Grad

P. Baillehache

March 25, 2018

Contents

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

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(int id, int nbLink, VecShort2D* 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;
  // Fee memory
  free(*that);
  *that = NULL;
}

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

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

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

// Get the appropriate deltas of positions according to the type of the
// Grad ’that’ and the position ’pos’
int* _GradGetDelta(Grad* that, VecShort2D* 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(VecShort2D* dim, GradType type, 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* 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(Grad* that, VecShort2D* pos) {
#if 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);
  }
#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(VecShort2D* dim, bool diagLink) {
#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);
  }
#endif
  // Allocate memory
  GradSquare *that = PBErrMalloc(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;
  // Fee memory
  GradFreeStatic(&((*that)->_grad));
  free(*that);
  *that = NULL;
}

// Create a new GradHexa with dimensions 'dim' and type 'type'
GradHexa* GradHexaCreate(VecShort2D* dim, GradHexaType type) {
#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);
  }
#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(VecShort2D* dim) {
#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);
  }
#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(VecShort2D* dim) {
#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);
  }
#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(VecShort2D* dim) {
#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);
  }
#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(VecShort2D* dim) {
#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);
  }
#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;
  // Fee memory
  GradFreeStatic(&((*that)->_grad));
  free(*that);
  *that = NULL;
}

// Get the GradCell at position 'pos' int the GradHexa 'that'
GradCell* _GradHexaCellAtPos(GradHexa* that, VecShort2D* pos) {
#if 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(Grad* that) {
#if 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(Grad* that, int from, int to, MatFloat* lookup) {
#if 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 onsult 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;
  int _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* that, VecShort* sources, float distMax,
  int stepMax) {
#if 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
  int nbSrc = VecGetDim(sources);
  // Declare a set of GradFloodPod
  GSet set = GSetCreateStatic();
  // For each sources
  for (int 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 = GSetGetElem(&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);
  }
}

void _GradFloodOld(Grad* that, VecShort* sources, float distMax,
  int stepMax) {
#if 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
  int nbSrc = VecGetDim(sources);
  // Get the area of the Grad
  int area = GradGetArea(that);
  // Declare two arrays to resolve the conflicting cells
  int* firstSrc = PBErrMalloc(GradErr, sizeof(int) * area);
  bool* flagConflict = PBErrMalloc(GradErr, sizeof(char) * area);
  for (int i = area; i--;) {
    firstSrc[i] = -1;
    flagConflict[i] = false;
  }
  // Declare two sets of GradFloodPod
  GSet setIn = GSetCreateStatic();
  GSet setOut = GSetCreateStatic();
  // For each sources
  for (int 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));
    GSetAddSort(&setIn, pod, 0.0);
    // Initialise the flood value of sources
    GradCellSetFlood(GradCellAt(that,
      VecGet(sources, iSource)), iSource);
  }
  // Declare a variable to memorize the nb of step
  int nbStep = 0;
  // Loop on nb of step
  while (GSetNbElem(&setIn) > 0 && (stepMax < 0 || nbStep < stepMax)) {
    // For each cell in the setIn
    while (GSetNbElem(&setIn) > 0) {
      // Get the distance up to this cell
      float dist = GSetGetElem(&setIn, 0)->_sortVal;
      // Pop the cell
      GradFloodPod* pod = GSetPop(&setIn);
      // 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) &&
            !(flagConflict[toCell])) {
            // If this cell has not been reached by another source yet
            if (firstSrc[toCell] == -1 ||
              firstSrc[toCell] == pod->_src) {
              // Add a new pod to the GSet ’setOut’
              GradFloodPod* npod =
                PBErrMalloc(GradErr, sizeof(GradFloodPod));
              npod->_src = pod->_src;
              npod->_cell = cellTo;
              GSetAddSort(&setOut, npod, d);
              firstSrc[toCell] = pod->_src;
            // Else it has been reached by another source
            } else {
              // Mark it has a conflicting cell
              flagConflict[toCell] = true;
            }
          }
        }
      }
    }
    // Free memory used by the GradFloodPod
    free(pod);
  }
  // Now the GSet ’setOut’ contains all the cell we try to reach.
  // Add them back to ’setIn’ while resolving conflict of
  // simultaneous arrival
  while (GSetNbElem(&setOut) > 0) {
    // Get the distance up to this cell
    float dist = GSetGetElem(&setOut, 0)->_sortVal;
    // Pop the cell
    GradFloodPod* pod = GSetPop(&setOut);
    int toCell = GradCellGetId(pod->_cell);
    // If it hasn’t been reached already and it’s not a
    // conflicting cell
    if (GradCellGetFlood(pod->_cell) == -1 &&
      !(flagConflict[toCell])) {
      // Flood it
      GradCellSetFlood(pod->_cell, pod->_src);
      // Add it to ’setIn’
      GSetAddSort(&setIn, pod, dist);
      // Else it has been reached
    } else {
      // Free memory used by the pod
      free(pod);
    }
  }
  // Increment the number of step
  ++nbStep;
}
// Free memory
while (GSetNbElem(&setIn) > 0) {
  GradFloodPod* pod = GSetPop(&setIn);
  free(pod);
}
```

```
  while (GSetNbElem(&setOut) > 0) {
    // Should never pass here
    GradFloodPod* pod = GSetPop(&setOut);
    free(pod);
  }
  free(flagConflict);
  free(firstSrc);
}

// Get the number of flooded cells from 'iSource'-th source in the Grad
// 'that'
int _GradGetFloodArea(Grad* that, int iSource) {
#if 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 serached 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(GradSquare* that) {
#if 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(GradHexa* 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(Grad* that, Grad* 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 symetric link is removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveLinkIndex(Grad* 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
  // 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 symetric link is removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveDirIndex(Grad* that, int fromCell, int dir,
  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 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 symetric links are removed too
// (only if the link from 'fromCell' exists)
void _GradRemoveAllLinkIndex(Grad* that, int fromCell,
  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* 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* that, int fromCell, int dir,
  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* that, int fromCell,
  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-INLINE.C ================

// ------------- GradCell

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

// Get the user data of the GradCell 'that'
#if BUILDMODE != 0
inline
#endif
void* GradCellData(GradCell* 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
inline
#endif
void GradCellSetData(GradCell* that, void* 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
inline
#endif
VecShort2D* GradCellPos(GradCell* 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
inline
#endif
```

```
int GradCellGetId(GradCell* 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
inline
#endif
int GradCellGetLink(GradCell* that, 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
inline
#endif
void GradCellSetLink(GradCell* that, int iLink, 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
inline
#endif
```

```
int GradCellGetNbLink(GradCell* 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
inline
#endif
float GradCellLinkVal(GradCell* that, 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
inline
#endif
void GradCellSetLinkVal(GradCell* that, int iLink, 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
inline
#endif
int GradCellGetFlood(GradCell* that) {
#if 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'
#if BUILDMODE != 0
inline
#endif
void GradCellSetFlood(GradCell* that, int iSource) {
#if 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'
#if BUILDMODE != 0
inline
#endif
bool GradCellIsBlocked(GradCell* that) {
#if 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'
#if BUILDMODE != 0
inline
#endif
void GradCellSetBlocked(GradCell* that, bool flag) {
#if 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'
#if BUILDMODE != 0
inline
#endif
GradCell* _GradCellAtIndex(Grad* that, 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
inline
#endif
GradCell* _GradCellAtPos(Grad* that, VecShort2D* 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
inline
#endif
GradType _GradGetType(Grad* 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
inline
#endif
GradHexaType GradHexaGetType(GradHexa* 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
inline
#endif
int _GradGetArea(Grad* 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
inline
#endif
VecShort2D* _GradDim(Grad* 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
inline
#endif
bool _GradIsPosInside(Grad* that, VecShort2D* 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
inline
#endif
void _GradResetFlagBlocked(Grad* 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
inline
#endif
bool GradSquareHasDiagonalLink(GradSquare* 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 symetric link is removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
inline
#endif
void _GradRemoveLinkPos(Grad* that, VecShort2D* fromCell,
  VecShort2D* toCell, 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 symetric link is removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
inline
#endif
void _GradRemoveDirPos(Grad* that, VecShort2D* fromCell, int dir,
  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 symetric links are removed too
// (only if the link from 'fromCell' exists)
#if BUILDMODE != 0
inline
#endif
void _GradRemoveAllLinkPos(Grad* that, VecShort2D* fromCell,
  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
inline
#endif
void _GradAddLinkPos(Grad* that, VecShort2D* fromCell,
  VecShort2D* toCell, 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
inline
#endif
void _GradAddDirPos(Grad* that, VecShort2D* fromCell, int dir,
  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
inline
#endif
void _GradAddAllLinkPos(Grad* that, VecShort2D* fromCell,
  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

```
#directory
PBERRDIR=../PBErr
GSETDIR=../GSet
PBMATHDIR=../PBMath

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

include $(PBERRDIR)/Makefile.inc

INCPATH=-I./ -I$(PBERRDIR)/ -I$(GSETDIR)/ -I$(PBMATHDIR)/
BUILDOPTIONS=$(BUILDPARAM) $(INCPATH)

# compiler
COMPILER=gcc

#rules
all : main

main: main.o pberr.o grad.o gset.o pbmath.o Makefile
$(COMPILER) main.o pberr.o grad.o gset.o pbmath.o $(LINKOPTIONS) -o main

main.o : main.c $(PBERRDIR)/pberr.h grad.h grad-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c main.c

grad.o : grad.c grad.h $(GSETDIR)/gset.h $(PBMATHDIR)/pbmath.h grad-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c grad.c

gset.o : $(GSETDIR)/gset.c $(GSETDIR)/gset-inline.c $(GSETDIR)/gset.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(GSETDIR)/gset.c

pbmath.o : $(PBMATHDIR)/pbmath.c $(PBMATHDIR)/pbmath-inline.c $(PBMATHDIR)/pbmath.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBMATHDIR)/pbmath.c

pberr.o : $(PBERRDIR)/pberr.c $(PBERRDIR)/pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBERRDIR)/pberr.c

clean :
rm -rf *.o main
```

```
valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main

unitTest :
main > unitTest.txt; diff unitTest.txt unitTestRef.txt
```

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, "GradHexaCreateOddRfailed");
    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, "GradHexaCreateEvenRfailed");
    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, -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, 0, 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
    };
  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] = {12,13,24,35,46,57,66,75,85};
  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,  0,  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
```