Grid Functions for Overture
User Guide, Version 1.0000000000000000000000000000000003

Bill Henshaw [1] Centre for Applied Scientific Computing

Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
http://www.llnl.gov/casc/people/henshaw
http://www.llnl.gov/casc/Overture May 20, 2011 UCRL-MA-132231

Abstract: We describe the grids and grid functions that can be used with Overture. The grid functions are based on the fabulous A++ array class library.

Contents

[1] This work was partially supported by grant N00014-95-F-0067 from the Office of Naval Research

List of Figures

1 Introduction

Figure 1 gives an overview of the classes that make up Overture. In this document we will discuss grids and grid functions.

Documentation can be found on the Overture home page, `http://www.llnl.gov/casc/Overture`, and includes the following documents that may be of interest

- A++ Quick Reference Card : `A++P++/DOCS/Quick_Reference_Card.tex`
- A primer for Overture[9].
- Grid and grid function documentation[3].
- Finite difference operators and boundary conditions[2].
- Finite volume operators [1].
- Mapping class documentation [4].
- Show file documentation [7].
- Interactive plotting[8].
- Oges "Equation Solver" documentation [6].
- Interactive grid generation documentation [5].
- The other stuff documentation[10].
- The OverBlown Navier-Stokes flow solver [12][11].

2 Grids

Grids and collections of grids are the fundamental objects that PDE solvers have to deal with. The grid classes that are described here are designed so that they can be used by a a wide variety of PDE solvers including

- solvers written for a single rectangular grid
- solvers written with AMR++ to perform adaptive computations
- solvers written with Overture for overlapping grids
- solvers that combine AMR++ and Overture

Here is a list of the various grid classes. The Generic grids are not really used for anything except to derive from.

- **GenericGrid** - Derive all grids from this class. This can be the base class for both structured and unstructured grids.
- **MappedGrid : GenericGrid** - Logically rectangular grid with a mapping. This class supports the creation of a variety of geometry arrays such as the `vertexCoord`, `vertexDerivatives` and also holds boundary condition information etc.
- **GenericGridGridCollection** - A collection of GenericGrid's. Base class for all collections of grids.
- **GridCollection : GenericGridCollection** - A collection of MappedGrid's. Also contains connection (interpolation) information and `mask` arrays and parent/child/sibling information for adaptive grids. May be a valid "overlapping" grid (grids cover the entire domain) or may just be a subset of the grids contained in a valid "overlapping" grid.
- **CompositeGrid : GridCollection** - This is a valid "overlapping" grid, grids cover the entire computational domain.

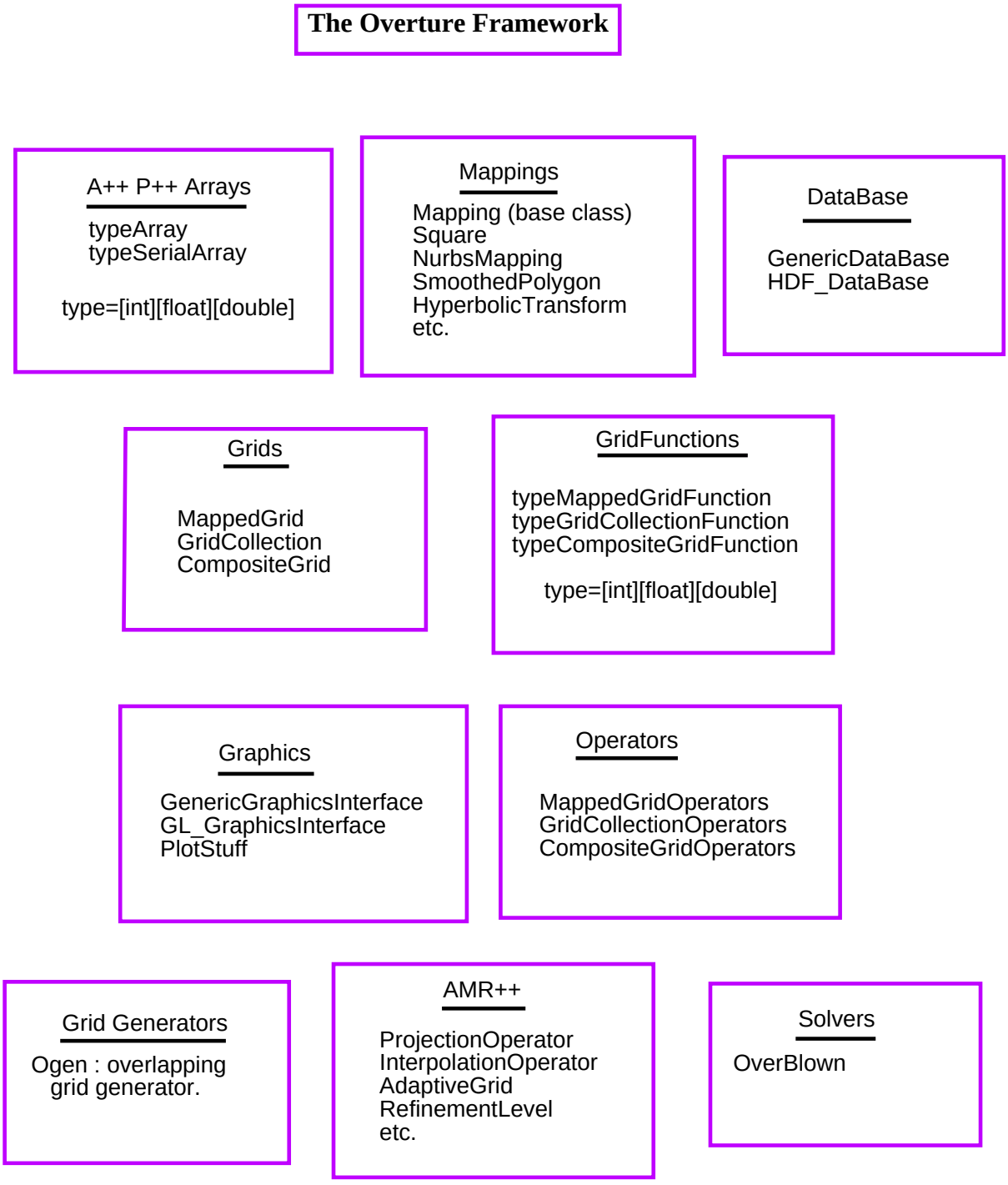

Figure 1: An overview of the Overture classes

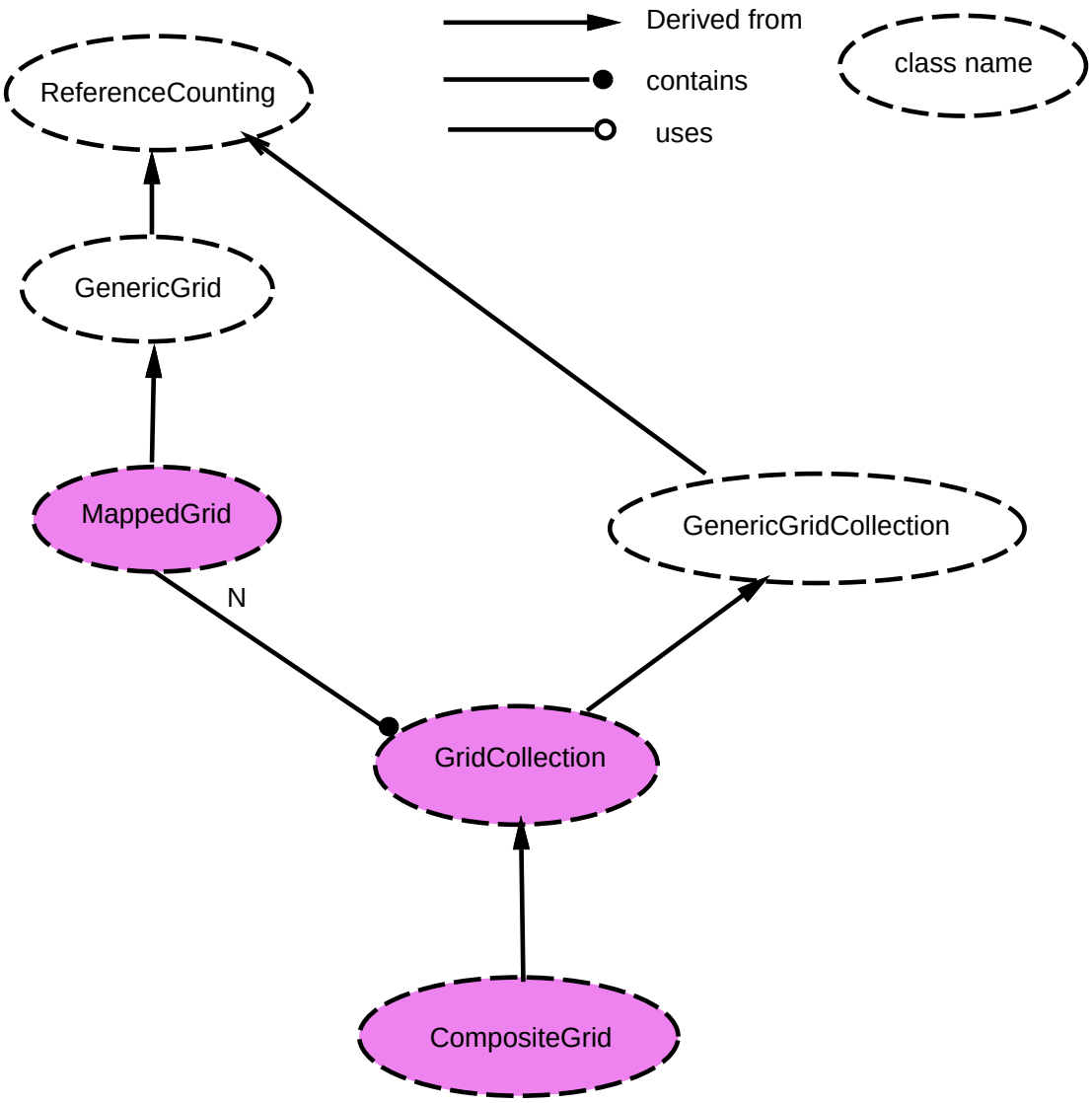

Figure 2: Class diagram for grid classes

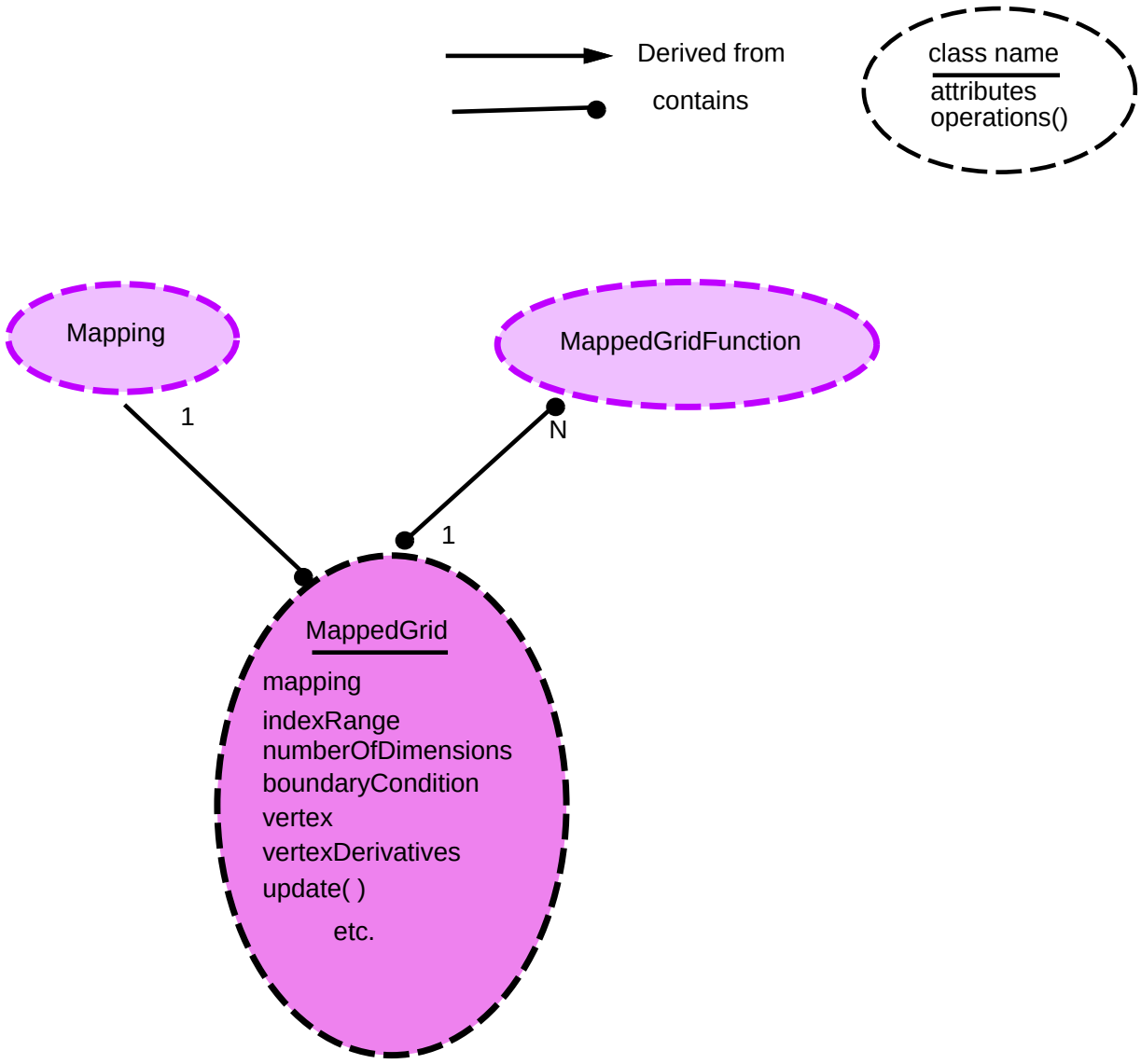

Figure 3: Class diagram for a MappedGrid

2.1 MappedGrid

***** This documentation is out of date *****

The mapped grid is a logically rectangular grid with a mapping function.
Define the Ranges $R1, R2, R3$ to define all points on a component grid:

```
const int Start=0, End=1, axis1=0, axis2=1, axis3=2;
CompositeGrid cg;
MappedGrid & mg = cg[grid];
Range R1(mg.dimension(Start,axis1),mg.dimension(End,axis1));
Range R2(mg.dimension(Start,axis2),mg.dimension(End,axis2));
Range R3(mg.dimension(Start,axis3),mg.dimension(End,axis3));
Range ND(0,cg.numberOfDimensions);
```

Recall that we denote the axes of the unit square (or cube) by r_1, r_2, (and r_3). Some arrays such as the `boundaryCondition` array, associate values with each side of a grid. The sides of the grid can be denoted by $r_i = 0$ or $r_i = 1$. These arrays are dimensioned as `boundaryCondition(0:1,0:2)` with

$$boundaryCondition(side, axis) = \text{ value for } r_{axis} = side \ \ , \ side = 0, 1 \ \ , \ axis = 0, 1, 2 \tag{1}$$

Some arrays, such as the array of vertex coordinates, come in three flavours, `vertex`, `vertex2D` and `vertex1D`. The first is dimensioned `vertex(R1,R2,R3,ND)` and thus looks like an array for a three dimensional grid. When the grid is two-dimensional the Range `R3` will only have 1 point. This array is useful when writing a code that will work in both 3D and 2D. The array `vertex2D(R1,R2,ND)` is only available when the grid is two-dimensional.

- **IntArray boundaryCondition(0:1,0:2)** Boundary condition flags, positive for a real boundary, negative for a periodic boundary and zero for an interpolation boundary.
- **IntArray boundaryDiscretizationWidth(0:2)** Width of the boundary condition stencil.
- **realMappedGridFunction center(R1,R2,R3,ND)** Coordinates of discretization centres.
- **realMappedGridFunction center2D(R1,R2,ND)** Coordinates of discretization centers, for a two-dimensional grid.
- **realMappedGridFunction center1D(R1,ND)** Coordinates of discretization centers, for a one-dimensional grid.
- **realMappedGridFunction centerDerivative(R1,R2,R3,ND,ND)** Derivative of the mapping at the discretization centers.
- **realMappedGridFunction centerDerivative2D(R1,R2,ND,ND)** Derivative at the discretization centers, for a two-dimensional grid.
- **realMappedGridFunction centerDerivative1D(R1,ND,ND)**
- **FloatMappedGridFunction centerJacobian(R1,R2,R3)** Determinant of centerDerivative.
- **IntArray dimension(0:1,0:2)** Dimensions of grid arrays – actual size of the A++ arrays, including ghost-points.
- **IntArray discretizationWidth(0:2)** Interior discretization stencil width (default=3)
- **IntArray gridIndexRange(0:1,0:2)** Index range of gridpoints, excluding ghost points.
- **realArray gridSpacing(0:2)** Grid spacing in the unit square, equal to 1 over the number of grid cells.
- **IntArray indexRange(0:1,0:2)** Index range of computational points, excluding ghostpoints and excluding periodic grid lines on the "End".
- **LogicalR isAllCellCentered** Grid is cell-centred in all directions (variable name misspelled for historial reasons, circa 1776)
- **LogicalR isAllVertexCentered** Grid is vertex-centred in all directions
- **LogicalArray isCellCentered(0:2)** Is this grid cell-centred in each direction.

- **IntArray isPeriodic(0:2)** Grid periodicity, equal one if notPeriodic, derivativePeriodic or functionPeriodic.
- **realMappedGridFunction inverseVertexDerivative(R1,R2,R3,ND,ND)** Inverse derivative of the mapping at the vertices. `inverseVertexDerivative(i1,i2,i3,axis,dir)` is the partial derivative of r_{axis} with respesct to x_{dir}.
- **realMappedGridFunction inverseVertexDerivative2D(R1,R2,ND,ND)** Inverse derivative at the vertices, for a two-dimensional grid.
- **realMappedGridFunction inverseVertexDerivative1D(R1,ND,ND)** Inverse derivative at the vertices, for a one-dimensional grid.
- **realMappedGridFunction inverseCenterDerivative(R1,R2,R3,ND,ND)** Inverse derivative at the discretization centers.
- **realMappedGridFunction inverseCenterDerivative2D(R1,R2,ND,ND)** Inverse derivative at the discretization centers, for a two-dimensional grid.
- **realMappedGridFunction inverseCenterDerivative1D(R1,ND,ND)** Inverse derivative at the discretization centers, for a one-dimensional grid.
- **IntMappedGridFunction mask(R1,R2,R3)** mask array that indicates which points are used and not used.
- **MappingRC mapping** Grid mapping (MappingRC is a reference counted Mapping which behaves like the Mapping class)
- **FloatArray minimumEdgeLength(0:2)** Minimum grid cell-edge length.
- **FloatArray maximumEdgeLength(0:2)** Maximum grid cell-edge length.
- **IntR numberOfDimensions** Number of space dimensions, an `IntR` is basically an `int` (used for reference counting).
- **IntArray numberOfGhostPoints(0:1,0:2)** number of ghost points on each side.
- **realMappedGridFunction vertex(R1,R2,R3,ND)** Vertex coordinates.
- **realMappedGridFunction vertex2D(R1,R2,ND)** Vertex coordinates, for a two-dimensional grid.
- **realMappedGridFunction vertex1D(R1,ND)** Vertex coordinates, for a one-dimensional grid.
- **FloatArray vertexBoundaryNormal[3][2]** Outward normal vectors at the vertices on each boundary. These arrays are dimensioned so that they lie on their respective boundary:
  - vertexBoundaryNormal[0][0](R1.getBase():R1.getBase(),R2,R3,ND),
  - vertexBoundaryNormal[0][1](R1.getBound():R1.getBound(),R2,R3,ND),
  - vertexBoundaryNormal[1][0](R1,R2.getBase():R2.getBase(),R3,ND),
  - vertexBoundaryNormal[1][1](R1,R2.getBound():R2.getBound(),R3,ND),
  - etc.
- **FloatArray centerBoundaryNormal[3][2]** Outward normal vectors at the centers on each boundary.
- **realMappedGridFunction vertexDerivative(R1,R2,R3,ND,ND)** Derivative of the mapping at the vertices, `vertexDeriavtive(i1,i2,i3,axis,dir)` is the partial derivative of x_{axis} with respesct to r_{dir}.
- **realMappedGridFunction vertexDerivative2D(R1,R2,ND,ND)** Derivative of the mapping at the vertices, for a two-dimensional grid.
- **realMappedGridFunction vertexDerivative1D(R1,ND,ND)** Derivative of the mapping at the vertices, for a one-dimensional grid.
- **FloatMappedGridFunction vertexJacobian(R1,R2,R3)** Determinant of vertexDerivative.

One may specify (or change) which arrays are to exist in the `MappedGrid` by calling the `update` function with an integer bit-flag. The values of the bit flag are determined from the following enumerator

```
enum {
  USEmask                    = USEgenericGrid             << 1,
  USEinverseVertexDerivative = USEmask                    << 1,
  USEinverseCenterDerivative = USEinverseVertexDerivative << 1,
  USEvertex                  = USEinverseCenterDerivative << 1,
  USEcenter                  = USEvertex                  << 1,
  USEvertexDerivative        = USEcenter                  << 1,
  USEcenterDerivative        = USEvertexDerivative        << 1,
  USEfaceNormal              = USEcenterDerivative        << 1,
  USEvertexJacobian          = USEfaceNormal              << 1,
  USEcenterJacobian          = USEvertexJacobian          << 1,
  USEvertexBoundaryNormal    = USEcenterJacobian          << 1,
  USEcenterBoundaryNormal    = USEvertexBoundaryNormal    << 1,
  USEmappedGrid              = USEcenterBoundaryNormal // Do not use.
};
```

- **MappedGrid(const String & file, const String & name)** Constructor from database file and name.
- **MappedGrid(Mapping & mapping)** Constructor from a mapping.
- **void updateReferences()** Set references to reference-counted data.
- **void update(const Int what = USEtheUsualSuspects)** Update the grid.

For further details consult the documentation sitting in the chair in Geoff's office.

2.2 GridCollection

See the description of a CompositeGrid.

2.3 CompositeGrid

***** This documentation is out of date *****

A CompositeGrid is a collection of MappedGrid's along with the information needed for interpolating between component grids.

Define the Ranges

```
const int Start=0, End=1, axis1=0, axis2=1, axis3=2;
CompositeGrid cg;
MappedGrid & mg = cg[grid];
Range R1(mg.dimension(Start,axis1),mg.dimension(End,axis1));
Range R2(mg.dimension(Start,axis2),mg.dimension(End,axis2));
Range R3(mg.dimension(Start,axis3),mg.dimension(End,axis3));
Range ND(0,cg.numberOfDimensions());
Range NG(0,cg.numberOfComponentGrids());
Range MG(0,cg.numberOfMultigridLevels());

Range NI(0,cg.numberOfInterpolationPoints(grid));
```

- **IntR numberOfComponentGrids** Number of component grids (MappedGrid's).
- **IntR numberOfDimensions** Number of space dimensions.
- **IntArray numberOfInterpolationPoints(NG)** The number of interpolation points on each component grid.
- **LogicalR interpolationIsAllExplicit**
- **LogicalArray interpolationIsImplicit(NG,NG)**
- **IntArray interpolationWidth(3,NG,NG)** The width of the interpolation stencil (direction, toGrid, fromGrid).
- **realArray interpolationOverlap(3,NG,NG)** The minimum overlap for interpolation (direction, toGrid, fromGrid).

- **ListOfReferenceCountedObjects<realArray> interpolationCoordinates[NG](NI,ND)** Coordinates of interpolation point on component grid "grid" are `interpolationCoordinates[grid](n,axis)` for $0 \le n \le$ `numberOfInterpolationPoints(grid)`.
- **ListOfReferenceCountedObjects<IntArray> interpoleeGrid[NG](NI)** Index of the "interpolee grid", i.e. this is the index of the grid from which we interpolate.
- **ListOfReferenceCountedObjects<IntArray> interpoleeLocation[NG](NI,ND)** Location of interpolation stencil on the interpolee grid, this is the index of the lower left corner of the stencil.
- **ListOfReferenceCountedObjects<IntArray> interpolationPoint[NG](NI,ND)** Indices of interpolation point.
- **ListOfReferenceCountedObjects<realArray> interpolationCondition[NG](NI)** Interpolation condition number.
- **IntGridCollectionFunction mask[NG](R1,R2,R3)** Flag array, positive for discretization point, negative for interpolation point, zero for unused point.
- **ListOfReferenceCountedObjects<MappedGrid> grid[NG]** Here is the list of MappedGrid's.

Here are variables related to multigrid levels

- **IntR numberOfMultigridLevels**
- **IntArray coarseToFineWidth(0:2,NG,MG)** Prolongation stencil width
- **IntArray coarseToFineIsImplicit(NG,MG)** Prolongation is always implicit.
- **IntArray fineToCoarseWidth(0:2,NG,MG)** Restriction stencil width
- **IntArray fineToCoarseIsImplicit(NG,MG)** Restriction is always implicit.
- **IntArray fineToCoarseFactor(0:2,NG,MG)** Ratio of this to coarser level
- **ListOfReferenceCountedObjects<CompositeGrid> compositeGrid**

3 GridFunctions

Almost all applications that use grids will also need to use grid functions. For example, a PDE solver will need to store the values of the solution (velocity, pressure, density, ...). These values are defined at each point on the grid. A grid function will thus hold one or more values for each point on the grid. The grid function will be associated with a grid and thus will know how to dimension itself. Grid functions also come with a collection of operations. These operations include the standard arithmetic operators as well as more sophisticated operations such as differentiation and interpolation.

Here is a list of the available grid functions. Basically each type of grid has a grid function associated with it. The keyword "`type`" can be any of `double`, `float` or `int`.

- **typeMappedGridFunction : typeArray** - this grid function is associated with a `MappedGrid` and is derived from an A++ array.
- **typeGridCollectionFunction** - this grid function is a collection of `typeMappedGridFunction`'s and is associated with a `GridCollection`.
- **typeCompositeFunction** - this grid function is a collection of `typeMappedGridFunction`'s and is associated with a `CompositeGrid`.

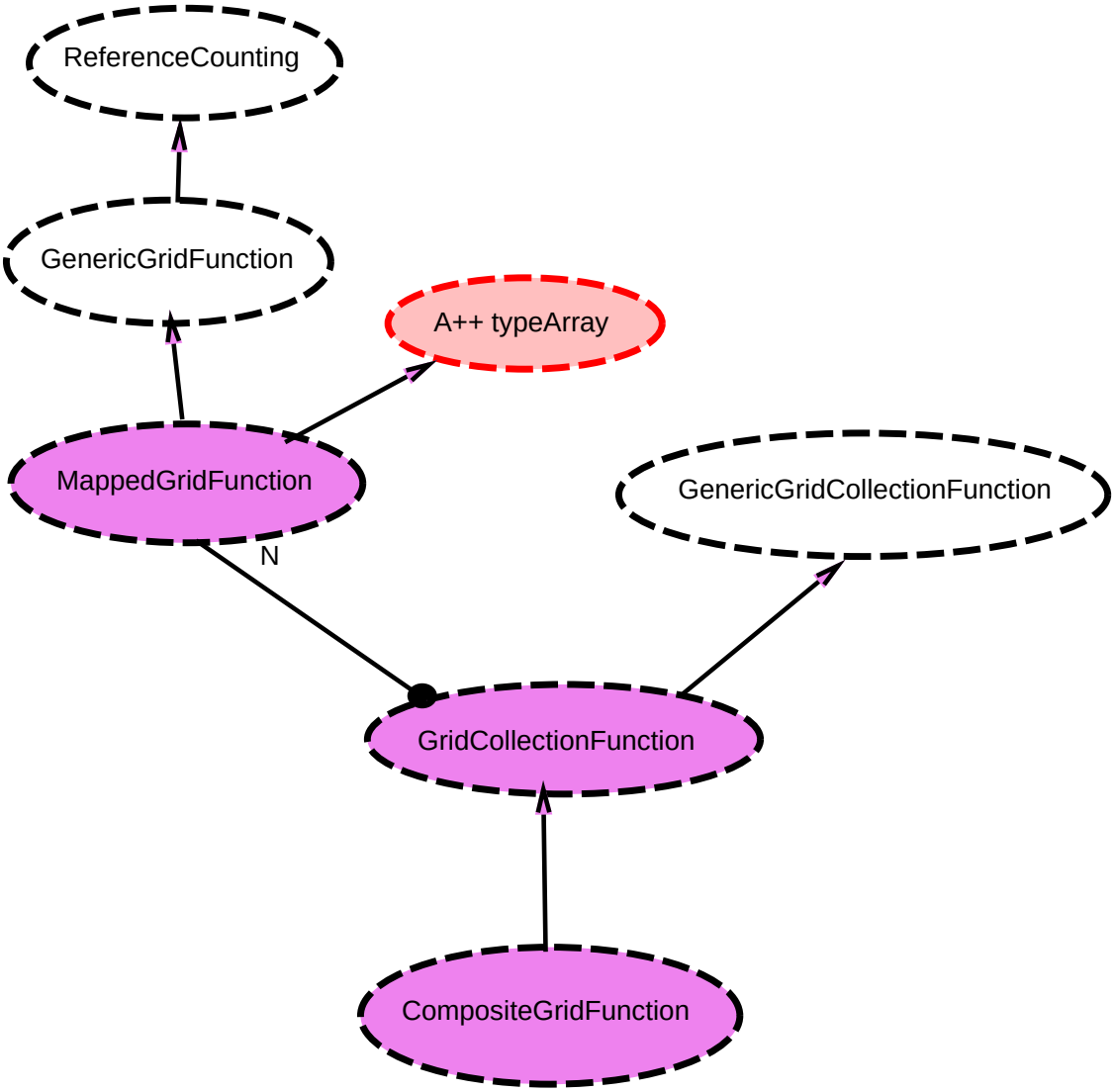

Figure 4: Class diagram for grid function classes

3.1 MappedGridFunction

This is a grid function that can be used with a MappedGrid. The main purpose of the `MappedGridFunction` is to act like a "smart" A++ array. (Not that A++ arrays are not already pretty smart). This class is derived from an A++ array so all A++ operations are defined. Since it is associated (has a pointer to) a `MappedGrid`, this grid function knows how to dimension itself and how to update periodic edges when the grid is periodic. It also knows how to update itself when the MappedGrid is changed (perhaps the number of points on the grid is increased). Here the update only involves redimensioning; not assigning values to the new grid function.

The types of MappedGridFunctions are `floatMappedGridFunction`, `doubleMappedGridFunction`, and `int-MappedGridFunction`.

This is a reference counted class so that there is no need to keep a pointer to a grid function. Use the `reference` member function to make one grid function reference another.

A grid function takes some of it's dimensions from the MappedGrid that it is associated with. The index positions in a MappedGridFunction corresponding to the dimensions of the grid are called the **coordinate** positions. A grid function can also have one or more **component** positions which indicate how many values are stored at each grid point. Consider an example of a MappedGrid in 2D with dimensions `(-1:11,-1:11)` (these values are stored in the MappedGrid in the `dimension` array). A MappedGridFunction defined on this MappedGrid could have dimensions `u(-1:11,-1:11,0:2)`. This grid function `u` has 3 components `(0:2)`, and `u.positionOfCoordinate(0)=0`, `u.positionOfCoordinate(1)=1` and `u.positionOfComponent(0)=2`. A MappedGridFunction could also be created as `u(0:1,-1:11,-1:11)` in which case `u.positionOfCoordinate(0)=1`, `u.positionOfCoordinate(1)=2` and `u.positionOfComponent(0)=0`.

In the description of the member functions that follow, `MappedGridFunction` will stand for one of `floatMappedGridFunction`, `doubleMappedGridFunction`, or `intMappedGridFunction`. In addition, any references to `double` will change to `float` or `int`.

3.1.1 Public enumerators

Here are the public enumerators:

edgeGridFunctionValues: Use these values to create special Range objects to define grid functions on boundaries.

```
enum edgeGridFunctionValues     // these enums are used to declare grid functions defined on faces or edges
{
  startingGridIndex   =-(INT_MAX/2),              // choose a big negative number assuming that
  biggerNegativeNumber=startingGridIndex/2,       // no grid will ever have dimensions in this range
  endingGridIndex     =biggerNegativeNumber/2,
  bigNegativeNumber   =endingGridIndex/2
};
```

stencilTypes: Here are somes standard stencil types for coefficient matrices.

```
enum stencilTypes               // if the grid function holds a coefficient matrix
{                               // these are the types of stencil that it may contain
  standardStencil,              // 3x3 int 2D or 3x3x3 in 3D (if 2nd order accuracy)
  starStencil,                  // 5 point star in 2D or 7pt star in 3D (if 2nd order accuracy)
  generalStencil
};
```

updateReturnValue: The value returned from the `updateToMatchGrid` and `updateToMatchGridFunction` is a mask formed by a bitwise `or` of the following values:

```
enum updateReturnValue  // the return value from updateToMatchGrid is a mask of the following values
{
  updateNoChange          = 0, // no changes made
  updateReshaped          = 1, // grid function was reshaped
  updateResized           = 2, // grid function was resized
  updateComponentsChanged = 4  // component dimensions may have changed (but grid was not resized or reshaped)
};
```

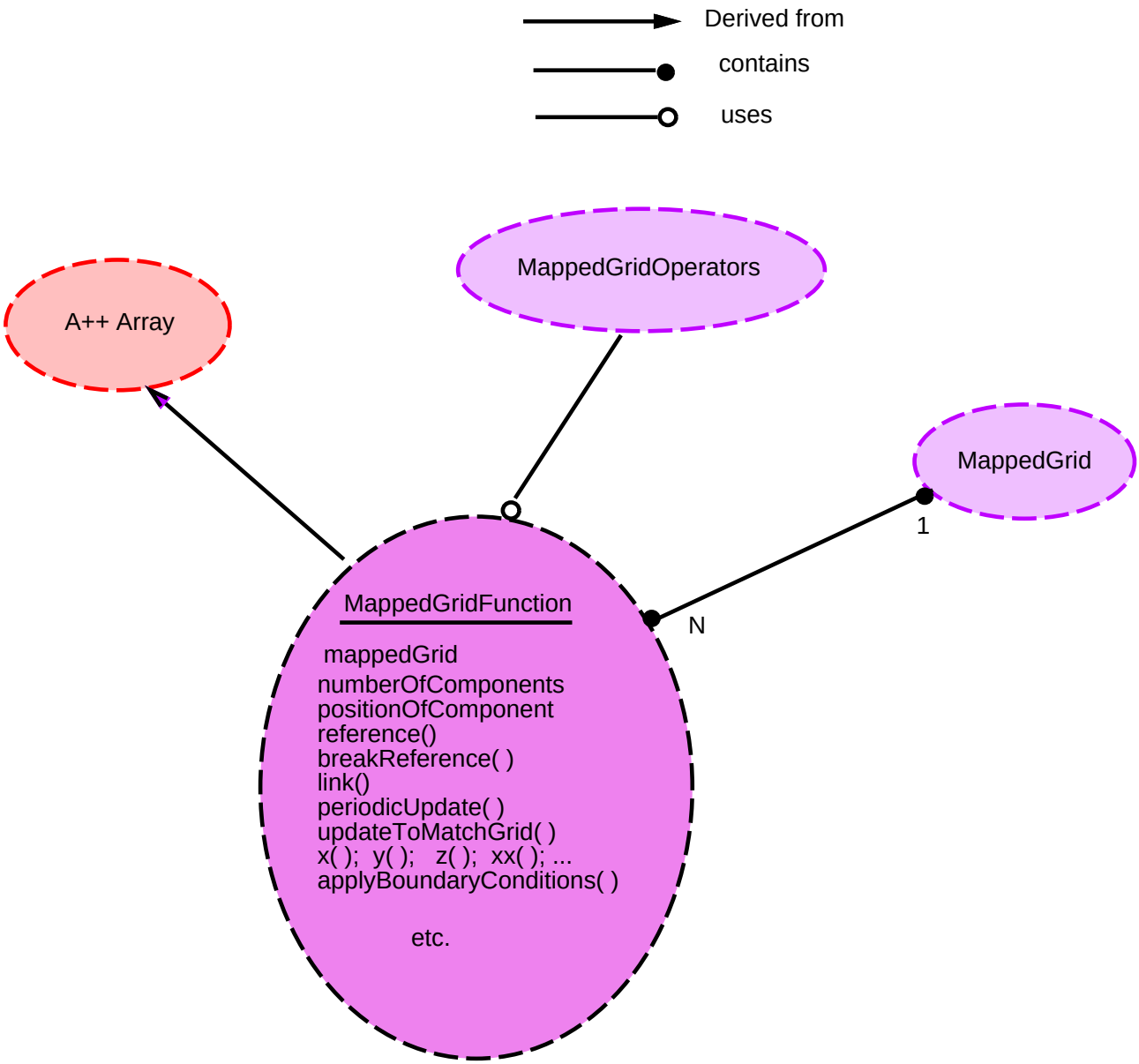

Figure 5: Class diagram for a MappedGridFunction

3.1.2 Constructors

MappedGridFunction ()

Description: Default constructor

Author: WDH

3.1.3 Constructors

MappedGridFunction(MappedGrid & grid0)

Description: Create a grid function and associate with a MappedGrid. The grid function will be a "scalar" as in the declaration:

```
Range all;
MappedGrid mg(...);
MappedGridFunction u(mg,all,all,all);
```

grid0 (input): grid to associate this grid function with.

Author: WDH

MappedGridFunction(MappedGrid & grid0,
const Range & R0,
const Range & R1 =nullRange,
const Range & R2 =nullRange,
const Range & R3 =nullRange,
const Range & R4 =nullRange,
const Range & R5 =nullRange,
const Range & R6 =nullRange,
const Range & R7 =nullRange)

Description: This constructor takes ranges, the first 3 "nullRange" values are taken to be the coordinate directions in the grid function. Each grid function is dimensioned according to the dimensions found with the `MappedGrid`, using the `dimension` values. Grid functions can have up to 8 dimensions, the index positions not used by the coordinate dimensions can be used to store different components. For example, a *vector* grid functions would use 1 index position for components while a *matrix* grid functions would use two index positions for components.

grid0 (input): MappedGrid to associate this grid function with.

R0, R1, R2, ... (input): Ranges to determine the shape and size of the grid function. An int can also be used instead of a Range.

Examples: Here are some examples

```
//  R1 = range of first dimension of the grid array
//  R2 = range of second dimension of the grid array
//  R3 = range of third dimension of the grid array

MappedGrid mg(...);

Range R1(mg.dimension()(Start,axis1),mg.dimension()(End,axis1));
Range R2(mg.dimension()(Start,axis2),mg.dimension()(End,axis2));
Range R3(mg.dimension()(Start,axis3),mg.dimension()(End,axis3));

Range all;    // null Range is used to specify where the coordinates are

MappedGridFunction u(mg);                          //  --> u(R1,R2,R3);

MappedGridFunction u(mg,all,all,all,1);            //  --> u(R1,R2,R3,0:1);
MappedGridFunction u(mg,all,all,Range(1,1));       //  --> u(R1,R2,1:1,R3);
```

```
MappedGridFunction u(mg,2,all);                    //  --> u(0:2,R1,R2,R3);
MappedGridFunction u(mg,Range(0,2),all,all,all);   //  --> u(0:2,R1,R2,R3);
MappedGridFunction u(mg,all,Range(3,3),all,all);   //  --> u(R1,3:3,R2,R3);
```

Author: WDH

MappedGridFunction(MappedGrid & grid0,
const GridFunctionParameters::GridFunctionType & type,
const Range & component0 =nullRange,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: This constructor is used to create a grid function of some standard type. The standard types are defined in the GridFunctionParameters::GridFunctionType enum,

- vertexCentered : grid function is vertex centred
- cellCentered : grid function is cell centred
- faceCenteredAll : grid function components are face centred in all directions
- faceCenteredAxis1 : grid function is face centred along axis1
- faceCenteredAxis2 : grid function is face centred along axis2
- faceCenteredAxis3 : grid function is face centred along axis3
- general : means same as vertexCentered when used in this constructor

grid0 (input): Use this MappedGrid

type (input): Make this type of grid function.

component0, component1,... (input): supply a Range for each component.

Examples: Here are some examples:

```
MappedGrid mg(...);
realMappedGridFunction u(mg,GridFunctionParameters::vertexCentered,2);   // u(mg,all,all,all,2);
realMappedGridFunction u(mg,GridFunctionParameters::cellCentered,2,3);   // u(mg,all,all,all,2,3);
realMappedGridFunction u(mg,GridFunctionParameters::faceCenteredAll,2);  // u(mg,all,all,all,2,faceRange);
realMappedGridFunction u(mg,GridFunctionParameters::faceCenteredAll,3,2);// u(mg,all,all,all,3,2,faceRange);
```

Remarks: A face centered grid function along axis=axis0 is vertex centered along axis0 and cell centered along the other axes.

Author: WDH

MappedGridFunction(const MappedGridFunction & cgf,
const CopyType copyType =DEEP)

Description: Copy constructor, deep copy by default

Notes: This routine was changes 011103 to call the underlying A++ copy constructor. This was necessary for functions that return a realMGF by value. On some compilers, like the Sun CC 4.2 These return by value temporaries would not be deleted immediately but rather stay around until the end of scope – this could result in a low of extra storage being required. See the test code otherStuff/memoryUsage.C for examples.

Author: WDH

3.1.4 applyBoundaryConditions

void
applyBoundaryConditions(const real & time = 0.)

Description: Apply the boundary conditions to this grid function. This routine just calls the function of the same name in the MappedGridOperators.

3.1.5 assignBoundaryConditionCoefficients

void
assignBoundaryConditionCoefficients(const real & time = 0.)

Description: Fill in the coefficients of the boundary conditions into this grid function. This routine just calls the function of the same name in the MappedGridOperators.

3.1.6 assignBoundaryCondition

void
applyBoundaryCondition(const Index & Components,
const BCTypes::BCNames & bcType = BCTypes::dirichlet,
const int & bc = BCTypes::allBoundaries,
const real & forcing =0.,
const real & time =0.,
const BoundaryConditionParameters &
bcParameters = Overture::defaultBoundaryConditionParameters(),
const int & grid_ =0)

void
applyBoundaryCondition(const Index & Components,
const BCTypes::BCNames & bcType,
const int & bc,
const RealArray & forcing,
const real & time =0.,
const BoundaryConditionParameters &
bcParameters = Overture::defaultBoundaryConditionParameters(),
const int & grid_ =0)

void
applyBoundaryCondition(const Index & Components,
const BCTypes::BCNames & bcType,
const int & bc,
const RealArray & forcing,
RealArray *forcinga[2][3],
const real & time =0.,
const BoundaryConditionParameters &
bcParameters = Overture::defaultBoundaryConditionParameters(),
const int & grid_ =0)

If forcinga[side][axis] !=NULL then use this array, otherwise use forcing.

void
applyBoundaryCondition(const Index & Components,
const BCTypes::BCNames & bcType,
const int & bc,
const MappedGridFunction & forcing,
const real & time =0.,
const BoundaryConditionParameters &
bcParameters = Overture::defaultBoundaryConditionParameters(),
const int & grid_ =0)

Description: Apply a boundary condition to the grid function. This function just calls the corresponding function in MappedGridOperators. See the operator documentation for further details.

3.1.7 breakReference

void
breakReference()

Description: This member function will cause the grid function to no longer be referenced. The grid function acquires its own copy of the data.

Author: WDH

3.1.8 dataCopy

int
dataCopy(const MappedGridFunction & mgf)

Description: copy the array data only

mgf (input): set the array data equal to the data in this grid function.

3.1.9 Derivatives: x,y,z,xx,xy,xz,yy,yz,zz,laplacian,grad,div,r1,r2,r3,r1r1,r1r2,...

MappedGridFunction
derivative(const Index & I1 =nullIndex ,
const Index & I2 =nullIndex ,
const Index & I3 =nullIndex ,
const Index & I4 =nullIndex ,
const Index & I5 =nullIndex ,
const Index & I6 =nullIndex ,
const Index & I7 =nullIndex ,
const Index & I8 =nullIndex
)

Description: Derivative equals one of x,y,z,xx,xy,xz,yy,yz,zz,laplacian,grad,div,r1,r2,r3,r1r1,r1r2,r1r3,r2r2,r2r3,r3r3. Return the derivative of this grid function. This routine just calls the function of the same name in the GenericMappedGridOperators (see also setOperators).

I1,I2,I3 (input) : optional arguments to specify where the derivatives are evaluated. In this case the returned grid function will only have values of the derivative computed at this subset of points, other values in the grid function will be zero.

I4 (input) : evaluate the derivative for these components, by default all components.

Return value: The derivative is returned as a new grid function. For all derivatives but `grad` and `div` the number of components in the result is equal to the number of components specified by I4 (if I4 not specified then the result will have the same number of components s `u`). The `grad` operator will have number of components equal to the number of space dimensions while the `div` operator will have only one component.

MappedGridFunction
derivative(const GridFunctionParameters & gfType,
const Index & I1 =nullIndex ,
const Index & I2 =nullIndex ,
const Index & I3 =nullIndex ,
const Index & I4 =nullIndex ,
const Index & I5 =nullIndex ,
const Index & I6 =nullIndex ,
const Index & I7 =nullIndex ,
const Index & I8 =nullIndex
)

Description: derivative equals one of x,y,z,xx,xy,xz,yy,yz,zz,laplacian,grad,div. Return the derivative of this grid function. The argument gfType determines the type of the grid function that is returned. This routine just calls the function of the same name in the GenericMappedGridOperators (see also setOperators).

gfType (input): The type of the grid function to be returned.

I1,I2,I3 (input) : optional arguments to specify where the derivatives are evaluated. In this case the returned grid function will only have values of the derivative computed at this subset of points, other values in the grid function will be zero.

I4 (input) : evaluate the derivative for these components, by default all components.

Return value: The derivative is returned as a new grid function. For all derivatives but `grad` and `div` the number of components in the result is equal to the number of components specified by I4 (if I4 not specified then the result will have the same number of components s `u`). The `grad` operator will have number of components equal to the number of space dimensions while the `div` operator will have only one component.

3.1.10 Derivative Coefficients: xCoefficient,yCoefficient,...

MappedGridFunction
Derivative(const Index & I1 =nullIndex,
const Index & I2 =nullIndex,
const Index & I3 =nullIndex,
const Index & I4 =nullIndex,
const Index & I5 =nullIndex,
const Index & I6 =nullIndex,
const Index & I7 =nullIndex,
const Index & I8 =nullIndex
)

Description: Derivative equals one of xCoefficient,yCoefficient,zCoefficient,xxCoefficient, xyCoefficient,xzCoefficient,yyCoefficient,yzCoefficient,zzCoefficient, laplacianCoefficient,gradCoefficient,divCoefficient. Return the coefficients of the derivative. This routine just calls the function of the same name in the MappedGridOperators (see also setOperators).

I1,I2,I3,... (input) : optional arguments to specify where the derivatives are evaluated. In this case the returned grid function will only have values of the derivative computed at this subset of points, other values in the grid function will be zero.

MappedGridFunction
Derivative(const GridFunctionParameters & gfType,
const Index & I1 =nullIndex,
const Index & I2 =nullIndex,
const Index & I3 =nullIndex,
const Index & I4 =nullIndex,
const Index & I5 =nullIndex,
const Index & I6 =nullIndex,
const Index & I7 =nullIndex,
const Index & I8 =nullIndex
)

Description: Derivative equals one of xCoefficient,yCoefficient,zCoefficient,xxCoefficient, xyCoefficient,xzCoefficient,yyCoefficient,yzCoefficient,zzCoefficient, laplacianCoefficient,gradCoefficient,divCoefficient. Return the coefficients of the derivative. This routine just calls the function of the same name in the MappedGridOperators (see also setOperators).

gfType (input): The type of the grid function to be returned.

I1,I2,I3,... (input) : optional arguments to specify where the derivatives are evaluated. In this case the returned grid function will only have values of the derivative computed at this subset of points, other values in the grid function will be zero.

3.1.11 destroy

int
destroy()

Description: Destroy this grid function. Release all memory, and reset the grid function properties to the default.

3.1.12 get

int
get(const GenericDataBase & dir, const aString & name)

Description: Get from a database file. Example:

```
HDF_DataBase db;
db.mount("myFile.hdf","R");
MappedGrid g;
realMappedGridFunction u;
initializeMappingList();
g.get(db,"my grid");
u.updateToMatchGrid(g);   // **NOTE**
u.get(db,"u");
```

dir (input): get from this directory of the database.

name (input): the name of the grid function on the database.

NOTE: This get function will not set the pointer to the MappedGrid associated with this grid function. You should call updateToMatchGrid(...) to set the grid BEFORE using this function.

3.1.13 getClassName

aString
getClassName() const

Description: Return the class name.

3.1.14 getSerialArray

doubleSerialArray &
getSerialArray()

Description: Return the grid function as a serial array. In parallel return the local array with ghost boundaries.

3.1.15 getComponentBase

int
getComponentBase(int component) const

Description: Get the base for the given component.

component (input): component number, 0,1,...

Return Values: The base for the component. Unused components have base=0 and bound=0

3.1.16 getComponentBound

int
getComponentBound(int component) const

Description: Get the bound for the given component.

component (input): component number, 0,1,...

Return Values: The bound for the component. Unused components have base=0 and bound=0

3.1.17 getComponentDimension

int
getComponentDimension(int component) const

Description: Get the dimension for the given component, dimension=bound-base+1

component (input): component number, 0,1,...

Return Values: The dimension for the component. Unused components have dimension=1

3.1.18 getCoordinateBase

int
getCoordinateBase(int coordinate) const

Description: Get the base for the given coordinate.

coordinate (input): component number, 0,1, or 2.

Return Values: The base for the coordinate. Unused coordinates have base=0 and bound=0

3.1.19 getCoordinateBound

int
getCoordinateBound(int coordinate) const

Description: Get the bound for the given coordinate.

coordinate (input): component number, 0,1, or 2.

Return Values: The bound for the coordinate. Unused coordinates have base=0 and bound=0

3.1.20 getCoordinateDimension

int
getCoordinateDimension(int coordinate) const

Description: Get the dimension for the given coordinate, dimension = bound-base+1

coordinate (input): component number, 0,1, or 2.

Return Values: The dimension for the coordinate. Unused coordinates have dimension=1

3.1.21 getDerivatives

void
getDerivatives(const Index & I1 =nullIndex,
const Index & I2 =nullIndex,
const Index & I3 =nullIndex,
const Index & I4 =nullIndex,
const Index & I5 =nullIndex,
const Index & I6 =nullIndex,
const Index & I7 =nullIndex,
const Index & I8 =nullIndex) const

Description: Get derivatives for this grid function. This routine just calls the function of the same name in the MappedGridOperators. See the documentation for operators for further details.

3.1.22 getFaceCentering

faceCenteringType
getFaceCentering() const

Description: Get the type of face centering. For further explanation see `setFaceCentering` and section4.

Errors: none.

Return Values: faceCenteringType.

Author: WDH

3.1.23 getGridFunctionType

GridFunctionType
getGridFunctionType(const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex) const

Description: Return the type of the grid function.

component0,component1,... (input): get type of the grid function corresponding to these components.

Return Values: The grid function type, one of the enums in GridFunctionType.

Author: WDH

3.1.24 getGridFunctionTypeWithComponents

GridFunctionTypeWithComponents
getGridFunctionTypeWithComponents(const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex) const

Description: Return the type of the grid function with the number of components.

component0,component1,... (input): get type of the grid function corresponding to these components. By default (if no arguments are given) the number of components will be equal to the number of components that the grid function was made with. Otherwise the number of components will equal the number of arguments that have been passed to this routine (actually the number of arguments that are not a nullIndex)

Return Values: The grid function type with number of components, one of the enums in GridFunctionParameters::GridFunctionTypeWithComponents.

Note: In a faceCenteredAll grid function, the position taken by the faceRange does not count as a component for the value returned by this routine.

```
MappedGrid mg(...);
Range all;
floatMappedGridFunction u(mg,floatMappedGridFunction::faceCenterAll,2);
u.getGridFunctionTypeWithComponents(); // == faceCenterAllWith1Component
u.getNumberOfComponents();             // == 1
```

Author: WDH

3.1.25 getIsCellCentered

bool
getIsCellCentered(const Index & axis0 =nullIndex,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex) const

Description: Determine the cell centeredness of a grid function.

axis0 (input): if axis0=nullIndex (default) then all axes are checked

component0 (input): if component0=nullIndex (default) then all components are checked

component1 (input): if component1=nullIndex (default) then all components are checked

component2 (input): if component2=nullIndex (default) then all components are checked

component3 (input): if component3=nullIndex (default) then all components are checked

component4 (input): if component4=nullIndex (default) then all components are checked

Return Values: TRUE or FALSE

Detailed Description: A `MappedGridFunction` can be used for finite difference and finite volume codes. Finite volume codes often require that the grid function be cell-centered. By default a `MappedGridFunction` will be cell-centered if the `MappedGrid` is cell-centered or vertex-centered if the `MappedGrid` is vertex-centered.

Finite-volume codes often require grid functions that are face-centered. In order to support all the various possibilities one can, in general, specify that a grid function be cell-centered (or not) in some or all of the coordinate directions. Use the member function `setIsCellCentered` to set the "centeredness" of each component of the grid function. Use `getIsCellCentered` function to inquire the centeredness of each component of a grid function.

Since face-centered grid functions are common, the function `setIsFaceCentered(axis,component)` can be used to create a face-centered grid function in the coordinate direction "axis" for a given component. (A face-centered grid function is vertex centered in the "axis-direction" and cell-centered in the other directions). The function `getIsFaceCentered` can be used to determine if a grid function is face centered in a given direction. For example

```
...
realMappedGridFunction u(mg,3);   // a grid function with 3 components
int axis=0, component=0;
u.setIsCellCentered(TRUE,axis,component); // make u cell centred along axis 0 for component 0

axis=1; component=1;
u.setIsCellCentered(FALSE,axis,component); // make u vertex centred along axis 0 for component 1

// inquire the cell-centredness
cout << "u.getIsCellCentered(axis,component) = " << u.getIsCellCentered(axis,0) << endl;

u.setIsFaceCentered( axis,component );   // make u face-centered along axis for a component
```

For further explanation see section4.

Author: WDH

3.1.26 getIsFaceCentered

bool
getIsFaceCentered(const int & axis0 =forAll,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex) const

Description: Determine if a given component of this grid function is face-centred along a given axis. By default check all axes and all components.

axis0: check if the components are face centred along this axis. By default check if the components are face centred in ANY direction.

component0, component1,... (input): check the value for these components, by default check all components.

3.1.27 getMappedGrid

MappedGrid*
getMappedGrid(const bool abortIfNull =TRUE) const

Description: Return a pointer to the MappedGrid that this grid function is asscoaiated with By default this function will abort if the pointer is NULL.

Return values: A pointer to a MappedGrid or NULL

3.1.28 getName

aString
getName(const int & component0 =defaultValue,
const int & component1 =defaultValue,
const int & component2 =defaultValue,
const int & component3 =defaultValue,
const int & component4 =defaultValue) const

Description: Get the name of the grid function or a component as in

```
aString nameOfGridFunction = u.getName();
aString nameOfComponent0   = u.getName(0);
aString nameOfComponent1   = u.getName(1);
```

name: the name of the grid function or component.

component0, component1, (input): get the name for this component. if all of component0,component1,component2 ==defaultValue then the name of the grid function is returned. Otherwise the default value becomes the base value for that component.

3.1.29 getNumberOfComponents

int
getNumberOfComponents() const

Description: return the number of components (0=scalar, 1=vector, 2=matrix, ...).

Return Values: Valid values are 0,...,5

Examples: Here are some examples. Note the special case for grid functions created with a `faceRange`, the `faceRange` position does NOT count as a component.

```
MappedGrid mg(...);
Range all;
floatMappedGridFunction u(mg);                              // 0 components
floatMappedGridFunction u(mg,all,all,all);                  // 0 components
floatMappedGridFunction u(mg,all,all,all,1);                // 1 component
floatMappedGridFunction u(mg,all,all,all,2,2);              // 2 components
floatMappedGridFunction u(mg,all,all,all,faceRange);        // 0 components
floatMappedGridFunction u(mg,all,all,all,3,faceRange);      // 1 component
```

Author: WDH

3.1.30 getOperators

MappedGridOperators*
getOperators() const

Description: get the operators used with this grid function. Return NULL if there are none.

3.1.31 isNull

bool
isNull()

Description: Return TRUE if this grid function is null (has no grid associated with it).

Return value: Return TRUE if this grid function is null, otherwise return FALSE.

3.1.32 link

void
link(const MappedGridFunction & mgf,
const Range & R0 =nullRange,
const Range & R1 =nullRange,
const Range & R2 =nullRange,
const Range & R3 =nullRange,
const Range & R4 =nullRange)

Description: The `link` member function can be used to link a grid function to a specific component of another grid function.

mgf (input): link to this

R0, R1, ..., R4 (input): indicate which components to link to. Note that the Ranges for the linked grid function always start at 0. Use updateToMatchGridFunction to change this.

Examples: A link is sort of like a reference since the array data is shared. NOTE that a link can only be made to a grid function whose components appear at the end of the array (position 3 for 3D grid functions or positions 2 or 3 for 2D grid functions). Links can also be made to more than one components, provided the components are contiguous.

(1) Linking to a vector grid function:

```
MappedGrid mg(...);
Range R0(0,3);
floatMappedGridFunction u(mg,all,all,all,R0);  // u is a vector grid function
floatMappedGridFunction v;
v.link(u,Range(0,0));   // link to component 0 of u         -> v(all,all,all,0:0)
v.link(u,Range(0,1));   // link to components 0 and 1 of u  -> v(all,all,all,0:1)
v.link(u,Range(2,2));   // link to component 2 u            -> v(all,all,all,0:0)
```

(2) Linking to a matrix grid function:

```
MappedGrid mg(...);
Range R0(0,3), R1(0,2);
floatMappedGridFunction u(mg,all,all,all,R0,R1);  // u is a 2D matrix grid function
v.link(u,Range(1,1));   // link to matrix element (1,0)  -> v(all,all,all,0:0,0:0)
v.link(u,Range(1,1),Range(2,2));   // link to component (1,2) -> v(all,all,all,0:0,0:0)
v.link(u,R0,Range(2,2));   // link to components (R0,2) -> v(all,all,all,0:3,0:0)

v.link(u,Range(1,1),Range(0,2));   // **ERROR** these values are not contiguous
```

Errors: Attempt to link to invalid components.

Return Values: none.

Notes: The linkee function will acquire the same operators as the function being linked to.

Author: WDH

3.1.33 multiply(a,coeff)

MappedGridFunction &
multiply(const MappedGridFunction & a_, const MappedGridFunction & coeff_)

Description: Multiply a grid function times a coefficient matrix. Use this function to multiply a scalar grid function "a" times a coefficient matrix "coeff". The result is saved in coeff and returned by reference.

```
coeff(M,I1,I2,I3) <- a(I1,I2,I3)*coeff(M,I1,I2,I3)
```

a_ (input) : a scalar grid function.

coeff_ (input/output) : a grid function in the shape a coefficient matrix (1 component in position 0) This argument is NOT const but it was made to to prevent some compiler warnings.

Return value: a reference to coeff

3.1.34 multiply(a,coeff)

MappedGridFunction &
multiply(const doubleDistributedArray & a_, const MappedGridFunction & coeff_)

Description: Multiply an array times a coefficient matrix. Use this function to multiply a "scalar" array "a" times a coefficient matrix "coeff". The result is saved in coeff and returned by reference.

```
coeff(M,I1,I2,I3) <- a(I1,I2,I3)*coeff(M,I1,I2,I3)
```

a_ (input) : an array with the same dimensions as a grid function.

coeff_ (input/output) : a grid function in the shape a coefficient matrix (1 component in position 0) This argument is NOT const but it was made to to prevent some compiler warnings.

3.1.35 numberOfComponents

const int&
numberOfComponents() const

Return value: the number of components (0=scalar, 1=vector, ...)

3.1.36 numberOfDimensions

const int&
numberOfDimensions() const

Return value: the numberOfDimensions of the grid function (equal to the domain dimension of the grid)

3.1.37 operator = MappedGridFunction

MappedGridFunction &
operator= (const MappedGridFunction & cgf)

Description: Set one grid function equal to another. This is a shallow copy where only the array data is copied. An error occurs if the two grid functions are not conformable. This operation has the same affect as the `dataCopy` memeber function. An exception to this rule is when the grid function to the left of the equals operator is a 'null' grid function (one that has no grid associated with it such as a grid function built by the default constructor). In this case a deep copy is performed.

Examples: Here are some examples

```
MappedGrid mg(...);
realMappedGridFunction u(mg),v(mg),w;
Index I;
...
u=1.;
v=u;                 // only the data is copied
w=u;                 // this is a deep copy since w is a 'null' grid function.
u=v+w;               // does NOT call this = operator, uses grid-function=A++ array
u=v(I)+w(I);         // does NOT call this = operator, uses grid-function=A++ array
u=3;                 // does NOT call this = operator, uses grid-function=scalar
u(I)=v(I)+v(I);      // does NOT call this = operator, uses A++ =
u.dataCopy(v+w);     // only copies array data (same as u=v+w; in this case)
u.updateToMatchGridFunction(v);  // this is a real deep copy.
realMappedGridFunction a = u;  // does NOT call this = operator, calls copy constructor
```

3.1.38 operator = double

MappedGridFunction &
operator= (const double x)

Description: Set the values of a grid function equal to a scalar.

3.1.39 operator = doubleDistributedArray

MappedGridFunction &
operator= (const doubleDistributedArray & X)

Description: Set the values of a grid function equal to an A++ array. The operation must be conformable or else an A++ error will be generated.

3.1.40 periodicUpdate

void
periodicUpdate(const Range & C0 =nullRange,
const Range & C1 =nullRange,
const Range & C2 =nullRange,
const Range & C3 =nullRange,
const Range & C4 =nullRange,
const bool & derivativePeriodic =FALSE)

Description: Swap periodic edges of the grid function. Assign values to `side=1` boundary lines

$$\texttt{i}_{\texttt{axis}} = \texttt{mg.gridIndexRange}()(1,\texttt{axis}) \quad \texttt{axis} = 0,1,..,\texttt{mg.numberOfDimensions}$$

(`mg` is the `MappedGrid` associated with this grid function) as well as all ghost lines on all sides that have periodic boundary conditions.

C0,C1,...C4 (input) : specify which components to update. By default update all components.

derivativePeriodic (input): if TRUE we assume that the grid function is not actually periodic but that only it's derivative is – like the grid function for the vetrex array on a periodic square.

3.1.41 positionOfFaceCentering

const int& positionOfFaceCentering() const

Return value: the index position, (0,1,2,..) of the face centering.

3.1.42 put

int
put(GenericDataBase & dir, const aString & name) const

Description: Output a grid function onto a database file

dir (input): put onto this directory of the database.

name (input): the name of the grid function on the database.

Notes:

First some definitions

- N=`maximumNumberOfIndicies` The maximum number of dimensions in a grid function (current value =8).
- N_A=`numberOfIndicies` The maximum number of A++ dimensions (current value =4).

Here are the items that are saved in a data base.

numberOfComponents (int) The number of component indices in the grid function: 0 for a scalar 1 for a vector, 2 for a matrix etc. Currently there can be at most 5 components. The default value and the value for unused entries is N=`maximumNumberOfIndicies`.

positionOfCoordinate (IntegerArray(N)) `positionOfCoordinate(i)` holds the index in the array (numbered starting from 0) of the 3 coordinate positions, $i = 0, 1, 2$. The default value and the value for unused entries is N=`maximumNumberOfIndicies`.

positionOfComponent (IntegerArray(N)) `positionOfComponent(i)` holds the index in the array (numbered starting from 0) of the component positions, $i = 0, 1, ..., N-1$. The default value and the value for unused entries is N=`maximumNumberOfIndicies`.

Examples:

```
MappedGrid mg(...);
Range all;
realMappedGridFunction u(mg,all,all,all);
--> numberOfComponents=0
--> positionOfCoordinate(0)=0, positionOfCoordinate(1)=1, positionOfCoordinate(2)=2
realMappedGridFunction u(mg,2,all,all,all);
--> numberOfComponents=1
--> positionOfCoordinate(0)=1, positionOfCoordinate(1)=2, positionOfCoordinate(2)=3
--> positionOfComponent(0)=0
realMappedGridFunction u(mg,all,2,all,3,all,4);
--> numberOfComponents=3
--> positionOfCoordinate(0)=0, positionOfCoordinate(1)=2, positionOfCoordinate(2)=4
--> positionOfComponent(0)=1,  positionOfComponent(2)=3,  positionOfComponent(3)=5
```

positionOfFaceCentering (int) For a face centred grid function of standard type this is the index position of the face centering. For all other types of grid functions this has a value of -1.

faceCentering (enum faceCenteringType) The face centering type for the grid function. Default value is `none`$= -1$.

numberOfDimensions The number of space dimensions, $1, 2$, or 3.

isACoefficientMatrix (bool) If TRUE (=1) then this is a coefficient matrix, default is FALSE (=0).

stencilType (enum StencilTypes) The type of stencil for a coefficient matrix, default is `standardStencil` (=0).

stencilWidth (int) The stencil width for a coefficient matrix, default value = 0.

R[i].base (int) (`i=0,1,...,N`) The base of the Range object `R[i]` which holds the base and bound for index position i. For unused positions the default is 0. There is one extra Range, `R[N]=Range(0,0)` which exists just for convenience.

R[i].bound (int) (`i=0,1,...,N`) The bound of the Range objects `R[i]`. For unused index positions the default is 0.

Ra[i].base (int) (`i=0,1,...,`N_A`-1`) The base of the Range objects `Ra[i]` which holds the actual base and bound for index position i of the A++ array (from which the grid function is derived). Currently A++ arrays have only 4 dimensions so we compress the final 5 dimensions of a grid function to be stored in the last A++ dimension. For unused positions the default is 0.

Ra[i].bound (int) (`i=0,1,...,`N_A`-1`) The bound of the Range object `Ra[i]`. For unused positions the default is 0.

Rc[i].base (int) (`i=0,1,2`) The base of the Range objects `Rc[i]` which hold special information about the base and bound for the coordinate directions. These are required for grid functions that only live on boundaries. The default value is 0.

Rc[i].bound (int) (`i=0,1,2`) The bound of the Range objects `Rc[i]`. The default value is -1.

numberOfNames (int) The number of names that are saved. (see next item).

name[i] (aString) (`i=0,1,...,numberOfNames-1`) The names for the grid function and its components.

isCellCentered (IntegerArray(3,C_0,C_1,C_2)) The cell centeredness (0/1) in each coordinate direction for each component. Currently we only save the info for 3 components (when A++ is fixed for 8 dimensions we will save the info for 5 components). For a vertex centered grid the default values are all 0. In terms of the other variables described here the isCellCentered array has dimensions:

```
isCellCentered.redim(3,R[positionOfComponent(0)],
                       R[positionOfComponent(1)],
                       R[positionOfComponent(2)]);
```

and thus C_i`=R[positionOfComponent(i)]`.

arrayData (A++ array) This is the A++ array that holds the actual array-data for this grid function.

3.1.43 reference

void
reference(const MappedGridFunction & cgf)

Description: Use this function to reference one MappedGridFunction to another. When two (or more) grid functions have been referenced they share the same array data so that changes to one grid function will change all the other referenced grid functions. Only the array data is referenced. Other properties of the grid function such as cell-centredness can be changed in the referenced grid function. The "shape" of the referenced grid function can also be changed without changing the referencee:`cgf`.

Author: WDH

3.1.44 Standard argument function, sa

double &
sa(const int & i0,
const int & i1,
const int & i2,
const int & c0 =0,
const int & c1 =0,
const int & c2 =0,
const int & c3 =0,
const int & c4 =0) const

Description: The sa, "standard argument" function permutes the arguments to that you can always refer to a function as u(coordinate(0),coordinate(1),corrdinate(2),component(0),component(1),...)

i0, i1, i2 (input): index values for the three coordinates

c0, c1,... (input): index values for the components

Return Values: The value of the grid function.

3.1.45 setFaceCentering

void
setFaceCentering(const int & axis =defaultValue)

Description: Set the type of face centering, the behaviour of this function depends on whether the argument "axis" has been specified or else if the current value for getFaceCentering().

axis (input):
1. if "axis" is given then make all components face centred in direction=axis
2. if getFaceCentering()==all : make components face centered in all directions, the grid function should have been contructed or updated using the faceRange to specify which Index is to be used for the "directions"

For further explanation see section4.

Author: WDH

3.1.46 setIsACoefficientMatrix

void
setIsACoefficientMatrix(const bool trueOrFalse =TRUE,
const int stencilSize0 = defaultValue,
const int numberOfGhostLines =1,
const int numberOfComponentsForCoefficients =1,
const int offset =0)

Description: Indicate whether a grid function holds a coefficient matrix. ALso use this routine to update the sparse matrix representation when the grid has changed. (Call this routine AFTER calling updateToMatchGrid)

trueOrFalse (input): TRUE means this grid function is a coefficient matrix

stencilSize0 (input): This is the stencil size for the coefficient matrix. By default the stencil size is 3 in 1D, 9 in 2D and 27 in 3D.

numberOfGhostLines (input): indicates the number of ghost-lines on which there will equations defined in the coefficient matrix.

numberOfComponentsForCoefficients (input): This is the dimension of the system of equations that is represented in the matrix.

offset (input): This is an offset to use when numbering the equations. This value would be used when the MappedGridFunction is really part of a CompositeGRidFunction.

Author: WDH

3.1.47 setIsACoefficientMatrix

void
setIsACoefficientMatrix(SparseRepForMGF *sparseRep)

Set the current sparse Representation. This is normally only used internally and by the gridCollectionFunction so that it can reference the multigrid and refinement level lists properly.

3.1.48 setIsCellCentered

void
setIsCellCentered(const bool trueOrFalse,
const Index & axis0 =nullIndex,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex)

Description: Change the cell centered-ness of the grid function. By default set all components.

trueOfFalse (input): make cell-centred or not.

axis0: set the value for this axis, by default set all axes.

component0, component1, (input): set the value for these components, by default set all components.

3.1.49 setIsFaceCentered

void
setIsFaceCentered(const int & axis0 =forAll,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex)

Description: Make a component of a grid function face centred along the given axis. A face centered grid function along axis0 is vertex centered along axis0 and cell centered along the other axes.

axis0: set the value for this axis, by default set all axes.

component0, component1, (input): set the value for these components, by default set all components.

3.1.50 setName

void
setName(const aString & name,
const int & component0 =defaultValue,
const int & component1 =defaultValue,
const int & component2 =defaultValue,
const int & component3 =defaultValue,
const int & component4 =defaultValue)

Description: Set the name of the grid function or a component as in

```
u.setName("nameOfGridFunction");
u.setName("nameOfComponent0",0);
u.setName("nameOfComponent1",1);
```

name: the name of the grid function or component.

component0, component1,... (input): give the name for this component. if all of component0,component1,component2 ==defaultValue then the name of the grid function is set. Otherwise the default value becomes the base value for that component.

3.1.51 setOperators

void
setOperators(GenericMappedGridOperators & operators0)

Description: Supply a derivative object to use for computing derivatives on all component grids. This operator is used for the member functions .x .y .z .xx .xy etc.

operators0: use these operators.

3.1.52 setUpdateToMatchGridOption

void
setUpdateToMatchGridOption(const UpdateToMatchGridOption & updateToMatchGridOption)

Description: Specify what should be updated when calls are made to updateToMatchGrid

updateToMatchGridOption (input): A combination (using the — operation) of the following options:

```
enum UpdateToMatchGridOption
{
  updateSize=1,
  updateCoefficientMatrix=2
};
```

The default is `updateToMatchGridOption= updateSize | updateCoefficientMatrix`.

3.1.53 updateToMatchGrid

updateReturnValue
updateToMatchGrid()

updateReturnValue
updateToMatchGrid(const Range & R0 = nullRange,
const Range & R1 = nullRange,
const Range & R2 = nullRange,
const Range & R3 = nullRange,
const Range & R4 = nullRange,
const Range & R5 = nullRange,
const Range & R6 = nullRange,
const Range & R7 = nullRange)

updateReturnValue
updateToMatchGrid(MappedGrid & grid0,
const Range & R0,
const Range & R1 = nullRange,
const Range & R2 = nullRange,
const Range & R3 = nullRange,
const Range & R4 = nullRange,
const Range & R5 = nullRange,
const Range & R6 = nullRange,
const Range & R7 = nullRange)

Description: Update a grid function. Optionally specify a new grid and new Ranges.

grid0 (input): update to match this grid.

R0, R1, ... (input): Use these Range objects to determine the grid function dimensions.

Return Values: Return a value from the enumerator `updateReturnValue`:

```
enum updateReturnValue  // the return value from updateToMatchGrid is a mask of the following val
{
  updateNoChange          = 0, // no changes made
  updateReshaped          = 1, // grid function was reshaped
  updateResized           = 2, // grid function was resized
  updateComponentsChanged = 4  // component dimensions may have changed (but grid was not resized
};
```

Author: WDH

updateReturnValue
updateToMatchGrid(MappedGrid & grid0,
const GridFunctionParameters::GridFunctionType & type,
const Range & component0,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: Use this update function to create a grid function of a given type. See the comments in the corresponding constructor.

updateReturnValue
updateToMatchGrid(MappedGridData & grid0,
const GridFunctionParameters::GridFunctionType & type,
const Range & component0,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: Use this update function to create a grid function of a given type. See the comments in the corresponding constructor.

updateReturnValue
updateToMatchGrid(const GridFunctionType & type,
const Range & component0,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: Use this update function to create a grid function of a given type. See the comments in the corresponding constructor.

updateReturnValue
updateToMatchGrid(MappedGrid & grid0,
const GridFunctionParameters::GridFunctionType & type)

Description: Use this update function to create a grid function of a given type, the components are left unchanged.

updateReturnValue
updateToMatchGrid(const GridFunctionType & type)

Description: Use this update function to create a grid function of a given type, the components are left unchanged.

3.1.54 updateToMatchGridFunction

updateReturnValue
updateToMatchGridFunction(const MappedGridFunction & cgf)

updateReturnValue
updateToMatchGridFunction(const MappedGridFunction & cgf,
const Range & R0,
const Range & R1 =nullRange,
const Range & R2 =nullRange,
const Range & R3 =nullRange,
const Range & R4 =nullRange,
const Range & R5 =nullRange,
const Range & R6 =nullRange,
const Range & R7 =nullRange)

Description: Update this grid function to match another grid function (this is like using the = operator but it avoids copying the array data)

cgf (input): match to this grid function.

R0, R1, ... (input): optional ranges to change the dimensions.

3.1.55 sizeOf

real
sizeOf(FILE *file = NULL) const

Description: Return number of bytes allocated by this object; optionally print detailed info to a file

file (input) : optinally supply a file to write detailed info to. Choose file=stdout to write to standard output.

Return value: the number of bytes.

3.1.56 fixupUnusedPoints

int
fixupUnusedPoints(const RealArray & value = nullRealArray(),
int numberOfGhostlines =1)

Description: Assign values to points on a grid function that correspond to unused points (mask==0). By default all unused points are set to zero. Use the value array to set unused points to particular values.

values (input) : if supplied, assign value(n) to unused points of component n and do not change any components not found in value. If not supplied set all unused points to zero.

numberOfGhostLines (input) : Indicate how many ghost lines are used in the computation. Other ghost line values will all be set to zero.

In the following, "`type`" will mean one of `float`, `double` or `int`. The most commonly used constructor takes a `MappedGrid` and an optional sequence of Range's (or int's). The optional `Range` arguments indicate the positions of the coordinates and the positions and dimensions of any components.

```
typeMappedGridFunction()                                    default constructor
typeMappedGridFunction(MappedGrid & mappedGrid,
                       Range & R0=nullRange,                this arg can be an int, or Range
                       Range & R1=nullRange,
                       Range & R2=nullRange,
                        ...
                       Range & R7=nullRange, )
typeMappedGridFunction(MappedGrid & mappedGrid,             Old style declaration, THIS WILL GO AWAY
                       int numberOfComponents=1,
                       int positionOfComponent=default )
```

Each grid function is dimensioned according to the dimensions found with the `MappedGrid`, using the `dimension` values. Grid functions can have up to 8 dimensions, the index positions not used by the coordinate dimensions can be used to store different components. For example, a *vector* grid functions would use 1 index position for components while a *matrix* grid functions would use two index positions for components. Here are some examples

```
//  R1 = range of first dimension of the grid array
//  R2 = range of second dimension of the grid array
//  R3 = range of third dimension of the grid array

MappedGrid mg(...);

Range R1(mg.dimension(Start,axis1),mg.dimension(End,axis1));
Range R2(mg.dimension(Start,axis2),mg.dimension(End,axis2));
Range R3(mg.dimension(Start,axis3),mg.dimension(End,axis3));

Range all;    // null Range is used to specify where the coordinates are

MappedGridFunction u(mg);                          //  --> u(R1,R2,R3);

MappedGridFunction u(mg,all,all,all,1);            //  --> u(n,R1,R2,R3,0:1);
MappedGridFunction u(mg,all,all,Range(1,1));       //  --> u(n,R1,R2,1:1,R3);

MappedGridFunction u(mg,2,all);                    //  --> u(n,0:2,R1,R2,R3);
MappedGridFunction u(mg,Range(0,2),all,all,all);   //  --> u(n,0:2,R1,R2,R3);
MappedGridFunction u(mg,all,Range(3,3),all,all);   //  --> u(n,R1,3:3,R2,R3);
```

3.2 Examples

In this example we show how to define a grid function using a `MappedGrid`, how to assign the grid function with A++ operations, how to reference one grid function to another and how to set and get names for the grid function and its components.

```
  ...
   MappedGrid cg(...);  // here is a mapped grid
   floatMappedGridFunction u(cg),v;
   u=5.;
   Index I(0,10);
   u(I,I)=3.;
   v.reference(u);   // v is referenced to u
   v=7.;             // changes u as well
   v.breakReference();  // v is no longer referenced to u
   v=10.;            // v changed but not u

   // Here is how to dimension a grid function after it has been declared:
   floatMappedGridFunction w;
   Range all;
   int numberOfComponents=2;
   w.updateToMatchGrid( cg,all,all,all,numberOfComponents );
   ...
   // give names to the grid function and components
   w.setName("w");      // name grid function
   w.setName("w.0",0);  // name component 0
   w.setName("w.1",1);  // name component 1
   cout << w.getName() << "," << w.getName(0) << "," << w.getName(1) << endl;
}
```

3.3 Grid functions defined on boundaries

Grid functions can be created so that they are defined on the boundary of a grid, or on the boundary and some number of neighbouring grid lines. For example, you may want to store the normal vectors on a given boundary of a grid.

To specify which boundary a grid function lives on one must create or update the grid function with a Range object that was created in a special way. This special Range object is used in place of an "all" Range object. Grid functions defined in this way can be correctly updated when the number of points on the grid changes – they will still live on the appropriate boundary.

Here are some examples (file `/home/henshaw.0/Overture/gf/edge.C`)

```
1   //================================================================================
2   // Demonstrate how to use grid function that are defined on the edge of grid
3   //================================================================================
4   #include "Overture.h"
5   #include "CompositeGridOperators.h"
6
7
8   int
9   main(int argc, char *argv[])
10  {
11    Overture::start(argc,argv);  // initialize Overture
12
13    cout << "Starting test...\n";
14
15    aString nameOfOGFile;
16    cout << " >>>>>Testing the Edge GridFunctions " << endl;
17
18    cout << "Enter the name of the composite grid file (in the ogen directory)" << endl;
19    cin >> nameOfOGFile;   nameOfOGFile="/users/henshaw/res/cgsh/" + nameOfOGFile;
20
21    CompositeGrid cg;
22    getFromADataBase(cg,nameOfOGFile);
23    cg.update();
24
25    MappedGrid & mg = cg[0];
26
27    Range all;
28
29     // first define a some local variables so we can shorten the names
30    realMappedGridFunction::edgeGridFunctionValues startIndex =realMappedGridFunction::startingGridIndex;
31    realMappedGridFunction::edgeGridFunctionValues endIndex   =realMappedGridFunction::endingGridIndex;
32
33    //
34    // define a grid function that lives on the face: (side,axis)=(0,0)
35    //
36    Range S(startIndex,startIndex);
37    realMappedGridFunction u(mg,S,all,all);
38    u=1.;
39    u.display("Here is u(mg,S,all,all)");
40    u.periodicUpdate();
41
42    //
43    // change the grid function to live on the face (side,axis)=(0,1) and include neighbouring grid lines
44    //
45    u.updateToMatchGrid(mg,all,Range(startIndex-1,startIndex+1));
46    u=2.;
47    u.display("u.updateToMatchGrid(mg,all,Range(startIndex-1,startIndex+1))");
48    u.periodicUpdate();
49
50    //
51    // define another grid function to live on the face (side,axis)=(1,0)
52    Range E(endIndex,endIndex);
53    realMappedGridFunction v(mg,E,all,all);
54    v=3.;
55    v.display("Here is v(mg,E,all,all)");
56    v.periodicUpdate();
57
58    v=3.*v;
59    v.display("Here is v=3*v =9?");
60    evaluate(3.*v).display(" evaluate(3.*v) =27?");
61
62    //
63    // Now make the grid function live on a corner (or edge in 3d)
64    //
65    v.updateToMatchGrid(mg,S,E,all);
66    v=4.;
67    v.display("Here is uLeftRight(mg,S,E,all)");
68    v.periodicUpdate();
69
70    Overture::finish();
71    return 0;
72
```

```
73  }
```

3.4 Grid Functions that hold coefficient matrices

A MappedGridFunction can be used to hold the coefficients for a sparse matrix.

********** **NOTE: This section is purely hypothetical. I am recording some possible ways to interface to the coefficient matrices** **********

Here is a standard way to store the coefficients:

```
****** this does not work yet *****

MappedGrid mg(...);
numberOfStencilCoefficients=pow(3,mg.numberOfDimensions);   // 9 or 27 points
realMappedGridFunction coeff(numberOfStencilCoefficients,all,all,all);

coeff.setIsACoefficientMatrix(standardStencil);             //

... fill in coeff matrix here ....

Index I1,I2,I3;
getIndex(mg.gridIndexRange,I1,I2,I3);
realMappedGridFunction u(mg),residual(mg);

intArray & so = coeff.stencilOffset;    // these values indicate the offsets
//
// Here is how the residual can be computed
//
//     residual(I1,I2,I3) = SUM coeff(m,I1,I2,I3) * u(I1+so(0,m),I2+so(1,m),I3+so(2,m))
//                          m=0
//
residual(I1,I2,I3) = coeff(0,I1,I2,I3)*u(I1+so(0,0),I2+so(1,0),I3+so(2,0))
                    +coeff(1,I1,I2,I3)*u(I1+so(0,1),I2+so(1,1),I3+so(2,1))
                    +coeff(2,I1,I2,I3)*u(I1+so(0,2),I2+so(1,2),I3+so(2,2))
                    +coeff(3,I1,I2,I3)*u(I1+so(0,3),I2+so(1,3),I3+so(2,3))
                    +coeff(4,I1,I2,I3)*u(I1+so(0,4),I2+so(1,4),I3+so(2,4))
                    +coeff(5,I1,I2,I3)*u(I1+so(0,5),I2+so(1,5),I3+so(2,5))
                        .... etc ....


 //
 //  Here is another way which assumes it is a standard 3x3x3 stencil
 //
 intArray value(Range(-1,1),Range(-1,1),Range(-1,1));
 for( int m3=-1; m3<=1; m3++ )
 for( int m2=-1; m2<=1; m2++ )
 for( int m1=-1; m1<=1; m1++ )
 {
   value(m1,m2,m3)=coeff.coefficientPosition(m1,m2,m3);
   assert( value(m1,m2,m3)>=0 );   // check for any errors in computing the coefficient position

 }

 residual(I1,I2,I3) = coeff(value(-1,-1,-1),I1,I2,I3)*u(I1-1,I2-1,I3-1)
                     +coeff(value( 0,-1,-1),I1,I2,I3)*u(I1  ,I2-1,I3-1)
                     +coeff(value(+1,-1,-1),I1,I2,I3)*u(I1+1,I2-1,I3-1)
                     +coeff(value(-1, 0,-1),I1,I2,I3)*u(I1-1,I2  ,I3-1)
                     +coeff(value( 0, 0,-1),I1,I2,I3)*u(I1  ,I2  ,I3-1)
                     +coeff(value(+1, 0,-1),I1,I2,I3)*u(I1+1,I2  ,I3-1)
                          ... etc ...
```

Here is how we handle systems of equations

```
****** this does not work yet *****

MappedGrid mg(...);
numberOfComponents=2;
numberOfStencilCoefficients=pow(3,mg.numberOfDimensions);   // 9 or 27 points
```

```
realMappedGridFunction coeff(numberOfComponents*numberOfStencilCoefficients,all,all,all);

coeff.setIsACoefficientMatrix(standardStencil,numberOfComponents);             //

... fill in coeff matrix here ....

Index I1,I2,I3;
getIndex(mg.gridIndexRange,I1,I2,I3);
realMappedGridFunction u(mg,all,all,all,2),residual(mg,all,all,all,2);

intArray & so = coeff.stencilOffset;    // these values indicate the offsets
intArray & c  = coeff.stencilComponent; // these values indicate the component
//
// Here is how the residual can be computed
//
//     residual(I1,I2,I3) = SUM coeff(m,I1,I2,I3) * u(I1+so(0,m),I2+so(1,m),I3+so(2,m))
//                          m=0
//
residual(I1,I2,I3) = coeff(0,I1,I2,I3)*u(I1+so(0,0),I2+so(1,0),I3+so(2,0),c(0))
                    +coeff(1,I1,I2,I3)*u(I1+so(0,1),I2+so(1,1),I3+so(2,1),c(1))
                    +coeff(2,I1,I2,I3)*u(I1+so(0,2),I2+so(1,2),I3+so(2,2),c(2))
                    +coeff(3,I1,I2,I3)*u(I1+so(0,3),I2+so(1,3),I3+so(2,3),c(3))
                    +coeff(4,I1,I2,I3)*u(I1+so(0,4),I2+so(1,4),I3+so(2,4),c(4))
                    +coeff(5,I1,I2,I3)*u(I1+so(0,5),I2+so(1,5),I3+so(2,5),c(5))
                        .... etc ....
 //
 //  Here is another way which assumes it is a standard 3x3x3 stencil
 //
 intArray value(Range(-1,1),Range(-1,1),Range(-1,1),numberOfComponents);
 for( int n=0; n<numberOfComponents; n++)
 for( int m3=-1; m3<=1; m3++ )
 for( int m2=-1; m2<=1; m2++ )
 for( int m1=-1; m1<=1; m1++ )
 {
   value(m1,m2,m3,n)=coeff.coefficientPosition(m1,m2,m3,n);
   assert( value(m1,m2,m3,n)>=0 );   // check for any errors in computing the coefficient position

 }
 residual(I1,I2,I3) = coeff(value(-1,-1,-1,0),I1,I2,I3)*u(I1-1,I2-1,I3-1,0)  // coeff's for component 0
                     +coeff(value( 0,-1,-1,0),I1,I2,I3)*u(I1  ,I2-1,I3-1,0)
                     +coeff(value(+1,-1,-1,0),I1,I2,I3)*u(I1+1,I2-1,I3-1,0)
                          ... etc ...
                     +coeff(value(-1,-1,-1,1),I1,I2,I3)*u(I1-1,I2-1,I3-1,1)  // coeff's for component 1
                     +coeff(value( 0,-1,-1,1),I1,I2,I3)*u(I1  ,I2-1,I3-1,1)
                     +coeff(value(+1,-1,-1,1),I1,I2,I3)*u(I1+1,I2-1,I3-1,1)
```

Other possible ways to store the coefficients

```
****** this does not work yet *****

coeff.setIsACoefficientMatrix(fiveOrSevenPointStencil);   // 5 point star in 2d, 7 point star in 3D


coeff.setIsACoefficientMatrix(generalStencil);   // user defined stencil
```

3.5 GridCollectionFunction and CompositeGridFunction

GridCollectionFunction's and the CompositeGridFunction's are very similar. The only difference is that a GridCollectionFunction is associated with a GridCollection while a CompositeGridFunction is associated with a CompositeGrid.

Here we describe the GridCollectionFunction. These remarks all apply to the CompositeGridFunction if "GridCollection" is replaced by "CompositeGrid".

This class contains a list of MappedGridFunction's and a GridCollection. A GridCollectionFunction knows how to dimension its member MappedGridFunction's and how to update itself when the GridCollection changes (for example, when a refinement patch is added or a grid is moved).

This is a reference counted class so that there is no need to keep a pointer to a grid function. Use the reference member function to make one grid function reference another.

3.5.1 Public data members

Here are the public data members:

intR numberOfComponentGrids: (CompositeGridFunction only) equals value found in the CompositeGrid, this is here for convenience

GridCollection *gridCollection: pointer to the GridCollection

GenericGridCollectionOperators *operators: pointer to operators used for derivatives etc.

3.5.2 Public enumerators

The following enumerators are equivalent to the ones appearing in the `MappedGridFunction`. See the MappedGridFunction documentation for futher details.

updateReturnValue: The value returned from the `updateToMatchGrid` and `updateToMatchGridFunction` is a mask formed by a bitwise `or` of the following values:

3.5.3 Arithmetic operators, max,min,abs

The arithmetic operators $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$ and $/=$ are defined for two `GridCollectionFunction`'s of the same type or for a `GridCollectionFunction` and a float, double, or int. The operators `max`, `min` and `fabs` (`abs` for the int case) are defined.

3.5.4 Constructors

GridCollectionFunction ()

Description: Default constructor

GridCollectionFunction(GridCollection & gc)

Description: Create a grid function and associate with a GridCollection. The grid function will be a "scalar" as in the declaration:

```
Range all;
GridCollection gc(...);
GridCollectionFunction u(gc,all,all,all);
```

gc (input): grid to associate this grid function with.

Author: WDH

GridCollectionFunction(GridCollection & gc,
const Range & R0 =nullRange,
const Range & R1 =nullRange,
const Range & R2 =nullRange,

const Range & R3 =nullRange,
const Range & R4 =nullRange,
const Range & R5 =nullRange,
const Range & R6 =nullRange,
const Range & R7 =nullRange)

Description: This constructor takes ranges, the first 3 "nullRange" values are taken to be the coordinate directions in the grid function. Each grid function is dimensioned according to the dimensions found with the `MappedGrid` found in the `GridCollection`. Grid functions can have up to 8 dimensions, the index positions not used by the coordinate dimensions can be used to store different components. For example, a *vector* grid functions would use 1 index position for components while a *matrix* grid functions would use two index positions for components.

grid0 (input): GridCollection to associate this grid function with.

R0, R1, R2, ... (input): Ranges to determine the shape and size of the grid function. An int can also be used instead of a Range.

Examples: Here are some examples

```
//  R1 = range of first dimension of the grid array
//  R2 = range of second dimension of the grid array
//  R3 = range of third dimension of the grid array

GridCollection gc(...);

Range all;    // null Range is used to specify where the coordinates are

GridCollectionFunction u(gc);                          //  --> u[grid](R1,R2,R3);

GridCollectionFunction u(gc,all,all,all,1);            //  --> u[grid](R1,R2,R3,0:1);
GridCollectionFunction u(gc,all,all,Range(1,1));       //  --> u[grid](R1,R2,1:1,R3);

GridCollectionFunction u(gc,2,all);                    //  --> u[grid](0:2,R1,R2,R3);
GridCollectionFunction u(gc,Range(0,2),all,all,all);   //  --> u[grid](0:2,R1,R2,R3);
GridCollectionFunction u(gc,all,Range(3,3),all,all);   //  --> u[grid](R1,3:3,R2,R3);
```

Author: WDH

GridCollectionFunction(GridCollection & gc,
const GridFunctionParameters::GridFunctionType & type,
const Range & component0 =nullRange,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: This constructor is used to create a grid function of some standard type. The standard types are defined in the GridFunctionParameters::GridFunctionType enum,

- vertexCentered : grid function is vertex centred
- cellCentered : grid function is cell centred
- faceCenteredAll : grid function components are face centred in all directions
- faceCenteredAxis1 : grid function is face centred along axis1
- faceCenteredAxis2 : grid function is face centred along axis2
- faceCenteredAxis3 : grid function is face centred along axis3
- general : means same as vertexCentered when used in this constructor

grid0 (input): Use this GridCollection.

type (input): Make this type of grid function.

component0, component1,... (input): supply a Range for each component.

Examples: Here are some examples:

```
GridCollection gc(...);
realGridCollectionFunction u(gc,GridFunctionParameters::vertexCentered,2);   // u(gc,all,all,all,2);
realGridCollectionFunction u(gc,GridFunctionParameters::cellCentered,2,3);   // u(gc,all,all,all,2,3);
realGridCollectionFunction u(gc,GridFunctionParameters::faceCenteredAll,2);  // u(gc,all,all,all,2,faceRange);
realGridCollectionFunction u(gc,GridFunctionParameters::faceCenteredAll,3,2);// u(gc,all,all,all,3,2,faceRange);
```

Author: WDH

3.5.5 breakReference

void
breakReference()

Description: This member function will cause the grid function to no longer be referenced. The grid function acquires its own copy of the data.

Author: WDH

3.5.6 operator()(Range,...)

GridCollectionFunction
operator()(const Range & component0,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: Return a new GridCollectionFunction that is linked (using the `link` function) to some specfied components of the current GridCollectionFunction. This is a convenient but **inefficient** way to easily access certain components of a multi-component GridCollectionFunction as in the example:

```
GridCollection gc(...);
floatGridCollectionFunction u(gc,all,all,all,2);
u(0)=1.;   // set component 0 of u to be 1.
u(1)=2.;   // set component 1 of u to be 2.
u(0)=u(0)*2.+u(1)*u(0);
```

The above code is inefficient since a new gridCollectionFunction is built every time an expression like `u(0)` appears.

This has the same effect as the following (more efficient but not as cute) code:

```
GridCollection gc(...);
floatGridCollectionFunction u(gc,all,all,all,2), u0,u1;
u0.link(u,Range(0,0));
u1.link(u,Range(1,1));
u0=1.;
u1=2.;
u0=u0*2.+u1*u0;
```

3.5.7 consistencyCheck

void
consistencyCheck() const

Description: Perform a consistency check on the grid function.

Return values: Return 0 if the grid function appears to be ok.

3.5.8 dataCopy

int
dataCopy(const GridCollectionFunction & gcf)

Description: copy the array data only

gcf (input): set the array data equal to the data in this grid function.

3.5.9 Derivatives: x,y,z,xx,xy,xz,yy,yz,zz,laplacian,grad,div

GridCollectionFunction
derivative(const Index & component0 =nullIndex ,
const Index & component1 =nullIndex ,
const Index & component2 =nullIndex ,
const Index & component3 =nullIndex ,
const Index & component4 =nullIndex
)

Description: derivative equals one of x,y,z,xx,xy,xz,yy,yz,zz,laplacian,grad,div. Return the derivative of this grid function. This routine just calls the function of the same name in the GenericGridCollectionOperators.

component0,component1,... (input) : optional arguments to specify which components should be computed. The other components will be returned as zero.

component0,component1,... (input) : optional arguments to specify which components should be computed. The other components will be returned as zero.

Return value: The derivative is returned as a new grid function. For all derivatives but `grad` and `div` the number of components in the result is equal to the number of components specified by component0,... (if component0 etc are not specified then the result will have the same number of components of the grid function being differentiated). The `grad` operator will have number of components equal to the number of space dimensions while the `div` operator will have only one component.

GridCollectionFunction
derivative(const GridFunctionType & gfType,
const Index & component0 =nullIndex ,
const Index & component1 =nullIndex ,
const Index & component2 =nullIndex ,
const Index & component3 =nullIndex ,
const Index & component4 =nullIndex
)

Description: derivative equals one of x,y,z,xx,xy,xz,yy,yz,zz,laplacian,grad,div. Return the derivative of this grid function. The argument gfType determines the type of the grid function that is returned. This routine just calls the function of the same name in the GenericGridCollectionOperators (see also setOperators).

gfType (input): The type of the grid function to be returned.

component0,component1,... (input) : optional arguments to specify which components should be computed. The other components will be returned as zero.

Return value: The derivative is returned as a new grid function. For all derivatives but `grad` and `div` the number of components in the result is equal to the number of components specified by component0,... (if component0 etc are not specified then the result will have the same number of components of the grid function being differentiated). The `grad` operator will have number of components equal to the number of space dimensions while the `div` operator will have only one component.

3.5.10 Derivative Coefficients: xCoefficient,yCoefficient,...

GridCollectionFunction
Derivative(const Index & component0 =nullIndex ,
const Index & component1 =nullIndex ,
const Index & component2 =nullIndex ,
const Index & component3 =nullIndex ,
const Index & component4 =nullIndex
)

Description: Derivative equals one of xCoefficient,yCoefficient,zCoefficient,xxCoefficient, xyCoefficient,xz-Coefficient,yyCoefficient,yzCoefficient,zzCoefficient, laplacianCoefficient,gradCoefficient,divCoefficient. Return the coefficients of the derivative. This routine just calls the function of the same name in the GenericGridCollectionOperators.

GridCollectionFunction
Derivative(const GridFunctionType & gfType,
const Index & component0 =nullIndex ,
const Index & component1 =nullIndex ,
const Index & component2 =nullIndex ,
const Index & component3 =nullIndex ,
const Index & component4 =nullIndex
)

Description: Derivative equals one of xCoefficient,yCoefficient,zCoefficient,xxCoefficient, xyCoefficient,xz-Coefficient,yyCoefficient,yzCoefficient,zzCoefficient, laplacianCoefficient,gradCoefficient,divCoefficient. Return the coefficients of the derivative. The argument gfType determines the type of the grid function that is returned. This routine just calls the function of the same name in the GenericGridCollectionOperators (see also setOperators).

gfType (input): The type of the grid function to be returned.

3.5.11 destroy

int
destroy()

Description: destroy this grid function. (Release all memory)

3.5.12 display

void
display(const aString & label =nullString,
const aString & format =nullString) const

void
display(const aString & label =nullString,
FILE *file = NULL,
const aString & format =nullString) const

Description: Display the grid function, print the values of in all the components.

label (input): optional label to print.

file (input): print to this file

format (input): use this format for printf

void
display(const aString & label, const DisplayParameters & displayParameters) const

Description: Display the grid function, print the values of in all the components.

label (input): optional label to print.

displayParameters (input): specify parameters

3.5.13 evaluate

GridCollectionFunction
evaluate(GridCollectionFunction & cgf)

Description: Due to the way that temporaries are handled it is necessary to use this function on expressions involving grid collection functions that are passed as arguments to function. Example:

```
realGridCollectionFunction u(gc),v(cg);
...
myFunction(evaluate(u+v));
...
```

If the `evaluate` function were not used there could be a possible memory leak.

cgf (input): If this grid function is a temporary,

Return value: A grid collection function equal to cgf which is not a temporary

3.5.14 get

int
get(const GenericDataBase & dir, const aString & name)

Description: Get from a database file. Example:

```
HDF_DataBase db;
db.mount("myFile.hdf","R");
GridCollection gc;
realGridCollectionFunction u;
initializeMappingList();
gc.get(db,"my grid");
u.updateToMatchGrid(gc);   // **NOTE**
u.get(db,"u");
```

dir (input): get from this directory of the database.

name (input): the name of the grid function on the database.

NOTE: This get function will not set the pointer to the Grid associated with this grid function. You should call updateToMatchGrid(...) to set the grid BEFORE using this function.

3.5.15 getClassName

aString
getClassName() const

Description: Return the class name.

3.5.16 getComponentBase

int
getComponentBase(int component) const

Description: Get the base for the given component.

component (input): component number, 0,1,...

Return Values: The base for the component. Unused components have base=0 and bound=0

3.5.17 getComponentBound

int
getComponentBound(int component) const

Description: Get the bound for the given component.

component (input): component number, 0,1,...

Return Values: The bound for the component. Unused components have base=0 and bound=0

3.5.18 getComponentDimension

int
getComponentDimension(int component) const

Description: Get the dimension for the given component, dimension=bound-base+1

component (input): component number, 0,1,...

Return Values: The dimension for the component. Unused components have dimension=1

3.5.19 getCoordinateBase

int
getCoordinateBase(int coordinate) const

Description: Get the base for the given coordinate.

coordinate (input): component number, 0,1, or 2.

Return Values: The base for the coordinate. Unused coordinates have base=0 and bound=0

3.5.20 getCoordinateBound

int
getCoordinateBound(int coordinate) const

Description: Get the bound for the given coordinate.

coordinate (input): component number, 0,1, or 2.

Return Values: The bound for the coordinate. Unused coordinates have base=0 and bound=0

3.5.21 getCoordinateDimension

int
getCoordinateDimension(int coordinate) const

Description: Get the dimension for the given coordinate, dimension = bound-base+1

coordinate (input): component number, 0,1, or 2.

Return Values: The dimension for the coordinate. Unused coordinates have dimension=1

3.5.22 getFaceCentering

faceCenteringType
getFaceCentering() const

Description: Get the type of face centering. For further explanation see `setFaceCentering` and section4.

Errors: none.

Return Values: faceCenteringType.

Author: WDH

3.5.23 getGridCollection

GridCollection*
getGridCollection(const bool abortIfNull =TRUE) const

Description: Return a pointer to the GridCollection that this grid function is asscoaiated with By default this function will abort if the pointer is NULL.

Return values: A pointer to a GridCollection or NULL

3.5.24 getGridFunctionType

GridFunctionType
getGridFunctionType(const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex) const

Description: return the type of the grid function

component0,component1,... (input): get type of the grid function corresponding to these components.

Return Values: The grid function type, one of the enums in GridFunctionParameters::GridFunctionType.

Author: WDH

3.5.25 getGridFunctionTypeWithComponents

GridFunctionTypeWithComponents
getGridFunctionTypeWithComponents(const Index & c0 =nullIndex,
const Index & c1 =nullIndex,
const Index & c2 =nullIndex,
const Index & c3 =nullIndex,
const Index & c4 =nullIndex) const

Description: return the type of the grid function with the number of components

c0,c1,... (input): get type of the grid function corresponding to these components.

Return Values: The grid function type with number of components, one of the enums in GridFunctionParameters::GridFunctionTypeWithComponents.

Note: In a faceCenteredAll grid function, the position taken by the faceRange does not count as a component.

```
GridCollection gc(...);
Range all;
floatGridCollectionFunction u(mg,floatGridCollectionFunction::faceCenterAll,2);
u.getGridFunctionTypeWithComponents(); // == faceCenterAllWith1Component
u.getNumberOfComponents();             // == 1
```

Author: WDH

3.5.26 getIsCellCentered

bool
getIsCellCentered(const Index & axis0 =nullIndex,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex,
const Index & grid0 =nullIndex) const

Description: Determine the cell centeredness of a grid function. See the detail description with te MappedGridFunction version of this function.

axis0 (input): if axis0=nullIndex (default) then all axes are checked

component0 (input): if component0=nullIndex (default) then all components are checked

component1 (input): if component1=nullIndex (default) then all components are checked

component2 (input): if component2=nullIndex (default) then all components are checked

component3 (input): if component3=nullIndex (default) then all components are checked

component4 (input): if component4=nullIndex (default) then all components are checked

Return Values: TRUE or FALSE

Author: WDH

3.5.27 getIsFaceCentered

bool
getIsFaceCentered(const int & axis0 =forAll ,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex,
const Index & grid0 =nullIndex) const

Description: Determine if a given component of this grid function is face-centred along a given axis. By default check all axes and all components.

axis0: check if the components are face centred along this axis. By default check if the components are face centred in ANY direction.

component0, component1,... (input): check the value for these components, by default check all components.

3.5.28 getName

aString
getName(const int & component0 =defaultValue,
const int & component1 =defaultValue,
const int & component2 =defaultValue,
const int & component3 =defaultValue,
const int & component4 =defaultValue) const

Description: Get the name of the grid function or a component as in

```
aString nameOfGridFunction = u.getName();
aString nameOfComponent0   = u.getName(0);
aString nameOfComponent1   = u.getName(1);
```

name: the name of the grid function or component.

component0, component1, (input): get the name for this component. if all of component0,component1,component2 ==defaultValue then the name of the grid function is returned. Otherwise the default value becomes the base value for that component.

3.5.29 getNumberOfComponents

int
getNumberOfComponents() const

Description: return the number of components (0=scalar, 1=vector, 2=matrix, ...).

Return Values: Valid values are 0,...,5

Examples: Here are some examples. Note the special case for grid functions created with a `faceRange`, the `faceRange` position does NOT count as a component.

```
GridCollection gc(...);
Range all;
floatGridCollectionFunction u(gc);                          // 0 components
floatGridCollectionFunction u(gc,all,all,all);              // 0 components
floatGridCollectionFunction u(gc,all,all,all,1);            // 1 component
floatGridCollectionFunction u(gc,all,all,all,2,2);          // 2 components
floatGridCollectionFunction u(gc,all,all,all,faceRange);    // 0 components
floatGridCollectionFunction u(gc,all,all,all,3,faceRange);  // 1 component
```

Author: WDH

3.5.30 getOperators

GridCollectionOperators*
getOperators() const

Description: get the operators used with this grid function. Return NULL if there are none.

3.5.31 interpolate

int
interpolate(const Range & C0 = nullRange,
const Range & C1 = nullRange,
const Range & C2 = nullRange)

Description: Interpolate using default Interpolant or one found in the grid collection

C0, C1, C2 (input): optionally specify components to interpolate. For example `u.interpolate(Range(1,2))` to interpolate components 1 and 2.

Return Values: 0=success, > 0 indicates an error.

Author: WDH

3.5.32 isNull

bool
isNull()

Description: Return TRUE if this grid function is null (has no grid associated with it).

Return value: Return TRUE if this grid function is null, otherwise return FALSE.

3.5.33 link

void
link(const GridCollectionFunction & gcf,
const Range & R0 = nullRange,
const Range & R1 = nullRange,
const Range & R2 = nullRange,
const Range & R3 = nullRange,
const Range & R4 = nullRange)

Description: The `link` member function can be used to link a grid function to a specific component of another grid function.

mgf (input): link to this

R0, R1, ..., R4 (input): indicate which components to link to. Note that the Ranges for the linked grid function always start at 0. Use updateToMatchGridFunction to change this.

Examples: See the examples in the documentation for the MappedGridFunction version of link.

Errors: Attempt to link to invalid components.

Return Values: none.

Notes: The linkee function will acquire the same operators as the function being linked to.

Author: WDH

3.5.34 multiply

GridCollectionFunction &
multiply(const GridCollectionFunction & a, const GridCollectionFunction & coeff_)

Description: Multiply a grid function times a coefficient matrix. Use this function to multiply a scalar grid function "a" times a coefficient matrix "coeff". The result is saved in coeff and returned by reference.

```
coeff[grid](M,I1,I2,I3) <- a[grid](I1,I2,I3)*coeff[grid](M,I1,I2,I3)
```

This is a non-member function and is called with

```
multiply(u,coeff)
```

a (input) : a scalar grid function.

coeff (input) : a grid function in the shape a coefficient matrix (1 component in position 0)

Return value: coeff is returned by reference

Notes: If "a" is an expression (`multiply(u+v,coeff)`) then this function will properly delete "a". Note that one should call "evaluate" on an expression that is being passed to a function that is not a member function of this class.

3.5.35 numberOfMultigridLevels

int
numberOfMultigridLevels() const

Description: Return the number of multigrid levels contained in this grid function. See the grid documentation for further details.

Author: WDH

3.5.36 numberOfRefinementLevels

int
numberOfRefinementLevels() const

Description: Return the number of refinement levels contained in this grid function. See the grid documentation for further details.

Author: WDH

3.5.37 operator = GridCollectionFunction

GridCollectionFunction &
operator= (const GridCollectionFunction & cgf)

Description: Set one grid-function equal to another. This is a shallow copy where only the array data is copied. An error occurs if the two grid functions do not have the same number of grids or the array data in each mappedGridFunction are not conformable. This operation has the same affect as the `dataCopy` member function. An exception to this rule is when the grid function to the left of the equals operator is a 'null' grid function (one that has no grid associated with it such as a grid function built by the default constructor). In this case a deep copy is performed.

Examples: Here are some examples for a realCompositeGridFunction. The examples are the same for GridCollectionFunction's.

```
CompositeGrid cg(...);
realCompositeGridFunction u(cg),v(cg),w.
u=1.;
v=u;                 // only the array data is copied.
w=u;                 // this is a DEEP copy since w is null.
u=v+w;               // u will steal the data from the temporary 'v+w'
u=3;                 // does NOT call this = operator, uses grid-function=scalar
u.dataCopy(v+w);     // only copies array data
u.updateToMatchGridFunction(v);  // this is a real deep copy.
realCompositeGridFunction a = u;  // does NOT call this = operator, calls copy constructor
```

Author: WDH

3.5.38 periodicUpdate

void
periodicUpdate(const Range & C0 =nullRange,
const Range & C1 =nullRange,
const Range & C2 =nullRange,
const Range & C3 =nullRange,
const Range & C4 =nullRange,
const bool & derivativePeriodic =FALSE)

Description: Swap periodic edges of the grid function. Assign values to `side=1` boundary lines

$$\texttt{i}_{\texttt{axis}} = \texttt{mg.gridIndexRange}()(1,\texttt{axis}) \quad \texttt{axis} = 0,1,..,\texttt{mg.numberOfDimensions}$$

(`mg` is the `MappedGrid` associated with this grid function) as well as all ghost lines on all sides that have periodic boundary conditions.

C0,C1,...C4 (input) : specify which components to update. By default update all components.

derivativePeriodic (input): if TRUE we assume that the grid function is not actually periodic but that only it's derivative is. *** This is not implemented yet ***

3.5.39 put

int
put(GenericDataBase & dir, const aString & name) const

Description: Output the grid function onto a database file.

dir (input): put onto this directory of the database.

name (input): the name of the grid function on the database.

Notes:

First some definitions

- N=`maximumNumberOfIndicies` The maximum number of dimensions in a grid function (current value =8).
- N_A=`numberOfIndicies` The maximum number of A++ dimensions (current value =4).

Here are the items that are saved in a data base.

numberOfComponentsGrids (int) The number of component grids. This is equal to the number of `mappedGridFunctions` that are in the grid collection.

positionOfComponent (IntegerArray(N)) The positions (base 0) of the component positions are saved in `positionOfComponent(i)`, $i = 0, 1, ..., N$.

positionOfCoordinate (IntegerArray(N)) The positions (base 0) of the 3 coordinate positions are saved in `positionOfCoordinate(i)`, $i = 0, 1, 2$. The default value and the value for unused entries is N=`maximumNumberOfIndicies`.

positionOfFaceCentering (int) For a face centred grid function of standard type this is the position of the face centering. For all other types of grid functions this has a value of -1.

faceCentering (enum faceCenteringType) The face centering type for the grid function. Default value is `none`= -1.

numberOfDimensions The number of space dimensions, 0, 1, or 2.

isACoefficientMatrix (bool) If TRUE (=1) then this is a coefficient matrix, default is FALSE (=0).

stencilType (enum StencilTypes) The type of stencil for a coefficient matrix, default is `standardStencil` (=0).

stencilOffset (int) The stencil offset for a coefficient matrix, default value = 0.

stencilWidth (int) The stencil width for a coefficient matrix, default value = 0.

R[i].base (int) (`i=0,1,...,N`) The base of the Range object `R[i]` which holds the base and bound for position i. For unused positions the default is 0. There is one extra Range, `R[N]=Range(0,0)` which exists just for convenience.

R[i].bound (int) (`i=0,1,...,N`) The bound of the Range objects `R[i]`. For unused positions the default is 0.

numberOfNames (int) The number of names that are saved. (see next item).

name[i] (aString) (`i=0,1,...,numberOfNames-1`) The names for the grid function and its components.

mappedGridFunctionList[i] (`i=0,1,...,numberOfGrids-1`) The `mappedGridFunctions` that are found in this grid collection function. See the documentation on the `put` member function for a `mappedGridFunction`

3.5.40 reference

void
reference(const GridCollectionFunction & cgf)

Description: Use this function to reference one GridCollectionFunction to another. When two (or more) grid functions have been referenced they share the same array data so that changes to one grid function will change all the other referenced grid functions. Only the array data is referenced. Other properties of the grid function such as cell-centredness can be changed in the referenced grid function. The "shape" of the referenced grid function can also be changed without changing the referencee:`cgf`.

Author: WDH

3.5.41 setBoundaryConditionValue

void
setBoundaryConditionValue(const real & value,
const int & component =forAll,
const int & index =forAll,
const int & side =forAll,
const int & axis =forAll,
const int grid0 =forAll
)

Description: Set some values for boundary conditions. This routine just calls the function of the same name in the MappedGridOperators.

3.5.42 setFaceCentering

void
setFaceCentering(const int & axis =defaultValue)

Description: Set the type of face centering, the behaviour of this function depends on whether the argument "axis" has been specified or else if the current value for getFaceCentering().

axis (input): 1. if "axis" is given then make all components face centred in direction=axis

2. if getFaceCentering()==all : make components face centered in all directions, the grid function should have been contructed or updated using the faceRange to specify which Index is to be used for the "directions"

For further explanation see section4.

Author: WDH

3.5.43 setInterpolant

void
setInterpolant(Interpolant *interpolant)

Description: Set a pointer to an interpolant to use. This will NOT change the interpolant associated with the GridCollection.

3.5.44 setIsCellCentered

void
setIsCellCentered(const bool trueOrFalse,
const Index & axis0 =nullIndex,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex,
const Index & grid0 =nullIndex)

Description: Change the cell centered-ness of the grid function. By default set all components and all grids.

trueOfFalse (input): make cell-centred or not.

axis0: set the value for this axis, by default set all axes.

component0, component1, (input): set the value for these components, by default set all components.

grid0 (input): set this s component grid.

3.5.45 setIsFaceCentered

setIsFaceCentered(const int & axis0 =forAll ,
const Index & component0 =nullIndex,
const Index & component1 =nullIndex,
const Index & component2 =nullIndex,
const Index & component3 =nullIndex,
const Index & component4 =nullIndex,
const Index & grid0 =nullIndex)

Description: Make a component of a grid function face centred along the given axis

axis0: set the value for this axis, by default set all axes.

component0, component1, (input): set the value for these components, by default set all components.

3.5.46 setName

void
setName(const aString & name,
const int & component0 =defaultValue,
const int & component1 =defaultValue,
const int & component2 =defaultValue,
const int & component3 =defaultValue,
const int & component4 =defaultValue)

Description: Set the name of the grid function or a component as in

```
u.setName("nameOfGridFunction");
u.setName("nameOfComponent0",0);
u.setName("nameOfComponent1",1);
```

name: the name of the grid function or component.

component0, component1,... (input): give the name for this component. if all of component0,component1,component2 ==defaultValue then the name of the grid function is set. Otherwise the default value becomes the base value for that component.

3.5.47 setOperators

void
setOperators(GenericCollectionOperators & operators0)

Description: Supply a derivative object to use for computing derivatives on all component grids. This operator is used for the member functions .x .y .z .xx .xy etc.

operators0: use these operators.

3.5.48 updateCollections

int
updateCollections()

Description: Update the refinementLevel (and eventually other collections) This is a protected member.

3.5.49 updateToMatchGrid

updateReturnValue
updateToMatchGrid()

updateReturnValue
updateToMatchGrid(GridCollection & gridCollection0)

updateReturnValue
updateToMatchGrid(const Range & R0 = nullRange,
const Range & R1 = nullRange,
const Range & R2 = nullRange,
const Range & R3 = nullRange,
const Range & R4 = nullRange,
const Range & R5 = nullRange,
const Range & R6 = nullRange,
const Range & R7 = nullRange)

updateReturnValue
updateToMatchGrid(GridCollection & gc,
const Range & R0,
const Range & R1 =nullRange,
const Range & R2 =nullRange,
const Range & R3 =nullRange,
const Range & R4 =nullRange,
const Range & R5 =nullRange,
const Range & R6 =nullRange,
const Range & R7 =nullRange)

Description: Update a grid function. Optionally specify a new grid and new Ranges.

gc (input): update to match this grid.

R0, R1, ... (input): Use these Range objects to determine the grid function dimensions.

Return Values: Return a value from the enumerator `updateReturnValue`:

```
enum updateReturnValue  // the return value from updateToMatchGrid is a mask of the following val
{
  updateNoChange          = 0, // no changes made
  updateReshaped          = 1, // grid function was reshaped
  updateResized           = 2, // grid function was resized
  updateComponentsChanged = 4  // component dimensions may have changed (but grid was not resized
};
```

Author: WDH

updateReturnValue
updateToMatchGrid(GridCollection & gridCollection0,
const GridFunctionParameters::GridFunctionType & type,
const Range & component0,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: Use this update function to create a grid function of a given type. See the comments in the corresponding constructor.

updateReturnValue
updateToMatchGrid(const GridFunctionType & type,
const Range & component0,
const Range & component1 =nullRange,
const Range & component2 =nullRange,
const Range & component3 =nullRange,
const Range & component4 =nullRange)

Description: Use this update function to create a grid function of a given type. See the comments in the corresponding constructor.

updateReturnValue
updateToMatchGrid(GridCollection & gridCollection0,
const GridFunctionParameters::GridFunctionType & type)

Description: Use this update function to create a grid function of a given type, the components are left unchanged.

updateReturnValue
updateToMatchGrid(const GridFunctionType & type)

Description: Use this update function to create a grid function of a given type, the components are left unchanged.

updateReturnValue
updateToMatchGridFunction(const GridCollectionFunction & cgf)

updateReturnValue
updateToMatchGridFunction(const GridCollectionFunction & cgf,
const Range & R0,
const Range & R1 =nullRange,
const Range & R2 =nullRange,
const Range & R3 =nullRange,
const Range & R4 =nullRange,
const Range & R5 =nullRange,
const Range & R6 =nullRange,
const Range & R7 =nullRange)

Description: Update this grid function to match another grid function (this is like using the = operator but it avoids copying the array data)

cgf (input): match to this grid function.

R0, R1, ... (input): optional ranges to change the dimensions.

updateToMatchnumberOfGrids

updateReturnValue
updateToMatchNumberOfGrids(GridCollection & gc)

Purpose: Update the GridCollectionFunction so that it has the correct number of components. The components are not dimensioned correctly.

updateToMatchComponentGrids

updateReturnValue
updateToMatchComponentGrids()

Purpose: Update the grid collection to match the component grids

3.5.50 sizeOf

real
sizeOf(FILE *file = NULL) const

Description: Return number of bytes allocated by this object; optionally print detailed info to a file

file (input) : optinally supply a file to write detailed info to. Choose file=stdout to write to standard output.

Return value: the number of bytes.

3.5.51 fixupUnusedPoints

int
fixupUnusedPoints(const RealArray & value = nullRealArray(),
int numberOfGhostlines =1)

Description: Assign values to points on a grid function that correspond to unused points (mask==0). By default all unused points are set to zero. Use the value array to set unused points to particular values.

value (input) : if supplied, assign value(n) to unused points of component n and do not change any components not found in value. If not supplied set all unused points to zero.

numberOfGhostLines (input) : Indicate how many ghost lines are used in the computation. Other ghost line values will all be set to zero.

3.5.52 Examples

In this example we make a GridCollectionFunction from a GridCollection. We assign the component grid functions (accessed using [ ]) in A++ style. We show how to use the `reference` and `breakReference` functions.

```
  ...

   GridCollection gc(...);  // here is a GridCollection or CompositeGrid
   doubleGridCollectionFunction u(gc),v;
   u[0]=5.;                 // mapped grid function for grid 0
   Index I(0,10);
   u[1](I,I)=3.;            // mapped grid function for grid 1
   v.reference(u);          // v is referenced to u
   v[1]=7.;                 // changes u as well
   v.breakReference();      // v is no longer referenced to u

   // Here is how to dimension a grid function after it has been declared:
   floatGridCollectionFunction w;
   w.updateToMatchGrid( gc,all,all,all,Range(0,1) );
   ...
   // give names to the grid function and components
   w.setName("w");      // name grid function
   w.setName("w.0",0);  // name component 0
   w.setName("w.1",1);  // name component 1
   cout << w.getName() << "," << w.getName(0) << "," << w.getName(1) << endl;
}
```

4 Cell-centred and Face-centred Grid Functions

In this section we discuss cell-centred and face-centred grid functions – how to create them and how to use them.

In a vertex-centred grid function the solution values are defined at the vertices of the grid (i.e. at the positions defined by the `vertex` array). By default a grid function defined on a vertex-centred grid will be vertex centered.

In a cell-centred grid function the solution values are assumed to be defined at the centers of the cells. By default a grid function defined on a cell-centred grid will be cell-centred.

In a face-centred grid function the solution values are defined on the face of the cell. In 2D there are two possible types of face-centered grid functions and in 3D there are three possibilities. A face centered grid function is vertex centred in one direction and cell centred in the others.

In three space dimensions a grid function could also be defined on an edge. An edge-centred grid function is vertex-centred in two directions and cell centred in one.

4.1 Creating face/cell/vertex centred grid functions in standard form

The easiest way to create any of the commonly used grid functions is to use the constructor that takes a `GridFunctionType` and optional Ranges to specify components:

```
MappedGridFunction(MappedGrid & mg,
                   const GridFunctionParameters::GridFunctionType & type,
                   const Range & component0=nullRange,
                   const Range & component1=nullRange,
                   const Range & component2=nullRange,
                   const Range & component3=nullRange,
                   const Range & component4=nullRange )
```

This constructor is used to create a grid function of some common types in *standard form*. A grid function in standard form will have the first 3 index positions occupied by the coordinate directions. Any component indices will follow the coordinate indicies. In addition to the constructor there is a corresponding `updateToMatchGrid` function that can be used to change the type and/or component dimensions of a grid function that has already been constructed.

The standard types of grid functions are defined in the enum GridFunctionType, (this enum is found in the class GridFunctionParameters),

```
enum GridFunctionType
{
  general,
  vertexCentered,
  cellCentered,
  faceCenteredAll,
  faceCenteredAxis1,
  faceCenteredAxis2,
  faceCenteredAxis3
  };
```

where

- vertexCentered : grid function is vertex centred
- cellCentered : grid function is cell centred
- faceCenteredAll : the grid function has components that are face centred in each direction
- faceCenteredAxis1 : grid function is face centred along axis1, (axis1==0)
- faceCenteredAxis2 : grid function is face centred along axis2, (axis2==1)
- faceCenteredAxis3 : grid function is face centred along axis3, (axis3==2)

A grid function with type `faceCenteredAxis1` will be vertex centered along `axis1` and cell-centered along the other axes. Thus in 2D the grid function will live on the vertical faces of the grid cells as shown in figure 6.
Similiarly a grid function with type `faceCenteredAxis2` will be vertex centered along `axis2` and cell-centered along the other axes. Thus in 2D the grid function will live on the horizontal faces of the grid cells as shown in figure 7.
A grid function that is defined to be `faceCenteredAll` will have components that live on each of the faces. This is accomplished by adding an extra component index to the grid function; the dimension of this extra component being

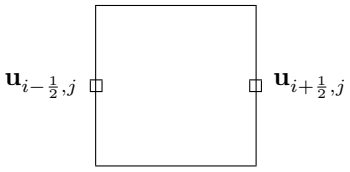

Figure 6: A grid function of type `faceCenteredAxis1`

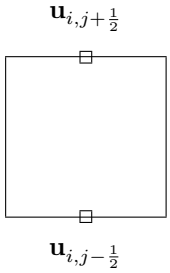

Figure 7: A grid function of type `faceCenteredAxis2`

equal to the number of space dimensions (i.e. the number of different faces). Thus for example, one could make a grid function, `faceNormals`, that holds vectors that are normals to the faces of every cell. Some of the normals are `faceCenteredAxis1`, some are `faceCenteredAxis2` and some are `faceCenteredAxis3`.

The next code fragment

```
MappedGrid mg(...);    // create a grid some-how
realMappedGridFunction u(mg,GridFunctionParameters::cellCentered,2);
```

will create a cell-centered grid function with two components. The name `cellCentered` is defined within the Class `GridFunctionParameters` (to avoid polluting the global name space) and so we must indicate its scope when we use it. Below we show how to do this in an easier way. The above declaration would be equivalent to

```
MappedGrid mg(...); // cell-centered MappedGrid
Range all;
realMappedGridFunction u(mg,all,all,all,2);
```

assuming that the `MappedGrid` were cell centered. Another way to do this would be to use the `updateToMatchGrid` member function

```
MappedGrid mg(...);
realMappedGridFunction u;
u.updateToMatchGrid(mg,GridFunctionParameters::cellCentered,2);
```

Before presenting any further examples let us first define some variables `vertexCentered`, `cellCentered`, etc. so that we do not have to indicate the scope, `GridFunctionParameters::vertexCentered`, when refering to entries in the GridFunctionType enum,

```
 const GridFunctionParameters::GridFunctionType vertexCentered    =GridFunctionParameters::vertexCentered;
 const GridFunctionParameters::GridFunctionType cellCentered      =GridFunctionParameters::cellCentered;
 const GridFunctionParameters::GridFunctionType faceCenteredAll   =GridFunctionParameters::faceCenteredAll;
 const GridFunctionParameters::GridFunctionType faceCenteredAll   =GridFunctionParameters::faceCenteredAll;
 const GridFunctionParameters::GridFunctionType faceCenteredAxis1 =GridFunctionParameters::faceCenteredAxis1;
 const GridFunctionParameters::GridFunctionType faceCenteredAxis2 =GridFunctionParameters::faceCenteredAxis2;
 const GridFunctionParameters::GridFunctionType faceCenteredAxis3 =GridFunctionParameters::faceCenteredAxis3;
```

Given these definitions here are some examples

```
realMappedGridFunction u(mg,vertexCentered,2);   // u(mg,all,all,all,0:1);
realMappedGridFunction u(mg,cellCentered,2,3);   // u(mg,all,all,all,0:1,0:2);

realMappedGridFunction u(mg,faceCenteredAxis1);  // u(mg,all,all,all);
realMappedGridFunction u(mg,faceCenteredAxis2,Range(2,3));  // u(mg,all,all,all,2:3);
```

```
realMappedGridFunction u(mg,faceCenteredAll);    // u(mg,all,all,all,0:d-1);
realMappedGridFunction u(mg,faceCenteredAll,2);  // u(mg,all,all,all,0:1,0:d-1);
realMappedGridFunction u(mg,faceCenteredAll,3,2);// u(mg,all,all,all,0:2,0:1,0:d-1);
```

The comment following each declaration defines the shape of the resulting grid function. (Here `d` is the number of space dimensions). Note that the grid functions created always have the 3 coordinate indices first, followed by the component indices (*standard form*). Also note that the grid functions with type `faceCenteredAll` have an extra index added to the end. The value of this index determines the face centered-ness. Thus, in the grid function declared `u(mg,faceCenteredAll)` the component,`u(all,all,all,i)`, will be face centered along axis=i, for $i = 0, 1, ..., d-1$, as shown again below

```
realMappedGridFunction u(mg,faceCenteredAll);

u(all,all,all,0)  <---> faceCenteredAxis1
u(all,all,all,1)  <---> faceCenteredAxis2   (if d>1)
u(all,all,all,2)  <---> faceCenteredAxis3   (if d>2)
```

Here are some more examples where we also indicate the value returned by some of the query member functions

```
realMappedGridFunction u(mg,faceCenteredAxis2,Range(0,1),Range(1,1)); // u(mg,all,all,all,0:1,1:1)
u.getGridFunctionType()               == faceCenteredAxis2
u.getGridFunctionTypeWithComponents() == faceCenteredAxis2With2Components
u.getNumberOfComponents()             == 2    // number of component indicies, 0=scalar, 1=vector, 2=matrix
u.getFaceCentering()                  == direction1


realMappedGridFunction u(mg,faceCenteredAxis1); // u(mg,all,all,all)
u.getGridFunctionType()               == faceCenteredAxis1
u.getGridFunctionTypeWithComponents() == faceCenteredAxis1With0Components
u.getNumberOfComponents()             == 0
u.getFaceCentering()                  == direction0


realMappedGridFunction u(mg,faceCenteredAll,3); // u(mg,all,all,all,0:2,0:d-1)
u.getGridFunctionType()               == faceCenteredAll
u.getGridFunctionTypeWithComponents() == faceCenteredAllWith1Component
u.getNumberOfComponents()             == 1
u.getFaceCentering()                  == all
```

If you need to determine the gridFunctionType for a subset of the components of a grid function then you can pass optional arguments to the `getGridFunctionType` member function to indicate which components you want to determine the type of,

```
realMappedGridFunction u(mg,faceCenteredAll,Range(0,1)); // u(mg,all,all,all,0:1,0:d-1)

u.getGridFunctionType()             == faceCenteredAll
u.getGridFunctionType(0)            == faceCenteredAll
u.getGridFunctionType(Range(0,1))   == faceCenteredAll

u.getGridFunctionType(0,0)          == faceCenteredAxis1
u.getGridFunctionType(Range(0,1),0) == faceCenteredAxis1
u.getGridFunctionType(0,1)          == faceCenteredAxis2    (if d>1)
u.getGridFunctionType(0,2)          == faceCenteredAxis3    (if d>2)
```

In the above example we see that a grid function of type `faceCenteredAll` has some components that are `faceCenteredAxis1`, some that are `faceCenteredAxis2` and some that are `faceCenteredAxis3`.

You can change the `gridFunctionType` of an existing grid function by using `updateToMatchGrid` as shown in the next example:

```
realMappedGridFunction u(mg,faceCenteredAxis1,Range(0,1)); // u(mg,all,all,all,0:1)

u.updateToMatchGrid(faceCenteredAxis2);                   // u(mg,all,all,all,0:1)
u.updateToMatchGrid(mg,faceCenteredAxis3);                // u(mg,all,all,all,0:1)

u.updateToMatchGrid(cellCentered,Range(1,3),Range(-1,1))    // u(mg,all,all,all,1:3,-1:1)
```

Note that the grid function retains its components if no new values are given.

4.2 Grid functions with arbitrary centredness

The normal user can probably ignore this section.

This section describes how to make grid function with arbitrary centeredness and also how to make face-centered grid functions that are not in standard form.

In order to represent all these types of centering a grid function has an intArray called `isCellCentered` which indicates the centering of the grid function in each coordinate direction. Actually each component of a grid function can have its own centering but we will ignore this for now. Thus, for example,

	`isCellCentered(axis)`		
type	axis=0	axis=1	axis=2
vertex centred	FALSE	FALSE	FALSE
cell centred	TRUE	TRUE	TRUE
face centred along axis=0	FALSE	TRUE	TRUE
face centred along axis=1	TRUE	FALSE	TRUE
face centred along axis=2	TRUE	TRUE	FALSE
edge centred	TRUE	FALSE	FALSE
edge centred	FALSE	TRUE	FALSE
edge centred	FALSE	FALSE	TRUE

The member functions `setIsCellCentered`, `setIsFaceCentered` and `setFaceCentering` can be used to create grid-functions with various centerings. The most general function is `setIsCellCentered`. The most general function for creating face centered grid functions is `setIsFaceCentered` while the function `setFaceCentering` can be used to create face-centred grid functions of some standard forms.

4.2.1 Semi-general face-centred grid functions

Although the grid functions support a very general type of face-centering, in practice one often uses more specialized forms. The two common forms of face-centred grid functions are

1. A grid function for which all components are face-centred in the same direction (i.e. along the same axis). For example

```
...
const int axis1=0, axis2=1, axis3=2;
MappedGrid mg(...);
Range all;                  // a null Range is used when constructing grid functions, it indicates
                            // the positions of the coordinate axes

realMappedGridFunction u(mg,all,all,all,Range(0,1));
u.setFaceCentering(axis1);              // u(I1,I2,I3,0:1) : all components face centred along direction 0

u.updateToMatchGrid(Range(0,2),all,all,all);
u.setFaceCentering(axis2);              // u(0:2,I1,I2,I3) : all components face centred along direction 1
```

2. A grid function where each component is face centred in all space directions, for example

```
...
MappedGrid mg(...);

realMappedGridFunction u(mg,all,all,all,Range(0,1),faceRange);
// u(I1,I2,I3,0:1,0)  : these components are face-centred along direction 0
// u(I1,I2,I3,0:1,1)  : these components are face-centred along direction 1
// u(I1,I2,I3,0:1,2)  : these components are face-centred along direction 2 (if the grid is 3D)

u.updateToMatchGrid(mg,all,all,all,faceRange,Range(0,0),Range(0,2));
// u(I1,I2,I3,0,0:0,0:2)  : these components are face-centred along direction 0
// u(I1,I2,I3,1,0:0,0:2)  : these components are face-centred along direction 1
// u(I1,I2,I3,2,0:0,0:2)  : these components are face-centred along direction 2 (if the grid is 3D)
```

Note that the special Range, `faceRange` is used to indicate the position of the face centering component. This component will be dimensioned with a Range = Range(0,numberOfDimensions-1). Note that only one occurence of `faceRange` must appear in the argument list.

To determine if a grid function is of a given type of face-centering use the function `getFaceCentering` which returns a value from the enumerator `faceCenteringType` (found in a MappedGridFunction):

```
  enum faceCenteringType    // Here are some standard types of face centred grid functions
  {
    none=-1,                // not face centred
    direction0=0,           // all components are face centred along direction (i.e. axis) = 0
    direction1=1,           // all components are face centred along direction (i.e. axis) = 1
    direction2=2,           // all components are face centred along direction (i.e. axis) = 2
    all=-2                  // components are face centred in all directions, positionOfFaceCentering determines
  };                        //   the Index position that is used for the "directions"
}
```

To further illustrate these ideas, consider the following function which determines whether a grid function is of one of the standard types:

```
void determineFaceCentering( realMappedGridFunction & u )
{
  switch (u.getFaceCentering())
  {
  case GridFunctionParameters::none :
    cout << " this is not a standard face centered variable \n";
    break;
  case GridFunctionParameters::all :
    cout << "this function has components that are face centred in all space dimensions\n";
    cout << "The value u.positionOfFaceCentering = " << u.positionOfFaceCentering << ", "
            "gives the position of the faceRange";
    break;
  case GridFunctionParameters::direction0 :
    cout << "all components are face centered along direction=0 \n";
    break;
  case GridFunctionParameters::direction1 :
    cout << "all components are face centered along direction=1 \n";
    break;
  case GridFunctionParameters::direction2 :
    cout << "all components are face centered along direction=2 \n";
    break;
  default:
    cout << "Unknown face-centering type! This case should not occur! \n";
  };
```

For further examples see the program `Overture/tests/cellFace.C` which tests various aspects of cell and face centred grid functions.

5 Interpolant: Interpolating Grid Functions

The Interpolant class is used for interpolating a CompositeGridFunction – i.e. obtaining values at the interpolation points in terms of the values at the other points. Once an Interpolant object has been made it can be used in one of two ways. One can either use the `interpolate` function in the Interpolant class or one can use the `interpolate` function that appears in the grid function class.

When an Interpolant is associated with a Composite Grid, the CompositeGrid will be changed and will hold a pointer to the Interpolant. Any GridFunction associated with the CompositeGrid will be able to use the Interpolant found there. This allows GridFunctions to magically know how to interpolant themselves.

5.1 Member Functions

For now it is only possible to interpolate a `realCompositeGridFunction` or a `realMultigridCompositeGrid-Function`.

5.1.1 Constructors

Interpolant()

Decsription: Default constructor.

Interpolant(CompositeGrid & cg0)

Decsription: Create an interpolant and associate with a Composite grid.

cg0 (input): assoicate the interpolant with this CompositeGrid.

5.1.2 interpolate a CompositeGridFunction

int
interpolate(realCompositeGridFunction & u,
const Range & C0 = nullRange,
const Range & C1 = nullRange,
const Range & C2 = nullRange)

Description: Interpolate the interpolation boundary of a CompositeGridFunction

u (input/output): fill in the values on the interpolation boundary using other values on the grid function.

C0, C1, C2 (input): optionally specify components to interpolate. For example `interpolant.interpolate(u,Range(1,2))` to interpolate components 1 and 2.

Return Values: 0 = success, positive value is an error.

5.1.3 interpolate a single grid from a CompositeGridFunction

int
interpolate(int gridToInterpolate only interpolate this grid.,
realCompositeGridFunction & u,
const Range & C0 = nullRangeoptionally specify components to interpolate,
const Range & C1 = nullRange,
const Range & C2 = nullRange)

Description: Interpolate the interpolation boundary of a CompositeGridFunction

gridToInterpolate (input) : interpolate this grid (-1 ==¿ do all grids.)

u (input/output): fill in the values on the interpolation boundary using other values on the grid function.

C0, C1, C2 (input): optionally specify components to interpolate. For example `interpolant.interpolate(u,Range(1,2))` to interpolate components 1 and 2.

Return Values: 0 = success, positive value is an error.

5.1.4 interpolate specified grids from a CompositeGridFunction

int
interpolate(realCompositeGridFunction & u,
const IntegerArray & gridsToInterpolate only interpolate these grids.,
const Range & C0 = nullRangeoptionally specify components to interpolate,
const Range & C1 = nullRange,
const Range & C2 = nullRange)

Description: Interpolate the interpolation boundary of a CompositeGridFunction

Note: No AMR style interpolation will be applied when only some grids are interpolated.

u (input/output): fill in the values on the interpolation boundary using other values on the grid function.

gridsToInterpolate (input) : an array of length cg.numberOfComponentGrids(), which specifies to interpolate grid g if gridsToInterpolate(g)!=0

C0, C1, C2 (input): optionally specify components to interpolate. For example `interpolant.interpolate(u,Range(1,2))` to interpolate components 1 and 2.

Return Values: 0 = success, positive value is an error.

5.1.5 interpolate specified grids from specified grids

int
interpolate(realCompositeGridFunction & u,
const IntegerArray & gridsToInterpolate specify which grids to interpolate,
const IntegerArray & gridsToInterpolateFrom specify which grids to interpolate from,
const Range & C0 = nullRangeoptionally specify components to interpolate,
const Range & C1 = nullRange,
const Range & C2 = nullRange)

Description: Interpolate the interpolation boundary of a CompositeGridFunction

u (input/output): fill in the values on the interpolation boundary using other values on the grid function.

gridsToInterpolate (input) : an array of length cg.numberOfComponentGrids(), which specifies to interpolate grid g if gridsToInterpolate(g)!=0

gridsToInterpolateFrom (input) : an array of length cg.numberOfComponentGrids(), which specifies to interpolate from grid g if gridsToInterpolateFrom(g)!=0

C0, C1, C2 (input): optionally specify components to interpolate. For example `interpolant.interpolate(u,Range(1,2))` to interpolate components 1 and 2.

Return Values: 0 = success, positive value is an error.

5.1.6 interpolate grid A from grid B

int
interpolate(realArray & ug,
int gridToInterpolate,
int interpoleeGrid,
realCompositeGridFunction & u,
const Range & C0 = nullRange,
const Range & C1 = nullRange,
const Range & C2 = nullRange)

Description: Interpolate points on grid "gridToInterpolate" that interpolate from grid "interpoleeGrid".

Note: No AMR style interpolation will be applied when only some grids are interpolated.

ug (output): fill in the interpolated values into this array. This array will be dimensioned to hold the proper number of interpolation point values.

gridToInterpolate (input) : interpolate points on this grid.

interpoleeGrid (input) : only compute points that interpolate from this grid.

C0, C1, C2 (input): optionally specify components to interpolate. For example `interpolant.interpolate(u,Range(1,2))` to interpolate components 1 and 2.

Return Values: 0 = success, positive value is an error.

5.1.7 interpolate a refinement level

int
interpolateRefinementLevel(const int refinementLevel,
realCompositeGridFunction & u,
const Range & C0 = nullRange,
const Range & C1 = nullRange,
const Range & C2 = nullRange)

Description: Interpolate points on the boundary of a refinement level – only interpolate overlapping grid points from other grids on the same refinement level.

refinementLevel (input) : interpolate this refinement level.

u (intput/output): fill in the interpolated values into this grid function.

C0, C1, C2 (input): optionally specify components to interpolate. For example `interpolant.interpolate(u,Range(1,2))` to interpolate components 1 and 2.

Return Values: 0 = success, positive value is an error.

5.1.8 interpolationIsExplicit

bool
interpolationIsExplicit() const

Return Values: true if the interpolation is explicit, false if implicit.

5.1.9 interpolationIsImplicit

bool
interpolationIsImplicit() const

Return Values: true if the interpolation is implicit, false if explicit.

5.1.10 setExplicitInterpolationStorageOption

int
setExplicitInterpolationStorageOption(ExplicitInterpolationStorageOptionEnum option)

Description: Define the storage option to use for explicit interpolation (or implicit iterative interpolation) There is a tradeoff between storage and the number of operations required to determine the interpolated values. For wider interpolation stencils one may want to use less storage. For quadratic interpolation (w=3) in 3D (d=3) the storage is not bad, 27 values per interpolation point. Interpolation on an eight-order grid with w=9 however requires 9*9*9=729 values per interpolation point. In this case the options requiring less storage may be better to use.

option (input) : one of: precomputeAllCoefficients : requires w^d coefficients per interp pt (w=width of interp stencil) precomputeSomeCoefficients : requires w*d coefficients per interp pt (d=dimension, 1,2, or 3) precomputeNoCoefficients : requires d coefficinets per interp point

5.1.11 setImplicitInterpolationTolerance

int
setImplicitInterpolationTolerance(real tol)

Description: Set the convergence tolerance for implicit interpolation when the implicit equations are solved by iteration.

tolerance (input) : tolerance on the residual of the interpolation equations.

5.1.12 setImplicitInterpolationMethod

int
setImplicitInterpolationMethod(ImplicitInterpolationMethodEnum method)

Description: Choose between different methods

5.1.13 setMaximumNumberOfIterations

int
setMaximumNumberOfIterations(int maximumNumberOfIterations_) for iterative interpolation

Description: Set the maximum number of iterations for iterative interpolation.

maximumNumberOfIterations_ (input) : maximum number of iterations for iterative interpolation.

5.1.14 getImplicitInterpolationMethod

ImplicitInterpolationMethodEnum
getImplicitInterpolationMethod() const

Description: return the current method for implicit interpolation.

5.1.15 setInterpolationOption

int
setInterpolationOption(InterpolationOptionEnum option, bool trueOrFalse)

Description: Set an interpolation option. Set options to determine which points should be interpolated (for CompositeGrid's that have refinement grids).

option (input) : interpolateOverlappingRefinementPoints : assign points on refinement grids that interpolate from (refinement) grids belong to different base grids. These are points that are usually determined by the Ogen function updateRefinement.

interpolateAllRefinementBoundaries : assign points on refinement boundaries (i.e. ghost lines) that interpolate from other grids on the same base grid.

interpolateHiddenRefinementPoints : assign points on coarser levels that interpolate from finer grid patches (from the same base grid).

5.1.16 getInterpolationOption

int
getInterpolationOption(InterpolationOptionEnum option)

Description: Get the value of an interpolation option.

option (input) : interpolateOverlappingRefinementPoints : assign points on refinement grids that interpolate from (refinement) grids belong to different base grids. These are points that are usually determined by the Ogen function updateRefinement.

interpolateAllRefinementBoundaries : assign points on refinement boundaries (i.e. ghost lines) that interpolate from other grids on the same base grid.

interpolateHiddenRefinementPoints : assign points on coarser levels that interpolate from finer grid patches (from the same base grid).

return value: value of the option.

5.1.17 setInterpolateRefinements

int
setInterpolateRefinements(InterpolateRefinements & interp)

Description: Supply an AMR interpolation object:

interp (input) : use this to interpolate refinement boundaries on adaptive grids.

5.1.18 breakReference

void
breakReference()

Description: Break any references.

5.1.19 reference

void
reference(const Interpolant & interpolant)

Description: Reference this Interpolant to another.

interpolant (input): reference to this Interpolant.

5.1.20 updateToMatchGrid

void
updateToMatchGrid(CompositeGrid & cg0, int refinementLevel =0)

Description: Associate this Interpolant with a CompositeGrid and compute the interpolation coefficients.

cg0 (input): associate the interpolant with this CompositeGrid.

refinementLevel : only grids on this refinement level and above have been changed.

5.1.21 updateToMatchAdaptiveGrid

void
updateToMatchAdaptiveGrid(CompositeGrid & cg0)

Description: Use this update when the grid has changed through the addition of removal of refinement grids (but the base grids have not changed).

On an adaptive grid we always interpolate refinement grids using explicit interpolation or iterative implicit interpolation. This means that we do not need to reform a matrix for the implicit interpolation. The matrix can be used on the base-grids

cg0 (input): associate the interpolant with this CompositeGrid.

5.2 Examples

Here is an example of using the Interpolant class (file `Overture/examplestestInterpolant.C`)

6 Other Interpolation Functions

6.1 interpolatePoints: Interpolate a CompositeGridFunction at some given points in space

int
interpolatePoints(const realArray & positionToInterpolate,
const realCompositeGridFunction & u,
realArray & uInterpolated,
const Range & R0=nullRange,
const Range & R1=nullRange,
const Range & R2=nullRange,
const Range & R3=nullRange,
const Range & R4=nullRange,
intArray & indexGuess=Overture::nullIntegerDistributedArray(),
intArray & interpoleeGrid=Overture::nullIntegerDistributedArray(),
intArray & wasInterpolated=Overture::nullIntegerDistributedArray())

Description: Given some points in space, determine the values of a grid function u. If interpolation is not possible then extrapolate from the nearest grid point. The extrapolation is zero-order so that the value is just set equal to the value from the boundary.

positionToInterpolate (input): positionToInterpolate(0:numberOfPointsToInterpolate-1,0:numberOfDimensions-1) : (x,y[,z]) positions to interpolate. The first dimension of this array determines how many points to interpolate.

u (input): interpolate values from this grid function

uInterpolated (output): uInterpolated(0:numberOfPointsToInterpolate-1,R0,R1,R2,R3,R4) : interpolated values

R0,R1,...,R4 (input): interpolate these components of the grid function. R0 is the range of values for the first component of u, R1 the values for the second component, etc. By default all components of u are interpolated.

indexGuess (input/ouput): indexGuess(0:numberOfPointsToInterpolate-1,0:numberOfDimensions-1) : (i1,i2[,i3]) values for initial guess for searches. Not required by default.

interpoleeGrid(.) (input/output): interpoleeGrid(0:numberOfPointsToInterpolate-1) : try this grid first. Not required by default.

wasInterpolated(.) (output) : If provided as an argument, on output wasInterpolated(i)=TRUE if the point was successfully interpolated, or wasInterpolated(i)=FALSE if the point was extrapolated.

Errors: This routine in principle should always be able to interpolate or extrapolate.

Return Values:
- 0 = success
- 1 = error, unable to interpolate (this should never happen)
- -N = could not interpolate N points, but could extrapolate – extrapolation was performed from the nearest grid point.

Author: WDH

Example: In this example we show how to interpolate a grid function at arbitrary points in space.

```
CompositeGrid cg(...);    // get a CompositeGrid from some-where
Range all;
realCompositeGridFunction u(cg,all,all,all,2);  // grid function with two components

int numberOfPointsToInterpolate=1;
realArray positionToInterpolate(numberOfPointsToInterpolate,3),
          uInterpolated(numberOfPointsToInterpolate,2);
for(;;)
```

```
{
  cout << "Enter a point to interpolate (x,y) \n";
  cin >> positionToInterpolate(0,0) >> positionToInterpolate(0,1) ;

  int extrap = interpolatePoints(positionToInterpolate,u, uInterpolated);

  if( extrap < 0 )
    cout << " point was extrapolated" << endl;

  uInterpolated.display("Here is uInterpolated:");
}
```

6.2 InterpolateAllPoints on one CompositeGridFunction from another CompositeGridFunction

int
interpolateAllPoints(const realCompositeGridFunction & uFrom,
realCompositeGridFunction & uTo, bool useNewWay =true)

Description: Interpolate all values on one CompositeGridFunction, `uTo`, from the values of another CompositeGridFunction, `uFrom`. Values on `uTo` are extrapolated if they lie outside the region covered by `uFrom`. This routine calls the `interpolatePoints` function.

uFrom (input): Use these values to interpolate from.

uTo (output): Fill in all values on this grid (including ghost-points).

Errors: This routine in principle should always be able to interpolate or extrapolate all values.

Return Values:
- 0 = success
- 1 = error, unable to interpolate
- -N = could not interpolate N points, but could extrapolate – extrapolation was performed from the nearest grid point.

Author: WDH

Example: Here is an example

```
CompositeGrid cgFrom(...);
realCompositeGridFunction uFrom(cgFrom);
uFrom=...;

...

CompositeGrid cgTo(...);
realCompositeGridFunction uTo(cgTo);

interpolateAllPoints(uFrom,uTo);
```

6.3 InterpolateExposedPoints of a CompositeGridFunction for a Moving CompositeGrid

int
interpolateExposedPoints(CompositeGrid & cg1,
CompositeGrid & cg2,
realCompositeGridFunction & u1,
OGFunction *TZFlow =NULL,
real t =0.,

const bool & returnIndexValues =FALSE,
IntegerArray & numberPerGrid0 = Overture::nullIntArray(),
intArray & ia0 = Overture::nullIntArray(),
int stencilWidth = -1)

Purpose: Assign values to exposed points in a moving grid

cg1 (input): grid and grid function at old time

cg2 (input): grid at new time

u1: A grid function on grid cg1. On output, exposed points are interpolated

TZFlow: If specified and non-NULL this pointer to a twilight-zone function will be used to compute the error in the interpolation. This is used for debugging.

t: Evaluate the twilight-zone function at this time (the time corresponding to cg1).

returnIndexValues (bool) : if TRUE return points that were interpolated in the array ia:

```
ia(i,0:2) = (i1,i2,i3) where the number of points from each grid is stored in numberPerGrid(grid)
 thus    ia(i,0:2) : points interpolated from grid=0 for i=0,...,numberPerGrid(0)-1
         ia(i,0:2) : points interpolated from grid=1 for i=numberPerGrid(0),...,numberPerGrid(1)-
```

stencilWidth (input): interpolate enough points to support a discrete stencil of this width. A second-order centered approximation, for example, would use stencilWidth=3. By default stencilWith equals the discretization width defined by the grid.

Remarks: Here is a picture of a 1D moving overlapping grid that illustrates the exposed point on the old grid that requires a value so the solution can be advanced to the new grid.

```
+---+---+---+---I                    Old grid, point * is unused
         *---I---+---+---+---+       Points marked I are interpolation
         1   2   3   4   5

+---+---+---+---I
 move-->   I---+---+---+---+---+     New grid, requires a value on the OLD grid
           1   2   3   4             at point * to compute a derivative at point 2
```

NOTE: This routine assumes (and checks) that the number of grid points has NOT changed!

References

[1] D. L. Brown, *Overture operator classes for finite volume computations on overlapping grids, user guide*, Tech. Rep. UCRL-MA-133649, Lawrence Livermore National Laboratory, 1998.

[2] W. D. Henshaw, *Finite difference operators and boundary conditions for Overture, user guide*, Research Report UCRL-MA-132231, Lawrence Livermore National Laboratory, 1998.

[3] ——, *Grid functions for Overture, user guide*, Research Report UCRL-MA-132231, Lawrence Livermore National Laboratory, 1998.

[4] ——, *Mappings for Overture, a description of the Mapping class and documentation for many useful Mappings*, Research Report UCRL-MA-132239, Lawrence Livermore National Laboratory, 1998.

[5] ——, *Ogen: An overlapping grid generator for Overture*, Research Report UCRL-MA-132237, Lawrence Livermore National Laboratory, 1998.

[6] ——, *Oges user guide, a solver for steady state boundary value problems on overlapping grids*, Research Report UCRL-MA-132234, Lawrence Livermore National Laboratory, 1998.

[7] ——, *Ogshow: Overlapping grid show file class, saving solutions to be displayed with plotStuff, user guide*, Research Report UCRL-MA-132235, Lawrence Livermore National Laboratory, 1998.

[8] ——, *Plotstuff: A class for plotting stuff from Overture*, Research Report UCRL-MA-132238, Lawrence Livermore National Laboratory, 1998.

[9] ——, *A primer for writing PDE codes with Overture*, Research Report UCRL-MA-132231, Lawrence Livermore National Laboratory, 1998.

[10] ——, *Other stuff for Overture, user guide, version* 1.0, Research Report UCRL-MA-134292, Lawrence Livermore National Laboratory, 1999.

[11] ——, *OverBlown: A fluid flow solver for overlapping grids, reference guide*, Research Report UCRL-MA-134289, Lawrence Livermore National Laboratory, 1999.

[12] ——, *OverBlown: A fluid flow solver for overlapping grids, user guide*, Research Report UCRL-MA-134288, Lawrence Livermore National Laboratory, 1999.

Index