreateOddQ(&dim);
    GradHexa* cloneHexa = GradHexaClone(gradHexa);
    if (cloneHexa == NULL) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradHexaClone failed");
        PBErrCatch(GradErr);
    }
    if (GradIsSameAs(gradHexa, cloneHexa) == false) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradIsSameAs failed");
        PBErrCatch(GradErr);
    }
    GradFree(&gradHexa);
    GradFree(&cloneHexa);
    printf("UnitTestGradCloneIsSame OK\n");
}

void UnitTestGradGetSet() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    GradSquare* gradSquare = GradSquareCreate(&dim, diagLink);
    GradHexa* gradHexa = GradHexaCreateOddQ(&dim);
    if (GradGetArea(gradSquare) != 6) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradGetArea failed");
        PBErrCatch(GradErr);
    }
    if (GradCellAt(gradSquare, 1) != gradSquare->_grad._cells + 1) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradGetCell failed");
        PBErrCatch(GradErr);
    }
}

```



```

}
VecShort2D pos = VecShortCreateStatic2D();
VecSet(&pos, 0, 1); VecSet(&pos, 1, 2);
if (GradCellAt(gradSquare, &pos) != gradSquare->_grad._cells + 5) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradGetCell failed");
    PBErrCatch(GradErr);
}
if (GradGetType(gradSquare) != GradTypeSquare) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradGetType failed");
    PBErrCatch(GradErr);
}
if (GradHexaGetType(gradHexa) != GradHexaTypeOddQ) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradHexaGetType failed");
    PBErrCatch(GradErr);
}
if (GradSquareHasDiagonalLink(gradSquare) != diagLink) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradSquareHasDiagonalLink failed");
    PBErrCatch(GradErr);
}
GradFree(&gradSquare);
GradFree(&gradHexa);
printf("UnitTestGradGetSet OK\n");
}

void UnitTestGradResetFlagBlocked() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    GradSquare* grad = GradSquareCreate(&dim, diagLink);
    for (int iCell = GradGetArea(grad); iCell--;)
        GradCellSetBlocked(GradCellAt(grad, iCell), true);
    GradResetFlagBlocked(grad);
    for (int iCell = GradGetArea(grad); iCell--;)
        if (GradCellIsBlocked(GradCellAt(grad, iCell)) != false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradResetFlagBlocked failed");
            PBErrCatch(GradErr);
        }
    GradFree(&grad);
    printf("UnitTestGradResetFlagBlocked OK\n");
}

void UnitTestGradEditLinks() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    GradSquare* grad = GradSquareCreate(&dim, diagLink);
    GradRemoveLinkTo(grad, 0, 1, false);
    if (GradCellGetLink(GradCellAt(grad, 0), GradSquareDirE) != -1 ||
        GradCellGetLink(GradCellAt(grad, 1), GradSquareDirW) != 0) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradRemoveLinkTo failed");
        PBErrCatch(GradErr);
    }
    GradAddLinkTo(grad, 0, 1, false);
    if (GradCellGetLink(GradCellAt(grad, 0), GradSquareDirE) != 1 ||
        GradCellGetLink(GradCellAt(grad, 1), GradSquareDirW) != 0) {
        GradErr->_type = PBErrTypeUnitTestFailed;
    }
}

```

```

    sprintf(GradErr->_msg, "GradAddLinkTo failed");
    PBErrCatch(GradErr);
}
GradRemoveLinkTo(grad, 2, 3, true);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != -1) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradRemoveLinkTo failed");
    PBErrCatch(GradErr);
}
GradAddLinkTo(grad, 2, 3, true);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != 3 ||
    GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != 2) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradAddLinkTo failed");
    PBErrCatch(GradErr);
}
VecShort2D from = VecShortCreateStatic2D();
VecSet(&from, 0, 0); VecSet(&from, 1, 2);
VecShort2D to = VecShortCreateStatic2D();
VecSet(&to, 0, 1); VecSet(&to, 1, 2);
GradRemoveLinkTo(grad, &from, &to, true);
if (GradCellGetLink(GradCellAt(grad, 4), GradSquareDirE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 5), GradSquareDirW) != -1) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradRemoveLinkTo failed");
    PBErrCatch(GradErr);
}
GradAddLinkTo(grad, &from, &to, true);
if (GradCellGetLink(GradCellAt(grad, 4), GradSquareDirE) != 5 ||
    GradCellGetLink(GradCellAt(grad, 5), GradSquareDirW) != 4) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradAddLinkTo failed");
    PBErrCatch(GradErr);
}
GradRemoveLinkToward(grad, 0, GradSquareDirE, false);
if (GradCellGetLink(GradCellAt(grad, 0), GradSquareDirE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 1), GradSquareDirW) != 0) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradRemoveLinkToward failed");
    PBErrCatch(GradErr);
}
GradAddLinkToward(grad, 0, GradSquareDirE, false);
if (GradCellGetLink(GradCellAt(grad, 0), GradSquareDirE) != 1 ||
    GradCellGetLink(GradCellAt(grad, 1), GradSquareDirW) != 0) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradAddLinkToward failed");
    PBErrCatch(GradErr);
}
GradRemoveLinkToward(grad, 2, GradSquareDirE, true);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != -1) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradRemoveLinkToward failed");
    PBErrCatch(GradErr);
}
GradAddLinkToward(grad, 2, GradSquareDirE, true);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != 3 ||
    GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != 2) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradAddLinkToward failed");
    PBErrCatch(GradErr);
}

```

```

}
GradRemoveLinkToward(grad, &from, GradSquareDirE, true);
if (GradCellGetLink(GradCellAt(grad, 4), GradSquareDirE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 5), GradSquareDirW) != -1) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradRemoveLinkToward failed");
    PBErrCatch(GradErr);
}
GradAddLinkToward(grad, &from, GradSquareDirE, true);
if (GradCellGetLink(GradCellAt(grad, 4), GradSquareDirE) != 5 ||
    GradCellGetLink(GradCellAt(grad, 5), GradSquareDirW) != 4) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradAddLinkToward failed");
    PBErrCatch(GradErr);
}
GradRemoveAllLink(grad, 2, false);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirN) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirNE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirSE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirS) != -1 ||
    GradCellGetLink(GradCellAt(grad, 0), GradSquareDirS) != 2 ||
    GradCellGetLink(GradCellAt(grad, 1), GradSquareDirSW) != 2 ||
    GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != 2 ||
    GradCellGetLink(GradCellAt(grad, 5), GradSquareDirNW) != 2 ||
    GradCellGetLink(GradCellAt(grad, 4), GradSquareDirN) != 2) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradRemoveAllLink failed");
    PBErrCatch(GradErr);
}
GradAddAllLink(grad, 2, false);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirN) != 0 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirNE) != 1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != 3 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirSE) != 5 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirS) != 4 ||
    GradCellGetLink(GradCellAt(grad, 0), GradSquareDirS) != 2 ||
    GradCellGetLink(GradCellAt(grad, 1), GradSquareDirSW) != 2 ||
    GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != 2 ||
    GradCellGetLink(GradCellAt(grad, 5), GradSquareDirNW) != 2 ||
    GradCellGetLink(GradCellAt(grad, 4), GradSquareDirN) != 2) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradAddAllLink failed");
    PBErrCatch(GradErr);
}
GradRemoveAllLink(grad, 2, true);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirN) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirNE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirSE) != -1 ||
    GradCellGetLink(GradCellAt(grad, 2), GradSquareDirS) != -1 ||
    GradCellGetLink(GradCellAt(grad, 0), GradSquareDirS) != -1 ||
    GradCellGetLink(GradCellAt(grad, 1), GradSquareDirSW) != -1 ||
    GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != -1 ||
    GradCellGetLink(GradCellAt(grad, 5), GradSquareDirNW) != -1 ||
    GradCellGetLink(GradCellAt(grad, 4), GradSquareDirN) != -1) {
    GradErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GradErr->_msg, "GradRemoveAllLink failed");
    PBErrCatch(GradErr);
}
GradAddAllLink(grad, 2, true);
if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirN) != 0 ||

```

```

        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirNE) != 1 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != 3 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirSE) != 5 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirS) != 4 ||
        GradCellGetLink(GradCellAt(grad, 0), GradSquareDirS) != 2 ||
        GradCellGetLink(GradCellAt(grad, 1), GradSquareDirSW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 5), GradSquareDirNW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 4), GradSquareDirN) != 2) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradAddAllLink failed");
        PBErrCatch(GradErr);
    }
    VecSet(&from, 0, 0); VecSet(&from, 1, 1);
    GradRemoveAllLink(grad, &from, false);
    if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirN) != -1 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirNE) != -1 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != -1 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirSE) != -1 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirS) != -1 ||
        GradCellGetLink(GradCellAt(grad, 0), GradSquareDirS) != 2 ||
        GradCellGetLink(GradCellAt(grad, 1), GradSquareDirSW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 5), GradSquareDirNW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 4), GradSquareDirN) != 2) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradRemoveAllLink failed");
        PBErrCatch(GradErr);
    }
    GradAddAllLink(grad, &from, false);
    if (GradCellGetLink(GradCellAt(grad, 2), GradSquareDirN) != 0 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirNE) != 1 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirE) != 3 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirSE) != 5 ||
        GradCellGetLink(GradCellAt(grad, 2), GradSquareDirS) != 4 ||
        GradCellGetLink(GradCellAt(grad, 0), GradSquareDirS) != 2 ||
        GradCellGetLink(GradCellAt(grad, 1), GradSquareDirSW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 3), GradSquareDirW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 5), GradSquareDirNW) != 2 ||
        GradCellGetLink(GradCellAt(grad, 4), GradSquareDirN) != 2) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradAddAllLink failed");
        PBErrCatch(GradErr);
    }
    GradFree(&grad);
    printf("UnitTestGradEditLinks OK\n");
}

void UnitTestGradLookupTable() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    GradSquare* grad = GradSquareCreate(&dim, diagLink);
    GradCellSetBlocked(GradCellAt(grad, 2), true);
    MatFloat* table = GradGetLookupTableMinDist(grad);
    if (table == NULL) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradGetLookupTableMinDist failed");
        PBErrCatch(GradErr);
    }
    float check[36] = {
        -1.000000, 1.000000, -1.000000, 1.414214, 2.828427, 2.414214, 1.000000, -1.000000, -1.000000, 1.000000, 2.414214
    }
}

```

```

    };
    for (int i = 0; i < 36; ++i) {
        if (ISEQUALF(table->_val[i], check[i]) == false) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradGetLookupTableMinDist failed");
            PBErrCatch(GradErr);
        }
    }
    GradFree(&grad);
    MatFree(&table);
    printf("UnitTestGradLookupTable OK\n");
}

void UnitTestGradFlood() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    GradSquare* grad = GradSquareCreate(&dim, diagLink);
    VecShort2D sources = VecShortCreateStatic2D();
    VecSet(&sources, 0, 12); VecSet(&sources, 1, 98);
    GradCellSetBlocked(GradCellAt(grad, 92), true);
    GradCellSetBlocked(GradCellAt(grad, 32), true);
    GradRemoveAllLink(grad, 8, true);
    GradRemoveAllLink(grad, 18, true);
    GradRemoveAllLink(grad, 19, true);
    float distMax = 20.0;
    int stepMax = 20;
    GradFlood(grad, (VecShort*)&sources, distMax, stepMax);
    VecShort2D pos = VecShortCreateStatic2D();
    int check[100] = {
        0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1,
        0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1,
        0, 0, -1, 0, 0, 0, 0, 0, 0, 1, 1,
        0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
        0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
        0, 0, -1, 1, 1, 1, 1, 1, 1, 1, 1,
        0, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        -1, 1, -1, 1, 1, 1, 1, 1, 1, 1, 1
    };
    int iCheck = 0;
    do {
        GradCell* cell = GradCellAt(grad, &pos);
        printf("%2d, ", GradCellGetFlood(cell));
        if (VecGet(&pos, 0) == 9) printf("\n");
        if (GradCellGetFlood(cell) != check[iCheck]) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradFlood failed");
            //PBErrCatch(GradErr);
        }
        ++iCheck;
    } while (VecPStep(&pos, &dim));
    int floodArea = GradGetFloodArea(grad, 0);
    if (floodArea != 52) {
        GradErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GradErr->_msg, "GradFloodArea failed");
        PBErrCatch(GradErr);
    }
    floodArea = GradGetFloodArea(grad, 1);
    if (floodArea != 38) {
        GradErr->_type = PBErrTypeUnitTestFailed;
    }
}

```

```

        sprintf(GradErr->_msg, "GradFloodArea failed");
        PBErrCatch(GradErr);
    }

    GradFree(&grad);
    printf("UnitTestGradFlood OK\n");
}

void UnitTestGradGetPath() {
    bool diagLink = true;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    GradSquare* grad = GradSquareCreate(&dim, diagLink);
    GradRemoveAllLink(grad, 51, true);
    GradRemoveAllLink(grad, 52, true);
    GradRemoveAllLink(grad, 53, true);
    GradRemoveAllLink(grad, 54, true);
    GradCellSetBlocked(GradCellAt(grad, 55), true);
    GradCellSetBlocked(GradCellAt(grad, 56), true);
    GradRemoveAllLink(grad, 58, true);
    GradRemoveAllLink(grad, 59, true);
    MatFloat* lookUp = GradGetLookupTableMinDist(grad);
    int from = 12;
    int to = 85;
    VecShort* path = GradGetPath(grad, from, to, lookUp);
    VecPrint(path, stdout); printf("\n");
    int check[9] = {12,23,24,35,46,57,66,75,85};
    for (int i = 9; i--;) {
        if (VecGet(path, i) != check[i]) {
            GradErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GradErr->_msg, "GradGetPath failed");
            PBErrCatch(GradErr);
        }
    }
    VecFree(&path);
    MatFree(&lookUp);
    GradFree(&grad);
    printf("UnitTestGradGetPath OK\n");
}

void UnitTestGrad() {
    UnitTestGradCreateFree();
    UnitTestGradCloneIsSame();
    UnitTestGradGetSet();
    UnitTestGradResetFlagBlocked();
    UnitTestGradEditLinks();
    UnitTestGradLookupTable();
    UnitTestGradFlood();
    UnitTestGradGetPath();

    printf("UnitTestGrad OK\n");
}

void UnitTestAll() {
    UnitTestGradCell();
    UnitTestGrad();
    printf("UnitTestAll OK\n");
}

int main() {
    UnitTestAll();
    // Return success code
}

```

```

    return 0;
}

```

## 6 Unit tests output

```

UnitTestGradCellCreateFree OK
UnitTestGradCellGetSet OK
UnitTestGradCell OK
UnitTestGradCreateFree OK
UnitTestGradCloneIsSame OK
UnitTestGradGetSet OK
UnitTestGradResetFlagBlocked OK
UnitTestGradEditLinks OK
UnitTestGradLookupTable OK
0, 0, 0, 0, 0, 0, 0, 0, -1, -1,
0, 0, 0, 0, 0, 0, 0, 0, -1, -1,
0, 0, 0, 0, 0, 0, 0, 0, 0, -1,
0, 0, -1, 0, 0, 0, 0, 0, 1, 1,
0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
0, 0, -1, 1, 1, 1, 1, 1, 1, 1,
0, -1, 1, 1, 1, 1, 1, 1, 1, 1,
-1, 1, -1, 1, 1, 1, 1, 1, 1, 1,
UnitTestGradFlood OK
<12,13,24,35,46,57,66,75,85>
UnitTestGradGetPath OK
UnitTestGrad OK
UnitTestAll OK

```