The Overture Grid Classes
Users' Guide, Version 1.0

Geoffrey S. Chesshire

Scientific Computing Group (CIC-19)
Los Alamos National Laboratory
Los Alamos, New Mexico 87545, USA

William D. Henshaw

Centre for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
http://www.llnl.gov/casc/people/henshaw
http://www.llnl.gov/casc/Overture May 20, 2011 UCRL-MA-134445

Abstract

Overture is a library containing classes for grids, overlapping grid generation and the discretization and solution of PDEs on overlapping grids. This document shows how the Overture grid classes may be used. The primary classes described are the **MappedGrid**, **GridCollection** and **CompositeGrid** classes. These classes hold the geometry arrays required by PDE solvers such as the **vertex** (grid vertices), **vertexDerivative** (jacobian derivatives), and **vertexBoundaryNormal** (normals on the boundary), etc. The geometry arrays can be optionally generated as required by the application. The grid classes have support for multigrid levels and for adaptive mesh refinement. While reading this users' guide, it may be useful to have a copy of the reference manual " The Overture Grid Classes, Reference Guide" at hand. Both this users' guide and the reference manual may be found on the Overture home page.

Contents

1 Introduction

The Overture framework contains C++ classes that describe grids for the discretization of partial differential equations on complex domains in one, two and three dimensions. While most of our work until now has been in the area of finite-difference and finite-volume methods using boundary-fitted curvilinear grids and overlapping collections of such grids, we anticipate supporting additional types of grids and the methods that use them. 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), which actually contains no geometric data. The class **MappedGrid** (§3) is derived from **GenericGrid** and is used to describe curvilinear grids. It contains the geometric data normally required for the implementation of finite-difference and finite-volume methods. 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 collection of **GenericGrid**s. The class **GridCollection** (§5) is derived from GenericGridCollection, and contains a collection of **MappedGrid**s. All other Overture grid classes that contain collections of **MappedGrid**s are derived from **GridCollection**. In particular, the class **CompositeGrid** (§6) is derived from **GridCollection**. All of the Overture grid classes are reference-counted, using the envelope-letter paradigm. To support this, the base grid classes **GenericGrid** and **GenericGridCollection** are derived from the base class **ReferenceCounting** (Appendix A). All of these classes are described in this document.

2 Class GenericGrid

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 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 yet anticipated. For example, the class **MappedGrid** (§3) is derived from **GenericGrid** in order to describe curvilinear structured grids. **GenericGrid** is derived from class **ReferenceCounting** (Appendix A), which provides a mechanism for implementing the envelope-letter paradigm for reference-counted objects. This allows several **GenericGrid**s to share the same data, in such a way that the data are deallocated only when the last **GenericGrid** that uses the data is destroyed, due either to its being deleted or to its going out of scope.

2.1 Data-sharing between GenericGrids

Example 2.1 shows several ways that a **GenericGrid** can be made either to share data or to contain a separate copy of the data of another grid. The copy constructor can be used to construct a "shallow copy" of another grid, so that the two grids share the same data. The three forms shown in Example 2.1 for the use of the copy constructor to construct a shallow copy are equivalent and may be used interchangeably. An existing grid can also be made to become a shallow copy of another grid by using the **reference**() member function, as shown in the example. The example also shows several ways to make a "deep copy," in which data are copied from one grid to another when the two grids do not share data. First, the copy constructor may be used, with the second argument set to **DEEP**, to construct a copy of a grid with separate data all set equal to that of the original grid. Second, **operator=**() may be used to copy one grid to another existing grid. This will of course also modify any other grids with which the latter shares data. Third, a deep copy can be accomplished by using the **reference**() member function, followed by using the **breakReference**() member function, as shown in Example 2.1. By contrast to the use of **operator=**(), this method does not modify any other grids with which the grid originally shared data before the call to **reference**(). The member functions **operator==**() and **operator!=**() answer the question of whether two **GenericGrid**s are references to the same grid, in the sense that they share the same data. This test is fast and easy, but is not equivalent to checking whether all of the data of two distinct grids are equal. All of the functions shown in this example are available in derived classes such as **MappedGrid**. In some cases they may be called directly, as for **operator==**() and **operator!=**(); in some cases as virtual functions, as for **breakReference**(); and in other cases as overloaded functions with arguments of the appropriate derived types, as for the copy constructor, **operator=**() and **reference**().

```
#include "GenericGrid.h"
void example(const GenericGrid & g0)
{
  GenericGrid                              // Construct some grids.
       g1,                                 // g1 uses the default constructor.
```

```
      g2 = g0, g3(g0), g4(g0, SHALLOW),  // g2, g3 and g4 are "shallow copies" sharing data with g0,
      g5(g0, DEEP);                      // g5 is a "deep copy" of g0, with its own separate data.
  g1.breakReference();
  g1 = g0;   assert(g1 != g0);           // Now g1 is a "deep copy" of g0.
  g1.breakReference}();
  g1.reference}(g0);   assert(g1 == g0); // Now g1 is a "shallow copy," sharing data with g0.
  g1.breakReference(); assert(g1 != g0); // Now g1 has a separate "deep copy" of its old data.
}
```

Deep and shallow copies of **GenericGrid**s

2.2 Geometry management

Some member functions of class **GenericGrid** are used to manage the geometric data of a grid. Several of these member functions are overloaded in derived classes such as **MappedGrid** (See also the discussion in §3.1.) They are defined in the base grid class in order to provide a consistent interface independent of the particular derived class. They use the C++ virtual function mechanism in order to manage the geometry of derived classes in situations where only the base grid class name is known. The member function **computedGeometry**() returns a mask that indicates which geometric data have been computed and are up-to-date. Since class **GenericGrid** contains no geometric data, this mask is meaningful only for an object belonging to a derived class. The member function **geometryHasChanged**() is used to mark geometric data out-of-date. It takes as an optional argument an integer mask that indicates which geometric data have become out-of-date. This function has a default argument value that indicates all geometric data, including the data of any derived class. Before attempting to use any geometric data, it is essential to ensure that the data have been computed, by calling the member function **update**() with arguments specifying exactly which geometric data are needed. For example, if the **GenericGrid** is actually a **MappedGrid** and some geometric data are needed which are defined in class **MappedGrid**, then **update**() must be called with arguments specifying which **MappedGrid** geometric data are needed. If the grid is changed in any way that could affect its geometric data, then these data are automatically marked out-of-date. The data are never recomputed until this is explicitly requested by a call to **update**(), with the arguments specifying exactly which geometric data are needed. This argument is an integer mask formed by taking the "bitwise or" of constants defined in the class **GenericGrid** and any class derived from it. These member functions may be used for any class derived from **GenericGrid**, such as class **MappedGrid**. Example 2.2 illustrates the use of these member functions. First, we find out which geometric data already have been computed. This information is returned by **computedGeometry**() as a mask of bits, where each bit refers to geometric data that are specific to the derived class of the grid. We store this mask in the integer **cg**. Next, we mark this geometric data out-of-date using **geometryHasChanged**(), as we might want to do if we had modified the grid in a way that would affect its geometric data. For example, if the grid was a **MappedGrid** and we had applied a rotation to its mapping, then all of the geometric data would become invalid and would need to be recomputed. Next, we destroy this geometric data using **destroy**(). We would not normally do this; we do it here only to show how it may be done. This reduces the storage used for the **GenericGrid** to the least possible. Finally, we recompute the geometric data specified by the mask **cg**.

```
#include "GenericGrid.h"
void example(GenericGrid g)
{
  Integer cg = g.computedGeometry();  // Check which geometric data are up-to-date.
  g.geometryHasChanged(cg);           // Mark this geometric data out-of-date.
  g.destroy(cg);                      // Deallocate storage for this geometric data.
  g.update(cg);                       // Reallocate storage and compute this geometric data.
}
```

Example 2.2: **GenericGrid** geometry management

2.3 Input/output and miscellaneous member functions

We list here some member functions of class **GenericGrid** for input/output of data structures, and any other member functions not mentioned above. Typically these member functions are overloaded in derived classes, and most of these are virtual member functions. The virtual member function **initialize**() is used to put the **GenericGrid** back into the state it was in after it was first constructed. However, this function is not particularly useful, because if the grid belongs to a derived grid class (as it usually does), it may have undesired side effects that leave the grid in a

state that is not self-consistent. Other examples are the member functions **get**() and **put**(), for reading and writing a **GenericGrid** to or from a database. An example of an overloaded function which is not a virtual function (and not even a member function) is the stream output operator. This function prints out a summary of the data of a **GenericGrid**. Example 2.3 illustrates the use of these functions.

```
#include "GenericGrid.h"
void example(GenericDataBase & file)
{
  GenericGrid g;
  g.get(file, "grid1");           // Read a GenericGrid named "grid1" from the file.
  g.put(file, "grid2");           // Write the GenericGrid to the file with name "grid2."
  cout << "g = " << endl << g << endl;  // Print out the contents of g.
}
```

Example 2.3: Input/output of **GenericGrid**s

3 Class MappedGrid

Class **MappedGrid** is used for all logically-rectangular grids. This includes cartesian, rectangular and curvilinear grids in one, two and three dimensions. 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 an interpolant function with this property. **MappedGrid** is derived from class **GenericGrid** (§2). It overloads some of the **GenericGrid** public constants, member data and member functions described in (§2). All of the member functions of class **GenericGrid** described there may be used; only note that now the copy constructor and the member functions **reference**() and **operator=**() take a **MappedGrid** as their argument. There is an additional constructor for **MappedGrid**, which takes a **Mapping** as argument, as shown in Example 3.1. Corresponding to this constructor, there is also a **reference**() function which takes a **Mapping** as argument.

3.1 Geometry management

Class **MappedGrid** provides access to geometric data at the grid vertices. In addition, it provides access to geometric data at cell centers and other locations, as well as some global geometric data. However, to start the discussion, we consider first only the the global geometric data and the geometric data that class **MappedGrid** provides at the grid vertices. In particular, it provides access to the coordinates of the vertices as computed from the mapping, the derivative, the inverse derivative and the jacobian determinant of the derivative of the mapping at the vertices, as well as the unit outward normal vector at the boundary vertices of the grid. The global information it provides is a bounding box for vertices of the grid and, for each direction in the parameter space of the grid, the minimum and maximum distance between adjacent grid vertices. The grid vertices and other local geometric data computed at the grid vertices are stored in **RealMappedGridFunction**s, which are derived from **A++** arrays. The first three indices of these arrays indicate the particular grid vertex. The fourth index refers, in the case of the vertex coordinates, the mapping derivative and the normal vector, to the coordinate (x, y, z); for the inverse mapping derivative it refers to the coordinate (r_1, r_2, r_3). The fifth index refers, in the case of the mapping derivative, to the partial derivative $(\partial/\partial r_1, \partial/\partial r_2, \partial/\partial r_3)$; in the case of the inverse mapping derivative it refers to the partial derivative $(\partial/\partial x, \partial/\partial y, \partial/\partial z)$. For example, if the mapping is $\mathbf{d}\colon [0,1]^3 \mapsto \mathbb{R}^3$ then

the coordinates of grid vertex (i_1, i_2, i_3) are

$$\mathbf{d}(r_1, r_2, r_3) = \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{i_1,i_2,i_3} = \begin{bmatrix} \mathbf{vertex}(i_1, i_2, i_3, 0) \\ \mathbf{vertex}(i_1, i_2, i_3, 1) \\ \mathbf{vertex}(i_1, i_2, i_3, 2) \end{bmatrix}$$

and the mapping derivative at grid vertex (i_1, i_2, i_3) is

$$\begin{aligned} \frac{\partial \mathbf{d}}{\partial \mathbf{r}}(r_1, r_2, r_3) &= \begin{bmatrix} \frac{\partial x}{\partial r_1} & \frac{\partial x}{\partial r_2} & \frac{\partial x}{\partial r_3} \\ \frac{\partial y}{\partial r_1} & \frac{\partial y}{\partial r_2} & \frac{\partial y}{\partial r_3} \\ \frac{\partial z}{\partial r_1} & \frac{\partial z}{\partial r_2} & \frac{\partial z}{\partial r_3} \end{bmatrix}_{i_1,i_2,i_3} \\ &= \begin{bmatrix} \mathbf{vertexDerivative}(i_1, i_2, i_3, 0, 0) & \mathbf{vertexDerivative}(i_1, i_2, i_3, 0, 1) & \mathbf{vertexDerivative}(i_1, i_2, i_3, 0, 2) \\ \mathbf{vertexDerivative}(i_1, i_2, i_3, 1, 0) & \mathbf{vertexDerivative}(i_1, i_2, i_3, 1, 1) & \mathbf{vertexDerivative}(i_1, i_2, i_3, 1, 2) \\ \mathbf{vertexDerivative}(i_1, i_2, i_3, 2, 0) & \mathbf{vertexDerivative}(i_1, i_2, i_3, 2, 1) & \mathbf{vertexDerivative}(i_1, i_2, i_3, 2, 2) \end{bmatrix}. \end{aligned}$$

The ranges of indices of grid vertices that lie in the interior or on the boundary of the grid are returned by the function **gridIndexRange**(i,j), where $i = 0$ refers to the lower bound for the index and $i = 1$ refers to the upper bound; $j = 0, 1, 2$ refers to the first, second or third index of the grid vertex. The geometric data are computed at grid vertices with indices in the range given by **dimension**(i,j), which determines the array dimensions of the **RealMappedGridFunction**s, and may extend outside the boundary of the grid. Access to vertex data is shown in Example 3.1.

```
#include "MappedGrid.h"
#include "Annulus.h"

int
main()
{

  AnnulusMapping map;
  MappedGrid g = map;    // Construct MappedGrid from a  Mapping
  g.update(MappedGrid::THEvertex |              // Update the grid vertex coordinates, the
     MappedGrid::THEvertexDerivative |          // derivative of the mapping at the vertices,
     MappedGrid::THEinverseVertexDerivative |   // the inverse derivative at the vertices, the
     MappedGrid::THEvertexJacobian |            // jacobian determinant of the derivative, the
     MappedGrid::THEvertexBoundaryNormal |      // unit outward normal at boundary vertices,
     MappedGrid::THEboundingBox |               // the bounding box for grid vertices, and the
     MappedGrid::THEminMaxEdgeLength);          // min.~and max.~distance between vertices.

  RealMappedGridFunction & vertex = g.vertex(),    // Access the grid vertices and the
    & vertexDerivative = g.vertexDerivative();     // mapping derivative at the grid vertices.

  Range d0 = g.numberOfDimensions(),                  // The indices of the vertices corresponding
    I1(g.gridIndexRange(0,0),g.gridIndexRange(1,0)),  // to interior and boundaries of the grid are
    I2(g.gridIndexRange(0,1),g.gridIndexRange(1,1)),  // given by g.gridIndexRange.  The
    I3(g.gridIndexRange(0,2),g.gridIndexRange(1,2));  // fourth index selects the (x,y,z) component.

  RealArray v;
  v = vertex(I1,I2,I3,d0);                  // Copy the grid vertices
  v.display("vertices");

  return 0;
}
```

Example 3.1: Access to **MappedGrid** grid vertex data

The data are computed and stored only when this is requested through a call to the **update**() member function with arguments specifying exactly which geometric data are needed. The "bitwise or" operator is used to combine several requests into one. The geometric data requested through the call to **update**() are computed only if it is determined that they are out-of-date. This would occur on the first call to **update**(), for example, or any time that the mapping is changed or the number of grid vertices is changed. (Note, however, that the **MappedGrid** is not able to determine if or when the user changes its mapping; in that case the user is responsible to call **geometryHasChanged**(), as explained in §2.2.) If **update**() were called again right away with the same arguments then no geometric data would be recomputed, because the data would already be marked up-to-date as a result of the previous call to **update**(). It is essential that **update**() be called in any function where geometric data are used, to update the specific geometric data needed in that function, if there is any possibility that the data may be out-of-date when the function is called. This is always safe to do and is never wasteful, because the only data recomputed are those which are still out-of-date.

The grid vertex data are used for graphics applications such as drawing a representation of the grid. They also could be used in the discretization of partial differential equations. However, we discourage their use in the discretization of PDEs. For this purpose, **MappedGrid** provides and we recommend instead the use of geometric data at points that we refer to as "discretization-cell centers," "discretization points," or simply "gridpoints." These are points where the discretization of a PDE is centered. For a vertex-centered discretization, as commonly used in finite-difference methods, the discretization points are identical to the grid vertices. For a cell-centered discretization, as commonly used in finite-volume methods, the discretization cells are grid cells, whose corners are the grid vertices.

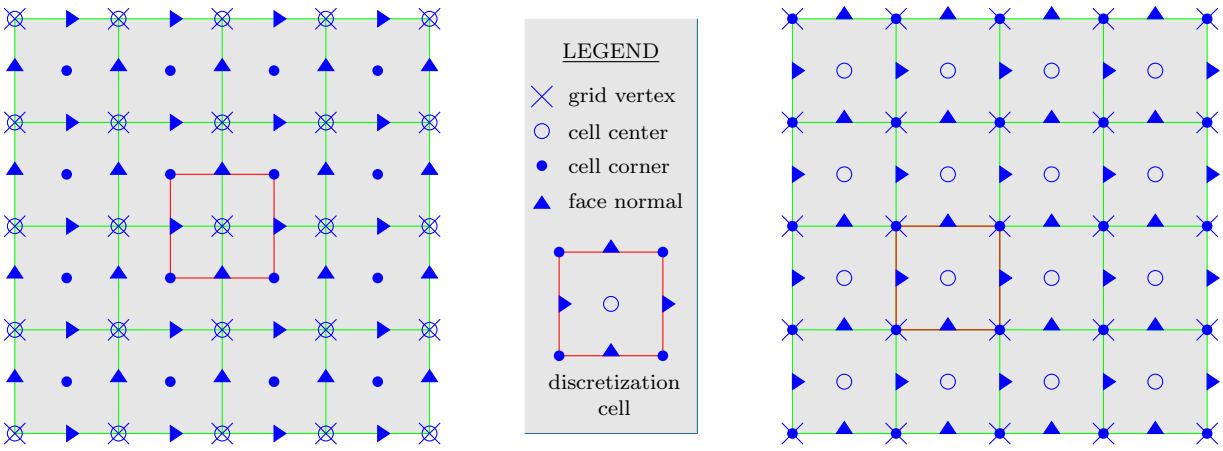


Figure 1: Some **MappedGrid** centering options

These two options for grid centering are illustrated in Figure 1. The **MappedGrid** contains the notion of how it is centered. This is given by the member function **isCellCentered**(i), which refers to the location of discretization points with respect to cells bounded by the grid vertices. For example, **isCellCentered**(i) = **LogicalFalse** for $i = 0, \ldots,$ **numberOfDimensions**() $- 1$ for a vertex-centered grid, used for a vertex-centered discretization, and **isCellCentered**(i) = **LogicalTrue** for $i = 0, \ldots,$ **numberOfDimensions**() $- 1$ for a cell-centered grid. Sometimes it is convenient to have available a PDE discretization that may be applied either at grid vertices or at grid-cell centers. To make it easy to implement such a discretization, **MappedGrid** provides geometric data at discretization points, and on faces and at corner of cells that are centered at discretization points. For a cell-centered grid, these cells are identical to the cells with grid vertices at their corners. However, for a vertex-centered grid, the corners of these cells form a dual grid. Some of the geometric data available at the discretization points are **center**, the coordinates of the discretization points, **centerDerivative**, **inverseCenterDerivative** and **centerJacobian**, the derivative, inverse derivative and the jacobian determinant of the derivative of the mapping computed at the discretization points. Example 3.1 shows how to access these data.

```
1   #include "MappedGrid.h"
2   #include "Annulus.h"
3
4   int
5   main()
6   {
7     AnnulusMapping map;
8     MappedGrid g = map;    // Construct MappedGrid from a  Mapping
9     g.update(MappedGrid::THEcenter |              // Update the discretization points, the
10       MappedGrid::THEcenterDerivative |          // derivative of the mapping,
11       MappedGrid::THEinverseCenterDerivative |   // the inverse derivative, and the
12       MappedGrid::THEcenterJacobian );           // jacobian determinant of the derivative
13
14    RealMappedGridFunction & center = g.center(); // Access the discretization points
15    center.display("center");
16    return 0;
17  }
```

Example 3.1: Access to **MappedGrid** discretization-point data

To aid in the implementation of finite-volume discretizations, **MappedGrid** provides additional geometric data about the discretization cells. In particular, it provides **faceNormal**, the inward normal vector to the left, bottom and back faces of the cell, normalized to the area of the face. (This assumes a right-handed coordinate system; for a left-handed coordinate system these vectors are outward normals.) It provides **faceArea**, the area of the left, bottom and back faces of the cell. Of course, the corresponding data on the right, top and front faces of the cell are

available also, as these faces are the left, bottom and back faces of neighbouring cells. Class **MappedGrid** provides also **cellVolume**, **centerNormal** and **centerArea**. The latter two refer to faces formed by cutting the cell through its center in each direction. Finally, on the boundary of the grid, it provides **centerBoundaryNormal**, the unit outward normal vector, and **centerBoundaryTangent**, unit vectors tangent to the faces of cells on the boundary. These tangent vectors are not necessarily orthogonal to each other, although each of them is orthogonal to the normal vector. Class **MappedGrid** also provides access to **corner**, the coordinates of corners of the discretization cells. In the case of a cell-centered grid, these are identical with the grid vertices, while in the case of a vertex-centered grid, these form a dual grid, as mentioned above. Example 3.1 shows how to access these data. Sometimes it is interesting to compute the geometric data not directly from the mapping, but instead by using appropriate finite differences and/or averaging of the coordinates of the discretization-cell corners. This is common when the finite-volume discretization is defined using a geometric construction. In order to force the geometry to be computed in this way, an optional second argument to **update**() may be passed the value **MappedGrid::USEdifferenceApproximation**, as shown in Example 3.1.

```
1   #include "MappedGrid.h"
2   #include "Annulus.h"
3
4   int
5   main()
6   {
7     AnnulusMapping map;
8     MappedGrid g = map;           // Construct MappedGrid from a  Mapping
9     g.changeToAllCellCentered();  // Make this a cell-centered grid.
10    g.update(MappedGrid::THEcorner       |   // Update the discretization cell corners,
11             MappedGrid::THEfaceNormal   |   // the cell face normal vectors,
12             MappedGrid::THEfaceArea     |   // the cell face areas,
13             MappedGrid::THEcellVolume   |   // the cell volumes,
14             MappedGrid::THEcenterNormal |   // the cell-centered normal vectors,
15             MappedGrid::THEcenterArea   |   // the cell-centered face areas,
16             MappedGrid::THEcenterBoundaryNormal |  // the unit normals to the grid boundaries, and
17             MappedGrid::THEcenterBoundaryTangent | // the unit tangents to the grid boundaries.
18             MappedGrid::USEdifferenceApproximation);  // Compute the the cell corners from the mapping,
19                                                       // and compute everything else from the corners by
20                                                       // finite differences and averaging of corner data.
21    RealMappedGridFunction & corner = g.corner(); // Access the discretization points
22    corner.display("corner");
23    return 0;
24  }
```

Example 3.1: Access to **MappedGrid** data used for finite-volume discretizations

3.2 Access to MappedGrid parameters

Class **MappedGrid** has many useful parameters that may accessed and modified in order to make the grid useful for the discretization of a given PDE problem. For example, as we mentioned above, a **MappedGrid** may be vertex-centered or cell-centered. There are parameters that may be modified by the user in order to change the grid from vertex-centered to cell-centered and vice versa. The parameters of a **MappedGrid** may be divided into several broad categories. First, there are parameters describing the dimensions of the index space of data (such as gridpoints) on the grid. Second, there are parameters describing the topology of the grid. Third, there are parameters describing discretization stencils used on the grid. The cell-centering is such a parameter. All of these parameters have reasonable default values; some of these are determined from the mapping that describes the grid. Many of these parameters may be set or changed by the user; the rest are recomputed whenever **update**() is called, in order to make all of the parameters consistent with each other. Whenever any of these parameters is changed, and the result of the change makes some of the geometric data invalid, the data that become invalid are marked out-of-date. For example, if the number of gridpoints is changed, then all geometric data that are computed on the gridpoints (such as the coordinates of the grid vertices) become invalid. It is the user's responsibility to call **update**() again with arguments that specify which geometric data are needed, in order to recompute geometric data that are still needed but have become out-of-date. For example, if **center**, the coordinates of the discretization points, is needed again after the number of gridpoints has changed, then it is necessary to call **update**() and pass **MappedGrid::THEcenter** as its argument. If this step is omitted, then **center** will either be empty, or it will

have the wrong dimensions, or it will contain incorrect data. There is no warning given when geometric data become out-of-date, because there is no assumption made about which data will be needed again unless and until **update**() is called to specify exactly which geometric data are still needed.

The **MappedGrid** parameters that describe the index space of grid data are **indexRange**, **extendedIndexRange**, **gridIndexRange**, **dimension** and **numberOfGhostPoints**. The parameters that may be set by the user are **gridIndexRange** and **numberOfGhostPoints**; the other parameters are automatically computed from these whenever **update**() is called. The parameter **gridIndexRange** gives the indices of grid vertices corresponding to each boundary of the grid. In the index space of the grid, where the grid vertices form an integer lattice, we refer to the left, right, bottom, top, back and front sides of the grid, where the first, second and third indices of grid vertices take their lower and upper bound, respectively. The lower and upper bounds of the first index of the grid vertices on the boundary are **gridIndexRange**(0,0) and **gridIndexRange**(1,0), respectively; the lower and upper bounds of the second index are **gridIndexRange**(0,1) and **gridIndexRange**(1,1), and the lower and upper bounds of the third index are **gridIndexRange**(0,2) and **gridIndexRange**(1,2). These are initially determined from the mapping that describes the grid; the user may change them using the function **setGridIndexRange**(). The parameter **numberOfGhostPoints** gives the width of a band of extra gridpoints that lie outside the boundary of the grid. These extra gridpoints are not usually considered to be part of the grid, but may be convenient to have for the discretization of a PDE. There is no guarantee that the mapping is well-behaved at these points. The number of ghost points on the left and right sides of the grid are given by **numberOfGhostPoints**(0,0) and **numberOfGhostPoints**(1,0), respectively; on the bottom and top, **numberOfGhostPoints**(0,1) and **numberOfGhostPoints**(1,1); and on the back and front, **numberOfGhostPoints**(0,2) and **numberOfGhostPoints**(1,2). These are initially set to one, which may be convenient for a discretization-stencil width of three. They may be set using the function **setNumberOfGhostPoints**(). All of the other parameters that describe the index space of grid data are recomputed whenever **update**() is called. For example, **dimension**, the dimensions used for the arrays of grid data, is set to **gridIndexRange** plus or minus **numberOfGhostPoints**. The parameter **indexRange** gives the lower and upper bounds for the discretization points in the interior and on the boundary of the grid. For a vertex-centered grid, **indexRange** is identical to **gridIndexRange**, because the discretization points are the grid vertices. The exception to this rule is a periodic grid, in which case the first and last gridpoints are equivalent; the last gridpoint is considered redundant for the purposes of discretization, so it is not included in the range of discretization points, and **indexRange**$(1,i)$ = **gridIndexRange**$(1,i) - 1$ in the periodic direction i. For a cell-centered grid, the discretization points are grid-cell centers. In each direction the number of cells is one fewer than the number of grid vertices. The cell centers are numbered the in same way as their lower-left corners (*i.e.*, the grid vertices), so **indexRange**$(0,i)$ = **gridIndexRange**$(0,i)$ and **indexRange**$(1,i)$ = **gridIndexRange**$(1,i) - 1$ on a cell-centered grid. To summarize, **indexRange** is the same as **gridIndexRange**, except in directions where the grid is either periodic or cell-centered, in which cases the upper bound **indexRange**(1,i) is less by one. The parameter **extendedIndexRange** is identical to **indexRange** with the possible exception that on sides of the grid where there is no boundary condition specified, **extendedIndexRange** may include a strip of ghost points as wide as (**discretizationWidth** $- 1)/2$. This strip of ghost points is included only if the flag **useGhostPoints**() is set to **LogicalTrue**. By default, this flag is set to LogicalFalse; it may be changed by calling the function **setUseGhostPoints**(). Example 3.2 shows how to access and set the parameters that describe the index space of grid data.

```
1   #include "MappedGrid.h"
2   #include "Annulus.h"
3
4   int
5   main()
6   {
7     AnnulusMapping map;
8     MappedGrid g = map;
9
10    printf("gridIndexRange=[%i,%i]x[%i,%i]x[%i,%i]\n",       // Print out the range of indices for grid vertices.
11           g.gridIndexRange(0,0),g.gridIndexRange(1,0),
12           g.gridIndexRange(0,1),g.gridIndexRange(1,1),
13           g.gridIndexRange(0,2),g.gridIndexRange(1,2));
14
15    printf("indexRange=[%i,%i]x[%i,%i]x[%i,%i]\n",          // Print out the range for discretization points.
16           g.indexRange(0,0),g.indexRange(1,0),
17           g.indexRange(0,1),g.indexRange(1,1),
18           g.indexRange(0,2),g.indexRange(1,2));
19
```

```
20     printf("numberOfGhostPoints=[%i,%i]x[%i,%i]x[%i,%i]\n",   // Print out the number of ghost points
21            g.numberOfGhostPoints(0,0),g.numberOfGhostPoints(1,0),
22            g.numberOfGhostPoints(0,1),g.numberOfGhostPoints(1,1),
23            g.numberOfGhostPoints(0,2),g.numberOfGhostPoints(1,2));
24
25     printf("dimension=[%i,%i]x[%i,%i]x[%i,%i]\n",            // Print out the grid dimensions
26            g.dimension(0,0),g.dimension(1,0),
27            g.dimension(0,1),g.dimension(1,1),
28            g.dimension(0,2),g.dimension(1,2));
29
30     int axis;
31     for( axis=0; axis<g.numberOfDimensions(); axis++ )
32     {
33       g.setIsCellCentered(axis,LogicalTrue); // Change the grid to cell-centered
34       g.setNumberOfGhostPoints(0,axis, 2);  // Change the number of ghost points on the left side
35       g.setNumberOfGhostPoints(1,axis, 2); // Change the number of ghost points on the right side
36     }
37
38     g.update(MappedGrid::THEvertex);  // Compute the geometry;
39
40     return 0;
41   }
```

```
----------------------- output -----------------------
gridIndexRange=[0,20]x[0,6]x[0,0]
indexRange=[0,19]x[0,6]x[0,0]
numberOfGhostPoints=[1,1]x[1,1]x[0,0]
dimension=[-1,21]x[-1,7]x[0,0]
```

Example 3.2: Access to **MappedGrid** index-space data

The **MappedGrid** parameters that describe the topology of the grid are **numberOfDimensions**, **isPeriodic**, **boundaryCondition**, **sharedBoundaryFlag** and **sharedBoundaryTolerance**. All of these parameters have default values determined from the mapping that describes the grid. These parameters may all be set by the user. However, it is important that they be set in such a way as to be consistent with the mapping. At this time I can think of no good reason to change **numberOfDimensions**; to do so would probably cause all kinds of trouble. Its initial value is that of the **Mapping** member function **getRangeDimension**(), which we assume is equal to **getDomainDimension**(). The parameter **isPeriodic** gives the periodicity of the grid in each coordinate direction. If the grid itself is periodic, then **isPeriodic** should be set to **Mapping::functionPeriodic**; otherwise, if the PDE problem to be solved on the grid is periodic, then **isPeriodic** should be set to **Mapping::derivativePeriodic**; otherwise **isPeriodic** should be set to **Mapping::notPeriodic**. The parameter **boundaryCondition** should be positive on each side of the grid that corresponds to a domain boundary. It should be negative on both opposite sides of the grid in each direction where either the domain is periodic or the PDE problem to be solved on the grid is periodic. In any case, the parameters **boundaryCondition** and **isPeriodic** must be set so that they are consistent with each other. The parameters **sharedBoundaryFlag** and **sharedBoundaryTolerance** are useful only in situations where several **MappedGrid**s overlap to cover the domain. Class **CompositeGrid** contains **MappedGrid**s used for this purpose, so the discussion of these parameters is deferred. The use of the **MappedGrid** parameters that describe the topology of the grid is shown in Example 3.2.

```
1    #include "MappedGrid.h"
2    #include "Annulus.h"
3
4    int
5    main()
6    {
7      AnnulusMapping map;
8      MappedGrid g = map;
9
10     printf("isPeriodic=[%i,%i,%i]\n",       // Print out the periodicity of each grid coordinate
11            g.isPeriodic(axis1),g.isPeriodic(axis2),g.isPeriodic(axis3));
12
13     printf("boundaryCondition=[%i,%i]x[%i,%i]x[%i,%i]\n",    // Print out the boundary condition for each side.
```

```
14            g.boundaryCondition(0,0),g.boundaryCondition(1,0),
15            g.boundaryCondition(0,1),g.boundaryCondition(1,1),
16            g.boundaryCondition(0,2),g.boundaryCondition(1,2));
17
18     printf("numberOfGhostPoints=[%i,%i]x[%i,%i]x[%i,%i]\n",     // Print out the shared boundary flag.
19            g.sharedBoundaryFlag(0,0),g.sharedBoundaryFlag(1,0),
20            g.sharedBoundaryFlag(0,1),g.sharedBoundaryFlag(1,1),
21            g.sharedBoundaryFlag(0,2),g.sharedBoundaryFlag(1,2));
22
23     printf("sharedBoundaryTolerance=[%f,%f]x[%f,%f]x[%f,%f]\n",  // Print out the shared boundary tolerance
24            g.sharedBoundaryTolerance(0,0),g.sharedBoundaryTolerance(1,0),
25            g.sharedBoundaryTolerance(0,1),g.sharedBoundaryTolerance(1,1),
26            g.sharedBoundaryTolerance(0,2),g.sharedBoundaryTolerance(1,2));
27
28     g.setIsPeriodic(axis1,Mapping::notPeriodic);  // Make the grid not periodic
29     g.setBoundaryCondition(0,axis1, 2);           // assign boundary condition values.
30     g.setBoundaryCondition(1,axis1, 3);
31
32     g.update(MappedGrid::THEvertex);
33     return 0;
34   }
```

```
----------------------- output ----------------------
isPeriodic=[2,0,1]
boundaryCondition=[-1,-1]x[1,2]x[-1,-1]
numberOfGhostPoints=[0,0]x[0,0]x[0,0]
sharedBoundaryTolerance=[0.100000,0.100000]x[0.100000,0.100000]x[0.000000,0.000000]
```

Example 3.2: Access to **MappedGrid** topology data

The **MappedGrid** parameters that describe discretization stencils used on the grid are **discretizationWidth**, **boundaryDiscretizationWidth**, **isCellCentered**, **isAllCellCentered**, **isAllVertexCentered** and **gridSpacing**. All of these have default values. The parameters **discretizationWidth**, **boundaryDiscretizationWidth** and **isCellCentered** may be set by the user; the others are automatically computed whenever **update**() is called. The parameter **discretizationWidth** refers to the width of the interior discretization stencil in each coordinate direction, and may be set to any positive odd integer. (A centered discretization is assumed.) The default value for **discretizationWidth** is three. The parameter **boundaryDiscretizationWidth** refers to the width, in the normal direction, of the discretization stencil of the boundary conditions on each side of the grid. This width does not include any ghost points that may or may not be used in the discretization of the boundary conditions. The stencil width in the tangential directions is assumed to be equal to the interior discretization-stencil width in those directions. This width may be any positive integer. The default value for **boundaryDiscretizationWidth** is three, which is convenient for a second-order one-sided approximation to the normal derivative. The parameter **isCellCentered** refers to the centering of discretization points within the cells of the grid. By default, **isCellCentered**(i) is **LogicalFalse** in each direction i, to indicate a vertex-centered grid. The parameter **isCellCentered**(i) may be set independently for each direction i, so that face-centered and edge-centered grids are possible in addition to vertex-centered and cell-centered grids. However, I can think of no possible use for a face-centered or edge-centered grid. Certainly some geometrical data such as face normal vectors may be face-centered, but this has nothing to do with whether or not the grid itself is face-centered. The parameter **isAllCellCentered** is computed by **update**() and is **LogicalTrue** only if **isCellCentered**(i) is **LogicalTrue** for all directions $0 < i <$ **numberOfDimensions**(). Likewise, the parameter **isAllVertexCentered** is computed by **update**() and is **LogicalTrue** only if **isCellCentered**(i) is **LogicalFalse** for all directions $0 < i <$ **numberOfDimensions**(). Finally, the parameter **discretizationWidth** is computed by **update**() to be the distance between gridpoints on the uniform rectangular grid on the unit-square parameter space of the grid and its mapping. The use of the **MappedGrid** parameters that describe discretization stencils on the grid is shown in Example 3.2.

```
1   #include "MappedGrid.h"
2   #include "Annulus.h"
3
4   int
5   main()
6   {
7     AnnulusMapping map;
```

```
8     MappedGrid g = map;
9
10    printf("discretizationWidth=[%i,%i,%i]\n",       // Print out the interior discretization width
11           g.discretizationWidth(axis1),g.discretizationWidth(axis2),g.discretizationWidth(axis3));
12
13
14    printf("boundaryDiscretizationWidth=[%i,%i]x[%i,%i]x[%i,%i]\n", // Print boundary discretization width
15           g.boundaryDiscretizationWidth(0,0),g.boundaryDiscretizationWidth(1,0),
16           g.boundaryDiscretizationWidth(0,1),g.boundaryDiscretizationWidth(1,1),
17           g.boundaryDiscretizationWidth(0,2),g.boundaryDiscretizationWidth(1,2));
18
19    printf("gridSpacing=[%f,%f,%f]\n",  // Print out the unit square spacing in each direction
20           g.gridSpacing(axis1),g.gridSpacing(axis2),g.gridSpacing(axis3));
21
22    printf("isCellCentered=[%i,%i,%i]\n",       // Print te cell-centering in each direction.
23           g.isCellCentered(axis1),g.isCellCentered(axis2),g.isCellCentered(axis3));
24
25
26    printf("isAllCellCentered=%i\n",g.isAllCellCentered());     // Print out the cell-centering of the grid
27    printf("isAllVertexCentered=%i\n",g.isAllVertexCentered());
28
29    int axis;
30    for( axis=0; axis<g.numberOfDimensions(); axis++ )
31      g.setIsCellCentered(axis,LogicalTrue); // Change the grid to cell-centered
32
33    g.update(MappedGrid::THEvertex);
34    return 0;
35  }
```

```
----------------------- output -----------------------
discretizationWidth=[3,3,1]
boundaryDiscretizationWidth=[3,3]x[3,3]x[1,1]
gridSpacing=[0.050000,0.166667,1.000000]
isCellCentered=[0,0,0]
isAllCellCentered=0
isAllVertexCentered=1
```

Example 3.2: Access to **MappedGrid** discretization-stencil data

There are two exceptions to the rule that geometric data invalidated by changes to parameters are not automatically updated at the same time as the parameter changes are made. The member functions **changeToAllCellCentered**() and **changeToAllVertexCentered**() cause any geometric data that were up-to-date before the call to be recomputed if the parameter changes caused them to become invalid. This is supposed to be useful in the case where the only parameter changes needed are to change the grid either to cell-centered or to vertex-centered. If any other parameters need to be changed also, then the geometric data may become invalid as a result of these other changes, and it would be a waste of time to have these member functions recompute the geometric data before all of the changes have been made. Also, it is wasteful to update the geometric data before changing the grid to cell-centered or vertex-centered, as some of the geometric data will usually need to be recomputed. Clearly, these member functions should be used with care if they are used at all.

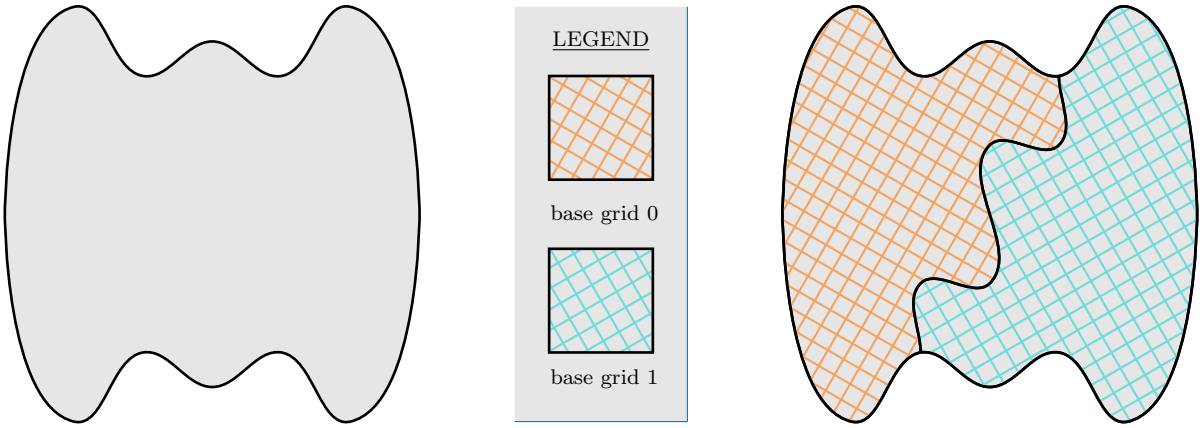
3.3 Miscellaneous member functions

Class **MappedGrid** has a few member functions not mentioned above. These functions are probably not needed by most people. The function **box**() returns a reference to a **Boxlib Box** object that contains index-space and cell-centering information about the **MappedGrid**. This information is redundant in the sense that it duplicates the information returned by **indexRange**() and **isCellCentered**(). However, it may be useful for interfacing with code that uses **Boxlib**. The function **adjustBoundary**() is used to force the parameter-space coordinates (r_1, r_2, r_3) of a point to zero or one, as appropriate, if the point is on a boundary of another grid that is shared with the corresponding boundary of this grid. The function **getInverseCondition**() computes a condition number for the sensitivity of the parameter-space coordinates (r_1, r_2, r_3) in this grid of a point from another grid, to errors in the coordinates of the point in the parameter space of the other grid. This condition number is used in estimates of the truncation error for interpolation of data from the other grid. The function **specifyProcesses**() is used to specify how data (such

as geometric data) on the **MappedGrid** are distributed over processes on a multiprocessor computer. The function **initialize**() is used to put the **MappedGrid** back into the state it was in after it was first constructed, supposing that it was constructed using its current mapping. In this state, the geometric data are marked out-of-date and their storage is dellocated. This function should be used with care, because if the grid belongs to a derived grid class, it may have undesired side effects that leave the grid in a state that is not self-consistent. In addition to these functions, the member functions of the base grid class **GenericGrid** are accessible from **MappedGrid**, either directly as they are described in §2 or as overloaded functions defined in class **MappedGrid**.

4 Class GenericGridCollection

Class **GenericGridCollection** is the base class for all Overture classes that contain collections of grids. It is derived from class **ReferenceCounting** (Appendix A), just as class **GenericGrid** is. As a consequence, the entire discussion of data-sharing in §2.1 and Example 2.1 applies directly to class **GenericGridCollection**. For each **GenericGrid** member function discussed there, there is a corresponding **GenericGridCollection** member function. The only difference is that the default constructor and member function **initialize**() take an optional argument specifying the number of grids initially in the collection, as shown in Example 4.1. The discussion of geometry management in §2.2 and Example 2.2 also applies directly to class **GenericGridCollection**. Whenever the **GenericGridCollection** member functions **update**(), **destroy**(), or **geometryHasChanged**() are called, these in turn call the corresponding function for each grid in the collection, passing on the arguments that specify the geometric data. In addition, these member functions are used to manage the geometric data discussed in §4.1, as shown in Example 4.1. The discussion of the miscellaneous functions in §2.3 and Example 2.3 also applies directly to class **GenericGridCollection**. It is a valuable exercise to review §2 in its entirety, mentally substituting **GenericGridCollection** everywhere that **GenericGrid** appears there.



The domain

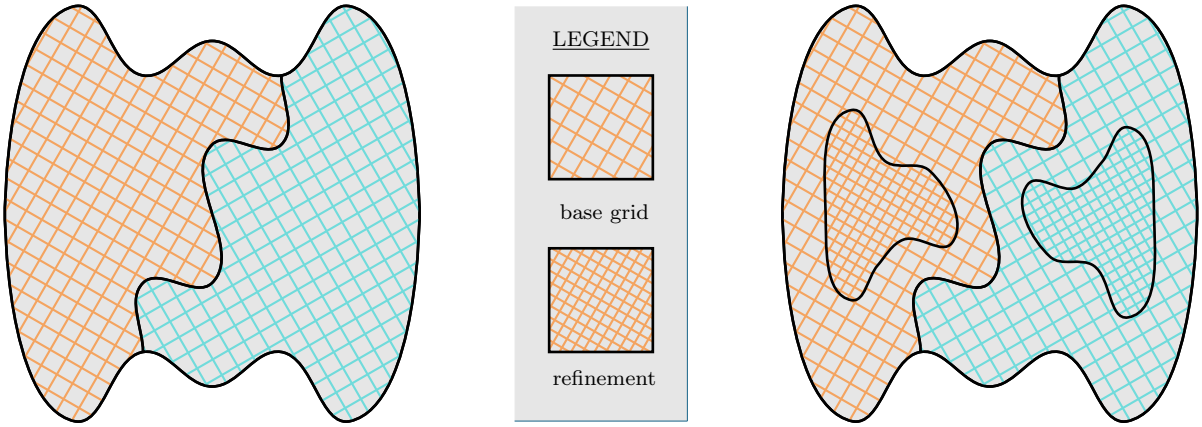
Two base grids covering the domain

Figure 2: A computational domain covered by a **GenericGridCollection** of two base grids. The cross-hatch patterns abstractly represent the base grids. However, since the grids are only **GenericGrid**s, which have no particular structure, these patterns are are not intended to imply the structure of the grids. The grids may be rectangular, unstructured triangular, or who knows what else.

Class **GenericGridCollection** contains a collection of **GenericGrid**s that may be indexed using the member function **operator[]**(), and it may be used as merely a collection of grids. However, it provides a mechanism for adding some structure to this collection. Each grid added to the collection is called a "base grid," to indicate that these grids are independent of and unrelated to each other. A base grid may cover the entire domain of the problem under consideration, or it may possibly cover only part of the domain, in which case we expect to have several base grids which together cover the entire domain. If there is more than one base grid, then each of the base grids is normally expected to cover a different part of the domain, although possibly with some overlap. It is expected that all of the base grids are needed in order to describe the domain accurately, and that if any of the base grids were omitted then the description of the domain would be incomplete or at best inaccurate. Figure 2 shows a cartoon of a domain covered by two base grids. The structure of the base grids is intended to be ambiguous, since these are **GenericGrid**s

and therefore have no particular structure. In a concrete example, these grids would belong to a derived grid class and would have some geometric structure. For example, the base grids might be **MappedGrid**s, in which case they would have logically-rectangular geometric structure that includes curvilinear gridlines, quadrilateral or hexahedral cells, and the like. This figure is supposed to convey the concept of a **GenericGridCollection** consisting of two base grids that do not overlap but which together cover the domain completely. Example 4.1 shows how a **GenericGridCollection** may be constructed with two grids that are also two base grids.

4.1 Local refinements and multigrid coarsenings



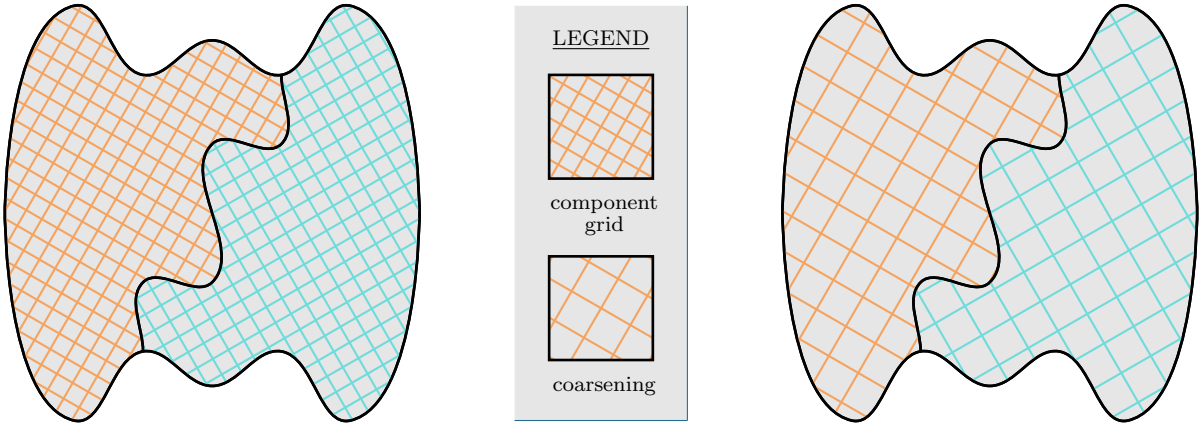
Two base grids

Base grids with one refinement each

Figure 3: Some **GenericGridCollection** base grids and local refinements. Each base grid has one level-one refinement grid. This **GenericGridCollection** has two base grids and two refinement levels.

Corresponding to each base grid in a **GenericGridCollection**, there may also be some local refinement grids. All of the refinements of a base grid are related to that base grid, in the sense that they each cover either the same sub-domain as that of the unrefined base grid, or perhaps only a part of that sub-domain. In other words, each refinement grid is a refinement of only one base grid, and does not cover any part of the domain which is not already covered by that base grid. Each base grid and refinement grid is labeled by a base grid number and a refinement level number. Each unrefined base grid has a unique base grid number and has refinement level number zero; its refinements have the same base grid number and have refinement level number one. The union of the level-one refinements of an unrefined base grid covers a sub-domain within that of the base grid. If this sub-domain is refined further, then these further refinements have refinement level number two. In this way an arbitrary number of nested refinement levels may be added to a base grid, each consisting of one or more local refinement grids. In other words, the refinement levels form successively finer representations of nested sub-domains. If the **GenericGridCollection** contains more than one base grid, then the collection of grids may be partitioned into disjoint subsets of grids that belong to each of the unrefined base grids (those grids which have in common their base grid number). Class **GenericGridCollection** contains a collection of **GenericGridCollection**s that hold the subsets of grids that form this partition. It also contains a collection of **GenericGridCollection**s that hold the disjoint subsets of grids that have in common their refinement level number, without consideration of which base grids they refine. Each of these subsets may contain refinements of more than one base grid, if it happens that different base grids each have refinements at the same refinement level. The two partitions of the collection of grids into subsets according to their base grid number and according to their refinement level number are arthogonal to each other in the sense that these two partitioning criteria are independent of each other. Figure 3 shows on the left a domain covered by two base grids; on the right it shows these base grids each partially covered by a level-one local refinement grid. Example 4.1 shows how a refinement may be added to each of two base grids, so that afterward the **GenericGridCollection** has two refinement levels and two base grids, each with one level-one refinement, for a total of four grids.

Each base grid or refinement grid in a **GenericGridCollection** may be coarsened for the purposes of multigrid, and may have several levels of multigrid coarsening. Each base grid or refinement grid, before coarsening for multigrid, is called a "component grid." Each multigrid coarsening covers the same subdomain as the component grid that it



Two component grids (base grids)

One coarsening of each component grid

Figure 4: Some **GenericGridCollection** component grids and multigrid coarsenings. The component grids in this case are base grids. Each component grid has one level-one multigrid coarsening. This **GenericGridCollection** has two component grids and two multigrid levels.

coarsens. If there is more than one level of multigrid coarsening, then each multigrid level is a coarsening of the grid that forms the preceding multigrid level. In other words, the multigrid levels form successively coarser representations of the sub-domain covered by their corresponding component grid (which may itself be either a base grid or a refinement grid). Each component grid and multigrid coarsening is labeled with a component grid number and a multigrid level number. Each component grid has a unique component grid number and has multigrid level number zero; its multigrid coarsenings have the same component grid number and have increasing multigrid level numbers according to the number of times they have been coarsened. If there is more than one component grid, the collection of grids may be partitioned into disjoint subsets of grids that have the same component grid number. Class **GenericGridCollection** contains a collection of **GenericGridCollection**s that hold the subsets of grids that form this partition. It also contains a collection of **GenericGridCollection**s that holds the disjoint subsets of grids that have in common their multigrid level number. The two partitions of the collection of grids into subsets according to their component grid number and according to their multigrid level number are arthogonal to each other in the sense that these two partitioning criteria are independent of each other. Figure 4 shows on the left a domain covered by two base grids, each of which is a component grid; on the right it shows level-one multigrid coarsenings of these component grids. Example 4.1 shows how a multigrid coarsenings may be added to each of four component grids, two of which are base grids and two are refinements, so that afterward the **GenericGridCollection** has two base grids at two refinement levels, for a total of four component grids, and each of these has three multigrid levels, for a total twelve grids.

```
1   #include "GenericGridCollection.h"
2
3   void
4   print(GenericGridCollection & gc)
5   {
6     printf("numberOfGrids=%i, numberOfBaseGrids=%i, numberOfComponentGrids=%i\n"
7            "numberOfRefinementLevels=%i, numberOfMultigridLevels=%i\n",
8            gc.numberOfGrids(),gc.numberOfBaseGrids(),gc.numberOfComponentGrids(),
9            gc.numberOfRefinementLevels(),gc.numberOfMultigridLevels());
10  }
11
12  int
13  main()
14  {
15    GenericGridCollection gc(2); // GenericGridCollectionwith 2 base grids
16
17    printf("Initial:\n"); print(gc);
```

```
18
19      int grid;
20      for( grid=0; grid<gc.numberOfBaseGrids(); grid++ )
21        gc.addRefinement(1,grid);
22
23      printf("After add refinement:\n"); print(gc);
24
25      int level;
26      for( level=1; level<3; level++ )
27        for( grid=0; grid<gc.numberOfComponentGrids(); grid++ )
28          gc.addMultigridCoarsening(level,grid);
29
30      printf("After add multigrid coarsenings:\n"); print(gc);
31
32
33      gc.update(GenericGridCollection::THEbaseGrid); // Partition g according to base grid number.
34      GenericGridCollection & gc1=gc.baseGrid[1];    // Access base grid one and its refinements.
35
36      gc1.update(GenericGridCollection::THEmultigridLevel); // Partition g1 according to multigrid level.
37
38      GenericGridCollection & gc12=gc1.multigridLevel[2]; // Access base grid one and its refinements
39                                                          // at multigrid level two.
40
41      return 0;
42    }
43
```

```
----------------------- output -----------------------
Initial:
numberOfGrids=2, numberOfBaseGrids=2, numberOfComponentGrids=2
numberOfRefinementLevels=1, numberOfMultigridLevels=1
After add refinement:
numberOfGrids=4, numberOfBaseGrids=2, numberOfComponentGrids=4
numberOfRefinementLevels=2, numberOfMultigridLevels=1
After add multigrid coarsenings:
numberOfGrids=12, numberOfBaseGrids=2, numberOfComponentGrids=4
numberOfRefinementLevels=2, numberOfMultigridLevels=3
```

Example 4.1: Access to partitions of a **GenericGridCollection**

To summarize, the collection of grids in a **GenericGridCollection** may be partitioned according to base grid or refinement level, and these two partitions are orthogonal to each other. Similarly, the collection may be partitioned according to component grid or multigrid level, and these two partitions also are orthogonal to each other. Since the collections forming the subsets in any of these four partitions are also **GenericGridCollection**s, they may be further partitioned in exactly the same ways. For example, it is possible in this way to form a collection of all the refinements of a particular base grid at a given multigrid level. Example 4.1 shows how member functions **update**() and **operator[]**() may be used to partition a **GenericGridCollection** in this way.

4.2 Miscellaneous member functions

We describe here some member functions of class **GenericGridCollection** that were not mentioned above. Typically these member functions are overloaded in derived classes, and most of these are virtual member functions. The virtual member functions **deleteMultigridCoarsening**() and **deleteMultigridLevels**() are used to delete multigrid coarsenings. The latter deletes all coarsenings whose multigrid levels are higher than the level specified. Similarly, member functions **deleteRefinement**() and **deleteRefinementLevels**() are used to delete refinement grids; the latter deletes all refinement grids whose refinement levels are higher than the level specified. Sometimes it is useful and efficient to be able to have a "new" **GenericGridCollection** that shares all of the unrefined grids of an "old" **GenericGridCollection**. A different set of refinements may be added to the new collection so that it becomes appropriately adapted to new data. At this point, the old and new **GenericGridCollection**s share the same base grids but have distinct sets of refinements. The virtual member function **referenceRefinementLevels**() provides this capability. It causes one **GenericGridCollection** to share with another **GenericGridCollection** all grids with refinement levels up to a given level of refinement. In order to share only the base grids, you would

pass arguments to specify level zero. The member function **getIndex**() is used to look for a **GenericGrid** within a **GenericGridCollection**. It returns the index of the grid within the collection, if the grid is found; otherwise it returns a negative number to indicate that the grid was not found. The **GenericGrid** member functions **operator==**() and **operator!=**() are used in this search. Example 4.2 shows how some of these member functions may be used.

```
1   #include "GenericGridCollection.h"
2
3   void
4   print(GenericGridCollection & gc)
5   {
6     printf("numberOfGrids=%i, numberOfBaseGrids=%i, numberOfComponentGrids=%i\n"
7            "numberOfRefinementLevels=%i, numberOfMultigridLevels=%i\n",
8            gc.numberOfGrids(),gc.numberOfBaseGrids(),gc.numberOfComponentGrids(),
9            gc.numberOfRefinementLevels(),gc.numberOfMultigridLevels());
10  }
11
12  int
13  main()
14  {
15    GenericGridCollection gc1(2), gc2;
16
17    int grid,level;
18    for( level=1; level<3; level++ )
19      for( grid=0; grid<gc1.numberOfBaseGrids(); grid++ )
20        gc1.addRefinement(level,grid);   // Add level-one and level-two refinements of each base grid.
21
22    for( level=1; level<3; level++ )
23      for( grid=0; grid<gc1.numberOfComponentGrids(); grid++ ) // Add level-one and level-two multigrid
24        gc1.addMultigridCoarsening(level,grid);                // coarsenings of each component grid.
25
26    printf("gc1 after adding grids:\n"); print(gc1);
27
28    gc2.referenceRefinementLevels(gc1, 1);  // Share grids with gc1 that have  refinement level at most one.
29
30    printf("gc2 after reference:\n"); print(gc2);
31
32    gc1.deleteRefinementLevels(1); // Delete refinements with refinement levels higher than one.
33    gc1.deleteMultigridLevels(1);  // Delete coarsenings with multigrid levels higher than one.
34
35    printf("gc1 after deleting grids:\n"); print(gc1);
36
37    GenericGrid & g1 = gc1[0]; assert(gc1.getIndex(g1) == 0);  // Check that g1 is in gc1 at index zero.
38    GenericGrid g2; assert(gc1.getIndex(g2) < 0);  // Check that g2 is not in gc1.
39
40    return 0;
41  }
42
```

```
----------------------- output -----------------------
gc1 after adding grids:
numberOfGrids=18, numberOfBaseGrids=2, numberOfComponentGrids=6
numberOfRefinementLevels=3, numberOfMultigridLevels=3
gc2 after reference:
numberOfGrids=12, numberOfBaseGrids=2, numberOfComponentGrids=4
numberOfRefinementLevels=2, numberOfMultigridLevels=3
gc1 after deleting grids:
numberOfGrids=8, numberOfBaseGrids=2, numberOfComponentGrids=4
numberOfRefinementLevels=2, numberOfMultigridLevels=2
```

Example 4.2: Miscellaneous **GenericGridCollection** member functions

5 Class GridCollection

Class **GridCollection** is the base class for all Overture classes that contain collections of **MappedGrid**s. It is derived from class **GenericGridCollection**. It overloads some of the **GenericGridCollection** public constants, member data and member functions described in §4. All of the member functions of class **GenericGridCollection** described there may be used; only note that the default constructor and member function **initialize**() take an additional optional argument specifying the number of dimensions of the grids initially in the collection, and the copy constructor and the member functions **reference**() and **operator=**() now take a **GridCollection** as their argument. Also, the overloaded member functions **addRefinement**() and **addMultigridCoarsening**() take additional arguments that describe how the refinements and coarsenings are constructed from the corresponding lower refinement-level or multigrid-level grids, as explained in §5.1. Class **GridCollection** has additional and member functions **boundingBox**, **numberOfDimensions**(), **changeToAllVertexCentered**() and **changeToAllCellCentered**() that correspond to the **MappedGrid** member functions of the same names. The member function **boundingBox** returns coordinate bounds of the smallest box containing the bounding boxes of all of the **MappedGrid**s in the collection; this bounding box is computed by **update**(). The member function **numberOfDimensions**() returns the same value as the corresponding member functions of all of the **MappedGrid**s in the collection (which must all agree). The member functions **changeToAllVertexCentered**() and **changeToAllCellCentered**() cause the corresponding member function to be called for each **MappedGrid** in the collection. Example 5 shows how some of these member functions may be used.

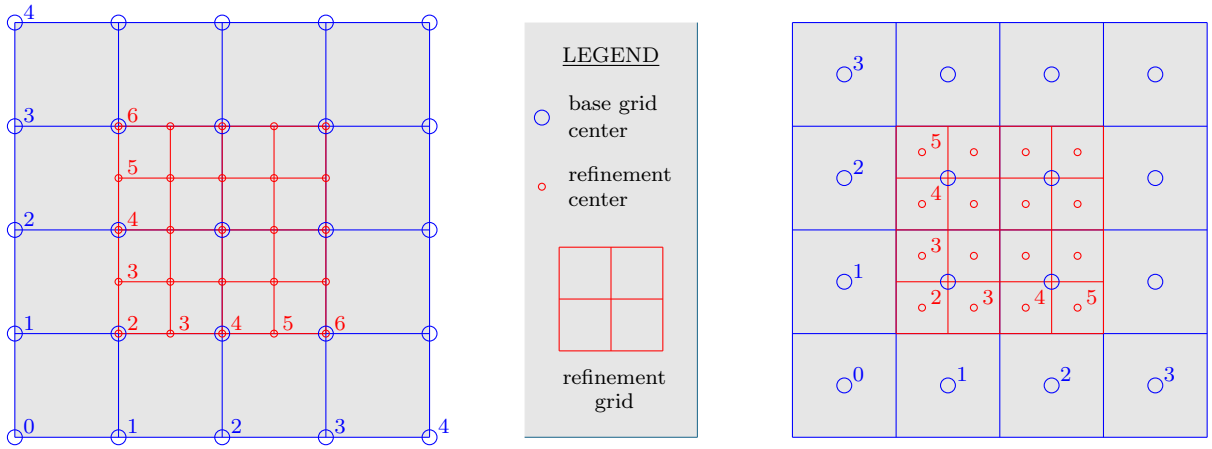
```
1   #include "Square.h"
2   #include "GridCollection.h"
3
4   int
5   main()
6   {
7     GridCollection g(2,1);                          // Start with one two-dimensional grid.
8     // GridCollection g; g.initialize(2,1);         // This would have the same effect.
9     int base = 0, ratio = 2, level = 1, refinement; // The base grid is grid zero.
10
11    SquareMapping square(0.,1.,0.,1.);         // A Mapping for the square [0,1]x[0,1].
12    g[base].reference(square);                 // Make the base grid use the square mapping.
13    g.update(GridCollection::THEboundingBox);  // Update and print out the bounding box
14
15    printf("boundingBox=[%f,%f]x[%f,%f]x[%f,%f]\n",
16           g.boundingBox(0,0),g.boundingBox(1,0),
17           g.boundingBox(0,1),g.boundingBox(1,1),
18           g.boundingBox(0,2),g.boundingBox(1,2));
19
20    g.changeToAllVertexCentered();                          // Make the base grid vertex-centered.
21    IntegerArray range = g[base].indexRange();                   // Find the index range of discretization points.
22    range(0,0) = range(0,1) = 1; range(1,0) = range(1,1) = 3;  // Refine the index range [1:3,1:3].
23    refinement = g.addRefinement(range, ratio, level, base);   // Add a level-one refinement grid.
24    g.deleteRefinement(refinement);                            // Delete the refinement grid.
25
26    g.changeToAllCellCentered();                   // Make the base grid cell-centered.
27    range = g[base].indexRange();                  // Find the index range of discretization points.
28    range(0,0) = range(0,1) = 1; range(1,0) = range(1,1) = 2; // Refine the index range [1:2,1:2].
29    refinement = g.addRefinement(range, ratio, level, base);  // Add a level-one refinement grid.
30    g.deleteRefinement(refinement);                            // Delete the refinement grid.
31
32    int component=0,coarsening;
33    coarsening = g.addMultigridCoarsening(ratio, level, component); // Add a level-one multigrid coarsening.
34    g.deleteMultigridCoarsening(coarsening);   // Delete the multigrid coarsening.
35
36    int grid;
37    for( grid=0; grid<g.numberOfGrids(); grid++ )
38    {
39      printf("refinementFactor=[%i,%i,%i]\n",               // Print the refinement factors
40             g.refinementFactor(axis1),g.refinementFactor(axis2),g.refinementFactor(axis3));
41      printf("multigridCoarseningFactor=[%i,%i,%i]\n",      // Print the multigrid coarsening factors
```

```
42              g.multigridCoarseningFactor(axis1),g.multigridCoarseningFactor(axis2),
43              g.multigridCoarseningFactor(axis3));
44        printf("refinementFactor=[%i,%i,%i]\n",                // Print the multigrid refinement factors
45              g.refinementFactor(axis1),g.refinementFactor(axis2),g.refinementFactor(axis3));
46
47      }
48      return 0;
49    }
```

```
----------------------- output ----------------------
boundingBox=[0.000000,1.000000]x[0.000000,1.000000]x[0.000000,0.000000]
refinementFactor=[1,1,1]
multigridCoarseningFactor=[1,1,1]
refinementFactor=[1,1,1]
```

Example 5 : Some **GridCollection** member functions

5.1 Local refinements and multigrid coarsenings



Vertex-centered grid

Cell-centered grid

Figure 5: Alignment of **MappedGrid** refinements for vertex-centered and for cell-centered grids. In this example, the refinement ratio in each direction is two. The range of indices of the base grid discretization points to be refined is indicated; for the vertex-centered case the range is [1:3,1:3], while for the cell-centered case the range is [1:2,1:2]. The base-grid and refinement-grid vertices with indices zero are aligned, even though these vertices may lie outside the base grid or the refinement grid.

Class **GridCollection** contains a collection of **MappedGrid**s. This is indexed using the overloaded member function **operator[]**(). It also contains **baseGrid**, **refinementLevel**, **componentGrid**, and **multigridLevel**, collections of **GridCollection**s that overload the corresponding member data of class **GenericGridCollection**. The discussion of base grids, refinements, component grids and multigrid coarsenings in §4 applies directly to class **GridCollection**. However, since each grid is a **MappedGrid**, we can say more about how the refinements and multigrid coarsenings are related to the base grids and component grids. In particular, each refinement must be generated by the mapping of its base grid, restricted to an aligned rectangular subset of the parameter space of the base grid. For each level-one refinement, this rectangular subset must be bounded exactly by cells of the base grid, and the cells of the refinement must refine the cells of the base grid by an integer refinement ratio in each direction, as shown in Figure 5. For each level-n refinement, this rectangular subset must be bounded exactly by cells of the union of level-$(n-1)$ refinements of the same base grid, and the cells of the level-n refinement must refine the cells of the level-$(n-1)$ refinements by an integer refinement ratio in each direction. While refinements of different base grids may use different refinement ratios, all same-level refinements of the same base grid must use the same refinement ratios. Each multigrid coarsenings of a component grid must be generated by the mapping of

its uncoarsened component grid (which is either a base grid or a refinement), and the cells of the level-n multigrid coarsening must coarsen the cells of the level-$(n-1)$ coarsening by an integer coarsening ratio in each direction. This places a restriction on the number of cells of the component grid, namely that in each direction it must be divisible by the product of the coarsening ratios of all of its multigrid coarsenings. The member function **refinementFactor** returns an the refinement factor of each grid in each direction relative to its base grid; this is the product of the refinement ratios of each refinement level up to that of the grid. For example, if level-one refinement grids are refined by a ratio of two and level-two refinement grids are refined by a ratio of three, then the level-two refinement grids have a refinement factor of $6 = 2 \times 3$. Similarly, the member function **multigridCoarseningFactor** returns the multigrid coarsening factor of each grid in each direction relative to its uncoarsened component grid; this is the product of the coarsening ratios of each multigrid level up to that of the grid. Example 5 shows how some of these member functions may be used. Figure 5 shows the base grid and refinement that are produced by the example, in cases where the base grid is vertex-centered or cell-centered.

6 Class CompositeGrid

Class **CompositeGrid** is the class used for composite overlapping grids; it is a collection of **MappedGrid**s with a description of how function values defined on the component grids are related through interpolation between component grids in their regions of overlap. Figure 6 shows a simple example of a **CompositeGrid** consisting of two overlapping component grids. **CompositeGrid** is derived from class **GridCollection**. It overloads some of the **GridCollection** public constants, member data and member functions described in §5. All of the member functions of class **GridCollection** described there may be used; only note that the copy constructor and the member functions **reference**() and **operator=**() now take a **CompositeGrid** as their argument. Also, **CompositeGrid** overloads the member data **multigridLevel**, which are now a collection of **CompositeGrid**s. Class **CompositeGrid** has member data that describe the criteria used in determining the overlap and interpolation of data between component grids, and the computed member data for interpolation at each interpolation point. It has member functions for use by grid-generation packages to aid in computing the overlap between component grids and determining which gridpoints should be interpolation points and from which component grid these should interpolate. Example 6 shows how some of these member data and member functions may be used.

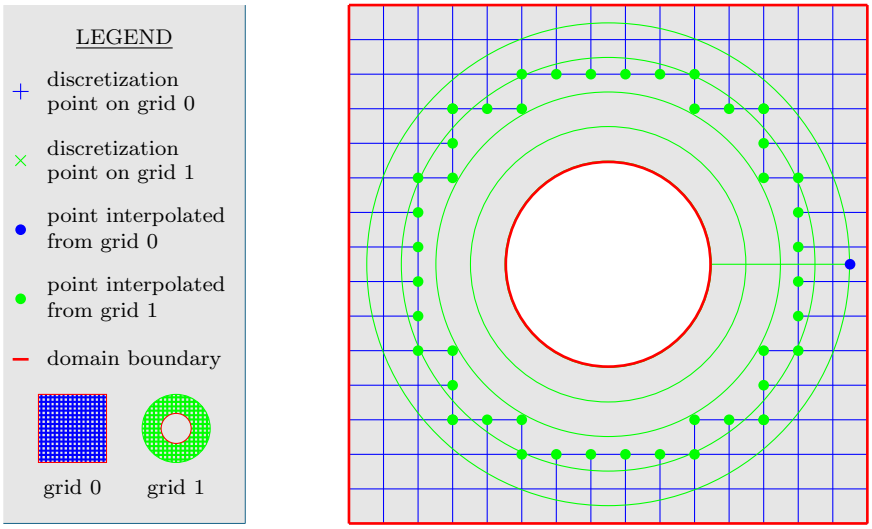


Figure 6: An overlapping **CompositeGrid**. Some gridpoints are discretization points, some are interpolation points, and some gridpoints of the square grid are unused.

```
#include "CompositeGrid.h"

void example( CompositeGrid & g)
{
```

```
  g.update(CompositeGrid::THEmask |                      // Update the interpolation data.
           CompositeGrid::THEinterpolationCoordinates | // This is usually done by a
           CompositeGrid::THEinterpoleeGrid |           // grid-generation package.
           CompositeGrid::THEinterpoleeLocation |
           CompositeGrid::THEinterpolationPoint |
           CompositeGrid::THEinterpolationCondition |
           CompositeGrid::THEinverseMap);

  for( Integer k=0; k<g.numberOfGrids(); k++)
  {
    Integer np = g.numberOfInterpolationPoints(k);    // Access the number of interpolation points,
    IntegerArray & ig = g.interpolationGrid[k],       // the interpolee grids,
                 & il = g.interpoleeLocation[k];      // the interpolee-stencil locations,
                 & ip = g.interpolationPoint[k];      // the interpolation points,
                 & ic = g.interpolationCoordinates[k];// the interpolation coordinates, and
    IntegerMappedGridFunction & mask = g. mask[k];    // the gridpoint mask.
    IntegerArray & dimension = g[k]. dimension,       // Access the outer dimensions of the grid.
    Range I1(dimension(0,0), dimension(1,0)),         // Construct  Ranges for gridpoints in the
          I2(dimension(0,1), dimension(1,1)),         // interior and on the boundary of grid k,
          I3(dimension(0,2), dimension(1,2));         // and ghost points outside the boundary.
    where (mask(I1,I2,I3) & CompositeGrid::ISdiscretizationPoint)
    {
       // ...           // These are discretization points.
    }
    elsewhere (mask(I1,I2,I3) & CompositeGrid::ISinterpolationPoint)
    {
       //  These are interpolation points.
    }
    elsewhere (mask(I1,I2,I3) & CompositeGrid::ISghostPoint)
    {
       // These are ghost points.
    }
    otherwise
    {
       // These are unused points.
    }
  }
}
```

Example 6: Some **CompositeGrid** member functions

The gridpoints of a **CompositeGrid** are classified as to whether they are discretization points, interpolation points, ghost points or unused points. Note that in order to make this distinction, we no longer use the terms "discretization-cell center" and "discretization point" synonymously with "gridpoint" as we did in §3.1. As before, a gridpoint is any point centered on the grid in the same way that the interior discretization stencil is centered. If the grid is vertex-centered then the gridpoints are the grid vertices, including any ghost points outside the grid boundaries; if the grid is cell-centered then the gridpoints are the grid-cell centers, including any ghost cells outside the grid boundaries. For the purposes of a **CompositeGrid**, "discretization-cell centers" and "discretization points" are those gridpoints in the interior or on the boundary of the grid which are labeled as discretization points in the **MappedGrid** member data **mask**, as shown in Example 6. "Interpolation points" are those interior or boundary gridpoints which are in the list of interpolation points for that grid and are labeled as interpolation points in **mask**. "Ghost points" are those gridpoints outside the grid boundaries for which the nearest interior or boundary gridpoint is either a discretization point or an interpolation point; ghost points are labeled as such in **mask**.

6.1 Interpolation criteria

To ensure stable and accurate interpolation of data between component grids, it is necessary to use criteria that impose some constraints on the overlap between the component grids. For example, if the interpolation points of two neighbouring component grids were perfectely aligned, then interpolation of data from each component grid to the other would consist of copying the data from the interpolation points of one component grid to those of the other and vice versa. If the interpolation points initially contained invalid data, then after interpolation they would still contain invalid data. The interpolation fails because there is insufficient overlap. With slightly more overlap, the interpolation stencils would include both interpolation points and discretization points. This is called implicit interpolation, and requires that the interpolation conditions be solved as a system of simultaneous linear equations. This system of equations becomes singular if any interpolation points of

neighbouring component grids are perfectly aligned. As the amount of overlap increases, the system of equations becomes well-conditioned. If the component grids overlap by more than half the width of the discretization stencil plus half the width of the interpolation stencil, then the interpolation stencil of each interpolation point is guaranteed to contain only discretization points of the other component grid and the system of equations becomes diagonal. This is called explicit interpolation, and allows the interpolation conditions to be applied independently at each interpolation point.

There are several **CompositeGrid** member data arrays and member functions that express these interpolation criteria. The array **interpolationWidth** indicates, for each pair of component grids at each multigrid level, the width of the interpolation stencil in each direction. The array **interpolationIsImplicit** indicates, for each pair of component grids at each multigrid level, whether or not the interpolation stencil for any interpolation point of the one component grid may contain interpolation points from the other component grid. If implicit interpolation is allowed, then less overlap is needed. The member functions **interpolationIsAllExplicit** and **interpolationIsAllImplicit** summarize this data, indicating whether interpolation from every pair of component grids at every multigrid level must be explicit, or whether interpolation between all pairs of component grids at all multigrid levels may be implicit. The array **interpolationOverlap** indicates, for each pair of component grids at each multigrid level, the minimum amount of overlap allowed in each direction. Occasionally it is useful to restrict interpolation so that interpolation is allowed only between certain pairs of grids. This constraint might be useful, for example, to avoid interpolation between grids that are known to have incompatible resolution. The array **mayInterpolate** indicates, for each pair of component grids at each multigrid level, whether interpolation is allowed between these component grids. In addition, it is possible to specify, for each component grid at each multigrid level, a sequence of preferences for how the gridpoints of that component grid should be used. This sequence should contain discretization as well as interpolation from any subset of the other component grids, in any sequence. For example, if discretization were specified to be the highest priority for all component grids, then the amount of overlap would be maximized, because only those points which could not be discretized would be considered for interpolation. The array **interpolationPreference** indicates, for each component grid at each multigrid level, the sequence of preferences for the gridpoints of that component grid. In order for gridpoints of grid k_1 to be allowed to interpolate from grid k_2, **mayInterpolate**(k_1,k_2) must be **LogicalTrue** and k_2 must appear in the **interpolationPreference** list for k_1. These and other member data arrays and member functions that describe constraints on interpolation are summarized in Table 1.

For diagnostic purposes, class **CompositeGrid** allows for a second set of interpolation criteria using less stringent constraints. These are useful when no configuration of gridpoints is possible that would satisfy the primary interpolation constraints. For example, given a set of component grids with insufficient overlap, it is impossible to interpolate data between the component grids in the areas of insufficient overlap. By allowing less stringent constraints to be used where the primary constraints fail, it is possible to identify these problem areas. The parameters that describe these constraints are **backupInterpolationIsImplicit**, **backupInterpolationWidth**, **backupInterpolationOverlap**, **backupInterpolationConditionLimit** and **mayBackupInterpolate**. The latter determines whether the backup interpolation criteria may be used.

6.2 Interpolation data

The interpolation data for a **CompositeGrid** consists of a collection of interpolation points, with data specifying from which component grids they interpolate, and their location within the parameter space of the components grid from which they interpolate. This data may be used to compute interpolation weights. The array **numberOfInterpolationPoints** contains the number of interpolation points of each component grid at each multigrid level. I should discuss the following member data and functions: enum ISgivenByInterpoleePoint; IntegerArray numberOfInterpolationPoints; IntegerArray numberOfInterpoleePoints; IntegerArray interpoleeGridRange; RVector<RealArray> interpolationCoordinates; RVector<IntegerArray> interpoleeGrid; RVector<IntegerArray> interpoleePoint; RVector<IntegerArray> interpoleeLocation; RVector<IntegerArray> interpolationPoint; RVector<RealArray> interpolationCondition;

6.3 Local refinements and multigrid coarsenings

I should discuss the following member data and functions: IntegerArray multigridCoarseningRatio; IntegerArray multigridProlongationWidth; IntegerArray multigridRestrictionWidth; RCVector<CompositeGrid> multigridLevel; virtual void makeCompleteMultigridLevels(); inline void adjustBoundary(const Integer& k1, const Integer& k2, const IntegerArray& i1, const RealArray& x);

6.4 Miscellaneous member data and functions

I should discuss the following member data and functions: RealCompositeGridFunction inverseCondition; RealCompositeGridFunction inverseCoordinates; IntegerCompositeGridFunction inverseGrid; inline void getInterpolationStencil(const Integer& k1, const Integer& k2, const RealArray& r, const IntegerArray& interpolationStencil, const LogicalArray& useBackupRules); inline void getInterpolationStencil(const MappedGrid& g, const Integer& k1, const Integer& k2, const RealArray& r, const IntegerArray& interpolationStencil, const LogicalArray& useBackupRules); inline Logical canInterpolate(const Integer& k1, const Integer& k2, const RealArray& r, const LogicalArray& ok, const LogicalArray& useBackupRules, const Logical checkForOneSided = LogicalFalse); inline Logical canInterpolate(const MappedGrid& g, CompositeMask& g_mask, const Integer& k1, const Integer& k2, const RealArray& r, const LogicalArray& ok, const LogicalArray& useBackupRules, const Logical

checkForOneSided = LogicalFalse); void isInteriorBoundaryPoint(const Integer& k1, const Integer& k2, const IntegerArray& i1, const RealArray& r2, const LogicalArray& ok); inline void adjustBoundary(const Integer& k1, const Integer& k2, const IntegerArray& i1, const RealArray& x);

A Class ReferenceCounting

I should discuss the following member functions: the default constructor, the copy constructor, the destructor, **operator=**(), **reference**(), **breakReference**(), **virtualConstructor**(), **incrementReferenceCount**(), **decrementReferenceCount**(), **getReferenceCount**(), **uncountedReferencesMayExist**(), **getClassName**(), **getGlobalID**(), and **consistencyCheck**(). One example of a virtual member function that is overloaded in derived classes is the function **getClassName**(), which returns the name of the most-derived class of the object.

Member data or member function	Description
epsilon()	The accuracy tolerance used in the computation of interpolation coordinates using the inverse mapping of each component grid.
interpolationIsImplicit(k_1, k_2, l) **backupInterpolationIsImplicit**(k_1, k_2, l)	**LogicalTrue** if interpolation of component grid k_1 from component grid k_2 at multigrid level l may be implicit (in other words, if the interpolation stencils of interpolation points of grid k_1 may include interpolation points of grid k_2); **LogicalFalse** otherwise.
interpolationIsAllExplicit()	**LogicalTrue** if **interpolationIsImplicit**(k_1, k_2, l) is **LogicalFalse** for all pairs (k_1, k_2) of grids at all multigrid levels l; **LogicalFalse** otherwise.
interpolationIsAllImplicit()	**LogicalTrue** if **interpolationIsImplicit**(k_1, k_2, l) is **LogicalTrue** for all pairs (k_1, k_2) of grids at all multigrid levels l; **LogicalFalse** otherwise.
interpolationWidth(i, k_1, k_2, l) **backupInterpolationWidth**(i, k_1, k_2, l)	The interpolation-stencil width in index direction i for interpolation of component grid k_1 from component grid k_2 at multigrid level l.
interpolationOverlap(i, k_1, k_2, l) **backupInterpolationOverlap**(i, k_1, k_2, l)	The minimum overlap in index direction i for interpolation of component grid k_1 from component grid k_2 at multigrid level l.
interpolationConditionLimit(k_1, k_2, l) **backupInterpolationConditionLimit**(k_1, k_2, l)	The maximum condition number allowed for interpolation of component grid k_1 from component grid k_2 at multigrid level l. The constraint on the condition number is ignored if this parameter is set to zero.
interpolationPreference(i, k, l)	The list of preferences for interpolation and discretization of component grid k at multigrid level l. For each grid k, the component grid numbers of interpolee grids (and for discretization, the component grid number k) are listed so that their order of preference increases with the index i. The end of the list is indicated by the value -1.
mayInterpolate(k_1, k_2, l) **mayBackupInterpolate**(k_1, k_2, l)	**LogicalTrue** if component grid k_1 may be interpolated from component grid k_2 at multigrid level l; **LogicalFalse** otherwise.
mayCutHoles(k_1, k_2)	**LogicalTrue** if the domain boundaries of component grid k_1 may cut holes in component grid k_2; **LogicalFalse** otherwise.
multigridCoarseningRatio(i, k, l)	The multigrid coarsening ratio in direction i of component grid k at multigrid levels $l-1$ and l.
multigridProlongationWidth(i, k, l)	The multigrid prolongation-stencil width in direction i for prolongation of component grid k from multigrid level l to multigrid level $l-1$.
multigridRestrictionWidth(i, k, l)	The multigrid restriction-stencil width in direction i for restriction of component grid k from multigrid level l to multigrid level $l+1$.
numberOfCompleteMultigridLevels()	The least number of multigrid levels of all component grids. All component grids are supposed to have at least this many multigrid levels. When interpolation is updated, it is updated only for the complete multigrid levels.

Table 1: Some **CompositeGrid** member data and member functions describing the criteria used in determining interpolation and overlap between component grids

Index