LA-UR 96-3708

Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: **OVERTURE: THE GRID CLASSES**

AUTHOR(S): K. Brislawn, D. Brown, G. Chesshire, W. Henshaw, K. Pao, D. Quinlan, E. Randall, and J. Saltzman

SUBMITTED TO: For Distribution on the Web

DISCLAIMER

Overture: The Grid Classes

K. Brislawn , D. Brown , G. Chesshire , W. Henshaw
K. Pao , D. Quinlan , E. Randall , J. Saltzman
Scientific Computing Group (CIC-19)
Los Alamos National Laboratory
Los Alamos, New Mexico 87545, USA

Abstract

Overture is a library containing classes for grids, overlapping grid generation and the discretization and solution of PDEs on overlapping grids. This document describes the Overture grid classes, including classes for single grids and classes for collections of grids.

Contents

1 Introduction

The Overture grid classes include classes for single grids and classes for collections of grids. The single-grid classes are related to each other through the C++ inheritance mechanism. The base class is **GenericGrid** (§2), and the class **MappedGrid** (§3) is derived from **GenericGrid**. The collections of grids are also related to each other through inheritance. The base class for collections of grids is **GenericGridCollection** (§4), which contains a list of **GenericGrid**s. The class **GridCollection** (§5) is derived from GenericGridCollection, and contains a list of **MappedGrid**s. All other Overture grid classes that contain collections of grids are derived from **GridCollection**. In particular, the classes **CompositeGrid** (§6), **MultigridMappedGrid** (§7) and **MultigridCompositeGrid** (§8) are all derived from **GridCollection**. All of these classes are described in this document.

2 Class GenericGrid

Note: You should not need to read this section unless you are designing a derived grid class.

Class **GenericGrid** is the base class for all of the Overture single-grid classes. By itself it does not contain any geometric data. It is useful only as a base class for other grid classes that may contain data to describe particular kinds of grids. We envision deriving from **GenericGrid** separate classes for structured and unstructured grids, and perhaps for other kinds of grids that we have not anticipated. For example, the class **MappedGrid** (§3) is derived from **GenericGrid** in order to describe curvilinear structured grids.

Many of the public constants, member data and member functions of class **GenericGrid** are overloaded in the derived classes. They are defined here in the base class for single grids because they are common to all single grid classes. The ordinary user (programmer) need not be concerned with these constants, data and member functions, or with the class **GenericGrid** at all, except where they are explicitly referred to in the descriptions of derived grid classes such as **MappedGrid** (§3).

2.1 Public constants

2.1.1 NOTHING

NOTHING = 0
NOTHING indicates no geometric data. See also **update(what,how)** (§2.4.9) and **destroy(what)** (§2.4.11).

2.1.2 THEusualSuspects

THEusualSuspects = **NOTHING** (§2.1.1)
THEusualSuspects indicates some of the geometric data of a **GenericGrid**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. In fact, a **GenericGrid** contains no geometric data, so all this is moot. This constant is typically overloaded in a derived class, to indicate some of the geometric data of that class, in addition to the geometric data indicated by the constant **THEusualSuspects** defined in its base class. See also **update(what,how)** (§2.4.9) and **destroy(what)** (§2.4.11).

2.1.3 EVERYTHING

EVERYTHING = **NOTHING** (§2.1.1)
EVERYTHING indicates all of the geometric data associated with a **GenericGrid**. In fact, a **GenericGrid** contains no geometric data. This constant is typically overloaded in a derived class, to indicate all of the geometric data of that class, in addition to the geometric data indicated by the constant **EVERYTHING** defined in its base class. See also **update(what,how)** (§2.4.9) and **destroy(what)** (§2.4.11).

2.1.4 COMPUTEnothing

COMPUTEnothing = 0
COMPUTEnotning indicates that no geometric data should be computed. See also **update(what,how)** (§2.4.9).

2.1.5 COMPUTEtheUsual

COMPUTEtheUsual = COMPUTEnothing (§2.1.4)
COMPUTEtheUsual indicates that computation of geometric data should proceed in the "usual way." In fact, a **GenericGrid** contains no geometric data, so this is irrelevant. This constant is typically overloaded in a derived class, to indicate the "usual way" of computing geometry relevant to that class, in addition to the usual way of computing the geometric data indicated by the constant **COMPUTEtheUsual** defined in and relevant to its base class. See also **update(what,how)** (§2.4.9).

2.1.6 COMPUTEfailed

COMPUTEfailed indicates that computation of some geometric data failed. See also **update(what,how)** (§2.4.9).

2.2 Public data

2.2.1 IntegerR computedGeometry

A bit mask that indicates which geometrical data has been computed. This must be reset to zero to invalidate the data when the geometry changes. It is recommended that this variable be used only by derived classes and grid-generation programs. See also **geometryHasChanged(what)** (§2.4.12).

2.3 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

2.3.1 GenericGridData* rcData

Pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§2.4.13) and **operator*()** (§2.4.14), which are provided for access to **rcData**.

2.3.2 Logical isCounted

Flag that indicates whether the data pointed to by **rcData** (§2.3.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

2.4 Public member functions

2.4.1 GenericGrid()

Default constructor.

Example

```
GenericGrid g;   // Construct a GenericGrid.
```

2.4.2 GenericGrid(const GenericGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default.

Example

```
GenericGrid g1;                                  // Construct a GenericGrid.
GenericGrid g2=g1, g3(g1);                       // Construct a GenericGrid using deep copy.
GenericGrid g4(g1,GenericGrid::SHALLOW);         // Construct using shallow copy; g2 shares the data of g1.
```

See also **operator=(x)** (§2.4.4) and **reference(x)** (§2.4.5).

2.4.3 virtual ~GenericGrid()

Destructor.

2.4.4 GenericGrid& operator=(const GenericGrid& x)

Assignment operator. This is also called a deep copy.

Example

```
GenericGrid g1, g2;   // Construct some GenericGrids.
g2 = g1;              // Copy data from g1 to g2.
```

2.4.5 void reference(const GenericGrid& x)

Make a reference. This is also called a shallow copy. This **GenericGrid** shares the data of **x**.

Example

```
GenericGrid g1, g2;   // Construct some GenericGrids.
g2.reference(g1);     // Now g2 shares the data of g1.
```

2.4.6 virtual void breakReference()

Break a reference. If this **GenericGrid** shares data with any other **GenericGrid**, then this function replaces it with a new copy that does not share data.

Example

```
GenericGrid g1, g2;   // Construct some GenericGrids.
g2.reference(g1);     // Now g2 shares the data of g1.
g2.breakReference();  // Now g2 has a new, unshared copy of the data.
```

2.4.7 virtual Integer get(const GenericDataBase& dir, const String& name)

Copy a **GenericGrid** into a file.

Example

```
HDF_DataBase dir;     // (derived from GenericDataBase)
dir.mount("g.hdf");   // Open a data file.
GenericGrid g;        // Construct a GenericGrid.
g.get(dir, "g");      // Copy g from dir.
dir.unmount();        // Close the data file.
```

2.4.8 virtual Integer put(GenericDataBase& dir, const String& name) const

Copy a **GenericGrid** into a file.

Example

```
HDF_DataBase dir;       // (derived from GenericDataBase)
dir.mount("g.hdf", "I"); // Open a data file.
GenericGrid g;          // Construct a GenericGrid.
g.put(dir, "g");        // Copy g into dir.
dir.unmount();          // Close the data file.
```

2.4.9 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **NOTHING** (§2.1.1), **THEusualSuspects** (§2.1.2) and **EVERYTHING** (§2.1.3) may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§2.1.6), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. Any combination of the constants **COMPUTEnothing** (§2.1.4) and **COMPUTEtheUsual** (§2.1.5) may be bitwise ORed together to form the optional second argument of **update**(). In fact, a **GenericGrid** contains no geometric data, so all this is irrelevant.

Example

```
GenericGrid g;                                  // Construct a GenericGrid.
g.update(GenericGrid::EVERYTHING);              // Update all of the GenericGrid geometric data.
```

2.4.10 virtual Integer update(GenericGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another **GenericGrid** (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contains the corresponding data, then the data for this **GenericGrid** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§2.4.9).

Example

```
GenericGrid g1, g2;                             // Construct some GenericGrids.
g1.update(GenericGrid::EVERYTHING);             // Update all of the GenericGrid geometric data of g1
g2.update(g1,GenericGrid::EVERYTHING);          // Update all of the GenericGrid data of g2, sharing g1 data.
```

2.4.11 virtual void destroy(const Integer what = NOTHING)

Destroy the indicated optional **GenericGrid** geometric data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **NOTHING** (§2.1.1), **THEusualSuspects** (§2.1.2) and **EVERYTHING** (§2.1.3) may be bitwise ORed together to form the first argument of **update**().

Example

```
GenericGrid g;                                  // Construct a GenericGrid.
g.destroy(GenericGrid::EVERYTHING);             // Destroy all of the GenericGrid geometric data.
```

2.4.12 void geometryHasChanged(const Integer what = ~NOTHING)

Mark the geometric data out-of-date. Any combination of the constants **NOTHING** (§2.1.1), **THEusualSuspects** (§2.1.2) and **EVERYTHING** (§2.1.3) may be bitwise ORed together to form the first argument of **geometryHasChanged**(). By default, all geometric data of this **GenericGrid** and all derived classes is marked out-of-date. It is recommended that this function be called only from derived classes and grid-generation

programs.

Example

```
GenericGrid g;                                          // Construct a GenericGrid.
g.geometryHasChanged(GenericGrid::EVERYTHING);          // Mark all GenericGrid data out-of-date.
```

2.4.13 GenericGridData* operator->()

Access the reference-counted data.

Example

```
GenericGrid g;                                  // Construct a GenericGrid.
Integer geometry = g->computedGeometry;         // Access the reference-counted data.
```

2.4.14 GenericGridData& operator*()

Access the pointer to the reference-counted data.

Example

```
GenericGrid g;                  // Construct a GenericGrid.
GenericGridData& data = *g;     // Access the pointer to the reference-counted data.
```

2.4.15 virtual String getClassName()

Get the class name of the most-derived class for this object.

Example

```
GenericGrid g;                              // Construct a GenericGrid.
String className = g.getClassName();        // Get the class name of g. It should be "GenericGrid".
```

2.5 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

2.5.1 void reference(GenericGridData& x)

Make a reference to an object of class **GenericGridData**. This **GenericGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

Example

```
GenericGrid g1, g2;     // Construct some GenericGrids.
g2.reference(*g1);      // Now g2 shares the data of g1.
```

2.5.2 void updateReferences()

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

Example

```
GenericGrid g;              // Construct a GenericGrid.
g.updateReferences();       // Update references to the reference-counted data.
```

3 Class MappedGrid

Class MappedGrid is used for all logically-rectangular grids. This includes cartesian, rectangular and curvilinear grids. Class MappedGrid allows for grids with holes, unused vertices or cells within a grid. It is assumed that a continuous function exists which maps the vertices of a uniform grid to the vertices of the MappedGrid. This is no restriction, becauase it is always possible to construct a function, for example an interpolant, with this property.

Class **MappedGrid** is derived from class **GenericGrid** (§2). It overloads some of the **GenericGrid** public constants, member data and member functions.

3.1 Public constants

3.1.1 MAXrefinementLevel

MAXrefinementLevel = 8
The maximimum number of refinement levels allowed for adaptive grid refinement. If possible, this constant should never be used. It will soon disappear.

3.1.2 THEmask

THEmask indicates the discretization point **mask** (§3.2.18). See also **THEkr** (§3.1.37), **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.3 THEvertex

vertex indicates **vertex** (§3.2.19), the locations of the vertices of the grid. See also **THExy** (§3.1.39), **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.4 THEvertex2D

THEvertex2D = **THEvertex** (§3.1.3)

3.1.5 THEvertex1D

THEvertex1D = **THEvertex** (§3.1.3)

3.1.6 THEcenter

center indicates **center** (§3.2.22), the locations of the discretization points of the grid. See also **THExyc** (§3.1.40), **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.7 THEcenter2D

THEcenter2D = **THEcenter** (§3.1.6)

3.1.8 THEcenter1D

THEcenter1D = **THEcenter** (§3.1.6)

3.1.9 THEvertexDerivative

THEvertexDerivative indicates **vertexDerivative** (§3.2.25), the derivative of the mapping at the vertices of the grid. See also **THExyrs** (§3.1.41), **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.10 THEvertexDerivative2D

THEvertexDerivative2D = **THEvertexDerivative** (§3.1.9)

3.1.11 THEvertexDerivative1D

THEvertexDerivative1D = **THEvertexDerivative** (§3.1.9)

3.1.12 THEcenterDerivative

THEcenterDerivative indicates **centerDerivative** (§3.2.28), the derivative of the mapping at the discretization points of the grid. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.13 THEcenterDerivative2D

THEcenterDerivative2D = THEcenterDerivative (§3.1.12)

3.1.14 THEcenterDerivative1D

THEcenterDerivative1D = THEcenterDerivative (§3.1.12)

3.1.15 THEinverseVertexDerivative

THEinverseVertexDerivative indicates **inverseVertexDerivative** (§3.2.31), the inverse of the mapping derivative, evaluated at the vertices of the grid. See also **THErsxy** (§3.1.38), **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.16 THEinverseVertexDerivative2D

THEinverseVertexDerivative2D = THEinverseVertexDerivative (§3.1.15)

3.1.17 THEinverseVertexDerivative1D

THEinverseVertexDerivative1D = THEinverseVertexDerivative (§3.1.15)

3.1.18 THEinverseCenterDerivative

THEinverseCenterDerivative indicates **inverseCenterDerivative** (§3.2.34), the inverse of the mapping derivative, evaluated at the discretization points of the grid. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.19 THEinverseCenterDerivative2D

THEinverseCenterDerivative2D = THEinverseCenterDerivative (§3.1.18)

3.1.20 THEinverseCenterDerivative1D

THEinverseCenterDerivative1D = THEinverseCenterDerivative (§3.1.18)

3.1.21 THEvertexJacobian

THEvertexJacobian indicates **vertexJacobian** (§3.2.37), the determinant of the derivative of the mapping at the vertices of the grid. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.22 THEcenterJacobian

THEcenterJacobian indicates **centerJacobian** (§3.2.38), the determinant of the derivative of the mapping at the discretization points of the grid. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.23 THEcellVolume

THEcellVolume indicates **cellVolume** (§3.2.39), the area (in two dimensions) or volume (in three dimensions) of the grid cells. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.24 THEfaceNormal

THEfaceNormal indicates **faceNormal** (§3.2.40), the normals to the grid cell edges (in two dimensions) or faces (in three dimensions), normalized to the length or area area of the corresponding edge or face. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.25 THEfaceNormal2D

THEfaceNormal2D = **THEfaceNormal** (§3.1.24)

3.1.26 THEfaceNormal1D

THEfaceNormal1D = **THEfaceNormal** (§3.1.24)

3.1.27 THEcenterNormal

THEcenterNormal indicates **centerNormal** (§3.2.43), the normals to constant parameter curves (in two dimensions) or surfaces (in three dimensions) passing through the grid cell centers, normalized to the length or area of that part of the curve or surface which lies inside the corresponding grid cell. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.28 THEcenterNormal2D

THEcenterNormal2D = **THEcenterNormal** (§3.1.27)

3.1.29 THEcenterNormal1D

THEcenterNormal1D = **THEcenterNormal** (§3.1.27)

3.1.30 THEfaceArea

THEfaceArea indicates **faceArea** (§3.2.46), the length (in two dimensions) or area (in three dimensions) of the grid cell edges or faces. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.31 THEfaceArea2D

THEfaceArea2D = **THEfaceArea** (§3.1.30)

3.1.32 THEfaceArea1D

THEfaceArea1D = **THEfaceArea** (§3.1.30)

3.1.33 THEvertexBoundaryNormal

THEvertexBoundaryNormal indicates **vertexBoundaryNormal** (§3.2.49), the unit outward normals to the grid boundary at the boundary vertices. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.34 THEcenterBoundaryNormal

THEcenterBoundaryNormal indicates **centerBoundaryNormal** (§3.2.50), the unit outward normals to the grid boundary at centers of the grid boundary cell edges (in two dimensions) or faces (in three dimensions). See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.35 THEminMaxEdgeLength

THEminMaxEdgeLength indicates **minimumEdgeLength** (§3.2.16) and **maximumEdgeLength** (§3.2.17), the minimum and maximum grid cell edge lengths. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.36 THEtrxyab

THEtrxyab indicates **trxyab** (§3.3.24), the coordinate bounds of a rectangular box that contains the vertices of the grid. See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.37 THEkr

THEkr = **mask** (§3.2.18)

3.1.38 THErsxy

THErsxy = **inverseVertexDerivative** (§3.2.31)

3.1.39 THExy

THExy = **vertex** (§3.2.19)

3.1.40 THExyc

THExyc = **center** (§3.2.22)

3.1.41 THExyrs

THExyrs = **vertexDerivative** (§3.2.25)

3.1.42 THEusualSuspects

THEusualSuspects = **GenericGrid::THEusualSuspects** (§2.1.2) | **THEmask** (§3.1.2) | **THEvertex** (§3.1.3) | **THEcenter** (§3.1.6) | **THEvertexDerivative** (§3.1.9)
THEusualSuspects indicates some of the geometric data of a **MappedGrid**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. **THEusualSuspects** overloads **GenericGrid::THEusualSuspects** (§2.1.2). See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.43 EVERYTHING

EVERYTHING = **GenericGrid::EVERYTHING** (§2.1.3) | **THEmask** (§3.1.2) | **THEvertex** (§3.1.3) | **THEcenter** (§3.1.6) | **THEvertexDerivative** (§3.1.9) | **THEcenterDerivative** (§3.1.12) | **THEinverseVertexDerivative** (§3.1.15) | **THEinverseCenterDerivative** (§3.1.18) | **THEvertexJacobian** (§3.1.21) | **THEcenterJacobian** (§3.1.22) | **THEcellVolume** (§3.1.23) | **THEfaceNormal** (§3.1.24) | **THEcenterNormal** (§3.1.27) | **THEfaceArea** (§3.1.30) | **THEvertexBoundaryNormal** (§3.1.33) | **THEcenterBoundaryNormal** (§3.1.34) | **THEminMaxEdgeLength** (§3.1.35) | **THEtrxyab** (§3.1.36)
EVERYTHING indicates all of the geometric data associated with a **MappedGrid**. **EVERYTHING** overloads **GenericGrid::EVERYTHING** (§2.1.3). See also **update(what,how)** (§3.5.13) and **destroy(what)** (§3.5.15).

3.1.44 USEdifferenceApproximation

USEdifferenceApproximation indicates that computation of all geometric data except for **vertex** (§3.2.19) should be done using discrete approximations such as finite-difference approximations. By default, if a **mapping** (§3.2.51) is available and is not of the base class "**Mapping**", then discrete approximations are not used. Instead, the mapping and its derivative are used to compute all of the geometric data. See also **update(what,how)** (§3.5.13).

3.1.45 COMPUTEgeometry

COMPUTEgeometry indicates that geometric data should be computed for each variable indicated, even if that data had already been computed and marked valid. See also **update(what,how)** (§3.5.13).

3.1.46 COMPUTEgeometryAsNeeded

COMPUTEgeometryAsNeeded indicates that geometric data should be computed only for those variables indicated, which either had not already been computed, were marked invalid, or for which new space needed to be allocated. See also **update(what,how)** (§3.5.13).

3.1.47 COMPUTEtheUsual

COMPUTEtheUsual = GenericGrid::COMPUTEtheUsual (§2.1.5) | **COMPUTEgeometryAsNeeded** (§3.1.

COMPUTEtheUsual indicates that computation of geometric data should proceed in the "usual way." Currently this means that geometric data is computed only for those variables indicated, which had not already been computed. However, this may change from time to time. **COMPUTEtheUsual** overloads **GenericGrid::COMPUTEtheUsual** (§2.1.5). See also **update(what,how)** (§3.5.13).

3.2 Public data

3.2.1 IntegerR numberOfDimensions

The number of dimensions of the domain. See also **nd** (§3.3.13).

3.2.2 IntegerArray dimension

Dimensions: (0:1, 0:2)
dimension holds the dimensions (lower and upper index bounds), for the indices corresponding to coordinates in the parameter space of the grid, of **MappedGridFunctions** of all types defined on the grid. **dimension**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. If the grid has ghost points, then **dimensions** $\neq$ **gridIndexRange**. For the extra dimensions **numberOfDimensions** $\leq j \leq 2$, if any, **dimension**$(i,j) = 0$. See also **ndrsab** (§3.3.14).

3.2.3 IntegerArray indexRange

Dimensions: (0:1, 0:2)
indexRange holds the range of indices of the discretization points. **indexRange**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. In those coordinate directions j where the grid is neither cell-centered nor periodic, and for $j \geq$ **numberOfDimensions**, the discretization points have the same index range as the vertices of the grid, so **indexRange**(i,j) = **gridIndexRange**(i,j) for $i = 0$ and for $i = 1$. In cell-centered or periodic coordinate directions j, the first discretization point has the same index as the first vertex, so **indexRange**$(0,j)$ = **gridIndexRange**$(0,j)$, but there is one discretization point fewer than the number of vertices, so **indexRange**$(1,j)$ = **gridIndexRange**$(1,j) - 1$. For the extra dimensions **numberOfDimensions** $\leq j \leq 2$, if any, **indexRange**$(i,j) = 0$. See also **mrsab** (§3.3.12).

3.2.4 IntegerArray gridIndexRange

Dimensions: (0:1, 0:2)
gridIndexRange holds the range of indices of the grid vertices. **gridIndexRange**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. For the extra dimensions **numberOfDimensions** $\leq j \leq 2$, if any, **gridIndexRange**$(i,j) = 0$. See also **nrsab** (§3.3.16).

3.2.5 IntegerArray numberOfGhostPoints

Dimensions: (0:1, 0:2)
numberOfGhostPoints holds the number of ghost point vertices on each side of the grid. **numberOfGhostPoints**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. The number of ghost points is the difference between the corresponding bounds on the grid vertex index range and on the dimensions, so

$$\textbf{numberOfGhostPoints}(i,j) = (-1)^i\big(\textbf{gridIndexRange}(i,j) - \textbf{dimensions}(i,j)\big).$$

See also **nxtra** (§3.3.17).

3.2.6 IntegerArray discretizationWidth

Dimensions: (0:2)
discretizationWidth holds the width of the interior discretization stencil. This means that every interior discretization point is guaranteed to have available to it a stencil of this width consisting of valid points for use in the discretization of a PDE. Points that are so close to the boundary that such a stencil would extend outside the grid are not considered to be interior discretization points, but may be boundary discretization points. **discretizationWidth**(i) refers to the width of the stencil in the direction corresponding to the coordinate r_i in the parameter space of the grid. For the extra dimensions **numberOfDimensions** $\leq i \leq 2$, if any, **discretizationWidth**$(i) = 0$. See also **dw** (§3.3.6).

3.2.7 IntegerArray boundaryDiscretizationWidth

Dimensions: (0:1, 0:2)
boundaryDiscretizationWidth holds the width of the boundary condition discretization stencil in the direction normal to the boundary, on each side of the grid. This means that every boundary discretization point is guaranteed to have available to it a one-sided stencil of this width consisting of valid points for use in the discretization of boundary conditions. This stencil includes points on the boundary and extends from there into the interior of the grid. **boundaryDiscretizationWidth**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. The boundary condition stencil width does not consider any ghost points. It considers only points on the boundary and inside the grid. In addition, any number of ghost points may be used as needed for the discretization of boundary conditions. For the extra dimensions **numberOfDimensions** $\leq j \leq 2$, if any, **boundaryDiscretizationWidth**$(i, j) = 0$. See also **bw** (§3.3.2).

3.2.8 IntegerArray boundaryCondition

Dimensions: (0:1, 0:2)
boundaryCondition holds the boundary condition flags, which indicate how each side of the grid is used; **boundaryCondition**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid.

$$\textbf{boundaryCondition}(i,j) \quad \begin{cases} < 0 & \Rightarrow \text{ The domain is periodic in the coordinate } r_j. \\ = 0 & \Rightarrow \text{ The side corresponding to } r_j = i \text{ may only interpolate.} \\ > 0 & \Rightarrow \text{ The side corresponding to } r_j = i \text{ is part of the domain boundary.} \end{cases}$$

For the extra dimensions **numberOfDimensions** $\leq j \leq 2$, if any, **boundaryCondition**$(i, j) = -1$. See also **bc** (§3.3.1).

3.2.9 IntegerArray sharedBoundaryFlag

Dimensions: (0:1, 0:2)
sharedBoundaryFlags holds the shared boundary flags, which may be used to indicate which sides of the grid correspond to the same feature of the domain boundary. Different features of the domain boundary may be distinguished from each other by their being separated by an edge or corner of the domain. Sides of grids which correspond to the same feature of the domain boundary ideally should match exactly where they overlap or should at least intersect only tangentially. In practice this is often impossible to ensure (especially in the case of grids whose mappings are defined discretely, for example, mappings based on splines), so this flag is useful in order to identify those cases where such was the intention. **sharedBoundaryFlag**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. A unique non-zero flag value should be assigned to **sharedBoundaryFlag**(i,j) for all of those sides of grids that correspond to the same feature of the domain boundary. For the extra dimensions **numberOfDimensions** $\leq j \leq 2$, if any, **sharedBoundaryFlag**$(i, j) = 0$. See also **share** (§3.3.22).

3.2.10 RealArray sharedBoundaryTolerance

Dimensions: (0:1, 0:2)
sharedBoundaryTolerance holds the shared boundary error tolerance, which indicates by how much the mapping that generates the grid may deviate from the ideal domain boundary on each side of the grid, normalized

to the width of grid cells in the direction normal to the boundary. In the case where the ideal domain boundary is unknown, **sharedBoundaryTolerance** is a useful estimate of the mismatch between sides of the grid that correspond to the same feature of the domain boundary, as identified by **sharedBoundaryFlag** (§3.2.9). **sharedBoundaryTolerance**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. See also **sheps** (§3.3.23).

3.2.11 RealArray gridSpacing

Dimensions: (0:2)
The grid spacing in the direction of the coordinate r_i in the parameter space of the grid is

$$\textbf{gridSpacing}(i) = \frac{1}{\textbf{gridIndexRange}(1,i) - \textbf{gridIndexRange}(0,i)}.$$

For the extra dimensions $\textbf{numberOfDimensions} \le i \le 2$, if any, $\textbf{gridSpacing}(i) = 1$. See also **drs** (§3.3.5).

3.2.12 LogicalArray isCellCentered

Dimensions: (0:2)
The flag **isCellCentered**(i) is **LogicalTrue** or non-zero if and only if $i <$ **numberOfDimensions** and the grid is cell-centered in the direction corresponding to the coordinate r_i in the parameter space of the grid. Cell-centered in direction i means that discretization points lie at positions

$$r_i = \begin{cases} \left(j + \frac{1}{2} - \textbf{indexRange}(0,i)\right) \textbf{gridSpacing}(i) & \text{if } \textbf{isCellCentered}(i) \\ \left(j - \textbf{indexRange}(0,i)\right) \textbf{gridSpacing}(i) & \text{otherwise} \end{cases}$$

for $\textbf{indexRange}(0,i) \le j \le \textbf{indexRange}(1,i)$. See also **cctype** (§3.3.3).

3.2.13 LogicalR isAllCellCentered

The flag **isAllCellCentered** is **LogicalTrue** or non-zero if and only if the grid is cell-centered in all directions. This means that the discretization points are grid cell centers. See also **isCellCentered** (§3.2.12), **cctype** (§3.3.3) and **CompositeGrid::cgtype** (§6.3.5).

3.2.14 LogicalR isAllVertexCentered

The flag **isAllVertexCentered** is **LogicalTrue** or non-zero if and only if the grid is vertex-centered in all directions. This means that the discretization points are grid vertices. See also **isCellCentered** (§3.2.12), **cctype** (§3.3.3) and **CompositeGrid::cgtype** (§6.3.5).

3.2.15 IntegerArray isPeriodic

Dimensions: (0:2)
The flag **isPeriodic**(i) describes the periodicity of the grid, the mapping that generates the grid, and all **MappedGridFunctions** defined on the grid. **isPeriodic**(i) is zero (non-periodic) if and only if $i <$ **numberOfDimensions** and either the derivative of the mapping that generates the grid is not periodic in the direction corresponding to the coordinate r_i in the parameter space of the grid, or the periodicity of the domain does not correspond to the periodicity of the derivative of the mapping in this direction. Two cases exist when the periodicity of the mapping corresponds to the periodicity of the domain: either the mapping itself is periodic or it is not. (In the latter case the mapping differs from a periodic function by a linear function.) These two cases are distinguished by different non-zero values of the flag **isPeriodic**(i). All **MappedGridFunctions** defined on the grid should have the same periodicity as the mapping; in each direction where the mapping is periodic, all **MappedGridFunctions** should be periodic, and in each direction where the derivative of the mapping is periodic, the derivatives of all **MappedGridFunctions** should be periodic. The possible values of **isPeriodic**(i) are

$$\textbf{isPeriodic}(i) = \begin{cases} \textbf{Mapping::notPeriodic} = 0 & \text{The derivative of the mapping is not periodic.} \\ \textbf{Mapping::derivativePeriodic} & \text{The derivative is periodic but the mapping is not.} \\ \textbf{Mapping::functionPeriodic} & \text{The mapping is periodic.} \end{cases}$$

See also **period** (§3.3.18).

3.2.16 RealArray minimumEdgeLength

Dimensions: (0:2)
minimumEdgeLength holds the minimum grid cell-edge length over all cell edges in the interior and the boundary of the grid, for each coordinate direction in the parameter space of the grid. **minimumEdgeLength**(i) refers to edges of the cells corresponding to the coordinate direction r_i. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                   // Construct a MappedGrid.
g.update(MappedGrid::THEminMaxEdgeLength);      // Update the minimum and maximum edge length.
```

See also **THEminMaxEdgeLength** (§3.1.35) and **update(what,how)** (§3.5.13).

3.2.17 RealArray maximumEdgeLength

Dimensions: (0:2)
maximumEdgeLength holds the maximum grid cell-edge length over all cell edges in the interior and the boundary of the grid, for each coordinate direction in the parameter space of the grid. **maximumEdgeLength**(i) refers to edges of the cells corresponding to the coordinate direction r_i. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                   // Construct a MappedGrid.
g.update(MappedGrid::THEminMaxEdgeLength);      // Update the minimum and maximum edge length.
```

See also **THEminMaxEdgeLength** (§3.1.35) and **update(what,how)** (§3.5.13).

3.2.18 IntegerMappedGridFunction mask

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} = \mathbf{dimension}(i,j)$.
The discretization point mask, which shows how each discretization point (*e.g.*, vertex or cell-center) is used.

$$\mathbf{mask}(i,j,k)\begin{cases} <0 & \Rightarrow \text{ interpolation point} \\ =0 & \Rightarrow \text{ unused point} \\ >0 & \Rightarrow \text{ discretization point} \end{cases}$$

This data may be updated as in the following example.

Example

```
MappedGrid g;                                   // Construct a MappedGrid.
g.update(MappedGrid::THEmask);                  // Update mask.
```

See also **THEmask** (§3.1.2), **update(what,how)** (§3.5.13) and **kr** (§3.3.11).

3.2.19 RealMappedGridFunction vertex

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1)$, where $d_{ij} = \mathbf{dimension}(i,j)$ and $n_1 = \mathbf{numberOfDimensions} - 1$.
vertex holds the coordinates of the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$\mathbf{vertex}(i_0,i_1,i_2,*) = \boldsymbol{g}(\mathbf{r}), \quad \text{where} \quad r_j = \big(i_j - \mathbf{indexRange}(0,j)\big)\,\mathbf{gridSpacing}(j),$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                              // Construct a MappedGrid.
g.update(MappedGrid::THEvertex);           // Update vertex.
```

See also **THEvertex** (§3.1.3), **update(what,how)** (§3.5.13) and **xy** (§3.3.26).

3.2.20 RealMappedGridFunction vertex2D

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
Vertex coordinates, for a one- or two-dimensional grid. **vertex2D** is aliased to **vertex** (§3.2.19), but has fewer dimensions, so that it may be used to simplify two-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                              // Construct a MappedGrid.
g.update(MappedGrid::THEvertex2D);         // Update vertex2D.
```

See also **THEvertex2D** (§3.1.4) and **update(what,how)** (§3.5.13).

3.2.21 RealMappedGridFunction vertex1D

Dimensions: $(d_{00}{:}\,d_{10}, 0{:}\,0)$, where $d_{ij} =$ **dimension**(i,j).
Vertex coordinates, for a one-dimensional grid. **vertex1D** is aliased to **vertex** (§3.2.19), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                              // Construct a MappedGrid.
g.update(MappedGrid::THEvertex1D);         // Update vertex1D.
```

See also **THEvertex1D** (§3.1.5) and **update(what,how)** (§3.5.13).

3.2.22 RealMappedGridFunction center

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
center holds the coordinates of the discretization points of the grid, (*e.g.*, the vertices or the grid cell-centers), including any ghost points (which lie outside the grid).

$$\textbf{center}(i_0,i_1,i_2,*) = \boldsymbol{g}(\mathbf{r}), \quad \text{where} \quad r_j = \begin{cases} \left(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\right)\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \left(i_j - \textbf{indexRange}(0,j)\right)\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **center** is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed by averaging **vertex** in each cell-centered direction:

$$\textbf{center}(i_0,i_1,i_2,*) = \Big(\prod_{\substack{j,\ \text{such that} \\ \textbf{isCellCentered}(j)}} \mu_{+j} \Big) \textbf{vertex}(i_0,i_1,i_2,*).$$

This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEcenter);                       // Update center, or ...
g.update(MappedGrid::THEcenter,                        // Force re-update of center,
         MappedGrid::COMPUTEgeometry |                 // this time using a
         MappedGrid::USEdiscreteApproximation);        // discrete approximation.
```

See also **THEcenter** (§3.1.6), **update(what,how)** (§3.5.13) and **xyc** (§3.3.28).

3.2.23 RealMappedGridFunction center2D

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
Coordinates of discretization points, for a one- or two-dimensional grid. **center2D** is aliased to **center** (§3.2.22), but has fewer dimensions, so that it may be used to simplify two-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                  // Construct a MappedGrid.
g.update(MappedGrid::THEcenter2D);             // Update center2D.
```

See also **THEcenter2D** (§3.1.7) and **update(what,how)** (§3.5.13).

3.2.24 RealMappedGridFunction center1D

Dimensions: $(d_{00}{:}\,d_{10}, 0{:}\,0)$, where $d_{ij} =$ **dimension**(i,j).
Coordinates of discretization points, for a one-dimensional grid. **center1D** is aliased to **center** (§3.2.22), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                  // Construct a MappedGrid.
g.update(MappedGrid::THEcenter1D);             // Update center1D.
```

See also **THEcenter1D** (§3.1.8) and **update(what,how)** (§3.5.13).

3.2.25 RealMappedGridFunction vertexDerivative

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
vertexDerivative holds the derivative of the mapping at the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$\textbf{vertexDerivative}(i_0,i_1,i_2,*,j) = \frac{\partial \boldsymbol{g}}{\partial r_j}, \quad \text{where} \quad r_j = \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j),$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **vertexDerivative** is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed by centered finite differences of **vertex**:

$$\textbf{vertexDerivative}(i_0,i_1,i_2,*,j) = \frac{1}{2\Delta r_j}\Delta_{0j}\textbf{vertex}(i_0,i_1,i_2,*),$$

where Δr_j = **gridSpacing**(j). This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEvertexDerivative);             // Update vertexDerivative.
```

See also **THEvertexDerivative** (§3.1.9), **update(what,how)** (§3.5.13) and **xyrs** (§3.3.29).

3.2.26 RealMappedGridFunction vertexDerivative2D

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, 0{:}\, n_1, 0{:}\, n_1)$, where d_{ij} = **dimension**(i,j) and n_1 = **numberOfDimensions** $-$ 1.
Derivative of the mapping at the vertices, for a one- or two-dimensional grid. **vertexDerivative2D** is aliased to **vertexDerivative** (§3.2.25), but has fewer dimensions, so that it may be used to simplify two-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEvertexDerivative2D);           // Update vertexDerivative2D.
```

See also **THEvertexDerivative2D** (§3.1.10) and **update(what,how)** (§3.5.13).

3.2.27 RealMappedGridFunction vertexDerivative1D

Dimensions: $(d_{00}{:}\, d_{10}, 0{:}\, 0, 0{:}\, 0)$, where d_{ij} = **dimension**(i,j).
Derivative of the mapping at the vertices, for a one-dimensional grid. **vertexDerivative1D** is aliased to **vertexDerivative** (§3.2.25), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEvertexDerivative1D);           // Update vertexDerivative1D.
```

See also **THEvertexDerivative1D** (§3.1.11) and **update(what,how)** (§3.5.13).

3.2.28 RealMappedGridFunction centerDerivative

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1, 0{:}\, n_1)$, where d_{ij} = **dimension**(i,j) and n_1 = **numberOfDimensions** $-$ 1.
centerDerivative holds the derivative of the mapping at the discretization points of the grid, including any ghost points (which lie outside the grid).

$$\textbf{centerDerivative}(i_0,i_1,i_2,*,j) = \frac{\partial \boldsymbol{g}}{\partial r_j}, \quad \text{where}$$

$$r_j = \begin{cases} \left(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\right)\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \left(i_j - \textbf{indexRange}(0,j)\right)\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **centerDerivative** is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed by averaging centered finite differences of **vertex**:

$$\textbf{centerDerivative}(i_0,i_1,i_2,*,j) = \Big(\prod_{\substack{k\neq j,\ \text{such that} \\ \textbf{isCellCentered}(k)}} \mu_{+k}\Big) \begin{cases} \frac{1}{\Delta r_j}\Delta_{+j}\textbf{vertex}(i_0,i_1,i_2,*) & \text{if } \textbf{isCellCentered}(j) \\ \frac{1}{2\Delta r_j}\Delta_{0j}\textbf{vertex}(i_0,i_1,i_2,*) & \text{otherwise,} \end{cases}$$

where Δr_j = **gridSpacing**(j). This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEcenterDerivative);             // Update centerDerivative.
```

See also **THEcenterDerivative** (§3.1.12) and **update(what,how)** (§3.5.13).

3.2.29 RealMappedGridFunction centerDerivative2D

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, 0{:}\, n_1, 0{:}\, n_1)$, where d_{ij} = **dimension**(i,j) and n_1 = **numberOfDimensions** $-$ 1.
Derivative at the discretization points, for a one- or two-dimensional grid. **centerDerivative2D** is aliased to **centerDerivative** (§3.2.28), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEcenterDerivative2D);           // Update centerDerivative2D.
```

See also **THEcenterDerivative2D** (§3.1.13) and **update(what,how)** (§3.5.13).

3.2.30 RealMappedGridFunction centerDerivative1D

Dimensions: $(d_{00}{:}\, d_{10}, 0{:}\, 0, 0{:}\, 0)$, where d_{ij} = **dimension**(i,j).
Derivative at the discretization points, for a one-dimensional grid. **centerDerivative1D** is aliased to **centerDerivative** (§3.2.28), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEcenterDerivative1D);           // Update centerDerivative1D.
```

See also **THEcenterDerivative1D** (§3.1.14) and **update(what,how)** (§3.5.13).

3.2.31 RealMappedGridFunction inverseVertexDerivative

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1, 0{:}\, n_1)$, where d_{ij} = **dimension**(i,j) and n_1 = **numberOfDimensions** $-$ 1.
inverseVertexDerivative holds the inverse of the derivative of the mapping at the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$[\textbf{inverseVertexDerivative}(i_0,i_1,i_2,*,*)] = [\textbf{vertexDerivative}(i_0,i_1,i_2,*,*)]^{-1}\,.$$

This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEinverseVertexDerivative);      // Update inverseVertexDerivative.
```

See also **vertexDerivative** (§3.2.25), **THEinverseVertexDerivative** (§3.1.15), **update(what,how)** (§3.5.13) and **rsxy** (§3.3.21).

3.2.32 RealMappedGridFunction inverseVertexDerivative2D

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
Inverse derivative at the vertices, for a one- or two-dimensional grid. **inverseVertexDerivative2D** is aliased to **inverseVertexDerivative** (§3.2.31), but has fewer dimensions, so that it may be used to simplify two-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                                    // Construct a MappedGrid.
g.update(MappedGrid::THEinverseVertexDerivative2D);              // Update inverseVertexDerivative2D.
```

See also **THEinverseVertexDerivative2D** (§3.1.16) and **update(what,how)** (§3.5.13).

3.2.33 RealMappedGridFunction inverseVertexDerivative1D

Dimensions: $(d_{00}{:}\,d_{10}, 0{:}\,0, 0{:}\,0)$, where $d_{ij} =$ **dimension**(i,j).
Inverse derivative at the vertices, for a one-dimensional grid. See also **inverseVertexDerivative** (§3.2.31). **inverseVertexDerivative1D** is aliased to **inverseVertexDerivative** (§3.2.31), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                                    // Construct a MappedGrid.
g.update(MappedGrid::THEinverseVertexDerivative1D);              // Update inverseVertexDerivative1D.
```

See also **THEinverseVertexDerivative1D** (§3.1.17) and **update(what,how)** (§3.5.13).

3.2.34 RealMappedGridFunction inverseCenterDerivative

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
inverseCenterDerivative holds the inverse of the derivative of the mapping at the discretization points of the grid, (*e.g.*, the vertices or the grid cell-centers), including any ghost points (which lie outside the grid).

$$[\textbf{inverseCenterDerivative}(i_0,i_1,i_2,*,*)] = [\textbf{centerDerivative}(i_0,i_1,i_2,*,*)]^{-1}\,.$$

This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                                    // Construct a MappedGrid.
g.update(MappedGrid::THEinverseCenterDerivative);                // Update inverseCenterDerivative.
```

See also **centerDerivative** (§3.2.28), **THEinverseCenterDerivative** (§3.1.18) and **update(what,how)** (§3.5.13).

3.2.35 RealMappedGridFunction inverseCenterDerivative2D

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
Inverse derivative at the discretization points, for a one- or two-dimensional grid. **inverseCenterDerivative2D** is aliased to **inverseCenterDerivative** (§3.2.34), but has fewer dimensions, so that it may be used to simplify two-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                                    // Construct a MappedGrid.
g.update(MappedGrid::THEinverseCenterDerivative2D);              // Update inverseCenterDerivative2D.
```

See also **THEinverseCenterDerivative2D** (§3.1.19) and **update(what,how)** (§3.5.13).

3.2.36 RealMappedGridFunction inverseCenterDerivative1D

Dimensions: $(d_{00}{:}\,d_{10}, 0{:}\,0, 0{:}\,0)$, where $d_{ij} =$ **dimension**(i,j).
Inverse derivative at the discretization points, for a one-dimensional grid. **inverseCenterDerivative1D** is aliased to **inverseCenterDerivative** (§3.2.34), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                                    // Construct a MappedGrid.
g.update(MappedGrid::THEinverseCenterDerivative1D);              // Update inverseCenterDerivative1D.
```

See also **THEinverseCenterDerivative1D** (§3.1.20) and **update(what,how)** (§3.5.13).

3.2.37 RealMappedGridFunction vertexJacobian

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} =$ **dimension**(i,j).
vertexDerivative holds the determinant of the derivative of the mapping at the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$\textbf{vertexJacobian}(i_0,i_1,i_2) = \det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right], \quad \text{where} \quad r_j = \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j),$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **vertexJacobian** is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed using the same approximation to the derivative as that used for **vertexDerivative** (§3.2.25). This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                                    // Construct a MappedGrid.
g.update(MappedGrid::THEvertexJacobian);                         // Update vertexJacobian.
```

See also **vertexDerivative** (§3.2.25), **THEvertexJacobian** (§3.1.21) and **update(what,how)** (§3.5.13).

3.2.38 RealMappedGridFunction centerJacobian

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} =$ **dimension**(i,j).
centerJacobian holds the determinant of the derivative of the mapping at the discretization points of the grid, including any ghost points (which lie outside the grid).

$$\textbf{centerJacobian}(i_0,i_1,i_2) = \det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right], \quad \text{where}$$

$$r_j = \begin{cases} \big(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **centerJacobian** is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed using the same approximation to the derivative as that used for **centerDerivative** (§3.2.28). This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                                    // Construct a MappedGrid.
g.update(MappedGrid::THEcenterJacobian);                         // Update centerJacobian.
```

See also **centerDerivative** (§3.2.28), **THEcenterJacobian** (§3.1.22) and **update(what,how)** (§3.5.13).

3.2.39 RealMappedGridFunction cellVolume

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} = \textbf{dimension}(i,j)$.
cellVolume holds the volumes of cells of the grid, including any ghost cells (which lie outside the grid).

$$\textbf{cellVolume}(i_0,i_1,i_2) = \Big(\prod_j \Delta r_j\Big) \det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right], \quad \text{where} \quad r_j = \Big(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\Big)\,\textbf{gridSpacing}(j),$$

$\Delta r_j = \textbf{gridSpacing}(j)$, $\textbf{dimension}(0,j) \le i_j \le \textbf{dimension}(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. **cellVolume** has the same sign as the determinant of the derivative of the mapping. (*I.e.*, it may be negative.) If **cellVolume** is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed in one dimension as the distance between surrounding vertices, approximated in two dimensions by the area of the polygon bounded by the surrounding vertices, and approximated in three dimensions by the volume of the solid bounded by the surrounding vertices, with the approximation that the four vertices of each face are assumed to be coplanar. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                  // Construct a MappedGrid.
g.update(MappedGrid::THEcellVolume);           // Update cellVolume.
```

See also **THEcellVolume** (§3.1.23) and **update(what,how)** (§3.5.13).

3.2.40 RealMappedGridFunction faceNormal

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} = \textbf{dimension}(i,j)$ and $n_1 = \textbf{numberOfDimensions} - 1$.
faceNormal holds vectors normal to cell faces (including any ghost cell faces), normalized to the cell-face area. The normal to cell face (i_0,i_1,i_2) corresponding to constant r_j is given by

$$\textbf{faceNormal}(i_0,i_1,i_2,*,j) = \Delta r_k \Delta r_l \frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l}, \quad \text{where } k = (j+1) \bmod 3,\ l = (j+2) \bmod 3,$$

where $\Delta r_j = 1$ for $j > n_0$, $\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 = \textbf{numberOfDimensions} - 1$, $\Delta r_j = \textbf{gridSpacing}(j)$,

$$r_k = \begin{cases} \big(i_k - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{if } k = j \\ \big(i_k + \frac{1}{2} - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{otherwise,} \end{cases}$$

$\textbf{dimension}(0,k) \le i_k \le \textbf{dimension}(1,k)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, $\textbf{faceNormal}(i_0,i_1,i_2,0,0) = 1$ if $\textbf{numberOfDimensions} = 1$. If **faceNormal** is updated using a discrete approximation, then the derivatives are not computed using the mapping, but are instead computed by centered finite differences of **vertex**, (averaged in three dimensions to the centers of cell faces). As a result, in two dimensions,

$$\begin{aligned} \textbf{faceNormal}(i,j,k,*,0) &= \Delta_{+j}\textbf{vertex}(i,j,k,*) \\ \textbf{faceNormal}(i,j,k,*,1) &= -\Delta_{+i}\textbf{vertex}(i,j,k,*) \end{aligned}$$

and in three dimensions,

$$\begin{aligned} \textbf{faceNormal}(i_0,i_1,i_2,*,j) &= \mu_{+i_l}\Delta_{+i_k}\textbf{vertex}(i_0,i_i,i_2,*) \times \mu_{+i_k}\Delta_{+i_l}\textbf{vertex}(i_0,i_i,i_2,*) \\ &= \frac{1}{2}\Delta_{\nearrow+i_k+i_l}\textbf{vertex}(i_0,i_1,i_2,*) \times \Delta_{\nwarrow+i_l+i_k}\textbf{vertex}(i_0,i_1,i_2,*) \end{aligned}$$

where $k = (j+1) \bmod 3$, $l = (j+2) \bmod 3$, $\Delta_{\nearrow+i+j}u_{ij} \equiv u_{i+1,j+1} - u_{ij}$ and $\Delta_{\nwarrow+i+j}u_{ij} \equiv u_{i,j+1} - u_{i+1,j}$. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                  // Construct a MappedGrid.
g.update(MappedGrid::THEfaceNormal);           // Update faceNormal.
```

See also **THEfaceNormal** (§3.1.24) and **update(what,how)** (§3.5.13).

3.2.41 RealMappedGridFunction faceNormal2D

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
Cell-face normal vector, for a one- or two-dimensional grid. **faceNormal2D** is aliased to **faceNormal** (§3.2.40), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                   // Construct a MappedGrid.
g.update(MappedGrid::THEfaceNormal2D);          // Update faceNormal2D.
```

See also **THEfaceNormal2D** (§3.1.25) and **update(what,how)** (§3.5.13).

3.2.42 RealMappedGridFunction faceNormal1D

Dimensions: $(d_{00}{:}\,d_{10}, 0{:}\,0, 0{:}\,0)$, where $d_{ij} =$ **dimension**(i,j).
Cell-face normal vector, for a one-dimensional grid. **faceNormal1D** is aliased to **faceNormal** (§3.2.40), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                   // Construct a MappedGrid.
g.update(MappedGrid::THEfaceNormal1D);          // Update faceNormal1D.
```

See also **THEfaceNormal1D** (§3.1.26) and **update(what,how)** (§3.5.13).

3.2.43 RealMappedGridFunction centerNormal

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
The normal to the surface corresponding to constant r_j and passing through the discretization point (i_0, i_1, i_2), normalized to the area of that portion of this surface which corresponds one cell, is given by

$$\textbf{centerNormal}(i_0,i_1,i_2,*,j) = \Delta r_k \Delta r_l \frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l}, \quad \text{where } k = (j+1) \bmod 3,\ l = (j+2) \bmod 3,$$

where $\Delta r_j = 1$ for $j > n_0$, $\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 =$ **numberOfDimensions** $- 1$, $\Delta r_j =$ **gridSpacing**(j),

$$r_k = \begin{cases} \left(i_k + \frac{1}{2} - \textbf{indexRange}(0,k)\right) \textbf{gridSpacing}(k) & \text{if } \textbf{isCellCentered}(k) \\ \left(i_k - \textbf{indexRange}(0,k)\right) \textbf{gridSpacing}(k) & \text{otherwise,} \end{cases}$$

dimension$(0,k) \le i_k \le$ **dimension**$(1,k)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, **centerNormal**$(i_0,i_1,i_2,0,0) = 1$ if **numberOfDimensions** $= 1$. In fact, **centerNormal** is related to **inverseCenterDerivative** (§3.2.34) and **centerJacobian** (§3.2.38):

$$[\textbf{centerNormal}(i_0,i_1,i_2,*,*)] = \Big(\prod_j \Delta r_j\Big) \textbf{centerJacobian}(i_0,i_1,i_2)\,[\textbf{inverseCenterDerivative}(i_0,i_1,i_2,*,*)]^{\mathrm{T}}.$$

If **centerNormal** is updated using a discrete approximation, then it is obtained by averaging **faceNormal** (§3.2.40) from the cell-face centers to the the discretization points. *Warning: if neither* **isAllCellCentered** *nor* **isAllVertexCentered** *then computation of* **centerNormal** *using a discrete approximation is not implemented. Please complain if you need this case, and I will implement it.* This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                   // Construct a MappedGrid.
g.update(MappedGrid::THEcenterNormal);          // Update centerNormal.
```

See also **THEcenterNormal** (§3.1.27) and **update(what,how)** (§3.5.13).

3.2.44 RealMappedGridFunction centerNormal2D

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
Cell-center normal vector, for a one- or two-dimensional grid. **centerNormal2D** is aliased to **centerNormal** (§3.2.43), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEcenterNormal2D);               // Update centerNormal2D.
```

See also **THEcenterNormal2D** (§3.1.28) and **update(what,how)** (§3.5.13).

3.2.45 RealMappedGridFunction centerNormal1D

Dimensions: $(d_{00}{:}\,d_{10}, 0{:}\,0, 0{:}\,0)$, where $d_{ij} =$ **dimension**(i,j).
Cell-center normal vector, for a one-dimensional grid. **centerNormal1D** is aliased to **centerNormal** (§3.2.43), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEcenterNormal1D);               // Update centerNormal1D.
```

See also **THEcenterNormal1D** (§3.1.29) and **update(what,how)** (§3.5.13).

3.2.46 RealMappedGridFunction faceArea

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
faceArea holds the area of cell faces (including any ghost cell faces). The area of the cell face (i_0, i_1, i_2) corresponding to constant r_j is given by

$$\textbf{faceArea}(i_0, i_1, i_2, j) = \left|\textbf{faceNormal}(i_0, i_1, i_2, *, j)\right|.$$

This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEfaceArea);                     // Update faceArea.
```

See also **faceNormal** (§3.2.40), **THEfaceArea** (§3.1.30) and **update(what,how)** (§3.5.13).

3.2.47 RealMappedGridFunction faceArea2D

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions** $- 1$.
Cell-face area, for a one- or two-dimensional grid. **faceArea2D** is aliased to **faceArea** (§3.2.46), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEfaceArea2D);                   // Update faceArea2D.
```

See also **THEfaceArea2D** (§3.1.31) and **update(what,how)** (§3.5.13).

3.2.48 RealMappedGridFunction faceArea1D

Dimensions: $(d_{00}{:}\,d_{10}, 0{:}\,0)$, where $d_{ij} = \textbf{dimension}(i,j)$.
Cell-face area, for a one-dimensional grid. **faceArea1D** is aliased to **faceArea** (§3.2.46), but has fewer dimensions, so that it may be used to simplify one-dimensional PDE code. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                   // Construct a MappedGrid.
g.update(MappedGrid::THEfaceArea1D);            // Update faceArea1D.
```

See also **THEfaceArea1D** (§3.1.32) and **update(what,how)** (§3.5.13).

3.2.49 RealArray vertexBoundaryNormal[3][2]

Dimensions of vertexBoundaryNormal[l][k]: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1)$, where

$$d_{ij} = \begin{cases} \textbf{dimension}(k,j) & \text{if } j = l \\ \textbf{dimension}(i,j) & \text{otherwise} \end{cases}$$

and $n_1 = \textbf{numberOfDimensions} - 1$.
centerBoundaryNormal holds unit outward normal vectors to the boundary at the boundary vertices. The normal corresponding to the side of the grid where $r_j = i$ is given by

$$\textbf{vertexBoundaryNormal}[j][i](i_0,i_1,i_2,*) = \pm(-1)^{i+1} \frac{\frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l}}{\left|\frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l}\right|}, \quad \text{where } k = (j+1) \bmod 3,\ l = (j+2) \bmod 3,$$

where $\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 = \textbf{numberOfDimensions} - 1$,

$$r_k = \big(i_k - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k),$$

$\textbf{dimension}(0,k) \le i_k \le \textbf{dimension}(1,k)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, $\textbf{vertexBoundaryNormal}[0][i](i_0,i_1,i_2,0) = \pm(-1)^{i+1}$ if $\textbf{numberOfDimensions} = 1$. The upper sign is taken if the coordinate system is right-handed and the lower sign if it is left-handed; the sign taken is that of the jacobian of the mapping, $\det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right]$ at the center of the grid $r_0 = r_1 = r_2 = \frac{1}{2}$. If **vertexBoundaryNormal** is updated using a discrete approximation, then the derivatives are not computed using the mapping, but are instead computed by centered finite differences of **vertex**. As a result, in two dimensions,

$$\begin{aligned} \textbf{vertexBoundaryNormal}[0][i](i_0,i_1,i_2,*) &= \pm(-1)^{i+1} \frac{\Delta_{0i_1} \textbf{vertex}(i_0,i_1,i_2,*)}{\left|\Delta_{0i_1} \textbf{vertex}(i_0,i_1,i_2,*)\right|} \\ \textbf{vertexBoundaryNormal}[1][i](i_0,i_1,i_2,*) &= \mp(-1)^{i+1} \frac{\Delta_{0i_0} \textbf{vertex}(i_0,i_1,i_2,*)}{\left|\Delta_{0i_0} \textbf{vertex}(i_0,i_1,i_2,*)\right|}. \end{aligned}$$

and in three dimensions,

$$\textbf{vertexBoundaryNormal}[j][i](i_0,i_1,i_2,*) = \pm(-1)^{i+1} \frac{\Delta_{0i_k} \textbf{vertex}(i_0,i_i,i_2,*) \times \Delta_{0i_l} \textbf{vertex}(i_0,i_i,i_2,*)}{\left|\Delta_{0i_k} \textbf{vertex}(i_0,i_i,i_2,*) \times \Delta_{0i_l} \textbf{vertex}(i_0,i_i,i_2,*)\right|}$$

where $k = (j+1) \bmod 3$, $l = (j+2) \bmod 3$, $\Delta_{\nearrow+i+j} u_{ij} \equiv u_{i+1,j+1} - u_{ij}$ and $\Delta_{\nwarrow+i+j} u_{ij} \equiv u_{i,j+1} - u_{i+1,j}$. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEvertexBoundaryNormal);         // Update vertexBoundaryNormal.
```

See also **THEvertexBoundaryNormal** (§3.1.33) and **update(what,how)** (§3.5.13).

3.2.50 RealArray centerBoundaryNormal[3][2]

Dimensions of centerBoundaryNormal[l][k]: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1)$, where

$$d_{ij} = \begin{cases} \textbf{dimension}(k,j) & \text{if } j = l \\ \textbf{dimension}(i,j) & \text{otherwise} \end{cases}$$

and $n_1 = \textbf{numberOfDimensions} - 1$.
centerBoundaryNormal holds unit outward normal vectors to the boundary at the discretization points of the boundary. Note that for a cell-centered grid, these points are not the cell-centers of boundary cells, but are the centers of the faces of boundary cells. The normal corresponding to the side of the grid where $r_j = i$ is given by

$$\textbf{centerBoundaryNormal}[j][i](i_0,i_1,i_2,*) = \pm(-1)^{i+1}\frac{\frac{\partial \mathbf{g}}{\partial r_k}\times\frac{\partial \mathbf{g}}{\partial r_l}}{\left|\frac{\partial \mathbf{g}}{\partial r_k}\times\frac{\partial \mathbf{g}}{\partial r_l}\right|}, \quad \text{where } k=(j+1) \bmod 3,\ l=(j+2) \bmod 3,$$

where $\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 = \textbf{numberOfDimensions} - 1$,

$$r_k = \begin{cases} i & \text{if } k = j, \text{ or else} \\ \left(i_k + \frac{1}{2} - \textbf{indexRange}(0,k)\right)\textbf{gridSpacing}(k) & \text{if } \textbf{isCellCentered}(k) \\ \left(i_k - \textbf{indexRange}(0,k)\right)\textbf{gridSpacing}(k) & \text{otherwise,} \end{cases}$$

$\textbf{dimension}(0,k) \le i_k \le \textbf{dimension}(1,k)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, $\textbf{centerBoundaryNormal}[0][i](i_0,i_1,i_2,0) = \pm(-1)^{i+1}$ if $\textbf{numberOfDimensions} = 1$. The upper sign is taken if the coordinate system is right-handed and the lower sign if it is left-handed; the sign taken is that of the jacobian of the mapping, $\det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right]$ at the center of the grid $r_0 = r_1 = r_2 = \frac{1}{2}$. If **centerBoundaryNormal** is updated using a discrete approximation, then the derivatives are not computed using the mapping, but are instead computed by centered finite differences of **vertex**. As a result, in two dimensions,

$$\begin{aligned} \textbf{centerBoundaryNormal}[0][i](i_0,i_1,i_2,*) &= \pm(-1)^{i+1}\frac{\Delta_{+i_1}\textbf{vertex}(i_0,i_1,i_2,*)}{\left|\Delta_{+i_1}\textbf{vertex}(i_0,i_1,i_2,*)\right|} \\ \textbf{centerBoundaryNormal}[1][i](i_0,i_1,i_2,*) &= \mp(-1)^{i+1}\frac{\Delta_{+i_0}\textbf{vertex}(i_0,i_1,i_2,*)}{\left|\Delta_{+i_0}\textbf{vertex}(i_0,i_1,i_2,*)\right|} \end{aligned}$$

if **isAllCellCentered** and

$$\begin{aligned} \textbf{centerBoundaryNormal}[0][i](i_0,i_1,i_2,*) &= \pm(-1)^{i+1}\frac{\Delta_{0i_1}\textbf{vertex}(i_0,i_1,i_2,*)}{\left|\Delta_{0i_1}\textbf{vertex}(i_0,i_1,i_2,*)\right|} \\ \textbf{centerBoundaryNormal}[1][i](i_0,i_1,i_2,*) &= \mp(-1)^{i+1}\frac{\Delta_{0i_0}\textbf{vertex}(i_0,i_1,i_2,*)}{\left|\Delta_{0i_0}\textbf{vertex}(i_0,i_1,i_2,*)\right|} \end{aligned}$$

if **isAllVertexCentered**. In three dimensions,

$$\begin{aligned} &\textbf{centerBoundaryNormal}[j][i](i_0,i_1,i_2,*) \\ &\quad= \pm(-1)^{i+1}\frac{\mu_{+i_l}\Delta_{+i_k}\textbf{vertex}(i_0,i_i,i_2,*)\times\mu_{+i_k}\Delta_{+i_l}\textbf{vertex}(i_0,i_i,i_2,*)}{\left|\mu_{+i_l}\Delta_{+i_k}\textbf{vertex}(i_0,i_i,i_2,*)\times\mu_{+i_k}\Delta_{+i_l}\textbf{vertex}(i_0,i_i,i_2,*)\right|} \\ &\quad= \pm(-1)^{i+1}\frac{\Delta_{\nearrow+i_k+i_l}\textbf{vertex}(i_0,i_1,i_2,*)\times\Delta_{\nwarrow+i_l+i_k}\textbf{vertex}(i_0,i_1,i_2,*)}{\left|\Delta_{\nearrow+i_k+i_l}\textbf{vertex}(i_0,i_1,i_2,*)\times\Delta_{\nwarrow+i_l+i_k}\textbf{vertex}(i_0,i_1,i_2,*)\right|} \end{aligned}$$

if **isAllCellCentered**, where $k = (j+1) \bmod 3$, $l = (j+2) \bmod 3$, $\Delta_{\nearrow+i+j}u_{ij} \equiv u_{i+1,j+1} - u_{ij}$ and $\Delta_{\nwarrow+i+j}u_{ij} \equiv u_{i,j+1} - u_{i+1,j}$, and

$$\textbf{centerBoundaryNormal}[j][i](i_0,i_1,i_2,*) = \pm(-1)^{i+1}\frac{\Delta_{0i_k}\textbf{vertex}(i_0,i_i,i_2,*)\times\Delta_{0i_l}\textbf{vertex}(i_0,i_i,i_2,*)}{\left|\Delta_{0i_k}\textbf{vertex}(i_0,i_i,i_2,*)\times\Delta_{0i_l}\textbf{vertex}(i_0,i_i,i_2,*)\right|}$$

if **isAllVertexCentered**. *Warning: if neither* **isAllCellCentered** *nor* **isAllVertexCentered** *then computation of* **centerBoundaryNormal** *using a discrete approximation is not implemented. Please complain if you need this case, and I will implement it.* This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                                          // Construct a MappedGrid.
g.update(MappedGrid::THEcenterBoundaryNormal);         // Update centerBoundaryNormal.
```

See also **THEcenterBoundaryNormal** (§3.1.34) and **update(what,how)** (§3.5.13).

3.2.51 MappingRC mapping

mapping is the reference-counted mapping that generates the grid. The mapping may be replaced as in the following example.

Example

```
MappedGrid g;                                          // Construct the MappedGrid g.
Mapping& square = *new SquareMapping(0.,1.,0.,1.);     // Construct the SquareMapping square.
square.incrementReferenceCount();                      // Increment the reference count of square.
g.reference(square);                                   // Replace g.mapping with square.
square.decrementReferenceCount();                      // Decrement the reference count of square.
```

See also **reference(const Mapping& x)** (§3.5.8) and **reference(const MappingRC& x)** (§3.5.9).

3.3 Public composite grid data used by Fortran programs

The composite grid data used by Fortran programs and listed in Table 3, page 6 of *Composite Grid Data: All You Never Wanted to Know and You were afraid to Ask* (*q.v.* for full details) are accessible to C++ classes through these public data members.

3.3.1 IntegerArray bc

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
$\mathbf{bc}(i,j) = \mathbf{boundaryCondition}(j,i)$. See also **boundaryCondition** (§3.2.8).

3.3.2 IntegerArray bw

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1, 0{:}\,1)$

$$\mathbf{bw}(i,j,k) = \begin{cases} \mathbf{boundaryDiscretizationWidth}(j,i) & \text{if } j = k \\ \mathbf{discretizationWidth}(k) & \text{otherwise} \end{cases}$$

See also **boundaryDiscretizationWidth** (§3.2.7).

3.3.3 LogicalArray cctype

Dimensions: $(0{:}\,\mathbf{nd}-1)$
$\mathbf{cctype}(i) = \mathbf{isCellCentered}(i)$. See also **isCellCentered** (§3.2.12).

3.3.4 RealArray ci

Dimensions: $(0{:}\,\mathbf{ni}-1, 0{:}\,\mathbf{nd}-1)$
$\mathbf{ci}(i,j) = \mathbf{CompositeGrid::interpolationCoordinates}[*](i,j)$.
See also **CompositeGrid::interpolationCoordinates** (§6.2.8).

3.3.5 RealArray drs

Dimensions: $(0{:}\,\mathbf{nd}-1)$
See also **gridSpacing** (§3.2.11).

3.3.6 IntegerArray dw

Dimensions: $(0{:}\,\mathbf{nd}-1)$
See also **discretizationWidth** (§3.2.6).

3.3.7 IntegerArray il

Dimensions: $(0{:}\,\mathbf{ni}-1, 0{:}\,\mathbf{nd})$

$$\mathbf{il}(i,j) = \begin{cases} \mathbf{CompositeGrid::interpoleeGrid}[*](i) & \text{if } j = \mathbf{nd} \\ \mathbf{CompositeGrid::interpoleeLocation}[*](i,j) & \text{otherwise} \end{cases}$$

See also **CompositeGrid::interpoleeGrid** (§6.2.9) and **CompositeGrid::interpoleeLocation** (§6.2.10)

3.3.8 IntegerArray ip

Dimensions: $(0{:}\,\mathbf{ni}-1, 0{:}\,\mathbf{nd}-1)$
$\mathbf{ip}(i,j) =$ **CompositeGrid::interpolationPoint**$[*](i,j)$.
See also **CompositeGrid::interpolationPoint** (§6.2.11).

3.3.9 RealArray iq

Dimensions: $(0{:}\,\mathbf{ni}-1)$
$\mathbf{ip}(i) =$ **CompositeGrid::interpolationPoint**$[*](i)$.
See also **CompositeGrid::interpolationCondition** (§6.2.12).

3.3.10 IntegerR kgrid

kgrid is not defined. It is recommended that **kgrid** not be used for any purpose.

3.3.11 IntegerMappedGridFunction kr

$$\text{Dimensions: } \begin{cases} (d_{00}{:}\,d_{01}) & \text{if } \mathbf{nd}=1 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}) & \text{if } \mathbf{nd}=2 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, d_{20}{:}\,d_{21}) & \text{if } \mathbf{nd}=3, \end{cases} \quad \text{where } d_{ij} = \mathbf{ndrsab}(i,j).$$

kr is aliased to **mask** (§3.2.18), although it has fewer dimensions than **mask** if $\mathbf{nd} < 3$. This data may be updated as in the following example.

Example

```
MappedGrid g;                          // Construct a MappedGrid.
g.update(MappedGrid::THEkr);           // Update kr.
```

See also **THEkr** (§3.1.37) and **update(what,how)** (§3.5.13).

3.3.12 IntegerArray mrsab

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
$\mathbf{mrsab}(i,j) =$ **indexRange**(j,i). See also **indexRange** (§3.2.3).

3.3.13 IntegerR nd

nd is aliased to **numberOfDimensions** (§3.2.1).

3.3.14 IntegerArray ndrsab

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
ndrsab(i,j) = **dimension**(j,i). See also **dimension** (§3.2.2).

3.3.15 IntegerR ni

ni = **CompositeGrid::numberOfInterpolationPoints**$(*)$.
See also **CompositeGrid::numberOfInterpolationPoints** (§6.2.3).

3.3.16 IntegerArray nrsab

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
nrsab(i,j) = **gridIndexRange**(j,i). See also **gridIndexRange** (§3.2.4).

3.3.17 IntegerArray nxtra

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
nxtra(i,j) = **numberOfGhostPoints**(j,i). See also **numberOfGhostPoints** (§3.2.5).

3.3.18 LogicalArray period

Dimensions: $(0{:}\,\mathbf{nd}-1)$

$$\mathbf{period}(i) = \begin{cases} \mathbf{LogicalFalse} & \text{if } \mathbf{isPeriodic}(i) = \mathbf{Mapping::notPeriodic} \\ \mathbf{LogicalTrue} & \text{otherwise} \end{cases}$$

See also **isPeriodic** (§3.2.15).

3.3.19 IntegerArray refine

Dimensions: $(0{:}\,2)$

3.3.20 RealArray rsab

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$

3.3.21 RealMappedGridFunction rsxy

$$\text{Dimensions: } \begin{cases} (d_{00}{:}\,d_{01}, 0{:}\,\mathbf{nd}-1, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=1 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, 0{:}\,\mathbf{nd}-1, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=2 \quad \text{where } d_{ij} = \mathbf{ndrsab}(i,j). \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, d_{20}{:}\,d_{21}, 0{:}\,\mathbf{nd}-1, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=3, \end{cases}$$

rsxy is aliased to **inverseVertexDerivative** (§3.2.31), although it has fewer dimensions than **inverseVertexDerivative** if **nd** < 3. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                          // Construct a MappedGrid.
g.update(MappedGrid::THErsxy);         // Update rsxy.
```

See also **THErsxy** (§3.1.38) and **update(what,how)** (§3.5.13).

3.3.22 IntegerArray share

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
See also **sharedBoundaryFlag** (§3.2.9).

3.3.23 RealArray sheps

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
See also **sharedBoundaryTolerance** (§3.2.10).

3.3.24 RealArray trxyab

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$

3.3.25 IntegerArray version

Dimensions: $(0{:}\,2)$

3.3.26 RealMappedGridFunction xy

Dimensions: $\begin{cases} (d_{00}{:}\,d_{01}, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=1 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=2 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, d_{20}{:}\,d_{21}, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=3, \end{cases}$ where $d_{ij} = \mathbf{ndrsab}(i,j)$.

xy is aliased to **vertex** (§3.2.19), although it has fewer dimensions than **vertex** if $\mathbf{nd} < 3$. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                          // Construct a MappedGrid.
g.update(MappedGrid::THExy);           // Update xy.
```

See also **THExy** (§3.1.39) and **update(what,how)** (§3.5.13).

3.3.27 RealArray xyab

Dimensions: $(0{:}\,\mathbf{nd}-1, 0{:}\,1)$
xyab contains coordinate bounds in space for the grid.

3.3.28 RealMappedGridFunction xyc

Dimensions: $\begin{cases} (d_{00}{:}\,d_{01}, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=1 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=2 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, d_{20}{:}\,d_{21}, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=3, \end{cases}$ where $d_{ij} = \mathbf{ndrsab}(i,j)$.

xyc is aliased to **center** (§3.2.22), although it has fewer dimensions than **center** if $\mathbf{nd} < 3$. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                          // Construct a MappedGrid.
g.update(MappedGrid::THExyc);          // Update xyc.
```

See also **THExyc** (§3.1.40) and **update(what,how)** (§3.5.13).

3.3.29 RealMappedGridFunction xyrs

Dimensions: $\begin{cases} (d_{00}{:}\,d_{01}, 0{:}\,\mathbf{nd}-1, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=1 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, 0{:}\,\mathbf{nd}-1, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=2 \\ (d_{00}{:}\,d_{01}, d_{10}{:}\,d_{11}, d_{20}{:}\,d_{21}, 0{:}\,\mathbf{nd}-1, 0{:}\,\mathbf{nd}-1) & \text{if } \mathbf{nd}=3, \end{cases}$ where $d_{ij} = \mathbf{ndrsab}(i,j)$.

xyrs is aliased to **vertexDerivative** (§3.2.25), although it has fewer dimensions than **vertexDerivative** if $\mathbf{nd} < 3$. This geometric data may be updated as in the following example.

Example

```
MappedGrid g;                          // Construct a MappedGrid.
g.update(MappedGrid::THExyrs);         // Update xyrs.
```

See also **THExyrs** (§3.1.41) and **update(what,how)** (§3.5.13).

3.3.30 RealArray xyzp

Dimensions: (0:2,0:2,0:1)
xyzp(*,j,1), contains a basis of three vectors ($0 < j \leq 2$) for space. If space is **xyzper**-periodic, then the first **xyzper** vectors are period vectors of space. **xyzp**(*,*,1) contains the matrix inverse of **xyzp**(*,*,0).

3.3.31 IntegerR xyzper

xyzper is the number of directions in which space is periodic.

3.4 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

3.4.1 MappedGridData* rcData

Pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§3.5.18) and **operator*()** (§3.5.19), which are provided for access to **rcData**.

3.4.2 Logical isCounted

Flag that indicates whether the data pointed to by **rcData** (§3.4.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

3.5 Public member functions

3.5.1 MappedGrid(const Integer numberOfDimensions_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null **MappedGrid**. Otherwise, create a **MappedGrid** with the given number of dimensions.

Example

```
MappedGrid(2) g;   // Construct a two-dimensional MappedGrid.
```

3.5.2 MappedGrid(const MappedGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default.

Example

```
MappedGrid g1;                                  // Construct a MappedGrid.
MappedGrid g2=g1, g3(g1);                       // Construct a MappedGrid using deep copy.
MappedGrid g4(g1,MappedGrid::SHALLOW);          // Construct using shallow copy; g2 shares the data of g1.
```

See also **operator=(x)** (§3.5.6) and **reference(const MappedGrid& x)** (§3.5.7).

3.5.3 MappedGrid(Mapping& mapping_)

Constructor from a mapping.

Example

```
Mapping& square = *new SquareMapping(0.,1.,0.,1.);   // Construct the SquareMapping square.
square.incrementReferenceCount();                    // Increment the reference count of square.
MappedGrid g(square);                                // Construct the MappedGrid g from square.
square.decrementReferenceCount();                    // Decrement the reference count of square.
```

Omission of the calls to **incrementReferenceCount**() and **decrementReferenceCount**() does not cause an error. It does cause a memory leak, however, as the **Mapping** will as a result be marked "uncounted" and the **delete** operator will never be called to match the call to the **new** operator. On the other hand, it is dangerous and is an error to construct a **MappedGrid** from a **Mapping** that was constructed on the stack (not using the **new** operator), because this **Mapping** may go out of scope before some object that references it, either directly or indirectly, as in the following example.

Example

```
// THIS EXAMPLE DEMONSTRATES INCORRECT USE OF MappedGrid::MappedGrid(Mapping&).
SquareMapping square(0.,1.,0.,1.);   // Construct the SquareMapping square.
MappedGrid g(square);                // Construct the MappedGrid g from square.
```

3.5.4 MappedGrid(MappingRC& mapping_)

Constructor from a mapping.

Example

```
Mapping& square = *new SquareMapping(0.,1.,0.,1.);   // Construct the SquareMapping square.
square.incrementReferenceCount();                    // Increment the reference count of square.
MappingRC map(square);                               // Construct the MappingRC map from square.
square.decrementReferenceCount();                    // Decrement the reference count of square.
MappedGrid g(map);                                   // Construct the MappedGrid g from map.
```

Omission of the calls to **incrementReferenceCount**() and **decrementReferenceCount**() does not cause an error. It does cause a memory leak, however, as the **Mapping** will as a result be marked "uncounted" and the **delete** operator will never be called to match the call to the **new** operator. On the other hand, it is dangerous and is an error to construct a **MappedGrid** from a **Mapping** that was constructed on the stack (not using the **new** operator), because this **Mapping** may go out of scope before some object that references it, either directly or indirectly, as in the following example.

Example

```
// THIS EXAMPLE DEMONSTRATES INCORRECT USE OF MappingRC::MappingRC(Mapping&).
SquareMapping square(0.,1.,0.,1.);   // Construct the SquareMapping square.
MappingRC map(square);               // Construct the MappingRC map from square.
MappedGrid g(map);                   // Construct the MappedGrid g from map.
```

3.5.5 virtual ~MappedGrid()

Destructor.

3.5.6 MappedGrid& operator=(const MappedGrid& x)

Assignment operator. This is also called a deep copy.

Example

```
MappedGrid g1, g2;   // Construct some MappedGrids.
g2 = g1;             // Copy data from g1 to g2.
```

3.5.7 void reference(const MappedGrid& x)

Make a reference. This is also called a shallow copy. This **MappedGrid** shares the data of **x**.

Example

```
MappedGrid g1, g2;   // Construct some MappedGrids.
g2.reference(g1);    // Now g2 shares the data of g1.
```

3.5.8 void reference(Mapping& x)

Use a given mapping.

Example

```
MappedGrid g;                                          // Construct the MappedGrid g.
Mapping& square = *new SquareMapping(0.,1.,0.,1.);     // Construct the SquareMapping square.
square.incrementReferenceCount();                      // Increment the reference count of square.
g.reference(square);                                   // Replace g.mapping with square.
square.decrementReferenceCount();                      // Decrement the reference count of square.
```

Omission of the calls to **incrementReferenceCount**() and **decrementReferenceCount**() does not cause an error. It does cause a memory leak, however, as the **Mapping** will as a result be marked "uncounted" and the **delete** operator will never be called to match the call to the **new** operator. On the other hand, it is dangerous and is an error to reference a **Mapping** that was constructed on the stack (not using the **new** operator), because this **Mapping** may go out of scope before some object that references it, either directly or indirectly, as in the following example.

Example

```
// THIS EXAMPLE DEMONSTRATES INCORRECT USE OF MappedGrid::reference().
MappedGrid g;                          // Construct the MappedGrid g.
SquareMapping square(0.,1.,0.,1.);     // Construct the SquareMapping square.
g.reference(square);                   // Replace g.mapping with square.
```

3.5.9 void reference(MappingRC& x)

Use a given reference-counted mapping.

Example

```
Mapping& square = *new SquareMapping(0.,1.,0.,1.);     // Construct the SquareMapping square.
square.incrementReferenceCount();                      // Increment the reference count of square.
MappingRC map(square);                                 // Construct the MappingRC map from square.
square.decrementReferenceCount();                      // Decrement the reference count of square.
g.reference(map);                                      // Replace g.mapping with map.
```

Omission of the calls to **incrementReferenceCount**() and **decrementReferenceCount**() does not cause an error. It does cause a memory leak, however, as the **Mapping** will as a result be marked "uncounted" and the **delete** operator will never be called to match the call to the **new** operator. On the other hand, it is dangerous and is an error to reference a **Mapping** that was constructed on the stack (not using the **new** operator), because this **Mapping** may go out of scope before some object that references it, either directly or indirectly, as in the following

example.

Example

```
// THIS EXAMPLE DEMONSTRATES INCORRECT USE OF MappedGrid::reference().
SquareMapping square(0.,1.,0.,1.);   // Construct the SquareMapping square.
MappingRC map(square);               // Construct the MappingRC map from square.
MappedGrid g;                        // Construct the MappedGrid g.
g.reference(map);                    // Replace g.mapping with map.
```

3.5.10 virtual void breakReference()

Break a reference. If this **MappedGrid** shares data with any other **MappedGrid**, then this function replaces it with a new copy that does not share data.

Example

```
MappedGrid g1, g2;    // Construct some MappedGrids.
g2.reference(g1);     // Now g2 shares the data of g1.
g2.breakReference();  // Now g2 has a new, unshared copy of the data.
```

3.5.11 virtual Integer get(const GenericDataBase& dir, const String& name)

Copy a **MappedGrid** into a file.

Example

```
HDF_DataBase dir;   // (derived from GenericDataBase)
dir.mount("g.hdf"); // Open a data file.
MappedGrid g;       // Construct a MappedGrid.
g.get(dir, "g");    // Copy g from dir.
dir.unmount();      // Close the data file.
```

3.5.12 virtual Integer put(GenericDataBase& dir, const String& name) const

Copy a **MappedGrid** into a file.

Example

```
HDF_DataBase dir;       // (derived from GenericDataBase)
dir.mount("g.hdf", "I"); // Open a data file.
MappedGrid g;           // Construct a MappedGrid.
g.put(dir, "g");        // Copy g into dir.
dir.unmount();          // Close the data file.
```

3.5.13 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **THEmask** (§3.1.2), **THEvertex** (§3.1.3), **THEvertex2D** (§3.1.4), **THEvertex1D** (§3.1.5), **THEcenter** (§3.1.6), **THEcenter2D** (§3.1.7), **THEcenter1D** (§3.1.8), **THEvertexDerivative** (§3.1.9), **THEvertexDerivative2D** (§3.1.10), **THEvertexDerivative1D** (§3.1.11), **THEcenterDerivative** (§3.1.12), **THEcenterDerivative2D** (§3.1.13), **THEcenterDerivative1D** (§3.1.14), **THEinverseVertexDerivative** (§3.1.15), **THEinverseVertexDerivative2D** (§3.1.16), **THEinverseVertexDerivative1D** (§3.1.17), **THEinverseCenterDerivative** (§3.1.18),

THEinverseCenterDerivative2D (§3.1.19), **THEinverseCenterDerivative1D** (§3.1.20), **THEvertexJacobian** (§3.1.21), **THEcenterJacobian** (§3.1.22), **THEcellVolume** (§3.1.23), **THEfaceNormal** (§3.1.24), **THEfaceNormal2D** (§3.1.25), **THEfaceNormal1D** (§3.1.26), **THEcenterNormal** (§3.1.27), **THEcenterNormal2D** (§3.1.28), **THEcenterNormal1D** (§3.1.29), **THEfaceArea** (§3.1.30), **THEfaceArea2D** (§3.1.31), **THEfaceArea1D** (§3.1.32), **THEvertexBoundaryNormal** (§3.1.33), **THEcenterBoundaryNormal** (§3.1.34), **THEminMaxEdgeLength** (§3.1.35), **THEtrxyab** (§3.1.36), **THEkr** (§3.1.37), **THErsxy** (§3.1.38), **THExy** (§3.1.39), **THExyc** (§3.1.40), **THExyrs** (§3.1.41), **THEusualSuspects** (§3.1.42) and **EVERYTHING** (§3.1.43), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.4.9), may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§2.1.6), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. Any combination of the constants **USEdifferenceApproximation** (§3.1.44), **COMPUTEgeometry** (§3.1.45), **COMPUTEgeometryAsNeeded** (§3.1.46), **COMPUTEtheUsual** (§3.1.47), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.4.9), may be bitwise ORed together to form the optional second argument of **update**(). The corresponding function **update(what,how)** (§2.4.9) is called with the same arguments for the base class **GenericGrid**.

Example

```
MappedGrid g;                                  // Construct a MappedGrid.
g.update(MappedGrid::EVERYTHING);              // Update all of the MappedGrid geometric data.
```

3.5.14 Integer update(MappedGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another grid (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contains the corresponding data, then the data for this **MappedGrid** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. The corresponding function **update(x,what,how)** (§2.4.10) is called with the same arguments for the base class **GenericGrid**. For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§3.5.13).

Example

```
MappedGrid g1, g2;                             // Construct some MappedGrids.
g1.update(MappedGrid::EVERYTHING);             // Update all of the MappedGrid geometric data of g1
g2.update(g1,MappedGrid::EVERYTHING);          // Update all of the MappedGrid data of g2, sharing g1 data.
```

3.5.15 void destroy(const Integer what = NOTHING)

Destroy the indicated optional geometric grid data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **THEmask** (§3.1.2), **THEvertex** (§3.1.3), **THEvertex2D** (§3.1.4), **THEvertex1D** (§3.1.5), **THEcenter** (§3.1.6), **THEcenter2D** (§3.1.7), **THEcenter1D** (§3.1.8), **THEvertexDerivative** (§3.1.9), **THEvertexDerivative2D** (§3.1.10), **THEvertexDerivative1D** (§3.1.11), **THEcenterDerivative** (§3.1.12), **THEcenterDerivative2D** (§3.1.13), **THEcenterDerivative1D** (§3.1.14), **THEinverseVertexDerivative** (§3.1.15), **THEinverseVertexDerivative2D** (§3.1.16), **THEinverseVertexDerivative1D** (§3.1.17), **THEinverseCenterDerivative** (§3.1.18), **THEinverseCenterDerivative2D** (§3.1.19), **THEinverseCenterDerivative1D** (§3.1.20), **THEvertexJacobian** (§3.1.21), **THEcenterJacobian** (§3.1.22), **THEcellVolume** (§3.1.23), **THEfaceNormal** (§3.1.24), **THEfaceNormal2D** (§3.1.25), **THEfaceNormal1D** (§3.1.26),

THEcenterNormal (§3.1.27), **THEcenterNormal2D** (§3.1.28), **THEcenterNormal1D** (§3.1.29), **THEfaceArea** (§3.1.30), **THEfaceArea2D** (§3.1.31), **THEfaceArea1D** (§3.1.32), **THEvertexBoundaryNormal** (§3.1.33), **THEcenterBoundaryNormal** (§3.1.34), **THEminMaxEdgeLength** (§3.1.35), **THEtrxyab** (§3.1.36), **THEkr** (§3.1.37), **THErsxy** (§3.1.38), **THExy** (§3.1.39), **THExyc** (§3.1.40), **THExyrs** (§3.1.41), **THEusualSuspects** (§3.1.42) and **EVERYTHING** (§3.1.43), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.4.9), may be bitwise ORed together to form the first argument of **update()**. The corresponding function **destroy(what)** (§2.4.11) is called with the same arguments for the base class **GenericGrid**.

Example

```
MappedGrid g;                                  // Construct a MappedGrid.
g.destroy(MappedGrid::EVERYTHING);             // Destroy all of the MappedGrid geometric data.
```

3.5.16 void adjustBoundary(MappedGrid& g2, const IntegerArray& i1, RealArray& r2)

Adjust the inverted coordinates of boundary points in cases where the points lie on a shared boundary.

3.5.17 void getInverseCondition(MappedGrid& g2, const RealArray& xr1, const RealArray& rx2, RealArray& condition)

g2 (INPUT) The **MappedGrid** whose inverse mapping is used.

xr1 (INPUT) Dimensions: $(0{:}2, 0{:}2, 0{:}n)$
The derivative of the mapping of this **MappedGrid**.

rx2 (INPUT) Dimensions: $(0{:}2, 0{:}2, 0{:}n)$
The inverse derivative of the mapping of **MappedGrid g2**.

condition (OUTPUT) Dimensions: $(0{:}n)$
The condition number of the inverse. The number of points n for which the inverse condition is computed is determined by the first dimension of **condition**. The third dimension of **xr1** and **rx2** should be the same as the dimension of **condition**.

Compute the condition number of the mapping inverse at n points.

$$\mathbf{condition}(i) = \left\| \mathrm{diag}\left(1/\mathbf{g2.gridSpacing}(*)\right) \left[\mathbf{rx2}(*,*,i)\right] \left[\mathbf{xr1}(*,*,i)\right] \mathrm{diag}\left(\mathbf{gridSpacing}(*)\right) \right\|_\infty$$

Example

```
MappedGrid g1, g2;                                                    // Construct some MappedGrids.
RealArray r1(3,1), x(3,1), xr1(3,3,1), r2(3,1), rx2(3,3,1), cond(1);  // Construct some RealArrays.
r1 = 0.5; g1.mapping.map(r1, x, xr1);                                 // Compute the location of a point in g1.
g2.mapping.inverseMap(x, r2, rx2);                                    // Invert the mapping of g2 at this point.
g1.getInverseCondition(g2, xf1, rx2, cond);                           // Compute the condition number.
```

3.5.18 MappedGridData* operator->()

Access the reference-counted data.

Example

```
MappedGrid g;                                  // Construct a MappedGrid.
Integer dimensions = g->numberOfDimensions;    // Access the reference-counted data.
```

3.5.19 MappedGridData& operator*()

Access the pointer to the reference-counted data.

Example

```
MappedGrid g;                    // Construct a MappedGrid.
MappedGridData& data = *g;       // Access the pointer to the reference-counted data.
```

3.5.20 virtual String getClassName()

Get the class name of the most-derived class for this object.

Example

```
MappedGrid g;                              // Construct a MappedGrid.
String className = g.getClassName();       // Get the class name of g. It should be "MappedGrid".
```

3.6 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

3.6.1 void reference(MappedGridData& x)

Make a reference to an object of type **MappedGridData**. This **MappedGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

Example

```
MappedGrid g1, g2;   // Construct some MappedGrids.
g2.reference(*g1);   // Now g2 shares the data of g1.
```

3.6.2 void updateReferences()

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

Example

```
MappedGrid g;            // Construct a MappedGrid.
g.updateReferences();    // Update references to the reference-counted data.
```

4 Class GenericGridCollection

Note: You should not need to read this section unless you are designing a derived grid class.

Class GenericGridCollection is the base class for all Overture classes that contain collections of grids.

4.1 Public constants

4.1.1 THEbaseGrid

THEbaseGrid indicates **baseGrid** (§4.2.5), the list of **GenericGridCollections** containing those **GenericGrids** which are descended from the same base grid. See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.2 THEmultigridLevel

THEmultigridLevel indicates **multigridLevel** (§4.2.8), the list of **GenericGridCollections** containing those **GenericGrids** which belong to the same multigrid level. See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.3 THErefinementLevel

THErefinementLevel indicates **refinementLevel** (§4.2.11), the list of **GenericGridCollections** containing those **GenericGrids** which belong to the same refinement level. See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.4 NOTHING

NOTHING = GenericGrid::NOTHING (§2.1.1) See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.5 THEusualSuspects

THEusualSuspects = GenericGrid::THEusualSuspects (§2.1.2)
THEusualSuspects indicates some of the geometric data of a **GenericGridCollection**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.6 THElists

THElists = THEbaseGrid (§4.1.1) | **THEmultigridLevel** (§4.1.2) | **THErefinementLevel** (§4.1.3)
See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.7 EVERYTHING

EVERYTHING = GenericGrid::EVERYTHING (§2.1.3) | **THEbaseGrid** (§4.1.1) | **THEmultigridLevel** (§4.1.2) | **THErefinementLevel** (§4.1.3)
EVERYTHING indicates all of the geometric data associated with a **GenericGridCollection**. See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.8 COMPUTEnothing

COMPUTEnothing = GenericGrid::COMPUTEnothing (§2.1.4)
See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.9 COMPUTEtheUsual

COMPUTEtheUsual = GenericGrid::COMPUTEtheUsual (§2.1.5)
See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.1.10 COMPUTEfailed

COMPUTEfailed = GenericGrid::COMPUTEfailed (§2.1.6)
See also **update(what,how)** (§4.4.10) and **destroy(what)** (§4.4.12).

4.2 Public data

4.2.1 IntegerR computedGeometry

A bit mask that indicates which geometrical data has been computed. This must be reset to zero to invalidate the data when the geometry changes. It is recommended that this variable be used only by derived classes and grid-generation programs. See also **geometryHasChanged(what)** (§4.4.13).

4.2.2 IntegerR numberOfGrids

The number of grids in the list **grid** (§4.2.3).

4.2.3 ListOfGenericGrid grid

Length: **numberOfGrids** (§4.2.2)
A list containing of all of the grids in the collection. The grids in this list are **GenericGrids**.

4.2.4 IntegerR numberOfBaseGrids

The number of **GenericGridCollections** in the list **baseGrid** (§4.2.5). This should be one greater than the maximum of **baseGridNumber** (§4.2.6).

4.2.5 ListOfGenericGridCollection baseGrid

A list of **GenericGridCollections** containing those **GenericGrids** which are descended from the same base grid. The length of this list is **numberOfBaseGrids** (§4.2.4). This data may be updated as in the following example.

Example

```
GenericGridCollection g;                                  // Construct a GenericGridCollection.
g.update(GenericGridCollection::THEbaseGrid);             // Update baseGrid.
```

4.2.6 IntegerArray baseGridNumber

The index of the base grid ancestor of each **GenericGrid** in the list **grid** (§4.2.3).

4.2.7 IntegerR numberOfMultigridLevels

The number of **GenericGridCollections** in the list **multigridLevel** (§4.2.8). This should be one greater than the maximum of **multigridLevelNumber** (§4.2.9).

4.2.8 ListOfGenericGridCollection multigridLevel

A list of **GenericGridCollections** containing those **GenericGrids** which belong to the same multigrid level. The length of this list is **numberOfMultigridLevels** (§4.2.7). This data may be updated as in the following example.

Example

```
GenericGridCollection g;                                  // Construct a GenericGridCollection.
g.update(GenericGridCollection::THEmultigridLevel);       // Update multigridLevel.
```

4.2.9 IntegerArray multigridLevelNumber

The index of the multigrid level of each **GenericGrid** in the list **baseGrid** (§4.2.5).

4.2.10 IntegerR numberOfRefinementLevels

The number of **GenericGridCollections** in the list **refinementLevel** (§4.2.11). This should be one greater than the maximum of **refinementLevelNumber** (§4.2.12).

4.2.11 ListOfGenericGridCollection refinementLevel

A list of **GenericGridCollections** containing those **GenericGrids** which belong to the same refinement level. The length of this list is **numberOfRefinementLevels** (§4.2.10). This data may be updated as in the following example.

Example

```
GenericGridCollection g;                                    // Construct a GenericGridCollection.
g.update(GenericGridCollection::THErefinementLevel);       // Update refinementLevel.
```

4.2.12 IntegerArray refinementLevelNumber

The index of the refinement level of each **GenericGrid** in the list **grid** (§4.2.3).

4.3 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

4.3.1 GenericGridCollectionData* rcData

Pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§4.4.18) and **operator*()** (§4.4.19), which are provided for access to **rcData**.

4.3.2 Logical isCounted

Flag that indicates whether the data pointed to by **rcData** (§4.3.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

4.4 Public member functions

4.4.1 GenericGridCollection(const Integer numberOfGrids_ = 0)

Default constructor. If numberOfGrids_==0 (*e.g.*, by default) then create a null **GenericGridCollection**. Otherwise, create a **GenericGridCollection** with the given number of grids.

Example

```
GenericGridCollection(2) g;   // Construct a GenericGridCollection with two GenericGrids.
```

4.4.2 GenericGridCollection(const GenericGridCollection& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default.

Example

```
GenericGridCollection g1;                  // Construct a GenericGridCollection.
GenericGridCollection g2=g1, g3(g1);       // Construct a GenericGridCollection using deep copy.
                                           // Construct using shallow copy; g2 shares the data of g1.
GenericGridCollection g4(g1,GenericGridCollection::SHALLOW);
```

See also **operator=(x)** (§4.4.4) and **reference(x)** (§4.4.6).

4.4.3 virtual ~GenericGridCollection()

Destructor.

4.4.4 GenericGridCollection& operator=(const GenericGridCollection& x)

Assignment operator. This is also called a deep copy.

Example

```
GenericGridCollection g1, g2;   // Construct some GenericGridCollections.
g2 = g1;                        // Copy data from g1 to g2.
```

4.4.5 GenericGrid& operator[](const int& i) const

Get a reference to the ith grid. If **g** is a **GenericGridCollection**, then **g**[i] is a reference to the **GenericGrid g.grid**[i].

Example

```
GenericGridCollection c(2);              // Construct a GenericGridCollection with two GenericGrids.
GenericGrid &g1 = c[0], &g2 = c[1];      // Declare references to the two GenericGrids in c.
```

4.4.6 void reference(const GenericGridCollection& x)

Make a reference. This is also called a shallow copy. This **GenericGridCollection** shares the data of **x**.

Example

```
GenericGridCollection g1, g2;   // Construct some GenericGridCollections.
g2.reference(g1);               // Now g2 shares the data of g1.
```

4.4.7 virtual void breakReference()

Break a reference. If this **GenericGridCollection** shares data with any other **GenericGridCollection**, then this function replaces it with a new copy that does not share data.

Example

```
GenericGridCollection g1, g2;   // Construct some GenericGridCollections.
g2.reference(g1);               // Now g2 shares the data of g1.
g2.breakReference();            // Now g2 has a new, unshared copy of the data.
```

4.4.8 virtual Integer get(const GenericDataBase& dir, const String& name)

Copy a **GenericGridCollection** into a file.

Example

```
HDF_DataBase dir;               // (derived from GenericDataBase)
dir.mount("g.hdf");             // Open a data file.
GenericGridCollection g;        // Construct a GenericGridCollection.
g.get(dir, "g");                // Copy g from dir.
dir.unmount();                  // Close the data file.
```

4.4.9 virtual Integer put(GenericDataBase& dir, const String& name) const

Copy a **GenericGridCollection** into a file.

Example

```
HDF_DataBase dir;          // (derived from GenericDataBase)
dir.mount("g.hdf", "I");   // Open a data file.
GenericGridCollection g;   // Construct a GenericGridCollection.
g.put(dir, "g");           // Copy g into dir.
dir.unmount();             // Close the data file.
```

4.4.10 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **THEbaseGrid** (§4.1.1), **THEmultigridLevel** (§4.1.2), **THErefinementLevel** (§4.1.3), **NOTHING** (§4.1.4), **THEusualSuspects** (§4.1.5), **THElists** (§4.1.6) and **EVERYTHING** (§4.1.7), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.4.9), may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§4.1.10), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. Any combination of the constants **COMPUTEnothing** (§4.1.8), **COMPUTEtheUsual** (§4.1.9), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.4.9), may be bitwise ORed together to form the optional second argument of **update**(). The corresponding function **update(what,how)** (§2.4.9) is called with the same arguments for each grid in the list **grid** (§4.2.3).

Example

```
GenericGridCollection g;   // Construct a GenericGridCollection.
                           // Update all of the GenericGridCollection geometric data.
g.update(GenericGridCollection::EVERYTHING);
```

4.4.11 virtual Integer update(GenericGridCollection& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another grid (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contains the corresponding data, then the data for this **GenericGridCollection** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. The corresponding function **update(x,what,how)** (§2.4.10) is called with the same arguments for each grid in the list **grid** (§4.2.3). For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§4.4.10).

Example

```
GenericGridCollection g1, g2;   // Construct some GenericGridCollections.
                                // Update all of the GenericGridCollection geometric data of g1
g1.update(GenericGridCollection::EVERYTHING);
                                // Update all of the GenericGridCollection data of g2, sharing g1 data.
g2.update(g1,GenericGridCollection::EVERYTHING);
```

4.4.12 void destroy(const Integer what = NOTHING)

Destroy the indicated optional geometric grid data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **THEbaseGrid** (§4.1.1), **THEmultigridLevel** (§4.1.2), **THErefinementLevel** (§4.1.3), **NOTHING** (§4.1.4), **THEusualSuspects** (§4.1.5), **THElists** (§4.1.6) and **EVERYTHING** (§4.1.7), as well as any of the corresponding constants allowed for **GenericGrid::destroy(what)** (§2.4.11), may be bitwise ORed together to form the first argument of **update**(). The corresponding function **destroy(what)** (§2.4.11) is called with the same arguments for each grid in the list **grid** (§4.2.3).

Example

```
GenericGridCollection g;   // Construct a GenericGridCollection.
                           // Destroy all of the GenericGridCollection geometric data.
g.destroy(GenericGridCollection::EVERYTHING);
```

4.4.13 void geometryHasChanged(const Integer what = ~NOTHING)

Mark the geometric data out-of-date. Any combination of the constants **THEbaseGrid** (§4.1.1), **THEmultigridLevel** (§4.1.2), **THErefinementLevel** (§4.1.3), **NOTHING** (§4.1.4), **THEusualSuspects** (§4.1.5), **THElists** (§4.1.6) and **EVERYTHING** (§4.1.7), as well as any of the corresponding constants allowed for **GenericGrid::geometryHasChanged(what)** (§2.4.12), may be bitwise ORed together to form the first argument of **geometryHasChanged**(). By default, all geometric data of this **GenericGridCollection** and all derived classes is marked out-of-date. The corresponding function **geometryHasChanged(what)** (§2.4.12) is called with the same argument for each grid in the list **grid** (§4.2.3). It is recommended that this function be called only from derived classes and grid-generation programs.

Example

```
GenericGridCollection g;   // Construct a GenericGridCollection.
                           // Mark all GenericGridCollection data out-of-date.
g.geometryHasChanged(GenericGridCollection::EVERYTHING);
```

4.4.14 void addRefinement(const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This refinement grid is marked as belonging to refinement level **level**. It is marked as having the same base grid as that of **grid**[**k**], which may be any sibling, parent, or other more remote ancestor. It is required that **level** > 0. There must already exist grids marked as belonging to refinement level **level** − 1.

Example

```
GenericGridCollection g(1);        // Construct a GenericGridCollection containing one GenericGrid.
g.addRefinement(1,0);              // Add a level-one refinement of g[0] to g.
g.update(THErefinementLevel);      // Update the list of GenericGridCollections at each refinement level.
GenericGridCollection &level_1
  = g.refinementLevel[1];          // Access refinement level one.
GenericGrid &g0 = level_1[0];      // Access the refinement grid.
```

4.4.15 void deleteRefinementLevels(const Integer& level)

Delete all grids in this collection marked as belonging to the specified refinement level and any grids marked as belonging to any finer refinement level. It is required that **level** > 0.

Example

```
GenericGridCollection g(1);   // Construct a GenericGridCollection containing one GenericGrid.
g.addRefinement(1,0);         // Add a level-one refinement of g[0] to g.
g.deleteRefinements(1);       // Delete refinement level 1 and all higher (finer) refinement levels.
```

4.4.16 void referenceRefinementLevels(GenericGridCollection& x, const Integer& level)

Make this collection contain exactly those grids from the collection **x** which are marked in **x** as belonging to refinement level **level** or to any coarser refinent level. It is required that **level** ≥ 0.

Example

```
GenericGridCollection g1(1), g2;   // Construct a GenericGridCollection containing one GenericGrid.
g1.addRefinement(1,0);             // Add a level-one refinement of g1[0] to g1.
g2.referenceRefinements(g1,1);     // Make g2 contain the refinement level 0 and 1 grids of g1.
```

4.4.17 Logical setNumberOfGrids(const Integer& numberOfGrids_)

Add or delete grids to/from this collection until there are **numberOfGrids_** grids. Any grids added are marked as belonging to base grid zero, multigrid level zero and refinement level zero. This function returns **LogicalTrue** (non-zero) if its invocation resulted in a change in the number of grids, and returns **LogicalFalse** (zero) otherwise. The use of this function is not recommended if the collection is intended to contain more than one base grid, multigrid level or refinement level. For example, if the collection is intended to contain more than one refinement level, it is recommended that the functions **addRefinement(level,k)** (§4.4.14) and **deleteRefinementLevels(level)** (§4.4.15) be used instead.

Example

```
GenericGridCollection g;   // Construct a GenericGridCollection.
g.setNumberOfGrids(2);     // Make this collection contain two grids.
```

4.4.18 GenericGridCollectionData* operator->()

Access the reference-counted data.

Example

```
GenericGridCollection g;                  // Construct a GenericGridCollection.
Integer grids = g->numberOfGrids;         // Access the reference-counted data.
```

4.4.19 GenericGridCollectionData& operator*()

Access the pointer to the reference-counted data.

Example

```
GenericGridCollection g;                  // Construct a GenericGridCollection.
GenericGridCollectionData& data = *g;     // Access the pointer to the reference-counted data.
```

4.4.20 virtual String getClassName()

Get the class name of the most-derived class for this object.

Example

```
GenericGridCollection g;                  // Construct a GenericGridCollection.
String className = g.getClassName();      // Get the class name of g. It should be "GenericGridCollection".
```

4.5 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

4.5.1 void reference(GenericGridCollectionData& x)

Make a reference to an object of type **GenericGridCollectionData**. This **GenericGridCollection** uses **x** for its data. It is recommended that this function be called only from derived classes.

Example

```
GenericGridCollection g1, g2;   // Construct some GenericGridCollections.
g2.reference(*g1);              // Now g2 shares the data of g1.
```

4.5.2 void updateReferences()

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

Example

```
GenericGridCollection g;   // Construct a GenericGridCollection.
g.updateReferences();      // Update references to the reference-counted data.
```

5 Class GridCollection

Class GridCollection is the base class for all Overture classes that contain collections of MappedGrids. Class GridCollection is derived from class GenericGridCollection.

5.1 Public constants

5.1.1 THEmask

THEmask = MappedGrid::mask (§3.2.18)

5.1.2 THEvertex

THEvertex = MappedGrid::THEvertex (§3.1.3)

5.1.3 THEvertex2D

THEvertex2D = MappedGrid::THEvertex2D (§3.1.4)

5.1.4 THEvertex1D

THEvertex1D = MappedGrid::THEvertex1D (§3.1.5)

5.1.5 THEcenter

THEcenter = **MappedGrid::THEcenter** (§3.1.6)

5.1.6 THEcenter2D

THEcenter2D = **MappedGrid::THEcenter2D** (§3.1.7)

5.1.7 THEcenter1D

THEcenter1D = **MappedGrid::THEcenter1D** (§3.1.8)

5.1.8 THEvertexDerivative

THEvertexDerivative = **MappedGrid::THEvertexDerivative** (§3.1.9)

5.1.9 THEvertexDerivative2D

THEvertexDerivative2D = **MappedGrid::THEvertexDerivative2D** (§3.1.10)

5.1.10 THEvertexDerivative1D

THEvertexDerivative1D = **MappedGrid::THEvertexDerivative1D** (§3.1.11)

5.1.11 THEcenterDerivative

THEcenterDerivative = **MappedGrid::THEcenterDerivative** (§3.1.12)

5.1.12 THEcenterDerivative2D

THEcenterDerivative2D = **MappedGrid::THEcenterDerivative2D** (§3.1.13)

5.1.13 THEcenterDerivative1D

THEcenterDerivative1D = **MappedGrid::THEcenterDerivative1D** (§3.1.14)

5.1.14 THEinverseVertexDerivative

THEinverseVertexDerivative = **MappedGrid::THEinverseVertexDerivative** (§3.1.15)

5.1.15 THEinverseVertexDerivative2D

THEinverseVertexDerivative2D = **MappedGrid::THEinverseVertexDerivative2D** (§3.1.16)

5.1.16 THEinverseVertexDerivative1D

THEinverseVertexDerivative1D = **MappedGrid::THEinverseVertexDerivative1D** (§3.1.17)

5.1.17 THEinverseCenterDerivative

THEinverseCenterDerivative = **MappedGrid::THEinverseCenterDerivative** (§3.1.18)

5.1.18 THEinverseCenterDerivative2D

THEinverseCenterDerivative2D = **MappedGrid::THEinverseCenterDerivative2D** (§3.1.19)

5.1.19 THEinverseCenterDerivative1D

THEinverseCenterDerivative1D = **MappedGrid::THEinverseCenterDerivative1D** (§3.1.20)

5.1.20 THEvertexJacobian

THEvertexJacobian = MappedGrid::THEvertexJacobian (§3.1.21)

5.1.21 THEcenterJacobian

THEcenterJacobian = MappedGrid::THEcenterJacobian (§3.1.22)

5.1.22 THEcellVolume

THEcellVolume = MappedGrid::THEcellVolume (§3.1.23)

5.1.23 THEfaceNormal

THEfaceNormal = MappedGrid::THEfaceNormal (§3.1.24)

5.1.24 THEfaceNormal2D

THEfaceNormal2D = MappedGrid::THEfaceNormal2D (§3.1.25)

5.1.25 THEfaceNormal1D

THEfaceNormal1D = MappedGrid::THEfaceNormal1D (§3.1.26)

5.1.26 THEcenterNormal

THEcenterNormal = MappedGrid::THEcenterNormal (§3.1.27)

5.1.27 THEcenterNormal2D

THEcenterNormal2D = MappedGrid::THEcenterNormal2D (§3.1.28)

5.1.28 THEcenterNormal1D

THEcenterNormal1D = MappedGrid::THEcenterNormal1D (§3.1.29)

5.1.29 THEfaceArea

THEfaceArea = MappedGrid::THEfaceArea (§3.1.30)

5.1.30 THEfaceArea2D

THEfaceArea2D = faceArea2D (§3.1.31)

5.1.31 THEfaceArea1D

THEfaceArea1D = MappedGrid::THEfaceArea1D (§3.1.32)

5.1.32 THEvertexBoundaryNormal

THEvertexBoundaryNormal = MappedGrid::THEvertexBoundaryNormal (§3.1.33)

5.1.33 THEcenterBoundaryNormal

THEcenterBoundaryNormal = MappedGrid::THEcenterBoundaryNormal (§3.1.34)

5.1.34 THEminMaxEdgeLength

THEminMaxEdgeLength = MappedGrid::THEminMaxEdgeLength (§3.1.35)

5.1.35 THEtrxyab

THEtrxyab = **MappedGrid::THEtrxyab** (§3.1.36)

5.1.36 THEusualSuspects

THEusualSuspects = **GenericGridCollection::THEusualSuspects** (§4.1.5) | **MappedGrid::THEusualSuspects** (§3.1.42)
THEusualSuspects indicates some of the geometric data of a **GridCollection**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. See also **update(what,how)** (§5.4.10) and **destroy(what)** (§5.4.12).

5.1.37 EVERYTHING

EVERYTHING = **GenericGridCollection::EVERYTHING** (§4.1.7) | **MappedGrid::EVERYTHING** (§3.1.43

EVERYTHING indicates all of the geometric data associated with a **GridCollection**. (overloaded) See also **update(what,how)** (§5.4.10) and **destroy(what)** (§5.4.12).

5.1.38 USEdifferenceApproximation

USEdifferenceApproximation = **MappedGrid::USEdifferenceApproximation** (§3.1.44)
See also **update(what,how)** (§5.4.10) and **destroy(what)** (§5.4.12).

5.1.39 COMPUTEgeometry

COMPUTEgeometry = **MappedGrid::COMPUTEgeometry** (§3.1.45)
See also **update(what,how)** (§5.4.10) and **destroy(what)** (§5.4.12).

5.1.40 COMPUTEgeometryAsNeeded

COMPUTEgeometryAsNeeded = **MappedGrid::COMPUTEgeometryAsNeeded** (§3.1.46)
See also **update(what,how)** (§5.4.10) and **destroy(what)** (§5.4.12).

5.1.41 COMPUTEtheUsual

COMPUTEtheUsual = **GenericGridCollection::COMPUTEtheUsual** (§4.1.9) | **MappedGrid::COMPUTEtheUsual** (§3.1.47)
See also **update(what,how)** (§5.4.10) and **destroy(what)** (§5.4.12).

5.1.42 GRIDnumberBits

GRIDnumberBits = $2^{24} - 1$

5.1.43 ISdiscretization

5.1.44 ISinterpolation

5.1.45 ISinteriorBoundary

5.1.46 ISinvalid

5.1.47 USESbackupRules

5.1.48 ISused

ISused = **ISdiscretization** (§5.1.43) | **ISinterpolation** (§5.1.44)

5.2 Public data

5.2.1 IntegerR numberOfDimensions

The number of dimensions of the domain. This should be the same as the number of dimensions of each grid: **numberOfDimensions** = **grid**[i].**numberOfDimensions** for $0 \leq i <$ **numberOfGrids**. See also **MappedGrid::numberOfDimensions** (§3.2.1)

5.2.2 IntegerArray refinementFactor

Dimensions: (0:2, 0:**numberOfGrids** − 1)
and **nd** (§6.3.27).

5.2.3 ListOfMappedGrid grid

Length: **numberOfGrids** (§4.2.2)
A list containing of all of the grids in the collection. The grids in this list are **MappedGrids**. **grid** overloads **GenericGridCollection::grid** (§4.2.3), which contains the same **MappedGrids** (referred to in the latter case as **GenericGrids**).

5.2.4 ListOfGridCollection baseGrid

A list of **GridCollections** containing those **MappedGrids** which are descended from the same base grid. The length of this list is **numberOfBaseGrids** (§4.2.4). **baseGrid** overloads **GenericGridCollection::baseGrid** (§4.2.5), which contains the same **GridCollections** (referred to in the latter case as **GenericGridCollections**). This data may be updated as in the following example.

Example

```
GridCollection g;                                  // Construct a GridCollection.
g.update(GridCollection::THEbaseGrid);             // Update baseGrid.
```

5.2.5 ListOfGridCollection multigridLevel

A list of **GridCollections** containing those **MappedGrids** which belong to the same multigrid level. The length of this list is **numberOfMultigridLevels** (§4.2.7). **multigridLevel** overloads **GenericGridCollection::multigridLevel** (§4.2.8), whose **GenericGridCollections** contain the same grids as the **GridCollections** of **multigridLevel** (referred to in the latter case as **GenericGrids**). This data may be updated as in the following example.

Example

```
GridCollection g;                                  // Construct a GridCollection.
g.update(GridCollection::THEmultigridLevel);       // Update multigridLevel.
```

5.2.6 ListOfGridCollection refinementLevel

A list of **GridCollections** containing those **MappedGrids** which belong to the same refinement level. The length of this list is **numberOfRefinementLevels** (§4.2.10). **refinementLevel** overloads **GenericGridCollection::refinementLevel** (§4.2.11), whose **GenericGridCollections** contain the same grids as the **GridCollections** of **refinementLevel**. This data may be updated as in the following example.

Example

```
GridCollection g;                                  // Construct a GridCollection.
g.update(GridCollection::THErefinementLevel);      // Update refinementLevel.
```

5.3 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

5.3.1 GridCollectionData* rcData

Pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§5.4.19) and **operator*()** (§5.4.20), which are provided for access to **rcData**.

5.3.2 Logical isCounted

Flag that indicates whether the data pointed to by **rcData** (§5.3.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

5.4 Public member functions

5.4.1 GridCollection(const Integer numberOfDimensions_ = 0, const Integer numberOfGrids_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null **GridCollection**. Otherwise, create a **GridCollection** with the given number of dimensions and number of grids.

Example

```
GridCollection(2, 3) g;   // Construct a two-dimensional GridCollection with three MappedGrids.
```

5.4.2 GridCollection(const GridCollection& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default.

Example

```
GridCollection g1;                                  // Construct a GridCollection.
GridCollection g2=g1, g3(g1);                       // Construct a GridCollection using deep copy.
GridCollection g4(g1,GridCollection::SHALLOW);      // Construct using shallow copy; g2 shares the data of g1.
```

See also **operator=(x)** (§5.4.4) and **reference(x)** (§5.4.6).

5.4.3 virtual ~GridCollection()

Destructor.

5.4.4 GridCollection& operator=(const GridCollection& x)

Assignment operator. This is also called a deep copy.

Example

```
GridCollection g1, g2;   // Construct some GridCollections.
g2 = g1;                 // Copy data from g1 to g2.
```

5.4.5 MappedGrid& operator[](const int& i) const

Get a reference to the ith grid. If **g** is a **GridCollection**, then **g**[i] is a reference to the **MappedGrid g.grid**[i].

5.4.6 void reference(const GridCollection& x)

Make a reference. This is also called a shallow copy. This **GridCollection** shares the data of **x**.

Example

```
GridCollection g1, g2;   // Construct some GridCollections.
g2.reference(g1);        // Now g2 shares the data of g1.
```

5.4.7 virtual void breakReference()

Break a reference. If this **MappedGrid** shares data with any other **GridCollection**, then this function replaces it with a new copy that does not share data.

Example

```
GridCollection g1, g2;   // Construct some GridCollections.
g2.reference(g1);        // Now g2 shares the data of g1.
g2.breakReference();     // Now g2 has a new, unshared copy of the data.
```

5.4.8 virtual Integer get(const GenericDataBase& dir, const String& name)

Copy a **GridCollection** into a file.

Example

```
HDF_DataBase dir;   // (derived from GenericDataBase)
dir.mount("g.hdf"); // Open a data file.
GridCollection g;   // Construct a GridCollection.
g.get(dir, "g");    // Copy g from dir.
dir.unmount();      // Close the data file.
```

5.4.9 virtual Integer put(GenericDataBase& dir, const String& name) const

Copy a **GridCollection** into a file.

Example

```
HDF_DataBase dir;       // (derived from GenericDataBase)
dir.mount("g.hdf", "I"); // Open a data file.
GridCollection g;       // Construct a GridCollection.
g.put(dir, "g");        // Copy g into dir.
dir.unmount();          // Close the data file.
```

5.4.10 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **THEusualSuspects** (§5.1.36) and **EVERYTHING** (§5.1.37), as well as any of the corresponding constants allowed for **GenericGridCollection::update(what,how)** (§4.4.10) or **MappedGrid::update(what,how)** (§3.5.13), may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§4.1.10), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. The constant **COMPUTEtheUsual** (§5.1.41), as

well as any of the corresponding constants allowed for **GenericGridCollection::update(what,how)** (§4.4.10) or **MappedGrid::update(what,how)** (§3.5.13), may be bitwise ORed together to form the optional second argument of **update**(). The corresponding function **update(what,how)** (§4.4.10) is called with the same arguments for the base class **GenericGridCollection**, and **update(what,how)** (§3.5.13) is called with the same arguments for each **MappedGrid** in the list **grid** (§5.2.3).

5.4.11 Integer update(GridCollection& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another grid (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contains the corresponding data, then the data for this **GridCollection** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. The corresponding function **update(x,what,how)** (§4.4.11) is called with the same arguments for the base class **GenericGridCollection**, and **update(what,how)** (§3.5.13) is called with the same arguments for each **MappedGrid** in the list **grid** (§5.2.3). For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§3.5.13).

5.4.12 void destroy(const Integer what = NOTHING)

Destroy optional grid data.

5.4.13 void getInterpolationStencil(const Integer& k1, const IntegerArray& k2, const RealArray& r, IntegerArray& interpolationStencil)

Return the bounds on the cube of points used for interpolation. Input: k1 Index of the grid containing the interpolation points. k2(p1:p2) Indices of the grids containing the interpolee points. The dimensions of this array determine p1 and p2. r(0:2,p1:p2) Interpolation coordinates of the interpolation points. Output: interpolationStencil(0:1,0:2,p1:p2) lower and upper index bounds of the stencils.

5.4.14 Logical canInterpolate(const Integer& k1, const IntegerArray& k2, const RealArray& r, IntegerArray& iab, const Logical checkForOneSided = LogicalFalse)

Determine whether points on grid k1 at r in the coordinates of grids k2 can be interpolated from grids k2.

5.4.15 void addRefinement(const IntegerArray& range, const IntegerArray& factor) const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This refinement grid is marked as belonging to refinement level **level**. It is marked as having the same base grid as that of **grid**[**k**], which may be any sibling, parent, or other more remote ancestor. It is required that **level** > 0. There must already exist grids marked as belonging to refinement level **level** − 1. The **indexRange** (§3.2.3) of the refinement is given by **range**, which must be a subset of the index range of a refinement of the same base grid (the refinement grid's most remote ancestor), at the same level of refinement, that would cover the entire base grid. The refinement factor **factor** is relative to any parent grid; any cell of any parent grid that is covered by cells of the new refinement grid is subdivided in each direction k into **factor**(k) cells of the new refinement grid.

Example

```
GridCollection g(2,1);                  // Construct a GridCollection containing one two-dimensional MappedGrid.
g.addRefinement(1,0);                   // Add a level-one refinement of g[0] to g.
IntegerArray range(2,3), factor(3);
g.update(THErefinementLevel);           // Update the list of GridCollections at each refinement level.
GridCollection &level_1
  = g.refinementLevel[1];               // Access refinement level one.
MappedGrid &g0 = level_1[0];            // Access the refinement grid.
```

5.4.16 void addRefinement(const IntegerArray& range, const Integer& factor) const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This function simply calls **addRefinement()** (§5.4.15) with **factor** replaced by an **IntegerArray** of length three set to the (scalar) **factor**..

5.4.17 void deleteRefinementLevels(const Integer& level)

Delete all grids in this collection marked as belonging to the specified refinement level and any grids marked as belonging to any finer refinement level. It is required that **level** > 0.

Example

```
GridCollection g(1);                // Construct a GridCollection containing one MappedGrid.
IntegerArray range;                 // Note: must first set range appropriately.
g.addRefinement(range,2,1,0);       // Add a level-one refinement of g[0] to g.
g.deleteRefinements(1);             // Delete refinement level 1 and all higher (finer) refinement levels.
```

5.4.18 Logical setNumberOfDimensionsAndGrids(const Integer& numberOfDimensios_, const Integer& numberOfGrids_)

Set the number of dimensions of the grids, and add or delete grids to/from this collection until there are **numberOfGrids_** grids. Any grids added are marked as belonging to base grid zero, multigrid level zero and refinement level zero. This function returns **LogicalTrue** (non-zero) if this resulted in a change in the number dimensions or the number of grids, and returns **LogicalFalse** (zero) otherwise. The use of this function is not recommended if the collection is intended to contain more than one base grid, multigrid level or refinement level. For example, if the collection is intended to contain more than one refinement level, it is recommended that the functions **addRefinement(range,factor,level,k)** (§5.4.15) and **deleteRefinementLevels(level)** (§5.4.17) be used instead.

Example

```
GridCollection g;                          // Construct a GridCollection.
g.setNumberOfDimensionsAndGrids(2,3);      // Make this collection contain three two-dimensional grids.
```

5.4.19 GridCollectionData* operator->()

Access the reference-counted data.

Example

```
GridCollection g;                          // Construct a GridCollection.
Integer grids = g->numberOfGrids;          // Access the reference-counted data.
```

5.4.20 GridCollectionData& operator*()

Access the pointer to the reference-counted data.

Example

```
GridCollection g;                          // Construct a GridCollection.
GridCollectionData& data = *g;             // Access the pointer to the reference-counted data.
```

5.4.21 virtual String getClassName()

Get the class name of the most-derived class for this object.

Example

```
GridCollection g;                          // Construct a GridCollection.
String className = g.getClassName();       // Get the class name of g. It should be "GridCollection".
```

5.5 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

5.5.1 void reference(GridCollectionData& x)

Make a reference to an object of type **GridCollectionData**. This **GridCollection** uses **x** for its data. It is recommended that this function be called only from derived classes.

Example

```
GridCollection g1, g2;   // Construct some GridCollections.
g2.reference(*g1);       // Now g2 shares the data of g1.
```

5.5.2 void updateReferences()

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

Example

```
GridCollection g;          // Construct a GridCollection.
g.updateReferences();      // Update references to the reference-counted data.
```

6 Class CompositeGrid

Class CompositeGrid is used for a composite overlapping grid, which is a collection of MappedGrids and a description of how function values defined on these grids are related through interpolation between grids in their regions of overlap. Class CompositeGrid is derived from class GridCollection.

6.1 Public constants

6.1.1 MAXrefinementLevel

6.1.2 THEinterpolationCoordinates

THEinterpolationCoordinates indicates **interpolationCoordinates** (§6.2.8), the coordinates of the inteprolation points in the parameter space of the grids from which they interpolate. See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.3 THEinterpoleeGrid

THEinterpoleeGrid indicates **interpoleeGrid** (§6.2.9), the indices of the grids from which the inteprolation points interpolate. See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.4 THEinterpoleeLocation

THEinterpoleeLocation indicates **interpoleeLocation** (§6.2.10), the indices of the lower-left corners of the interpolation stencils in the grids from which the inteprolation points interpolate. See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.5 THEinterpolationPoint

THEinterpolationPoint indicates **interpolationPoint** (§6.2.11), the indices of the interpolation points. See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.6 THEinterpolationCondition

THEinterpolationCondition indicates **interpolationCondition** (§6.2.12), the condition numbers for interpolation of the interpolation points. See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.7 THEinverseCondition

THEinverseCondition indicates **inverseCondition** (§6.4.3). See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.8 THEinverseCoordinates

THEinverseCoordinates indicates **inverseCoordinates** (§6.4.1). See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.9 THEinverseGrid

THEinverseGrid indicates **inverseGrid** (§6.4.2). See also **update(what,how)** (§6.6.9) and **destroy(what)** (§6.6.11).

6.1.10 THEci

6.1.11 THEil

6.1.12 THEip

6.1.13 THEiq

6.1.14 THEkr

6.1.15 THErsxy

6.1.16 THExy

6.1.17 THExyc

6.1.18 THExyrs

6.1.19 THEusualSuspects

THEusualSuspects = GridCollection::THEusualSuspects (§5.1.36) | **THEinterpolationCoordinates** (§6.1.2) | **THEinterpoleeGrid** (§6.1.3) | **THEinterpoleeLocation** (§6.1.4) | **THEinterpolationPoint** (§6.1.5)

6.1.20 EVERYTHING

EVERYTHING = GridCollection::EVERYTHING (§5.1.37) | **THEinterpolationCoordinates** (§6.1.2) | **THEinterpoleeGrid** (§6.1.3) | **THEinterpoleeLocation** (§6.1.4) | **THEinterpolationPoint** (§6.1.5) | **THEinterpolationCondition** (§6.1.6) | **THEinverseCondition** (§6.1.7) | **THEinverseCoordinates** (§6.1.8) | **THEinverseGrid** (§6.1.9)

6.1.21 COMPUTEtrxyab

6.1.22 COMPUTEtheUsual

6.2 Public data

6.2.1 IntegerR numberOfComponentGrids

The number of component grids.

6.2.2 RealR epsilon

6.2.3 IntegerArray numberOfInterpolationPoints

Dimensions: (0: **numberOfGrids** − 1)
The number of interpolation points on each component grid. See also **MappedGrid::ni** (§3.3.15) and **CompositeGrid::ni** (§6.3.30).

6.2.4 LogicalR interpolationIsAllExplicit

The type of interpolation between all pairs of grids. See also **CompositeGrid::it** (§6.3.19).

6.2.5 LogicalArray interpolationIsImplicit

Dimensions: (0: **numberOfComponentGrids** − 1, 0: **numberOfComponentGrids** − 1)
The type of interpolation (toGrid, fromGrid). See also **CompositeGrid::it** (§6.3.19).

6.2.6 IntegerArray interpolationWidth

Dimensions: (0: 2, 0: **numberOfComponentGrids** − 1, 0: **numberOfComponentGrids** − 1)
The width of the interpolation stencil (direction, toGrid, fromGrid). See also **CompositeGrid::iw** (§6.3.21).

6.2.7 RealArray interpolationOverlap

Dimensions: (0: 2, 0: **numberOfComponentGrids** − 1, 0: **numberOfComponentGrids** − 1)
The minimum overlap for interpolation (direction, toGrid, fromGrid). See also **CompositeGrid::ov** (§6.3.33).

6.2.8 ListOfRealArray interpolationCoordinates

Length: **numberOfGrids** (§4.2.2)
Dimensions of **interpolationCoordinates**(i): (0: **numberOfInterpolationPoints**(i) − 1, 0: **numberOfDimensions** − 1)
Coordinates of each interpolation point. See also **CompositeGrid::ci** (§6.3.6).

6.2.9 ListOfIntegerArray interpoleeGrid

Length: **numberOfGrids** (§4.2.2)
Dimensions of **interpolationCoordinates**(i): (0: **numberOfInterpolationPoints**(i) − 1)
Index of interpolee grid for each interpolation point. See also **CompositeGrid::il** (§6.3.15).

6.2.10 ListOfIntegerArray interpoleeLocation

Length: **numberOfGrids** (§4.2.2)
Dimensions of **interpolationCoordinates**(i): (0: **numberOfInterpolationPoints**(i) − 1, 0: **numberOfDimensions** − 1)
Location of interpolation stencil for each interpolation point. See also **CompositeGrid::il** (§6.3.15).

6.2.11 ListOfIntegerArray interpolationPoint

Length: **numberOfGrids** (§4.2.2)
Dimensions of **interpolationCoordinates**(i): (0: **numberOfInterpolationPoints**(i) − 1, 0: **numberOfDimensions** − 1)
Indices of each interpolation point in its grid. See also **CompositeGrid::ip** (§6.3.17).

6.2.12 ListOfRealArray interpolationCondition

Length: **numberOfGrids** (§4.2.2)
Dimensions of **interpolationCoordinates**(i): (0: **numberOfInterpolationPoints**(i) − 1)
Interpolation condition number of each interpolation point. See also **CompositeGrid::iq** (§6.3.18).

6.3 Public composite grid data used by Fortran programs

The composite grid data used by Fortran programs and listed in Table 2, page 5 of *Composite Grid Data: All You Never Wanted to Know and You were afraid to Ask* (*q.v.* for full details) are accessible to C++ classes through these public data members.

6.4.1 RealCompositeGridFunction inverseCoordinates

Dimensions of **inverseCoordinates**[k]: $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12}, 0\colon n_1)$, where d_{ij} = **grid**[k].**dimension**(i,j) and n_1 = **numberOfDimensions** − 1.
Inversion coordinates.

6.4.2 IntegerCompositeGridFunction inverseGrid

Dimensions of **inverseGrid**[k]:: $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12})$, where d_{ij} = **grid**[k].**dimension**(i,j).
Inverted grid.

6.4.3 RealCompositeGridFunction inverseCondition

Dimensions of **inverseCondition**[k]: $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12})$, where d_{ij} = **grid**[k].**dimension**(i,j).
Inverse quality.

6.5 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

6.5.1 CompositeGridData* rcData

Pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§6.6.14) and **operator*()** (§6.6.15), which are provided for access to **rcData**.

6.5.2 Logical isCounted

Flag that indicates whether the data pointed to by **rcData** (§6.5.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

6.6 Public member functions

6.6.1 CompositeGrid(const Integer numberOfDimensions_ = 0, const Integer numberOfComponentGrids_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null CompositeGrid. Otherwise create a CompositeGrid with the given number of dimensions and number of component grids.

Example

```
CompositeGrid(2, 3) g;   // Construct a two-dimensional CompositeGrid with three component grids.
```

6.6.2 CompositeGrid(const CompositeGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default.

Example

```
CompositeGrid g1;                                  // Construct a CompositeGrid.
CompositeGrid g2=g1, g3(g1);                       // Construct a CompositeGrid using deep copy.
CompositeGrid g4(g1,CompositeGrid::SHALLOW);       // Construct using shallow copy; g2 shares the data of g1.
```

See also **operator=(x)** (§6.6.4) and **reference(x)** (§6.6.5).

6.6.3 virtual ~CompositeGrid()

Destructor.

6.6.4 CompositeGrid& operator=(const CompositeGrid& x)

Assignment operator. This is also called a deep copy.

Example

```
CompositeGrid g1, g2;   // Construct some CompositeGrids.
g2 = g1;                // Copy data from g1 to g2.
```

6.6.5 void reference(const CompositeGrid& x)

Make a reference. This is also called a shallow copy. This **CompositeGrid** shares the data of **x**.

Example

```
CompositeGrid g1, g2;   // Construct some CompositeGrids.
g2.reference(g1);       // Now g2 shares the data of g1.
```

6.6.6 virtual void breakReference()

Break a reference. If this **GenericGrid** shares data with any other **CompositeGrid**, then this function replaces it with a new copy that does not share data.

Example

```
CompositeGrid g1, g2;   // Construct some CompositeGrids.
g2.reference(g1);       // Now g2 shares the data of g1.
g2.breakReference();    // Now g2 has a new, unshared copy of the data.
```

6.6.7 virtual Integer get(const GenericDataBase& dir, const String& name)

Copy a **CompositeGrid** into a file.

Example

```
HDF_DataBase dir;   // (derived from GenericDataBase)
dir.mount("g.hdf");  // Open a data file.
CompositeGrid g;    // Construct a CompositeGrid.
g.get(dir, "g");    // Copy g from dir.
dir.unmount();      // Close the data file.
```

6.6.8 virtual Integer put(GenericDataBase& dir, const String& name) const

Copy a **CompositeGrid** into a file.

Example

```
HDF_DataBase dir;       // (derived from GenericDataBase)
dir.mount("g.hdf", "I"); // Open a data file.
CompositeGrid g;        // Construct a CompositeGrid.
g.put(dir, "g");        // Copy g into dir.
dir.unmount();          // Close the data file.
```

6.6.9 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update the grid.

6.6.10 Integer update(CompositeGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update the grid, sharing the data of another grid.

6.6.11 void destroy(const Integer what = NOTHING)

Destroy optional grid data.

6.6.12 void getInterpolationStencil(const Integer& k1, const IntegerArray& k2, const RealArray& r, IntegerArray& interpolationStencil, const Logical useBackupRules = LogicalFalse)

Return the bounds on the cube of points used for interpolation. Input: k1 Index of the grid containing the interpolation points. k2(p1:p2) Indices of the grids containing the interpolee points. The dimensions of this array determine p1 and p2. r(0:2,p1:p2) Interpolation coordinates of the interpolation points. useBackupRules Must be LogicalFalse unless backup interpolation rules should be used to determine the stencil. Output: interpolationStencil(0:1,0:2,p1:p2) lower and upper index bounds of the stencils.

6.6.13 Logical canInterpolate(const Integer& k1, const IntegerArray& k2, const RealArray& r, IntegerArray& iab, const Logical checkForOneSided = LogicalFalse, const Logical useBackupRules = LogicalFalse)

Determine whether points on grid k1 at r in the coordinates of grids k2 can be interpolated from grids k2.

6.6.14 CompositeGridData* operator->()

Access the reference-counted data.

Example

```
CompositeGrid g;                                  // Construct a CompositeGrid.
Integer grids = g->numberOfComponentGrids;        // Access the reference-counted data.
```

6.6.15 CompositeGridData& operator*()

Access the pointer to the reference-counted data.

Example

```
CompositeGrid g;                  // Construct a CompositeGrid.
CompositeGridData& data = *g;     // Access the pointer to the reference-counted data.
```

6.6.16 virtual String getClassName()

Get the class name of the most-derived class for this object.

Example

```
CompositeGrid g;                          // Construct a CompositeGrid.
String className = g.getClassName();      // Get the class name of g. It should be "CompositeGrid".
```

6.7 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

6.7.1 void reference(CompositeGridData& x)

Make a reference to an object of type **CompositeGridData**. This **CompositeGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

Example

```
CompositeGrid g1, g2;   // Construct some CompositeGrids.
g2.reference(*g1);      // Now g2 shares the data of g1.
```

6.7.2 void updateReferences()

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

Example

```
CompositeGrid g;        // Construct a CompositeGrid.
g.updateReferences();   // Update references to the reference-counted data.
```

7 Class MultigridMappedGrid

Class MultigridMappedGrid is used for a set of MappedGrids that discretize the same region at various nested levels of refinement. Class MultigridMappedGrid is derived from class GridCollection.

7.1 Public constants

7.1.1 THEusualSuspects

7.1.2 EVERYTHING

7.1.3 COMPUTEtheUsual

7.2 Public data

7.2.1 IntegerR numberOfMultigridLevels

(aliased to numberOfComponentGrids)

7.2.2 IntegerArray fineToCoarseFactor

Ratio of this to the next coarser level.

7.3 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

7.3.1 MultigridMappedGridData* rcData

Pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§7.4.15) and **operator*()** (§7.4.16), which are provided for access to **rcData**.

7.3.2 Logical isCounted

Flag that indicates whether the data pointed to by **rcData** (§7.3.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

7.4 Public member functions

7.4.1 MultigridMappedGrid(const Integer numberOfDimensions_ = 0, const Integer numberOfMultigridLevels_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null MultigridMappedGrid. Otherwise, create a MultigridMappedGrid with the given number of dimensions and number of multigrid levels.

Example

```
MultigridMappedGrid(2, 3) g;   // Construct a two-dimensional MultigridMappedGrid
                               // with three multigrid levels.
```

7.4.2 MultigridMappedGrid(const MultigridMappedGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default.

Example

```
MultigridMappedGrid g1;                          // Construct a MultigridMappedGrid.
MultigridMappedGrid g2=g1, g3(g1);               // Construct a MultigridMappedGrid using deep copy.
MultigridMappedGrid g4(g1,                       // Construct using shallow copy; g2 shares the data of g1.
  g4(g1,MultigridMappedGrid::SHALLOW);
```

See also **operator=(x)** (§7.4.6) and **reference(x)** (§7.4.8).

7.4.3 MultigridMappedGrid(Mapping& mapping_, const Integer numberOfMultigridLevels_ = 1)

Constructor from a mapping.

7.4.4 MultigridMappedGrid(MappingRC& mapping_, const Integer numberOfMultigridLevels_ = 1)

Constructor from a mapping.

7.4.5 virtual ~MultigridMappedGrid()

Destructor.

7.4.6 MultigridMappedGrid& operator=(const MultigridMappedGrid& x)

Assignment operator. This is also called a deep copy.

Example

```
MultigridMappedGrid g1, g2;   // Construct some MultigridMappedGrids.
g2 = g1;                      // Copy data from g1 to g2.
```

7.4.7 MappedGrid& operator[](const int& i) const

Get a reference to a multigrid level using C or Fortran indexing.

7.4.8 void reference(const MultigridMappedGrid& x)

Make a reference. This is also called a shallow copy. This **MultigridMappedGrid** shares the data of **x**.

Example

```
MultigridMappedGrid g1, g2;   // Construct some MultigridMappedGrids.
g2.reference(g1);             // Now g2 shares the data of g1.
```

7.4.9 virtual void breakReference()

Break a reference. If this **MultigridMappedGrid** shares data with any other **MultigridMappedGrid**, then this function replaces it with a new copy that does not share data.

Example

```
MultigridMappedGrid g1, g2;   // Construct some MultigridMappedGrids.
g2.reference(g1);             // Now g2 shares the data of g1.
g2.breakReference();          // Now g2 has a new, unshared copy of the data.
```

7.4.10 virtual Integer get(const GenericDataBase& dir, const String& name)

Copy a **MultigridMappedGrid** into a file.

Example

```
HDF_DataBase dir;             // (derived from GenericDataBase)
dir.mount("g.hdf");           // Open a data file.
MultigridMappedGrid g;        // Construct a MultigridMappedGrid.
g.get(dir, "g");              // Copy g from dir.
dir.unmount();                // Close the data file.
```

7.4.11 virtual Integer put(GenericDataBase& dir, const String& name) const

Copy a **MultigridMappedGrid** into a file.

Example

```
HDF_DataBase dir;             // (derived from GenericDataBase)
dir.mount("g.hdf", "I");      // Open a data file.
MultigridMappedGrid g;        // Construct a MultigridMappedGrid.
g.put(dir, "g");              // Copy g into dir.
dir.unmount();                // Close the data file.
```

7.4.12 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update the grid.

7.4.13 Integer update(MultigridMappedGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update the grid, sharing the data of another grid.

7.4.14 void destroy(const Integer what = NOTHING)

Destroy optional grid data.

7.4.15 MultigridMappedGridData* operator->()

Access the reference-counted data.

Example

```
MultigridMappedGrid g;                          // Construct a MultigridMappedGrid.
Integer levels = g->numberOfMultigridLevels;    // Access the reference-counted data.
```

7.4.16 MultigridMappedGridData& operator*()

Access the pointer to the reference-counted data.

Example

```
MultigridMappedGrid g;                  // Construct a MultigridMappedGrid.
MultigridMappedGridData& data = *g;     // Access the pointer to the reference-counted data.
```

7.4.17 virtual String getClassName()

Get the class name of the most-derived class for this object.

Example

```
MultigridMappedGrid g;                      // Construct a MultigridMappedGrid.
String className = g.getClassName();        // Get the class name of g. It should be "MultigridMappedGrid".
```

7.5 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

7.5.1 void reference(MultigridMappedGridData& x)

Make a reference to an object of type **MultigridMappedGridData**. This **MultigridMappedGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

Example

```
MultigridMappedGrid g1, g2;     // Construct some MultigridMappedGrids.
g2.reference(*g1);              // Now g2 shares the data of g1.
```

7.5.2 void updateReferences()

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

Example

```
MultigridMappedGrid g;      // Construct a MultigridMappedGrid.
g.updateReferences();       // Update references to the reference-counted data.
```

8 Class MultigridCompositeGrid

Class MultigridCompositeGrid is used for collections of CompositeGrids that discretize the same region at various nested levels of refinement. Each CompositeGrid contains the same number of MappedGrids, and every MappedGrid in one CompositeGrid is related to a MappedGrid in each of the other CompositeGrids. The related MappedGrids each discretize the same sub-region at the various nested levels of refinement. Class MultigridCompositeGrid is derived from class GridCollection.

8.1 Public constants

8.1.1 MAXrefinementLevel

8.1.2 THEci

8.1.3 THEil

8.1.4 THEip

8.1.5 THEiq

8.1.6 THEkr

8.1.7 THErsxy

8.1.8 THExy

8.1.9 THExyc

8.1.10 THExyrs

8.1.11 THEinverseCoordinates

8.1.12 THEinverseGrid

8.1.13 THEinverseCondition

8.1.14 THEusualSuspects

8.1.15 EVERYTHING

8.1.16 COMPUTEtrxyab

8.1.17 COMPUTEtheUsual

8.2 Public data

8.2.1 IntegerR numberOfComponentGrids

(overloaded) ng

8.2.2 IntegerR numberOfMultigridLevels

mg

8.2.3 IntegerArray coarseToFineWidth

Prolongation stencil width. cfw

8.2.4 LogicalArray coarseToFineIsImplicit

Prolongation is always implicit. cft

8.2.5 IntegerArray fineToCoarseWidth

Restriction stencil width. fcw

8.2.6 LogicalArray fineToCoarseIsImplicit

Restriction is always implicit. fct

8.2.7 IntegerArray fineToCoarseFactor

Ratio of this to coarser level. fcf

8.2.8 ListOfCompositeGrid compositeGrid

8.3 Public composite grid data used by Fortran programs

The composite grid data used by Fortran programs and listed in Table 1, page 4 of *Composite Grid Data: All You Never Wanted to Know and You were afraid to Ask* (*q.v.* for full details) are accessible to C++ classes through these public data members.

8.4.1 MultigridCompositeGridData* rcData

Pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§8.5.15) and **operator*()** (§8.5.16), which are provided for access to **rcData**.

8.4.2 Logical isCounted

Flag that indicates whether the data pointed to by **rcData** (§8.4.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

8.5 Public member functions

8.5.1 MultigridCompositeGrid(const Integer numberOfDimensions_ = 0, const Integer numberOfComponentGrids_ = 0, const Integer numberOfMultigridLevels_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null MultigridCompositeGrid. Otherwise create a MultigridCompositeGrid with the given number of dimensions, number of component grids and number of multigrid levels.

Example

```
MultigridCompositeGrid(2, 3, 4) g;   // Construct a two-dimensional MultigridCompositeGrid
                                     // with three component grids and four multigrid levels.
```

8.5.2 MultigridCompositeGrid(const MultigridCompositeGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default.

Example

```
MultigridCompositeGrid g1;                                          // Construct a MultigridComposite
MultigridCompositeGrid g2=g1, g3(g1);                               // Construct a MultigridComposite
MultigridCompositeGrid g4(g1,MultigridCompositeGrid::SHALLOW);      // Construct using shallow copy; g2 sh
```

See also **operator=(x)** (§8.5.4) and **reference(x)** (§8.5.6).

8.5.3 virtual ~MultigridCompositeGrid()

Destructor.

8.5.4 MultigridCompositeGrid& operator=(const MultigridCompositeGrid& x)

Assignment operator. This is also called a deep copy.

Example

```
MultigridCompositeGrid g1, g2;   // Construct some MultigridCompositeGrids.
g2 = g1;                         // Copy data from g1 to g2.
```

8.5.5 CompositeGrid& operator[](const Integer& i) const

Get a reference to a component grid using C or Fortran indexing.

8.5.6 void reference(MultigridCompositeGrid& x)

Make a reference. This is also called a shallow copy. This **MultigridCompositeGrid** shares the data of **x**.

Example

```
MultigridCompositeGrid g1, g2;   // Construct some MultigridCompositeGrids.
g2.reference(g1);                // Now g2 shares the data of g1.
```

8.5.7 virtual void breakReference()

Break a reference. If this **MultigridCompositeGrid** shares data with any other **MultigridCompositeGrid**, then this function replaces it with a new copy that does not share data.

Example

```
MultigridCompositeGrid g1, g2;   // Construct some MultigridCompositeGrids.
g2.reference(g1);                // Now g2 shares the data of g1.
g2.breakReference();             // Now g2 has a new, unshared copy of the data.
```

8.5.8 virtual Integer get(const GenericDataBase& dir, const String& name)

Copy a **MultigridCompositeGrid** into a file.

Example

```
HDF_DataBase dir;                // (derived from GenericDataBase)
dir.mount("g.hdf");              // Open a data file.
MultigridCompositeGrid g;        // Construct a MultigridCompositeGrid.
g.get(dir, "g");                 // Copy g from dir.
dir.unmount();                   // Close the data file.
```

8.5.9 virtual Integer put(GenericDataBase& dir, const String& name) const

Copy a **MultigridCompositeGrid** into a file.

Example

```
HDF_DataBase dir;                // (derived from GenericDataBase)
dir.mount("g.hdf", "I");         // Open a data file.
MultigridCompositeGrid g;        // Construct a MultigridCompositeGrid.
g.put(dir, "g");                 // Copy g into dir.
dir.unmount();                   // Close the data file.
```

8.5.10 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update the grid.

8.5.11 Integer update(MultigridCompositeGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update the grid, sharing the data of another grid.

8.5.12 void destroy(const Integer what = NOTHING)

Destroy optional grid data.

8.5.13 void insertLevel(const Integer& i)

Manage Level and other arrays and lists.

8.5.14 void deleteLevel(const Integer& i)

Manage Level and other arrays and lists.

8.5.15 MultigridCompositeGridData* operator->()

Access the reference-counted data.

Example

```
MultigridCompositeGrid g;                          // Construct a MultigridCompositeGrid.
Integer levels = g->numberOfMultigridLevels;       // Access the reference-counted data.
```

8.5.16 MultigridCompositeGridData& operator*()

Access the pointer to the reference-counted data.

Example

```
MultigridCompositeGrid g;                          // Construct a MultigridCompositeGrid.
MultigridCompositeGridData& data = *g;             // Access the pointer to the reference-counted data.
```

8.5.17 virtual String getClassName()

Get the class name of the most-derived class for this object.

Example

```
MultigridCompositeGrid g;                          // Construct a MultigridCompositeGrid.
String className = g.getClassName();               // Get the class name of g. It should be "MultigridCompositeGrid".
```

8.6 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

8.6.1 void reference(MultigridCompositeGridData& x)

Make a reference to an object of type **MultigridCompositeGridData**. This **MultigridCompositeGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

Example

```
MultigridCompositeGrid g1, g2;                     // Construct some MultigridCompositeGrids.
g2.reference(*g1);                                 // Now g2 shares the data of g1.
```

8.6.2 void updateReferences()

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

Example

```
MultigridCompositeGrid g;                          // Construct a MultigridCompositeGrid.
g.updateReferences();                              // Update references to the reference-counted data.
```