Finite Difference Operators and Boundary Conditions for Overture User Guide, Version 1.00

Bill 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-132232

Abstract: We describe some finite difference operators and boundary conditions for use with the Overture grid functions. Second and fourth order accurate approximations are available for general curvilinear grids. For rectangular periodic domains the pseudo-spectral approximations are also available.

Contents

1 Introduction

We describe some finite difference operators and boundary conditions for use with the Overture grid functions. The derivative operators allow one to take first and second order derivatives (∂_x, ∂_y, ∂_z, ∂_{xx}, ∂_{xy} etc.) with second order, fourth order or spectral accuracy. (Spectral accuracy is for rectangular periodic domains only). The derivative operators can also be used to generate the matrix (9 point stencil, for example) corresponding to a derivative operator. These "coefficient" operators can be used to generate a sparse matrix.

The boundary condition operators define a "library" of elementary boundary condition operations that can be used to implement application specific boundary conditions. Examples of elementary boundary conditions include Dirichlet, Neumann and mixed conditions, extrapolation, setting the normal component of a vector and so on. A solver can apply one or more elementary boundary conditions to the different sides of a grid.

The class `MappedGridOperators` defines operators for differentiating MappedGridFunction's and operators for applying boundary conditions to MappedGridFunction's.

The classes `GridCollectionOperators`, `CompositeGridOperators` and `MultigridComposite-GridOperators` use the operators in the class `MappedGridOperators` (or a class derived there-of) to define differential and boundary condition operators for `GridCollection`'s, `CompositeGrid`'s and `MultigridCompositeGrid`'s.

The `MappedGridOperators` class can be used to compute spatial derivatives of a `realMappedGridFunction` including all first and second order derivatives with respect to x, y and z.

This class can also be used to define boundary conditions and to evaluate the boundary conditions.

There may be one or more "flavour" of this class. One flavour will define derivatives in the "standard" finite difference manner using the "mapping method". Another flavour will define derivatives using a finite volume approach. Yet other flavours can be defined (by derivation from this class).

The grid function classes `realMappedGridFunction` and `realCompositeGridFunction` have member functions for differentiation and applying boundary conditions. A `MappedGridFunction` has a pointer to an object of the `MappedGridOperators` class. It uses this object to perform the differentiation or to apply boundary conditions. To use a different "flavour" of differentiation one must tell the grid function using the `setMappedGridOperators` member function. Similarly a `GridCollectionFunction` has a pointer to a `GridCollectionOperators` and a `CGF` has a pointer to a `CompositeGridOperators`.

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

- A++ Quick Reference Card : `A++P++/DOCS/Quick_Reference_Card.tex`
- A primer for Overture[9].
- Grid and grid function documentation[3].
- Finite difference operators and boundary conditions[2].
- Finite volume operators [1].
- Mapping class documentation [4].

- Show file documentation [7].
- Interactive plotting[8].
- Oges "Equation Solver" documentation [6].
- Interactive grid generation documentation [5].
- The other stuff documentation[10].
- The OverBlown Navier-Stokes flow solver [12][11].

1.1 Differentiation

A number of different approximations are provided for a variety of differential operators. Given a vector-valued grid-function u, one may evaluate the first derivatives:

$$\frac{\partial u}{\partial x}\ ,\ \frac{\partial u}{\partial y}\ ,\ \frac{\partial u}{\partial z}$$

second-derivatives:

$$\frac{\partial^2 u}{\partial x^2}\ ,\ \frac{\partial^2 u}{\partial y^2}\ ,\ \frac{\partial^2 u}{\partial z^2}\ ,\ \frac{\partial^2 u}{\partial x \partial y}\ ,\ \frac{\partial^2 u}{\partial x \partial z}\ ,\ \frac{\partial^2 u}{\partial y \partial z}$$

as well as other second-order operators:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$$

$$\nabla \cdot (s(\mathbf{x})\nabla) = \frac{\partial}{\partial x}(s(\mathbf{x})\frac{\partial u}{\partial x}) + \frac{\partial}{\partial y}(s(\mathbf{x})\frac{\partial u}{\partial y}) + \frac{\partial}{\partial z}(s(\mathbf{x})\frac{\partial u}{\partial z})$$

$$\frac{\partial}{\partial x}(s(\mathbf{x})\frac{\partial u}{\partial x})\ ,\ \frac{\partial}{\partial x}(s(\mathbf{x})\frac{\partial u}{\partial y})\ ,\ \frac{\partial}{\partial z}(s(\mathbf{x})\frac{\partial u}{\partial x})\ ,\ \frac{\partial}{\partial y}(s(\mathbf{x})\frac{\partial u}{\partial x})\ ,\ \ldots$$

There are second-order, fourth-order, sixth-order and eight-order accurate approximations. For many operators there are conservative (finite-volume) and non-conservative approximations.

There are a number of different ways to evaluate derivatives of a grid function.

- Use the member function found in the `MappedGridOperators` object.
- Use the member function found in the `realMappedGridFunction`
- Use the member function found in the `realCompositeGridFunction`
- Use the member function `getDerivatives` found in the `MappedGridOperators` class to evaluate a set of derivatives all at once. This is more efficient than the previous approaches.
- Use the function `derivative` to directly compute the derivative. This is more efficient than the previous approaches.

Currently the most natural way is not the most efficient because it involves extra computation and extra data movement. All of these approaches are illustrated in the examples that follow.

2 Class MappedGridOperators

2.1 Public member function and member data descriptions

2.1.1 Public enumerators

Here are the public enumerators:

derivativeTypes: This enumerator contains a list of all the derivatives that we know how to evaluate

```
enum derivativeTypes
{
  xDerivative,
  yDerivative,
  zDerivative,
  xxDerivative,
  xyDerivative,
  xzDerivative,
  yxDerivative,
  yyDerivative,
  yzDerivative,
  zxDerivative,
  zyDerivative,
  zzDerivative,
  laplacianOperator,
  r1Derivative,
  r2Derivative,
  r3Derivative,
  r1r1Derivative,
  r1r2Derivative,
  r1r3Derivative,
  r2r2Derivative,
  r2r3Derivative,
  r3r3Derivative,
  gradient,
  divergence,
  divergenceScalarGradient,
  scalarGradient,
  identityOperator,
  vorticityOperator,
  xDerivativeScalarXDerivative,
  xDerivativeScalarYDerivative,
  yDerivativeScalarYDerivative,
  yDerivativeScalarZDerivative,
  zDerivativeScalarZDerivative,
  divVectorScalarDerivative,
  numberOfDifferentDerivatives   // counts number of entries in this list
};
```

BCNames: This enum (which for technical reasons is in the BCTypes Class, NOT the MappedGridOperators) defines the different types of elementary boundary conditions that have been implemented:

```
enum BCNames
```

```
{
  dirichlet,
  neumann,
  extrapolate,
  normalComponent,
  mixed,
  generalMixedDerivative,
  normalDerivativeOfNormalComponent,
  normalDerivativeOfADotU,
  aDotU,
  aDotGradU,
  evenSymmetry,
  vectorSymmetry,
  TangentialComponent0,
  TangentialComponent1,
  normalDerivativeOfTangentialComponent0,
  normalDerivativeOfTangentialComponent1,
  numberOfDifferentBoundaryConditionTypes   // counts number of entries in this list
};
```

2.1.2 Constructors

MappedGridOperators()

MappedGridOperators(MappedGrid & mg)

Description: Construct a MappedGridOperators

mg (input): Associate this grid with the operators.

Author: WDH

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

MappedGridFunction
"derivative"(const realMappedGridFunction & u,
const Index & I0 =nullIndex ,
const Index & I1 =nullIndex ,
const Index & I2 =nullIndex ,
const Index & I3 =nullIndex ,
const Index & I4 =nullIndex ,
const Index & I5 =nullIndex ,
const Index & I6 =nullIndex ,
const Index & I7 =nullIndex
)

Description: "derivative" equals one of x, y, z, xx, xy, xz, yy, yz, zz, laplacian, grad, div.

u (input): Take the derivative of this grid function.

I0,I1,I3 (input): evaluate the derivatives at these points.

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

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

2.1.4 Derivative Coefficients

MappedGridFunction
"derivativeCoefficients"(const Index & I0 =nullIndex ,
const Index & I1 =nullIndex ,
const Index & I2 =nullIndex ,
const Index & I3 =nullIndex ,
const Index & I4 =nullIndex ,
const Index & I5 =nullIndex ,
const Index & I6 =nullIndex ,
const Index & I7 =nullIndex
)

Description: "derivativeCoefficients" equals one of xCoefficients, yCoefficients, zCoefficients, xxCoefficients, xyCoefficients, xzCoefficients, yyCoefficients, yzCoefficients, zzCoefficients, laplacianCoefficients, gradCoefficients, divCoefficients, identityCoefficients. Compute the coefficients of the specified derivative.

I0,I1,... (input): determine the coefficients at these points.

return Value: The derivative coefficients.

2.1.5 get

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

Description: Get from a database file

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

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

2.1.6 getFourierOperators

FourierOperators*
getFourierOperators(const bool abortIfNull =true) const

Description: Return a pointer to the Fourier operators used by this class to perform pseudo-spectral derivatives. **NOTE:** This pointer will not be assigned until the first derivative operation is applied.

abortIfNull (input) : by default this routine will abort if the pointer is null

2.1.7 put

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

Description: output onto a database file

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

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

2.1.8 setOrderOfAccuracy

void
setOrderOfAccuracy(const int & orderOfAccuracy0)

Description: set the order of accuracy

orderOfAccuracy0 (input): valid values are 2 or 4 or MappedGridOperators::spectral. Choosing spectral means that derivatives are computed with the pseudo-spectral method. This is only valid for rectangular periodic grids.

2.1.9 setStencilSize

void
setStencilSize(const int stencilSize0)

Description: Indicate the stencil size for functions returning coefficients

2.1.10 setTwilightZoneFlow

void
setTwilightZoneFlow(const int & twilightZoneFlow0)

Description: Indicate if twilight-zone forcing should be added to boundary conditions

twilightZoneFlow0 (input): if true then add the twilight-zone forcing (see also setTwilightZoneFlowFunction and the section on boundary conditions)

2.1.11 isRectangular

bool
isRectangular()

Description: Return true if the grid is rectangular

2.1.12 updateToMatchGrid

void
updateToMatchGrid(MappedGrid & mg)

Description: associate a new MappedGrid with this object

mg (input): use this MappedGrid.

Notes: perform computations here that only depend on the grid

2.1.13 updateToMatchGrid

void
updateToMatchUnstructuredGrid(MappedGrid & mg)

Description: associate a new unstructured MappedGrid with this object

mg (input): use this MappedGrid.

Notes: perform computations here that only depend on the grid

2.1.14 sizeOf

real
sizeOf(FILE *file = NULL) const

Description: Return size of this object

2.1.15 setTwilightZoneFlow

void
setTwilightZoneFlow(const int & twilightZoneFlow_)

Description: Indicate if twilight-zone forcing should be added to boundary conditions

twilightZoneFlow_ (input): if 1 then add the twilight-zone forcing to all boundary conditions except for extrapolation. If 2 then also add to extrapolation. (see also setTwilightZoneFlow-Function and the section on boundary conditions)

2.1.16 setTwilightZoneFlowFunction

void
setTwilightZoneFlowFunction(OGFunction & twilightZoneFlowFunction0)

Description: Supply a twilight-zone forcing to use for boundary conditions

twilightZoneFlowFunction0 (input): use this class for twilight-zone forcing (see also setTwilightZoneFlow and the section on boundary conditions)

2.1.17 useConservativeApproximations

void
useConservativeApproximations(bool trueOrFalse = TRUE)

Description: Indicate whether to use the *conservative* approximations to the operators `div`, `laplacian`, `divScalarGrad` and `scalarGrad` and correspoding boundary conditions

trueOrFalse (input): TRUE means use conservative approximations.

2.1.18 usingConservativeApproximations

bool
usingConservativeApproximations() const

Description: Return TRUE if we are using conservative approximations.

2.1.19 setAveragingType

void
setAveragingType(const AveragingType & type)

Description: Set the averaging type for certain operators such as `divScalarGrad`. The default is `arithmeticAverage`. The `harmonicAverage` is often used for problems with discontinuos coefficients. Recall that

$$\begin{aligned} \text{arithmetic average } &= \frac{a+b}{2} \\ \text{harmonic average } = \frac{2}{\frac{1}{a}+\frac{1}{b}} &= \frac{2ab}{a+b} \end{aligned}$$

type (input) : one of `arithmeticAverage` or `harmonicAverage`.

2.1.20 getAveragingType

AveragingType
getAveragingType() const

Description: Return the current averaging type.

2.1.21 isRectangular

bool
isRectangular()

Description: Return true if the grid is rectangular

2.1.22 finishBoundaryConditions

void
finishBoundaryConditionsOld(realMappedGridFunction & u,
const BoundaryConditionParameters & bcParameters =
Overture::defaultBoundaryConditionParameters(),
const Range & C0 =nullRange)

Description: Call this routine when all boundary conditions have been applied. This function will update periodic edges and fix up the solution values in the ghost points outside corners which are not assigned by `applyBoundaryCondition` (i.e. the ghost points that lie outside the corners in 2D or the ghost points that lie outside the edges and the vertices in 3D). This routine wil also fill in extrapolation equations at ghost points that correspond to interpolation points on physical boundaries.

More precisely,

1. First call `u.periodicUpdate()` to assign values to `side=1` boundary lines

   $$\mathtt{i_{axis}} = \mathtt{mg.gridIndexRange()(1,axis)} \quad \mathtt{axis} = 0,1,..,\mathtt{mg.numberOfDimensions}$$

   (`mg` is the `MappedGrid` associated with the grid function `u`) as well as all ghost lines on all sides that have periodic boundary conditions.
2. Extrapolate corner ghost points which are not assigned by step 1 using extrapolation to order bcParameters.orderOfExtrapolation (orderOfAccuray+1)
   - In 2D extrapolate the corner ghost points along the diagonal. For example, if

     $$\mathtt{bcParameters.orderOfExtrapolation} = 3 \quad (\mathtt{defaultfor2ndorderaccuracy})$$

     then the value at the lower left corner ghost point

     $$(i_1, i_2) = (\mathtt{mg.indexRange()(Start,axis1)} - 1, \mathtt{mg.indexRange()(Start,axis2)} - 1)$$

     will be given by

     $$u(i_1, i_2) = 3u(i_1 + 1, i_2 + 1) - 3u(i_1 + 2, i_2 + 2) + u(i_1 + 3, i_2 + 3)$$

     If there are two ghost lines then also assign points (i_1-1, i_2),(i_1, i_2-1),(i_1-1, i_2-1). And so on, if there are more than 2 ghost lines.
   - In 3D extrapolate the ghost points next to edges and the ghost points next to vertices. Obtain values by extrapolating into the interior as much as possible.
3. extrapolate ghost points that lies outside of interpolation points on the physical boundary, mg.boundaryCondition(side,aixs)¿0.

For even more details you can look at the code in `Overture/GridFunction/GenericMappedGridOperators.C`

Note: When applied to a coefficient matrix the above operations will generate new equations in the coefficient matrix rather than be applied directly to the grid function.

u (input/output): Grid function to which boundary conditions were applied.

bcParameters (input): Supply parameters such as bcParameters.orderOfExtrapolation which indicates the order of extrapolation to use.

C0 (input) : apply to these components

2.1.23 divScalarGrad

realMappedGridFunction
divScalarGrad(const realMappedGridFunction & u,
const realMappedGridFunction & scalar,
const Index & I1_,
const Index & I2_,
const Index & I3_,
const Index & I4_,
const Index & I5_,
const Index & I6_,
const Index & I7_,
const Index & I8_)

Description: Evaluate the derivative $\nabla \cdot (\mathrm{scalar} \nabla \mathrm{u})$.

u (input):

scalar (input) : The coefficient appearing in the derivative expression.

Author: WDH

2.1.24 scalarGrad

```
realMappedGridFunction
scalarGrad( const realMappedGridFunction & u,
            const realMappedGridFunction & scalar,
            const Index & I1_,
            const Index & I2_,
            const Index & I3_,
            const Index & I4_,
            const Index & I5_,
            const Index & I6_,
            const Index & I7_,
            const Index & I8_)
```

Description: Evaluate the derivative $\mathrm{scalar} \nabla \mathrm{u}$.

u (input):

scalar (input) : The coefficient appearing in the derivative expression.

Author: WDH

2.1.25 derivativeScalarDerivative

```
realMappedGridFunction
derivativeScalarDerivative( const realMappedGridFunction & u,
                            const realMappedGridFunction & scalar,
                            const int & direction1,
                            const int & direction2,
                            const Index & I1_,
                            const Index & I2_,
                            const Index & I3_,
                            const Index & I4_,
                            const Index & I5_,
                            const Index & I6_,
                            const Index & I7_,
                            const Index & I8_)
```

Description: Evaluate the derivative

$$\frac{\partial}{\partial x_{\mathrm{direction1}}}(\mathrm{scalar} \frac{\partial}{\partial \mathrm{x}_{\mathrm{direction2}}} \mathrm{u})$$

u (input):

scalar (input) : The coefficient appearing in the derivative expression.

direction1,direction2 (input) : specify the derivatives to use.

2.1.26 divVectorScalar

realMappedGridFunction
divVectorScalar(const realMappedGridFunction & u,
const realMappedGridFunction & s,
const Index & I1,
const Index & I2,
const Index & I3,
const Index & I4,
const Index & I5,
const Index & I6,
const Index & I7,
const Index & I8)

Description: Evaluate the divergence of a known vector times the dependent variable u:

$$\nabla \cdot (\mathbf{S}u)$$

u (input):

s (input) : The coefficient appearing in the derivative expression, number of components equal to the number of space dimensions.

2.1.27 setNumberOfDerivativesToEvaluate

void
setNumberOfDerivativesToEvaluate(const int & numberOfDerivatives)

Description: Specify how many derivatives are to be evaluated

numberOfDerivatives (input): Indicate how many derivatives that you want to evaluate in the call to `getDerivatives`.

Author: WDH

2.1.28 setDerivativeType

void
setDerivativeType(const int & index, const derivativeTypes & derivativeType0, RealDistributedArray & ux1x2)

Description: Specify which derivative to evaluate and provide an array to save the results in.

index (input): Specify this derivative. $0 \le index <$ `numberOfDerivatives` where `numberOfDerivatives` was specified with `setNumberOfDerivativesToEvaluate`.

derivativeType0 (input): indicates which derivative to evaluate, from the enum `derivativeTypes`.

ux1x2 (input): Here is the array that the function `getDerivatives` will save the derivative in. This class keeps a reference to the array `ux1x1`. This array will be automatically be made large enough to hold the result.

Author: WDH

2.1.29 getDerivatives

void
getDerivatives(const realMappedGridFunction & u,
const Index & I1_ =nullIndex,
const Index & I2_ =nullIndex,
const Index & I3_ =nullIndex,
const Index & N =nullIndex,
const Index & Evaluate =nullIndex)

Description: This is an efficient way to compute derivatives. Compute the derivatives of u that were specified with `setNumberOfDerivativesToEvaluate` and `setDerivativeType`.

u (input): Compute the derivatives of this grid function.

I1,I2,I3 (input): evaluate the derivatives at these coordinate Index values (by default evaluate at as many points as is possible; for second-order discretization all points but the last ghost line are evaluated, for fourth order all points but the 2 last ghostlines are evaluated).

N (input): Evaluate the derivatives of these components of u (by default all components are evaluated).

Evaluate (input): evaluate this subset of the derivatives. The derivatives to be evaluated are numbered from 0,1,2,... For example, suppose you used setDerivativeType to specify:

```
setDerivativeType(0,MappedGridOperators::xDerivative,ux);
setDerivativeType(1,MappedGridOperators::yDerivative,uy);
setDerivativeType(2,MappedGridOperators::xxDerivative,uxx);
setDerivativeType(3,MappedGridOperators::yyDerivative,uyy);
```

If you only want to evaluate the second derivatives you can choose `Evaluate=Index(2,2)` to only evaluate derivatives 2 and 3.

Notes: This is an efficient way to compute many derivatives because computations can be shared This routine first computes u.r, u.s, [u.t] for efficiency

WARNING on each call to getDerivatives, the arrays used to hold the results will be redimensioned if the new results do not fit into the existing array (not just the size but the (base,bound) values for each dimension). Thus if you call `getDerivatives` in consecutive statements with different values for N and Evaluate, then the results from the first call may be destroyed if the arrays were not big enough. You can either explicitly dimension the arrays to be large enough or else initially call getDerivatives with the default values for N and Evaluate so the arrays are dimensioned to be full size.

2.1.30 divScalarGradCoefficients

```
realMappedGridFunction
divScalarGradCoefficients(const realMappedGridFunction & scalar,
                          const Index & I1_ = nullIndex,
                          const Index & I2_ = nullIndex,
                          const Index & I3_ = nullIndex,
                          const Index & I4_ = nullIndex,
                          const Index & I5_ = nullIndex,
                          const Index & I6_ = nullIndex,
                          const Index & I7_ = nullIndex,
                          const Index & I8_ = nullIndex)
```

Description: Form the coefficient matrix for the operator $\nabla \cdot (\text{scalar}\nabla)$.

scalar (input) : coefficient that appears in the operator.

Author: WDH

2.1.31 derivativeScalarDerivativeCoefficients

```
realMappedGridFunction
derivativeScalarDerivativeCoefficients(const realMappedGridFunction & scalar,
                                       const int & direction1,
                                       const int & direction2,
                                       const Index & I1 = nullIndex,
                                       const Index & I2 = nullIndex,
                                       const Index & I3 = nullIndex,
                                       const Index & I4 = nullIndex,
                                       const Index & I5 = nullIndex,
                                       const Index & I6 = nullIndex,
                                       const Index & I7 = nullIndex,
                                       const Index & I8 = nullIndex)
```

Description: Form the coefficient matrix for the operator

$$\frac{\partial}{\partial x_{\text{direction1}}}(\text{scalar}\frac{\partial}{\partial \text{x}_{\text{direction2}}}\text{u})$$

scalar (input) : coefficient that appears in the operator.

direction1,direction2 (input) : specify the derivatives to use.

2.1.32 scalarGradCoefficients

```
realMappedGridFunction
scalarGradCoefficients(const realMappedGridFunction & scalar,
                       const Index & I1_ = nullIndex,
                       const Index & I2_ = nullIndex,
                       const Index & I3_ = nullIndex,
```

const Index & I4_ = nullIndex,
const Index & I5_ = nullIndex,
const Index & I6_ = nullIndex,
const Index & I7_ = nullIndex,
const Index & I8_ = nullIndex)

Description: Form the coefficient matrix for the operator scalar∇.

scalar (input) : coefficient that appears in the operator.

Author: WDH

2.1.33 divVectorScalarCoefficients

realMappedGridFunction
divVectorScalarCoefficients(const realMappedGridFunction & s,
const Index & I1_ = nullIndex,
const Index & I2_ = nullIndex,
const Index & I3_ = nullIndex,
const Index & I4_ = nullIndex,
const Index & I5_ = nullIndex,
const Index & I6_ = nullIndex,
const Index & I7_ = nullIndex,
const Index & I8_ = nullIndex)

Description: Form the coefficient matrix for the operator $\nabla \cdot (\mathbf{S})$.

s (input) : The coefficient appearing in the derivative expression, number of components equal to the number of space dimensions.

2.1.34 applyBoundaryCondition

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

void
applyBoundaryCondition(realMappedGridFunction & u,
const Index & Components,
const BCTypes::BCNames & bcType,
const int & bc,

const RealArray & forcing,
const real & time =0.,
const BoundaryConditionParameters &
bcParameters = Overture::defaultBoundaryConditionParameters(),
const int & grid =0)

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

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

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

Description: Apply a boundary condition to a grid function. This routine implements every boundary condition known to man (ha!)

u (input/output): apply boundary conditions to this grid function.

Components (input): apply to these components

bcType (input): the name of the boundary condition to apply (dirichlet, neumann,...)

bc (input): apply the boundary condition on all sides of the grid where the boundaryCondition array (in the MappedGrid) is equal to this value. By default `bc=BCTypes allBoundaries` apply to all boundaries (with a positive value for boundaryCondition). To apply a boundary condition to a specified side use

- `bc=BCTypes::boundary1` for $(side, axis) = (0, 0)$
- `bc=BCTypes::boundary2` for $(side, axis) = (1, 0)$
- `bc=BCTypes::boundary3` for $(side, axis) = (0, 1)$
- `bc=BCTypes::boundary4` for $(side, axis) = (1, 1)$

- `bc=BCTypes::boundary5` for $(side, axis) = (0, 2)$
- `bc=BCTypes::boundary6` for $(side, axis) = (1, 2)$

or use `bc=BCTypes::boundary1+side+2*axis` for given values of $(side, axis)$ (this could be used in a loop, for example).

forcing (input): This value is used as a forcing for the boundary condition, if needed.

time (input): apply boundary conditions at this time (used by twilightZoneFlow)

bcParameters (input): optional parameters are passed using this object. See the examples for how to pass parameters with this argument.

Limitations: only second order accurate.

2.1.35 applyBoundaryConditionCoefficients

void
applyBoundaryConditionCoefficients(realMappedGridFunction & uCoeff,
const Index & E,
const Index & C,
const BCTypes::BCNames &
bcType = BCTypes::dirichlet,
const int & bc = allBoundaries,
const BoundaryConditionParameters &
bcParams = Overture::defaultBoundaryConditionParameters(),
const int & grid =0)

Description: Fill in the coefficients of the boundary conditions.

uCoeff (input/output): grid function to hold the coefficients of the BC.

E (input): apply to these equations (for a system of equations)

C (input): apply to these components

t (input): apply boundary conditions at this time.

Notes: If you supply Range objects for `E` and `C` then the boundary conditions are filled in for all equations and components indicated by the Ranges and NOT just the "diagonal" entries (as might be first expected). Thus normally you will want to specify `E` and `C` to just be int's.

Limitations: too many to write down.

2.2 Example 1: Differentiation of a `realMappedGridFunction`

In this first example we show to evaluate derivatives of a MappedGridFunctionin a few different ways. The recommended efficient method of evaluation is demonstrated near the end of the example code. (file `Overture/examples/tmgo.C`)

```
#include "Overture.h"
#include "MappedGridOperators.h"
#include "Square.h"
//================================================================================
//  Examples showing how to differentiate realMappedGridFunctions
//     o evaluate using the x,y,... member functions
//     o evaluate in an effficient manner by computing many derivatives at once.
//================================================================================
int
main(int argc, char *argv[])
{
  Overture::start(argc,argv);  // initialize Overture

  SquareMapping square(0.,1.,0.,1.);                   // Make a mapping, unit square
  square.setGridDimensions(axis1,11);                  // axis1==0, set no. of grid points
  square.setGridDimensions(axis2,11);                  // axis2==1, set no. of grid points
  MappedGrid mg(square);                               // MappedGrid for a square
  mg.update();                                         // create default variables

  Index I1,I2,I3;
  Range all;                                             // null Range
  realMappedGridFunction u(mg,all,all,all,Range(0,0)),   // define some component grid functions,
                         v(mg,all,all,all,Range(0,0)),   // in 3D
                         w(mg,all,all,all,Range(0,1));

  MappedGridOperators operators(mg);                     // define some differential operators
  u.setOperators(operators);                             // Tell u which operators to use
  v.setOperators(operators);
  w.setOperators(operators);                             // Tell u which operators to use

  getIndex(mg.dimension(),I1,I2,I3);                                                 // assign I1,I2,I3
  u(I1,I2,I3)=sin(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2));   // u=sin(x)*cos(y)
  w(I1,I2,I3,0)=sin(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2));   // first component
  w(I1,I2,I3,1)=sin(mg.vertex()(I1,I2,I3,axis1))*sin(mg.vertex()(I1,I2,I3,axis2));   // second component

  u.display("here is u");

  // compute the derivatives at interior and boundary points (there is 1 ghost line by default)
  getIndex(mg.indexRange(),I1,I2,I3);                                   // assign I1,I2,I3

  operators.x(u).display("Here is operators.x(u)");                // one way to compute u.x
  u.x().display("Here is u.x");                                    // another way to compute u.x

  v=u;
  v.x().display("v=u; here is v.x");


  real error = max(fabs(u.x()(I1,I2,I3)- cos(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2
  cout << "Maximum error (2nd order) = " << error << endl;

  // here we compute the derivatives of only some components of w
  v=w.x(all,all,all,0)+w.y(all,all,all,1);
```

```
v.display("here is w.x(0)+w.y(1)");


// now compute to 4th order
operators.setOrderOfAccuracy(4);
// 4th order has a 5 point stencil -- therefore on compute on interior points
getIndex(mg.indexRange(),I1,I2,I3,-1);

// compute the derivatives at interior and boundary points (there is 1 ghost line by default)

error = max(fabs(u.x()(I1,I2,I3)-cos(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2))));
cout << "Maximum error (4th order) = " << error << endl;

// --- Here is a more complicated expression:
v(I1,I2,I3)=u(I1,I2,I3)*u.x()(I1,I2,I3)+v(I1,I2,I3)*u.y()(I1,I2,I3)-.1*(u.xx()(I1,I2,I3)+u.yy()(I1,I2,

// --- make a list of derivatives to evaluate all at once (this is more efficient) ---
RealArray ux,uy;                              // these arrays will hold the answers
operators.setNumberOfDerivativesToEvaluate( 2 );
operators.setDerivativeType( 0, MappedGridOperators::xDerivative, ux );
operators.setDerivativeType( 1, MappedGridOperators::yDerivative, uy );

// reset order of accuracy to 2
u.getOperators()->setOrderOfAccuracy(2);  // This is the same as operators.setOrderOfAccuracy(2);

// compute the x and y derivatives of u and save in the arrays ux and uy
operators.getDerivatives(u,I1,I2,I3);
// this next line is another way to do exactly the same thing
u.getDerivatives(I1,I2,I3);

error = max(fabs(ux(I1,I2,I3)-cos(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2))));
cout << "Maximum error in ux: (2nd order) = " << error << endl;
error = max(fabs(uy(I1,I2,I3)+sin(mg.vertex()(I1,I2,I3,axis1))*sin(mg.vertex()(I1,I2,I3,axis2))));
cout << "Maximum error in uy: (2nd order) = " << error << endl;


// compute the y derivative only
ux=-123.;  // init with bogus values
uy=-123.;
u.getDerivatives(I1,I2,I3,all,1);    // all=all components, 1=derivative number 1 (yDerivative)

error = max(fabs(ux(I1,I2,I3)-cos(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2))));
cout << "Maximum error in ux: (2nd order) (should be bad, only uy computed)= " << error << endl;
error = max(fabs(uy(I1,I2,I3)+sin(mg.vertex()(I1,I2,I3,axis1))*sin(mg.vertex()(I1,I2,I3,axis2))));
cout << "Maximum error in uy: (2nd order) = " << error << endl;

// ***** now compute derivatives of a grid function with multiple components

getIndex(mg.indexRange(),I1,I2,I3);                                 // assign I1,I2,I3
w.getDerivatives(I1,I2,I3);

ux=-123.;  // init with bogus values
uy=-123.;
w.getDerivatives(I1,I2,I3,0,1);    // 0=component, 1=yDerivative
w.getDerivatives(I1,I2,I3,1,0);    // 1=component, 0=xDerivative

ux.display("ux for w");
```

```
    uy.display("uy for w");

    error = max(fabs(uy(I1,I2,I3,0)+sin(mg.vertex()(I1,I2,I3,axis1))*sin(mg.vertex()(I1,I2,I3,axis2))));
    cout << "Maximum error in w(0).y: (2nd order) = " << error << endl;
    error = max(fabs(ux(I1,I2,I3,1)-cos(mg.vertex()(I1,I2,I3,axis1))*sin(mg.vertex()(I1,I2,I3,axis2))));
    cout << "Maximum error in w(1).x: (2nd order) = " << error << endl;



    cout << "Program Terminated Normally! \n";
    Overture::finish();
    return 0;
  }
```

In this example we create a `MappedGridOperators` object and associate it with a grid function. We compute the x-derivative of a `realMappedGridFunction`. The member function "x" in the grid function returns the x derivative of the grid function as a new grid function. It uses the derivative defined in the `MappedGridOperators` object. Note that by default the derivative of a `realMappedGridFunction` is only computed at interior and boundary points (indexRange). Thus to access (make a view) of the derivative values of the grid function `u.x()` at the Index's `(I1,I2,I3)` it is necessary to say `u.x()(I1,I2,I3)`. On the other hand the statement `u.x(I1,I2,I3)` will evaluate the derivatives on the points defined by `(I1,I2,I3)`, but will return a grid function that is dimensioned for the entire grid. Thus in general on could say `u.x(I1,I2,I3)(J1,J2,J3)` to evaluate the derivatives at points `(I1,I2,I3)` but to use (take a view) of the grid function at the Index's `(J1,J2,J3)`.

The example code also shows how to compute the derivatives of just some components of a grid function. The grid function `w` has 2 components. The expression `w.x(all,all,all,0)` computes the derivative of component '0' of `w` and returns the result as a grid function with 1 component.

The efficient method for computing derivatives is shown at the bottom of this example. First one must indicate how many derivatives will be evaluated, `setNumberOfDerivativesToEvaluate`, and which derivatives should be evaluated, `setDerivativeType`, and also supply A++ arrays to hold the results in (`ux,uy`). These arrays will automatically be made large enough to hold the results if they are not already large enough. The call to `getDerivatives` will evaluate all the derivatives all at once (thus saving computations) and place the results in the user supplied arrays (thus saving memory allocation overhead).

See also section (3.2) for a similar example that uses `CompositeGridFunction`'s.

2.3 Derivatives Defined Using Finite Differences

The class `MappedGridOperators` defines derivatives using finite differences and the "mapping method". Simply put, each derivative is written, using the chain use, in terms of derivatives on the unit square (or cube). The derivatives on the unit square are discretized using standard central finite differences.

Each `MappedGrid`, **M**, consists of a set of grid points,

$$\mathbf{M} = \{\texttt{vertex}_i \mid i = (i_1, i_2, i_3) \quad \texttt{dimension(Start,m)} \le i_m \le \texttt{dimension(End,m)}\ ,\ m = 0, 1, 2\}\ .$$

One or two extra lines of fictitious points are added for convenience in discretizing to second or fourth-order. Boundaries of the computational domain will coincide with the boundaries of the unit cubes, $i_m = \texttt{gridIndexRange(Start,m)}$ or $i_m = \texttt{gridIndexRange(End,m)}$.

The derivatives are discretized with second or fourth-order accurate central differences applied to the equations written in the unit cube coordinates, as will now be outlined. Define the shift operator in the coordinate direction m by

$$E_{+m}\mathbf{U}_i = \begin{cases} \mathbf{U}_{i_1+1,i_2,i_3} & \text{if } m=0 \\ \mathbf{U}_{i_1,i_2+1,i_3} & \text{if } m=1 \\ \mathbf{U}_{i_1,i_2,i_3+1} & \text{if } m=2 \end{cases} , \tag{1}$$

and the difference operators

$$\begin{aligned} D_{+r_m} &= \frac{E-1}{\Delta r_m} \\ D_{-r_m} &= \frac{1-E^{-1}}{\Delta r_m} \\ D_{0r_m} &= \frac{E-E^{-1}}{2\Delta r_m} \\ D_{+m} &= E_{+m}-1 \\ D_{+m_1,m_2} &= \frac{E_{+m_1}E_{+m_2}-1}{\Delta r_m} . \end{aligned}$$

Let D_{2r_m}, $D_{2r_m r_n}$, D_{2x_m} and $D_{2x_m x_n}$ denote second-order accurate derivatives with respect to $\mathbf{r}$ and $\mathbf{x}$. The derivatives with respect to $\mathbf{r}$ are the standard centred difference approximations. For example

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial r_m} \approx D_{2r_m}\mathbf{U}_i &:= \frac{(E_{+m}-E_{+m}^{-1})\mathbf{U}_i}{2(\Delta r_m)} \\ \frac{\partial^2 \mathbf{u}}{\partial r_m^2} \approx D_{2r_m r_m}\mathbf{U}_i &:= \frac{(E_{+m}-2+E_{+m}^{-1})\mathbf{U}_i}{(\Delta r_m)^2} \end{aligned}$$

Let D_{4r_m}, $D_{4r_m r_n}$, D_{4x_m} and $D_{4x_m x_n}$ denote fourth order accurate derivatives with respect to $\mathbf{r}$ and $\mathbf{x}$. The derivatives with respect to $\mathbf{r}$ are the standard fourth-order centred difference approximations. For example

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial r_m} \approx D_{4r_m}\mathbf{U}_i &:= \frac{(-E_{+m}^2+8E_{+m}-8E_{+m}^{-1}+E_{+m}^{-2})\mathbf{U}_i}{12(\Delta r_m)} \\ \frac{\partial^2 \mathbf{u}}{\partial r_m^2} \approx D_{4r_m r_m}\mathbf{U}_i &:= \frac{(-E_{+m}^2+16E_{+m}-30+16E_{+m}^{-1}-E_{+m}^{-2})\mathbf{U}_i}{24(\Delta r_m)^2} \end{aligned}$$

where $\Delta r_m = 1/(n_{m,b}-n_{m,a})$.

The derivatives with respect to $\mathbf{x}$ are defined by the chain rule. For the fourth-order approximations, for example,

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial x_m} &= \sum_n \frac{\partial r_n}{\partial x_m}\frac{\partial \mathbf{u}}{\partial r_n} \quad \approx D_{4x_m}\mathbf{U}_i := \sum_n \frac{\partial r_n}{\partial x_m} D_{4r_n}\mathbf{U}_i \\ \frac{\partial^2 \mathbf{u}}{\partial x_m^2} &= \sum_{n,l} \frac{\partial r_n}{\partial x_m}\frac{\partial r_l}{\partial x_m}\frac{\partial^2 \mathbf{u}}{\partial r_n r_l} + \sum_n \frac{\partial^2 r_n}{\partial x_m^2}\frac{\partial \mathbf{u}}{\partial r_n} \\ &\approx D_{4x_m x_m}\mathbf{U}_i := \sum_{n,l} \frac{\partial r_n}{\partial x_m}\frac{\partial r_l}{\partial x_m} D_{4r_n r_l}\mathbf{U}_i + \sum_n \Big(D_{4x_m}\frac{\partial r_n}{\partial x_m}\Big) D_{4r_n}\mathbf{U}_i \end{aligned}$$

The entries in the Jacobian matrix, $\partial r_m/\partial x_n$, are assumed to be known at the vertices of the grid; these values are obtained from the `MappedGrid` in the array `inverseVertexDerivatives`.

2.4 Conservative Difference Approximations

The `MappedGridOperators` also supply some conservative difference approximations. *** this is new ***

Define J to be the determinant of the Jacobian matrix of the transformation derivatives

$$J = \det \left[\frac{\partial \mathbf{x}}{\partial \mathbf{r}} \right]$$

Note that $J d\mathbf{r}$ is a measure of the local volume element.

The divergence operator is

$$\begin{aligned} \nabla_{\mathbf{x}} \cdot \mathbf{u} &= \sum_i \frac{\partial u_i}{\partial x_i} \\ &= \sum_j \sum_i \frac{\partial r_j}{\partial x_i} \frac{\partial u_i}{\partial r_j} \end{aligned}$$

The divergence operator can be written in **conservation form** for the computational variables $\mathbf{r}$

$$\begin{aligned} \nabla_{\mathbf{x}} \cdot \mathbf{u} &= \frac{1}{J} \sum_j \frac{\partial}{\partial r_j} \left[\sum_i J \frac{\partial r_j}{\partial x_i} u_i \right] \\ &= \frac{1}{J} \nabla_{\mathbf{r}} \cdot \mathbf{U} \\ \text{where } U_i &= J \sum_k \frac{\partial r_i}{\partial x_k} u_k \end{aligned}$$

This is called conservation form for the variables $\mathbf{r}$ since integrals over $d\mathbf{r}$ space can be expressed in a simple form from which the divergence theorem can be applied:

$$\begin{aligned} \int \nabla_{\mathbf{x}} \cdot \mathbf{u} \, d\mathbf{x} &= \int \nabla_{\mathbf{x}} \cdot \mathbf{u} \, J d\mathbf{r} \\ &= \int \nabla_{\mathbf{r}} \cdot \mathbf{U} \, d\mathbf{r} \end{aligned}$$

The laplacian operator in divergence form now follows easily,

$$\begin{aligned} \Delta \phi &= \frac{1}{J} \sum_j \frac{\partial}{\partial r_j} \left[\sum_i J \frac{\partial r_j}{\partial x_i} \frac{\partial \phi}{\partial x_i} \right] \\ &= \frac{1}{J} \sum_j \frac{\partial}{\partial r_j} \left[\sum_i J \frac{\partial r_j}{\partial x_i} \sum_k \frac{\partial r_k}{\partial x_i} \frac{\partial \phi}{\partial r_k} \right] \end{aligned}$$

The **conservative difference approximations** to the divergence and laplacian are obtained by discretizing the above expressions.

Similarly the operator $\nabla \cdot (a(\mathbf{x}) \nabla \phi)$ is

$$\nabla \cdot (a \nabla \phi) = \frac{1}{J} \sum_j \frac{\partial}{\partial r_j} \left[\sum_i J \frac{\partial r_j}{\partial x_i} a \sum_k \frac{\partial r_k}{\partial x_i} \frac{\partial \phi}{\partial r_k} \right]$$

A general second-order derivative, $\partial_{x_m}(a\partial_{x_n}\phi)$, can be written from the expression for the divergence of a vector whose m^{th} component is $a\partial_{x_n}\phi$ (and other components zero),

$$\begin{aligned}\frac{\partial}{\partial x_m}\left[a\frac{\partial\phi}{\partial x_n}\right] &= \frac{1}{J}\sum_j \frac{\partial}{\partial r_j}\left[J\frac{\partial r_j}{\partial x_m}a\frac{\partial\phi}{\partial x_n}\right] \\ &= \frac{1}{J}\sum_j \frac{\partial}{\partial r_j}\left[J\frac{\partial r_j}{\partial x_m}a\sum_i\frac{\partial r_i}{\partial x_n}\frac{\partial\phi}{\partial r_i}\right]\end{aligned}$$

In two dimensions we write the expression for $\nabla\cdot(a\nabla\phi)$ in more detail

$$\begin{aligned}\nabla\cdot(a\nabla\phi) = \frac{1}{J}\Big\{&\frac{\partial}{\partial r_1}\left(aJ\left[\frac{\partial r_1}{\partial x_1}^2+\frac{\partial r_1}{\partial x_2}^2\right]\frac{\partial\phi}{\partial r_1}\right)+\frac{\partial}{\partial r_2}\left(aJ\left[\frac{\partial r_2}{\partial x_1}^2+\frac{\partial r_2}{\partial x_2}^2\right]\frac{\partial\phi}{\partial r_2}\right)+ \\ &\frac{\partial}{\partial r_1}\left(aJ\left[\frac{\partial r_1}{\partial x_1}\frac{\partial r_2}{\partial x_1}+\frac{\partial r_1}{\partial x_2}\frac{\partial r_2}{\partial x_2}\right]\frac{\partial\phi}{\partial r_2}\right)+\frac{\partial}{\partial r_2}\left(aJ\left[\frac{\partial r_1}{\partial x_1}\frac{\partial r_2}{\partial x_1}+\frac{\partial r_1}{\partial x_2}\frac{\partial r_2}{\partial x_2}\right]\frac{\partial\phi}{\partial r_1}\right)\Big\}\end{aligned}$$

This expression can be written in the simplified form

$$\nabla\cdot(a\nabla\phi) = \frac{1}{J}\Big\{\frac{\partial}{\partial r_1}\left(A^{11}\frac{\partial\phi}{\partial r_1}\right)+\frac{\partial}{\partial r_2}\left(A^{22}\frac{\partial\phi}{\partial r_2}\right)+\frac{\partial}{\partial r_1}\left(A^{12}\frac{\partial\phi}{\partial r_2}\right)+\frac{\partial}{\partial r_2}\left(A^{21}\frac{\partial\phi}{\partial r_1}\right)\Big\}$$

where $A^{12}=A^{21}$. A **second-order accurate** compact discretization to this expression is

$$\nabla\cdot(a\nabla\phi) \approx \frac{1}{J}\Big\{D_{+r_1}\left(A^{11}_{i_1-\frac12}D_{-r_1}\phi\right)+D_{+r_2}\left(A^{22}_{i_2-\frac12}D_{-r_2}\phi\right)+D_{0r_1}\left(A^{12}D_{0r_2}\phi\right)+D_{0r_2}\left(A^{21}D_{0r_1}\phi\right)\Big\}$$

where we can define the cell average values for A^{mn} by

$$\begin{aligned}A^{11}_{i_1-\frac12} &\approx \frac12(A^{11}_{i_1}+A^{11}_{i_1-1}) \\ A^{22}_{i_2-\frac12} &\approx \frac12(A^{22}_{i_2}+A^{22}_{i_2-1})\end{aligned}$$

We may also want to use the **harmonic** average

$$A^{11}_{i_1-\frac12} \approx \frac{2A^{11}_{i_1}A^{11}_{i_1-1}}{A^{11}_{i_1}+A^{11}_{i_1-1}}$$

which is appropriate if the coefficients vary rapidly.

A **fourth-order accurate** approximation can be derived as follows. A fourth-order accurate discretization to the second derivative is

$$\frac{\partial^2 u}{\partial r^2} = D_+D_-(1-\frac{h^2}{12}D_+D_-)u_i \ +O(h^4)$$

which can be approximately factored into the product

$$\frac{\partial^2 u}{\partial r^2} = \left[D_+(1-\frac{h^2}{24}D_+D_-)\right]\left[D_-(1-\frac{h^2}{24}D_+D_-)\right]u_i \ +O(h^4)$$

where

$$D_+(1-\frac{h^2}{24}D_+D_-)u_i = \frac{\partial u}{\partial r}(x_{i+\frac12}) \ +O(h^4)$$

is a fourth order accurate approximation to the first derivative at $x_{i+\frac{1}{2}}$. Thus

$$\frac{\partial}{\partial r}\left(A\frac{\partial u}{\partial r}\right) = \left[D_+(1-\frac{h^2}{24}D_+D_-)\right]\left[A_{i-\frac{1}{2}}D_-(1-\frac{h^2}{24}D_+D_-)\right]u_i \ +O(h^4)$$

is a fourth-order accurate conservative approximation. We can make this a compact 5 point scheme by dropping the highest order differences (which are $O(h^4)$ anyway) to give

$$\frac{\partial}{\partial r}\left(A\frac{\partial u}{\partial r}\right) = \left[D_+(1-\frac{h^2}{24}D_+D_-)\right]\left[A_{i-\frac{1}{2}}D_-\right]u_i - [D_+]\left[A_{i-\frac{1}{2}}D_-\frac{h^2}{24}D_+D_-\right]u_i \ +O(h^4)$$

or

$$\frac{\partial}{\partial r}\left(A\frac{\partial u}{\partial r}\right) = \left[D_+\left(A_{i-\frac{1}{2}} - (\frac{h^2}{24}D_+D_-)\tilde{A}_{i-\frac{1}{2}} - \tilde{A}_{i-\frac{1}{2}}\frac{h^2}{24}D_+D_-\right)D_-\right]u_i \ +O(h^4)$$

We approximate

$$\begin{aligned} A_{i-\frac{1}{2}} &= \frac{9}{16}(A_i+A_{i-1}) - \frac{1}{16}(A_{i+1}+A_{i-2}) \ +O(h^4) \qquad **checkthis** \\ \tilde{A}_{i-\frac{1}{2}} &= \frac{1}{2}(A_i+A_{i-1}) \ +O(h^2) \end{aligned}$$

A consistent approximation to the boundary condition $\mathbf{n}_m\cdot(a\nabla\phi)$, where $\mathbf{n}_m$ is the normal to the boundary with $r_m = constant$, can be obtained from the expressions

$$\begin{aligned} \mathbf{n}_m &= \frac{\nabla_{\mathbf{x}} r_m}{\|\nabla_{\mathbf{x}} r_m\|} \\ \mathbf{n}_m\cdot(a\nabla\phi) &= a\frac{\nabla_{\mathbf{x}} r_m}{\|\nabla_{\mathbf{x}} r_m\|}\left(\nabla_{\mathbf{x}} r_1\phi_{r_1} + \nabla_{\mathbf{x}} r_2\phi_{r_2}\right) \\ &\equiv B^1\phi_{r_1} + B^2\phi_{r_2} \end{aligned}$$

where we note that the operator $\nabla\cdot(a\nabla\phi)$ contains this expression:

$$\nabla\cdot(a\nabla\phi) = \frac{1}{J}\sum_m\left(\frac{\partial}{\partial r_m}J\|\nabla_{\mathbf{x}} r_m\|\left[\mathbf{n}_m\cdot(a\nabla\phi)\right]\right)$$

(consistent with the divergence theorem). Thus we should approximate the normal derivative at the boundary point i as an average of the approximations to $\mathbf{n}_m\cdot(a\nabla\phi)$ at the points $i-\frac{1}{2}$ and $i+\frac{1}{2}$

$$\mathbf{n}_m\cdot(a\nabla\phi)_i = \frac{1}{2}\left(B^1_{i+\frac{1}{2}}D_{+r_1}\phi_i + B^1_{i-\frac{1}{2}}D_{+r_1}\phi_{i-1}\right) + \frac{1}{2}\left(B^2_{j+\frac{1}{2}}D_{+r_2}\phi_j + B^1_{j-\frac{1}{2}}D_{+r_2}\phi_{j-1}\right)$$

These approximations implicitly appear in the discretization of the operator $\nabla\cdot(a\nabla\phi)$. If we choose the same approximations in the boundary condition then terms will cancel appropriately.

3 Class GridCollectionOperators and Class CompositeGridOperators

This class is used to define differential operators for `realCompositeGridFunction`'s. It uses the `MappedGridOperators` class to do this. The class `CompositeGridOperators` is actually derived from the class `GridCollectionOperators`. Most of the member functions are defined in the base class. For the discussion here, however, we will pretend that the functions are defined in class `CompositeGridOperators`.

3.1 Public member function and member data descriptions

3.1.1 Public enumerators

Here are the public enumerators:

3.1.2 Constructors

GridCollectionOperators()

GridCollectionOperators(GridCollection & gridCollection0)

Description: Construct a GridCollectionOperators

gridCollection0 (input): Associate this grid with the operators.

Author: WDH

GridCollectionOperators(MappedGridOperators & op)

Description: Construct a GridCollectionOperators using a MappedGridOperators

op (input): Associate this grid with these operators.

Author: WDH

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

GridCollectionFunction
"derivative"(const realGridCollectionFunction & u,
const Index & N =nullIndex
)

Description: "derivative" equals one of x, y, z, xx, xy, xz, yy, yz, zz, laplacian, grad, div.

u (input): Take the derivative of this grid function.

N (input): evaluate the derivatives for these components.

return Value: The derivative.

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

3.1.4 Derivative coefficients

GridCollectionFunction
"derivativeCoefficients"(const Index & N =nullIndex)

Description: "derivativeCoefficients" equals one of xCoefficients, yCoefficients, zCoefficients, xxCoefficients, xyCoefficients, xzCoefficients, yyCoefficients, yzCoefficients, zzCoefficients, laplacianCoefficients, gradCoefficients, divCoefficients. Compute the coefficients of the specified derivative.

N (input): evaluate the coefficients for these components.

return Value: The derivative coefficients.

3.1.5 get

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

Description: Get from a database file

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

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

3.1.6 put

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

Description: output onto a database file

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

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

3.1.7 applyBoundaryCondition

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

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

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

Description: Apply a boundary condition to a grid function. This routine implements every boundary condition known to man (ha!)

u (input/output): apply boundary conditions to this grid function.

Components (input): apply to these components

bcType (input): the name of the boundary condition to apply (dirichlet, neumann,...)

bc (input): apply the boundary condition on all sides of the grid where the boundaryCondition array (in the MappedGrid) is equal to this value. By default `bc=BCTypes allBoundaries` apply to all boundaries (with a positive value for boundaryCondition). To apply a boundary condition to a specified side use

- `bc=BCTypes::boundary1` for $(side, axis) = (0, 0)$
- `bc=BCTypes::boundary2` for $(side, axis) = (1, 0)$
- `bc=BCTypes::boundary3` for $(side, axis) = (0, 1)$
- `bc=BCTypes::boundary4` for $(side, axis) = (1, 1)$
- `bc=BCTypes::boundary5` for $(side, axis) = (0, 2)$
- `bc=BCTypes::boundary6` for $(side, axis) = (1, 2)$

or use `bc=BCTypes::boundary1+side+3*axis` for given values of $(side, axis)$ (this could be used in a loop, for example).

forcing (input): This value is used as a forcing for the boundary condition, if needed.

time (input): apply boundary conditions at this time (used by twilightZoneFlow)

bcParameters (input): optional parameters are passed using this object. See the examples for how to pass parameters with this argument.

Limitations: only second order accurate.

3.1.8 applyBoundaryConditionCoefficients

void
applyBoundaryConditionCoefficients(realGridCollectionFunction & coeff,
const Index & Equations,
const Index & Components,
const BCTypes::BCNames &
bcType = BCTypes::dirichlet,
const int & bc = BCTypes::allBoundaries,
const BoundaryConditionParameters &
bcParameters = Overture::defaultBoundaryConditionParameters())

Description: Fill in the coefficients of the boundary conditions.

coeff (input/output): grid function to hold the coefficients of the BC.

t (input): apply boundary conditions at this time.

Limitations: too many to write down.

3.1.9 finishBoundaryConditions

void
finishBoundaryConditions(realGridCollectionFunction & u,
const BoundaryConditionParameters & bcParameters =
Overture::defaultBoundaryConditionParameters(),
const Range & C0 =nullRange,
const IntegerArray & gridsToUpdate =
Overture::nullIntArray)

Description: Call this routine when all boundary conditions have been applied. This function will fix up the solution values in corners and update periodic edges.

u (input/output): Grid function to which boundary conditions were applied.

bcParameters (input): Supply parameters such as bcParameters.orderOfExtrapolation which indicates the order of extrapolation to use.

C0 (input) : apply to these components.

gridsToUpdate (input) : optionally supply a list of grids to update. Bu default all grids are updated.

3.2 Example 1: Operators applied to a `realCompositeGridFunction`

In this example we use the CompositeGridOperators to compute some derivatives. This example is similar to the example described in section (2.2), see the comments there for more information. (file `Overture/examples/tcgo.C`)

```
#include "Overture.h"
#include "CompositeGridOperators.h"
#include "display.h"

//================================================================================
//  Examples showing how to differentiate realCompositeGridFunctions
//     o evaluate using the x,y,... member functions
//     o evaluate in an effficient manner by computing many derivatives at once.
//================================================================================
main(int argc, char *argv[])
{
  Overture::start(argc,argv);  // initialize Overture

  aString nameOfOGFile;
  cout << "Enter the name of the overlapping grid data base file " << endl;
  cin >> nameOfOGFile;

  // create and read in a CompositeGrid
  CompositeGrid cg;
  getFromADataBase(cg,nameOfOGFile);
  cg.update(MappedGrid::THEvertex | MappedGrid::THEcenter | MappedGrid::THEinverseVertexDerivative );

  Index I1,I2,I3;
  Range all;                                          // null Range (defaults to entire Range when
  realCompositeGridFunction  u(cg,all,all,all,Range(0,0)),   // define some component grid functions in
                            v2(cg,all,all,all,Range(0,0)),
                            v4(cg,all,all,all,Range(0,0)),
                             q(cg,all,all,all,Range(0,1));   // q has 2 components

  CompositeGridOperators operators(cg);               // define some differential operators
  u.setOperators(operators);                          // Tell u which operators to use
  q.setOperators(operators);

  for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
  {
    MappedGrid & mg = cg[grid];                                           // mg is an alias for c
    getIndex(mg.dimension(),I1,I2,I3);                                     // assign I1,I2,I3
    u[grid](I1,I2,I3)=sin(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2));   // u=sin(x)*c

//     realMappedGridFunction ivd;
//     ivd=cg[grid].inverseVertexDerivative();
//     ::display(ivd,"ivd");

  }

  u.display("here is u");
  operators.x(u).display("Here is operators.x(u)");              // one way to compute u.x
  u.x().display("Here is u.x");                                  // another way to compute u.x

  v2=u.x();                                    // save x derivative (2nd-order)

  Range c0(0,0),c1(1,1);
```

```
q(c0)=1.;                            // assign component 0 of q. This is cute but relatively expensive
q(c1)=2.;                            // assign component 1 of q.
q.display("here is q");
q(c0)=q(c0)*q.x(c0)+q(c1)*q.y(c0);

operators.setOrderOfAccuracy(4);                // now compute to 4th order
v4=u.x();                                       //  save x derivative (4th-order)

operators.setOrderOfAccuracy(2);             // reset back to 2nd order

// print the errors
real error;
for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
{
  MappedGrid & mg = cg[grid];                                                  // mg is an alias for c
  // compute errors on interior points and boundary
  getIndex(mg.indexRange(),I1,I2,I3);                               // assign I1,I2,I3
  error = max(fabs(v2[grid](I1,I2,I3)- cos(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2
  cout << "Maximum error (2nd order) = " << error << endl;

  error = max(fabs(v4[grid](I1,I2,I3)-cos(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2)
  cout << "Maximum error (4th order) = " << error << endl;
}

// Now we compute the derivatives in a more efficient way. To do this we loop over the
// component grids.

// The arrays ux and uy are used to save the results in. These arrays are re-used for all
// the different component grids (thus saving space)
RealArray ux,uy;
// --- make a list of derivatives to evaluate on each component grid
for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
{
  operators[grid].setNumberOfDerivativesToEvaluate( 2 );
  operators[grid].setDerivativeType( 0, MappedGridOperators::xDerivative, ux );
  operators[grid].setDerivativeType( 1, MappedGridOperators::yDerivative, uy );
  operators[grid].setOrderOfAccuracy(2);
}

// Now evaluate the derivatives
for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
{
  MappedGrid & mg = cg[grid];

  // compute the x and y derivatives of u and save in the arrays ux and uy
  operators[grid].getDerivatives(u[grid],I1,I2,I3);
  // this next line is another way to do exactly the same thing
  u[grid].getDerivatives(I1,I2,I3);

  error = max(fabs(ux(I1,I2,I3)-cos(mg.vertex()(I1,I2,I3,axis1))*cos(mg.vertex()(I1,I2,I3,axis2))));
  cout << "Maximum error in ux: (2nd order) = " << error << endl;
  error = max(fabs(uy(I1,I2,I3)+sin(mg.vertex()(I1,I2,I3,axis1))*sin(mg.vertex()(I1,I2,I3,axis2))));
  cout << "Maximum error in uy: (2nd order) = " << error << endl;
}

Overture::finish();
cout << "Program Terminated Normally! \n";
```

```
  return 0;
}
```

4 Boundary Conditions

The boundary condition operators define a "library" of elementary boundary condition operations that can be used to implement application specific boundary conditions. Examples of elementary boundary conditions include Dirichlet, Neumann and mixed conditions, extrapolation, setting the normal component of a vector and so on.

Here are the elementary boundary conditions that are supported

$u = g$	dirichlet
$\partial_n u = g$	neumann
$a_0 u + a_1 \partial_n u = g$	mixed
$(D_+)^p u = 0$	extrapolation (to p^{th} order)
$(D_+)^p \mathbf{n} \cdot \mathbf{u} = 0$	extrapolate normal component (to p^{th} order)
$(D_+)^p \mathbf{t}_m \cdot \mathbf{u} = 0$	extrapolate tangential component, m=0,1
$\mathbf{n} \cdot \mathbf{u} = g$	normalComponent
$\mathbf{a} \cdot \mathbf{u} = g$	aDotU
$a_0 \partial_x u_1 + a_1 \partial_y u_2 + a_2 \partial_z u_3 = g$	generalizedDivergence
$a_0 u + a_1 u_x + a_2 u_y + a_3 u_z = g$	generalMixedDerivative
$u(-m) = u(+m)$	evenSymmetry
$\mathbf{n} \cdot \mathbf{u}(-m) = \mathbf{n} \cdot (2\mathbf{u}(0) - \mathbf{u}(+m))$, $\mathbf{t} \cdot \mathbf{u}(-m) = \mathbf{t} \cdot \mathbf{u}(+m)$	vectorSymmetry
$\mathbf{u} \leftarrow (\mathbf{n} \cdot \mathbf{u})\mathbf{n} + \mathbf{g}$	tangentialComponent
$\mathbf{t}_m \cdot \mathbf{u} = g$	tangentialComponent{m}, m=0,1
$\mathbf{n} \cdot \partial_n \mathbf{u} = g$	normalDerivativeOfNormalComponent
$\mathbf{t}_m \cdot \partial_n \mathbf{u} = g$	normalDerivativeOfTangentialComponent{m}, m=0,1
$\mathbf{n} \cdot a \nabla u = g$	normalDerivativeScalarGrad

Here are possible future ones (let me know if you need something)

$(\mathbf{a} \cdot \nabla) u = g$	aDotGradU
$\partial_n (\mathbf{a} \cdot \mathbf{u}) = g$	normalDerivativeOfADotU

The notation $u(-m) = u(+m)$ means that the value of the solution on ghost line m is set equal to the value on the m^{th} line inside the domain. Here $\mathbf{n}$ is the unit OUTWARD normal and ∂_n is the normal derivative, $\partial_n = \mathbf{n} \cdot \nabla$, and $\mathbf{t}_m$ represents the tangent vector(s).

There is also a `extrapolateInterpolationNeighbours` boundary condition described below.

There are two common approaches to implementing boundary conditions

- Use ghost points
- Do not use ghost points; instead use one sided differences.

On curvilinear grids my experience is that the first approach is easier. Moreover, using one sided differences is equivalent to using a centred difference on the boundary and extrapolating the ghost point(s). Thus we will only discuss how to assign boundary conditions assuming that we are using ghost points.

Consider first the case of a second order accurate method. Suppose that all variables have Dirichlet boundary conditions. In this case the ghost points are probably not used; if they are it is usually good enough just to extrapolate the ghost points.

$$\text{Dirichlet:} \left\{ \begin{array}{l} \text{1. extrapolate ghost points} \\ \text{2. apply Dirichlet boundary conditions} \end{array} \right.$$

Now suppose that all variables have a Neumann boundary condition. In this case the equation can be applied up to and including the boundary. The Neumann boundary condition can be thought of as giving the value at the fictitious points.

$$\text{Neumann:}\begin{cases}\text{1. apply interior equation on the boundary}\\ \text{2. apply Neumann boundary conditions}\end{cases}$$

When a boundary condition consists of some variables being given Dirichlet and some given Neumann boundary conditions it is often appropriate to

$$\text{Neumann/Dirichlet:}\begin{cases}\text{1. apply interior equation on the boundary}\\ \text{2. apply the Dirichlet Boundary conditions}\\ \text{3. extrapolate variables with Dirichlet boundary conditions}\\ \text{4. apply Neumann boundary conditions}\end{cases}$$

Note that the order of applying the conditions is important. For example, the Neumann condition may use values of the Dirichlet variables on the boundary or on the ghost points. In this case the Neumann condition should be applied last.

Now let us see some examples of how we can actually implement the above procedures...

4.1 Example: apply boundary conditions to a MappedGridFunction

The `applyBoundaryCondition` member function of the `MappedGridOperators` or a `MappedGridFunction` will assign an elementary boundary condition, such as `dirichlet`, to all sides of a `MappedGrid mg` where the values of `mg.boundaryCondition(side,axis)` are equal to a specified positive integer. Usually a solver will define integer values for non-elementary boundary conditions such as

```
const int inflow=1,
          outflow=2,
          wall=3;
```

The values of `mg.boundaryCondition(side,axis)` will then be assigned with the appropriate values such as

```
mg.boundaryCondition(Start,axis1)=inflow;
mg.boundaryCondition(End  ,axis1)=outflow;
mg.boundaryCondition(Start,axis2)=wall;
etc.
```

A function call of the form

```
realMappedGridFunction u(...)
...
int component=0;
u.applyBoundaryCondition(component,dirichlet,inflow,1.);
```

will assign a Dirichlet boundary condition, $u = 1$, to *component* $= 0$ of u, on all sides of the grid where `mg.boundaryCondition(side,axis)=inflow`.

When the `MappedGridOperators` `applyBoundaryCondition` function is called it loops through all the boundaries in the following fashion:

```
  ...
  ForBoundary(side,axis) // loop over all faces
  {
    if( c.boundaryCondition(side,axis)==bc
        || ( bc==allBoundaries && c.boundaryCondition(side,axis) > 0) )
    {
      switch ( bcType )
      {
      case dirichlet:
        // assign dirichlet BC on this side
        break;
      case neumann:
        // assign dirichlet BC on this side
        break;
      ...
      }
    }
  }
...
```

The enumator `allBoundaries` is a default argument.

The `finishBoundaryConditions` function should be called when all boundary conditions have been applied. This routine will assign values in corners and update periodic boundaries.

In this example code we show how to assign and evaluate boundary conditions. Applying boundary conditions to a `realCompositeGridFunction` works in the same way. (file `Overture/examples/bcgf.C`)

4.2 Boundary Condition Descriptions

In this section we describe in some detail how each elementary boundary condition is applied.

Define the following values which are functions of the input parameters to `applyBoundaryCondition`:

```
void MappedGridOperators::
applyBoundaryCondition(realMappedGridFunction & u,
       const Index & Components,
       const BCTypes::BCNames & bcType,  /* = BCTypes::dirichlet */
       const int & bc,                   /* = allBoundaries */
       const real & forcing,             /* =0. */
       const real & time,                /* =0. */
       const BoundaryConditionParameters &
                             bcParameters /* = defaultBoundaryConditionParameters */,
                       const int & grid /* =0 */ )

  MappedGrid & mg = *u.getMappedGrid();
  Range C = Components;
  int nc = Components.getLength();  // number of components
  int nd = number of space dimensions
  intArray & components = bcParameters.components;
  bool componentsSpecified = components.getLength(0) > 0;
```

```
int lineToAssign = bcParameters.lineToAssign;
Index I1,I2,I3;
int side,axis; // defines the face of the grid we are on
int grid;      // defines the grid number if from a gridCollectionFunction
getBoundaryIndex(mg.gridIndexRange,side,axis,I1,I2,I3,lineToAssign);
Range C1 = C-C.getBase()+forcing.getBase();
OGFunction e = twilight zone function (if specified)
```

There are also versions of `applyBoundaryCondition` where `forcing` is a `realArray`, or a `realMappedGridFunction` or and array of `realArray`'s.
Note: For boundary conditions that normally assign the value on the boundary (such as `dirichlet` or `normalComponent` a value can be assigned on a line other than the boundary by setting `bcParameters.lineToAssign` – a value of zero is the boundary, 1 the first ghost line and -1 the first interior line etc.

4.2.1 dirichlet

By default the dirichlet boundary condition assigns values on the boundary according to the following

$$u(I1,I2,I3,uC) = \begin{cases} e.u(mg,I1,I2,I3,fC,t) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ forcing(fC) & \text{if forcing is a realArray with 1 array dimension} \\ forcing(I1,I2,I3,fC) & \text{if forcing is a realArray that is big enough} \\ forcing(fC,side,axis,grid) & \text{if forcing is a realArray that is big enough} \\ forcing(I1,I2,I3,fC) & \text{if forcing is a gridFunction} \end{cases}$$

Here `uC` and `fC` are intArrays and

$$u(I1,I2,I3,uC) = e.u(mg,I1,I2,I3,fC,t)$$

means

$$u(I1,I2,I3,uC(i)) = e.u(mg,I1,I2,I3,fC(i),t) \quad \text{for} \quad i = uC.getBase(0),\ldots,uC.getBound(0)$$

The values found in the intArrays `uC` and `fC` depend on the arguments to `applyBoundaryConditions`. By default

$$\begin{aligned} uC(i) &= i \quad \text{for} \quad i = C.getBase(),\ldots,C.getBound() \\ fC(i) &= i \quad \text{for} \quad i = C.getBase(),\ldots,C.getBound() \end{aligned}$$

However, if the argument `forcing` is a grid function then fC is defined so that it's base is the same as the base of the grid function `forcing`:

$$fC(i) = i - C.getBase() + forcing.getBase() \quad \text{for} \quad i = C.getBase(),\ldots,C.getBound()$$

For arbitrary control of which components to use one can dimension and set one or both of the intArrays `bcParameters.uComponents` and `bcParameters.fComponents`. When either of these intArrays is given the argument C is ignored. The following statements define how `uC` and `fC` are determined in all cases (with `uComponents`:=`bcParameters.uComponents` and `fComponents`:=`bcParameters.fComponents`)

$$uC = \begin{cases} C & \text{if neither uComponents nor fComponents is specified} \\ \texttt{uComponents} & \text{if } \texttt{uComponents} \text{ is given} \\ b, b+1, ... & \text{if } \texttt{fComponents} \text{ is specified but not } \texttt{uComponents}\text{, b=u.getComponentBase(0)} \end{cases}$$

and

$$fC = \begin{cases} C & \text{if neither } \texttt{uComponents} \text{ nor } \texttt{fComponents} \text{ is specified} \\ b, b+1, ... & \text{if as above case but with grid function forcing, b=forcing.getComponentBase(0)} \\ \texttt{fComponents} & \text{if } \texttt{fComponents} \text{ is given} \\ b, b+1, ... & \text{if } \texttt{uComponents} \text{ is specified but not } \texttt{fComponents}\text{, b=forcing.getComponentBase(0)} \end{cases}$$

A value can be assigned on a line other than the boundary by setting `bcParameters.lineToAssign` – a value of zero is the boundary, 1 the first ghost line and -1 the first interior line etc.

Sometimes a given boundary condition such as `dirichlet` will need to use different forcing values on different sides of different grids. Maybe the dirichlet value on one face is 1 while on another face it is 2. These different values can be passed with a `realArray` forcing (they can also be passed more generally with a grid function). If the forcing function `force` is a `realArray` with dimensions that are large enough then the forcing for a given face `(side,axis)` belonging to a given `grid` will be taken as `force(fC,side,axis,grid)`. If the `force` array is not dimensioned large enough for the given index values of `(side,axis,grid)` then `force(fC)` will be used.

4.2.2 neumann

For second-order accuracy the neumann boundary condition will assign the value on the first ghost line from $\mathbf{n} \cdot \nabla u = g$. Recall that $\mathbf{n}$ is the outward normal.

Define

```
Index Ig1,Ig2,Ig3;
getGhostIndex(mg.gridIndexRange,side,axis,Ig1,Ig2,Ig3);    // first ghost line
Index Ip1,Ip2,Ip3;
getGhostIndex(mg.gridIndexRange,side,axis,Ip1,Ip2,Ip3,-1);  // first line in
```

On a rectangular grid the neumann condition is computed as

$$u(Ig1, Ig2, Ig3, uC) = u(Ip1, Ip2, Ip3, uC) \pm 2\Delta x_{\text{axis}}\; g \;,$$

where Δx_{axis} is the grid spacing in the direction normal to the boundary and

$$g = \begin{cases} \mathbf{n} \cdot (e.uGrad(mg, I1, I2, I3, fC, t)) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ forcing(fC) & \text{if forcing is a realArray with 1 array dimension} \\ forcing(I1, I2, I3, fC) & \text{if forcing is a realArray that is big enough} \\ forcing(fC, side, axis, grid) & \text{if forcing is a realArray that is big enough} \\ forcing(I1, I2, I3, fC) & \text{if forcing is a gridFunction} \end{cases}$$

and

$$e.uGrad(mg, I1, I2, I3, fC, t) = (e.ux(mg, I1, I2, I3, fC, t), e.uy(mg, I1, I2, I3, fC, t), e.uz(mg, I1, I2, I3, fC,$$

The definition of the intArray's `uC` and `fC` are given in the comments for Dirichlet boundary conditions.

On a curvlinear grid $u(Ig1,Ig2,Ig3,uC)$ is determined by imposing the condition $\mathbf{n}\cdot\nabla u = g$ (on the boundary). This is done by forming the matrix coefficients for $\mathbf{n}\cdot\nabla$ (on the boundary)

$$c(M,I1,I2,I3) = \mathbf{n}\cdot(op.xCoefficients(), op.yCoefficientsI(), op.zCoefficients())$$

(M represents the stencil, 9 points or 27 points). Then we have an equation of the form

$$c(m_0,I1,I2,I3)u(Ig1,Ig2,Ig3,C) = \sum_{m\neq m_0} c(m,I1,I2,I3)u(I1(m),I2(m),I3(m),C) + \text{forcing}$$

that determines $u(Ig1,Ig2,Ig3,uC)$ (m_0 is the stencil index corresponding to the ghost line value). (The coefficients are only computed once for efficiency).

4.2.3 mixed

For second-order accuracy the mixed boundary condition will assign the value on the first ghost line from the discretization of

$$a_0u + a_1(\mathbf{n}\cdot\nabla)u = g$$

where $\mathbf{n}$ is the outward normal. It is assumed that $a_1 \neq 0$. The values of a_0 and a_1 are found in `bcParameters.a`. If `bcParameters.a` is dimensioned to be at least as large as `bcParameters.a(2,2,numberOfDimensions,numberOfGrids)` then the values for a_0 and a_1 will be (a_0,a_1)=`bcParameters.a(0:1,side,axis,grid)` where `side,axis,grid` denote the particular boundary we are on. In this way different values can be used on different sides of different grids. Otherwise a_0 and a_1 will be (a_0,a_1)=`bcParameters.a(0:1)` and the same values will be used on all boundaries.

Since for non-rectangular grids the matrix representing the boundary operator is saved (for efficiency) it is currently assumed that the values $a(0:1)$ do not change from one call to the next .

The mixed boundary condition is applied in basically the same way as the `neumann` boundary condition (see above for more details).

4.2.4 extrapolate

Extrapolation determines a value on a ghostline by extrapolating along the coordinate direction normal to the boundary. By default the value on the first ghostline is determined using second order extrapolation:

$$u(Ig1,Ig2,Ig3,uC) = 2u(I1,I2,I3,uC) - u(Ip1,Ip2,Ip3,uC) + g\ ,$$

or more generally using p^{th}-order extrapolation (p=1,...,10)

$$u(Ig1,Ig2,Ig3,uC) = D_{\pm}^{p}(u(I1,I2,I3,uC)) + g\ ,$$

Here the extrapolation operator is either D_-^p or D_+^p, chosen so we extrapolate into the interior of the grid, and

$$g = \begin{cases} e.u(mg,Ig1,Ig2,Ig3,fC,t) - D_{\pm}^{p}(e.u(mg,I1,I2,I3,fC,t)) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ forcing(fC) & \text{if forcing is a realArray with 1 array dim} \\ forcing(I1,I2,I3,fC) & \text{if forcing is a realArray that is big enoug} \\ forcing(fC,side,axis,grid) & \text{if forcing is a realArray that is big enoug} \\ forcing(I1,I2,I3,fC) & \text{if forcing is a gridFunction} \end{cases}$$

The definition of the intArray's `uC` and `fC` are given in the comments for Dirichlet boundary conditions.

To extrapolate a different line change `bcParameters.ghostLineToAssign` (default=1). To change the order of extrapolation set `bcParameters.orderOfExtrapolation` (default=2).

4.2.5 normalComponent

The `normalComponent` boundary condition changes the values of u on the boundary (or some other line) to satisfy $\mathbf{n}\cdot\mathbf{u}=g$. This can be done by the projection

$$\mathbf{u}(I1,I2,I3,uC) \leftarrow \mathbf{u}(I1,I2,I3,uC)+[g-(\mathbf{n}\cdot\mathbf{u}(I1,I2,I3,uC))]\mathbf{n}$$

The forcing for this boundary condition is determined from

$$g(I1,I2,I3)=\begin{cases} \mathbf{n}\cdot e.u(mg,I1,I2,I3,fC,t) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ \mathbf{n}\cdot forcing(fC) & \text{if forcing is a realArray with 1 array dimension} \\ \mathbf{n}\cdot forcing(fC,side,axis,grid) & \text{if forcing is a realArray that is big enough} \\ forcing(I1,I2,I3) & \text{if forcing is a scalar gridFunction} \\ \mathbf{n}\cdot forcing(I1,I2,I3,fC) & \text{if forcing is a vector gridFunction} \end{cases}$$

The definition of the intArray's `uC` and `fC` are given in the comments for Dirichlet boundary conditions.

4.2.6 tangentialComponent0, tangentialComponent1

The `tangentialComponent0` and `tangentialComponent1` boundary conditions change the value of $\mathbf{u}$ on the boundary (or some other line) to satisfy $\mathbf{t}_m\cdot\mathbf{u}=g$ for $m=0$ or $m=1$.

There are two (or one in 2D) tangent vectors on a given boundary. Label the boundary with the two integerers $(axis, side)$ where $(axis=0,1,2, side=0,1)$ for the 6 faces. The two tangent vectors are the derivatives with respect to the two tangential unit square coordinates, r_k, where the values for k are obtained as a cyclic permutation starting from the value of $axis+1$,

$$k = axis+m+1 \quad \text{mod } \texttt{numberOfDimensions},$$

The tangent vectors are normalized to be unit length

$$\mathbf{t}_m=\frac{\partial\mathbf{x}/\partial r_k}{||\partial\mathbf{x}/\partial r_k||}, \qquad m=0,1, \quad k=1,2\ (axis=0) \text{ or } k=2,0\ (axis=1) \text{ or } k=0,1\ (axis=2)$$

and are accessible in a `MappedGrid` as the `centerBoundaryTangent[axis][side](I1,I2,I3,0:nd-1,m)` (where `nd=numberOfSpaceDimensions`).

These boundary conditions are applied in the same manner as the `normalComponent` boundary condition, see the comments there for further details.

4.2.7 normalDerivativeOfTangentialComponent[0,1]

The `normalDerivativeOfTangentialComponent0` (or `normalDerivativeOfTangentialComponent1`) boundary condition changes the values of u on the ghost line to satisfy

$$\mathbf{t}_m\cdot(\frac{\partial}{\partial n}\mathbf{u})=g.$$

where $\mathbf{t}_m$, $m = 0$ (or $m = 1$) is the tangent vector as defined in section (4.2.6) This is not really the normal derivative of the tangential component:

$$\frac{\partial}{\partial n}(\mathbf{t}_m \cdot \mathbf{u}) = g \qquad \text{(not this!)}$$

unless the tangent vector is constant, but it is close and probably good enough for most purposes (?).

The forcing functions for this boundary condition can be of one of the following forms

$$g(I1,I2,I3) = \begin{cases} \mathbf{t}_m \cdot (\mathbf{n} \cdot \nabla(e.u(mg,I1,I2,I3,fC,t)) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ \mathbf{t} \cdot forcing(fC) & \text{if forcing is a realArray with 1 array dimension} \\ \mathbf{t} \cdot forcing(fC,side,axis,grid) & \text{if forcing is a realArray that is big enough} \\ forcing(I1,I2,I3) & \text{if forcing is a scalar gridFunction} \\ \mathbf{t}_m \cdot forcing(I1,I2,I3,fC) & \text{if forcing is a vector gridFunction} \end{cases}$$

The definition of the intArray's `uC` and `fC` are given in the comments for Dirichlet boundary conditions.

4.2.8 extrapolateNormalComponent, extrapolateTangentialComponent[0,1]

The `extrapolateNormalComponent` boundary condition changes the value of the normal component of $\mathbf{u}$ on a ghost line by extrapolation from interior values. This can be done by the projection

$$\mathbf{u}(Ig1,Ig2,Ig3,uC) \leftarrow \mathbf{u}(Ig1,Ig2,Ig3,uC) + [g - (\mathbf{n} \cdot \mathbf{u}(Ig1,Ig2,Ig3,uC))]\mathbf{n}$$

where $((Ig1,Ig2,Ig3)$ are the indices of the ghost line and g is the extrapolated value from interior points, for example,

$$g = 2\mathbf{n} \cdot \mathbf{u}(I1g+1,I2g,I3g,uC) - \mathbf{n} \cdot \mathbf{u}(I1g+1,I2g,I3g,uC).$$

The definition of the intArray's `uC` and `fC` are given in the comments for Dirichlet boundary conditions.

To extrapolate a different line change `bcParameters.ghostLineToAssign` (default=1). To change the order of extrapolation set `bcParameters.orderOfExtrapolation` (default=2).

The `extrapolateTangentialComponent0` and `extrapolateTangentialComponent1` are the same as`extrapolateNormalComponent` except that the normal vector is replaced by the tangent vector $\mathbf{t}_m$ for $m = 0$ or $m = 1$.

4.2.9 extrapolateTangentialComponent0, extrapolateTangentialComponent0,

The tangential components of a vector grid function can also be extrapolated in a similar fashion to the `extrapolateNormalComponent` boundary condition.

4.2.10 tangentialComponent

The `tangentialComponent` boundary condition sets the value of the tangential component(s).
WARNING: You cannot in general use this condition on two adjacent sides of a grid and expect that the value at the corner is correct since there are two equations defining the corner value and only the last one applied will be satisfied (in general).

It changes the value of $\mathbf{u}$ on the boundary to satisfy $\mathbf{u} - (\mathbf{n}\cdot\mathbf{u})\mathbf{n} = g$. This is done (without having to know tangential vectors) by setting

$$\mathbf{u}(I1,I2,I3,uC) \leftarrow [\mathbf{n}\cdot\mathbf{u}(I1,I2,I3,uC)]\mathbf{n} + g$$

If uC specifies more values than the number of space dimensions then the extra values are ignored. The forcing for this boundary condition is determined from

********************* finish this *************************

$$g(I1,I2,I3) = \begin{cases} \mathbf{n}\cdot e.u(mg,I1,I2,I3,fC,t) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ \mathbf{n}\cdot forcing(fC) & \text{if forcing is a realArray} \\ forcing(I1,I2,I3) & \text{if forcing is a scalar gridFunction} \\ \mathbf{n}\cdot forcing(I1,I2,I3,fC) & \text{if forcing is a vector gridFunction} \end{cases}$$

4.2.11 evenSymmetry

The evenSymmetry boundary condition determines the values on the n^{th} ghostline by setting them equal to the values on the n^{th} line in:

$$u(Ig1,Ig2,Ig3,uC) = u(Ip1,Ip2,Ip3,uC) + g$$

where

$$g = e.u(mg,Ig1,Ig2,Ig3,fC,t) - e.u(mg,Ip1,Ip2,Ip3,fC,t) \quad \text{if twightZoneFlow==TRUE}$$

By default the first ghostline is assigned. To assign a different ghostline set `bcParameters.ghostLineToAssign` (default=1).

4.2.12 vectorSymmetry

Apply a symmetry condition to a vector $\mathbf{u} = (u1,u2,u3)$ by making $\mathbf{n}\cdot\mathbf{u}$ an odd function with respect to the boundary and $\mathbf{t}\cdot\mathbf{u}$ an even function:

$$\begin{aligned} \mathbf{t}\cdot\mathbf{u}(-m) &= \mathbf{t}\cdot\mathbf{u}(+m) \\ \mathbf{n}\cdot\mathbf{u}(-m) &= \mathbf{n}\cdot(2\mathbf{u}(0) - \mathbf{u}(+m)) \end{aligned}$$

This condition can be used, for example, in a fluids computation as a boundary condition for the velocity at a symmetry wall - the velocity normal to the wall is odd will the velocities tangential to the walls are even.

The components of u that are changed are given by $u(I1,I2,I3,uC)$. If uC specifies more values than the number of space dimensions then the extra values are ignored.

To implement the boundary condition we first set all components on the ghost line:

$$u(Ig1,Ig2,Ig3,uC) = u(Ip1,Ip2,Ip3,uC)\ .$$

This will make all components even. We then change the normal component on the ghostline to make the normal component odd:

$$\mathbf{n}\cdot u(Ig1,Ig2,Ig3,uC) = \mathbf{n}\cdot(2u(I1,I2,I3,uC) - u(Ip1,Ip2,Ip3,uC)) + g$$

where

$$\begin{aligned} g = \mathbf{n}\cdot(&e.u(mg,Ig1,Ig2,Ig3,fC(0),t) - 2e.u(mg,I1,I2,I3,fC(1),t) \\ &+e.u(mg,Ip1,Ip2,Ip3,fC(2),t)) \qquad \text{if twightZoneFlow==TRUE} \end{aligned}$$

This can be done by the projection

$$\begin{aligned} \mathbf{u}(Ig1,Ig2,Ig3,uC) \leftarrow \mathbf{u}(Ig1,Ig2,Ig3,uC) + (g - (\mathbf{n}\cdot(\mathbf{u}(Ig1,Ig2,Ig3,uC) - (2\mathbf{u}(I1,I2,I3,uC) \\ -\mathbf{u}(Ip1,Ip2,Ip3,uC)))))\mathbf{n} \end{aligned}$$

4.2.13 aDotU

The `aDotU` boundary condition changes the values of u on the boundary to satisfy $\mathbf{a}\cdot\mathbf{u} = g$. This can be done by the projection

$$\mathbf{u}(I1,I2,I3,uC) \leftarrow \mathbf{u}(I1,I2,I3,uC) + [g - (\mathbf{a}\cdot\mathbf{u}(I1,I2,I3,uC))]\frac{\mathbf{a}}{||\mathbf{a}||^2}$$

The values of the vector $\mathbf{a}$ are found in the array `bcParameters.a(0:)`. If uC specifies more values than the number of space dimensions then the extra values are ignored. The forcing for this boundary condition is determined from

$$g(I1,I2,I3) = \begin{cases} \mathbf{a}\cdot e.u(mg,I1,I2,I3,fC,t) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ \mathbf{a}\cdot forcing(fC) & \text{if forcing is a realArray with 1 array dimension} \\ \mathbf{a}\cdot forcing(fC,side,axis,grid) & \text{if forcing is a realArray that is big enough} \\ forcing(I1,I2,I3) & \text{if forcing is a scalar gridFunction} \\ \mathbf{a}\cdot forcing(I1,I2,I3,fC) & \text{if forcing is a vector gridFunction} \end{cases}$$

4.2.14 generalMixedDerivative

The general mixed derivative boundary condition is

$$a(0)u + a(1)u_x + a(2)u_y + a(3)u_z = g\ .$$

For a second-order accurate discretization this condition will determine the value of u on the first ghostline. The values of the vector $\mathbf{a}$ are found in the array `bcParameters.a(0:)`. (To be well defined this means that $\mathbf{a}\cdot\mathbf{n}\neq 0$)

The right-hand side is given by

$$g = \begin{cases} a(0)e.u(mg,Ig1,Ig2,Ig3,fC,t) & \\ \quad + a(1:3)\cdot e.uGrad(I1,I2,I3,fC,t) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ forcing(fC) & \text{if forcing is a realArray with 1 array dimension} \\ forcing(fC,side,axis,grid) & \text{if forcing is a realArray that is big enough} \\ forcing(I1,I2,I3,fC) & \text{if forcing is a gridFunction} \end{cases}$$

To impose this condition the matrix of coefficients for

$$a(0)I + \mathbf{a}(1:3)\cdot\nabla$$

is formed...

4.2.15 generalizedDivergence

This boundary condition can be used to set the divergence, $\nabla \cdot \mathbf{u} = g$, of vector grid function, or more generally to set

$$a(0)u(0)_x + a(1)u(1)_y + a(2)u(2)_z = g$$

Note that this is a single condition imposed on a vector. The values of the vector $\mathbf{a}$ are found in the array `bcParameters.a(0:)`. If `bcParameters.a` is not dimensioned then by default $\mathbf{a} = (1, 1, 1)$ (in which case this condition sets the divergence : $\nabla \cdot \mathbf{u} = g$).

If uC specifies more values than the number of space dimensions then the extra values are ignored. The forcing for this boundary condition is determined from

$$g = \begin{cases} a(0:2) \cdot e.uGrad(mg, I1, I2, I3, fC(0), t) & \text{if twightZoneFlow==TRUE} \\ forcing & \text{if forcing is a real} \\ \mathbf{a} \cdot forcing(fC) & \text{if forcing is a realArray} \\ forcing(I1, I2, I3) & \text{if forcing is a scalar gridFunction} \\ \mathbf{a} \cdot forcing(I1, I2, I3, fC) & \text{if forcing is a vector gridFunction} \end{cases}$$

NOTE: This boundary condition uses some values on the ghostlines of adjacent boundaries when applying this equation at corners. Thus you **must make sure that ghostline values on adjacent boundaries have been assigned** before applying this boundary condition.

Method: In the case of a rectangular grid this condition is rather easy to apply. For example, for the boundary with x =constant and a second-order difference approximation we would solve

$$a(0)D_{0x}u(0) \equiv a(0)\frac{(u(0)_{i+1} - u(0)_{i-1})}{2\Delta x} = -a(1)D_{0,y}u(1) - a(2)D_{0z}u(2) + g$$

for the value on the ghost line, $u(0)_{i-1}$ (left edge) or $u(0)_{i+1}$ (right edge). Here D_{0x}, D_{0y} and D_{0z} are the centered difference operators in the $x, y, z-$directions.

For a general curvilinear grid we must project the values of $\mathbf{u}$ on the ghost line so the condition is satisfied. To do this we form the discrete approximation to

$$a(0)u(0)_x + a(1)u(1)_y + a(2)u(2)_z = g$$

on the boundary. This gives a stencil operator at each boundary point $\mathbf{i} = (i_1, i_2, i_3)$ of the form

$$\sum_{\mathbf{m}} \mathbf{c_m} \cdot \mathbf{u_{i+m}} = g_{\mathbf{i}} \qquad , \ \mathbf{m} = (m_1, m_2, m_3)$$

where for a 27 point stencil (or 9 point in 2D) each component of $\mathbf{m}$ ranges over $-1 \le m_\mu \le +1$. (Note that the corner points in the stencil $\mathbf{c_m}$ are actually zero since only first derivatives appear in this boundarry condition so the stencil is really 7 point (or 5 point).) If we solve this equation for the unknown value of $\mathbf{c_m} \cdot \mathbf{u_{i+m}}$ on the ghost point, say, $\mathbf{c}_{(-1,0,0)} \cdot \mathbf{u}_{\mathbf{i}+(-1,0,0)}$, in terms of the known values of $\mathbf{u}$ on the boundary and the interior then we are led to the equation

$$\mathbf{c}_{(-1,0,0)} \cdot \mathbf{u}_{\mathbf{i}+(-1,0,0)} = g_{\mathbf{i}} - \sum_{\mathbf{m} \neq (-1,0,0)} \mathbf{c_m} \cdot \mathbf{u_{i+m}} \qquad (2)$$

that must be satisfied. This equation looks just like our $\mathbf{a} \cdot \mathbf{u} = g$ boundary condition so we can apply the same formula

$$\mathbf{u}_g \leftarrow \mathbf{u}_g - (\tilde{\mathbf{g}} - \mathbf{a} \cdot \mathbf{u}_g)\frac{\mathbf{a}}{\|\mathbf{a}\|^2}$$

where $\mathbf{a} = \mathbf{c}_{(-1,0,0)}$ and $\tilde{\mathbf{g}}$ is the right hand side of (2). Note that we are able to change the appropriate component of $\mathbf{u}_{(-1,0,0)}$ without having to decompose the operator into tangential and normal components.

4.3 extrapolateInterpolationNeighbours

Extrapolate the unused points that lie next to interpolation points. This boundary condition is useful if one has a second order method with fourth-order artificial viscosity. This routine will fill in values needed by the larger stencil of the fourth-order artificial viscosity. This is often a good enough solution, rather than creating an overlapping grid with two lines of interpolation (discretization width = 5).

Note: the "corners" next to interpolation points are not assigned, only the neighbours that lie along one of the coordinate directions. So the points marked "e" below are assigned

```
      e e e
    e I I I       e=extrapolate
  e I I X X       I= interpolation pt
  e I X X X       X= discretaization pt
  e I X X X
```

4.4 Boundary conditions at corners (and edges in 3D)

The corners of a grid are assigned by `finishBoundaryConditions`. By default the corners are extrapolated but there are other options for assigning the corners given by the following enum found in the `BoundaryConditionParameters` class

```
enum CornerBoundaryConditionEnum
{
  extrapolateCorner=0,
  symmetryCorner,  // name deprecated, use evenSymmetryCorner
  taylor2ndOrder,  // name deprecated, use taylor2ndOrderEvenCorner
  evenSymmetryCorner,
  oddSymmetryCorner,
  taylor2ndOrderEvenCorner,
  taylor4thOrderEvenCorner
};
```

To set the conditions used on a particular corner first set the property in a BoundaryConditionParameters object and then use this object when assigning boundary conditions:

```
bcParams.setCornerBoundaryCondition(cornerBC,side1,side2,side3);
```

Here `side1`,`side2`,`side3` equal one of $= -1, 0, 1$. If all three values are from $0, 1$ then this defines a corner. If one of the values is -1 then this defines an edge along that axis.

The `evenSymmetry` boundary condition sets

$$u(i1-m1, i2-m2, i3-m3) = u(i1+m1, i2+m2, i3+m3)$$

where $u(i1, i2, i3)$ is a point on the boundary. The `oddSymmetry` boundary condition sets

$$u(i1-m1, i2-m2, i3-m3) = 2u(i1, i2, i3) - u(i1+m1, i2+m2, i3+m3)$$

The `taylor2ndOrderEvenCorner` boundary condition uses (in 2D)

$$\begin{aligned}
u(+1,+1) &= u(0,0) + \Delta r u_r + \Delta s u_s + \Delta r^2/2 u_{rr} + \Delta r \Delta s u_{rs} + \Delta s^2/2 u_{ss} + \ldots \\
u(-1,-1) &= u(0,0) - \Delta r u_r - \Delta s u_s + \Delta r^2/2 u_{rr} + \Delta r \Delta s u_{rs} + \Delta s^2/2 u_{ss} + \ldots \\
u(-1,-1) &= u(1,1) - 2\Delta r u_r - 2\Delta s u_s + O(\Delta r^3 + \ldots) \\
u_r &= (u(1,0) - u(-1,0))/(2\Delta r) + O(\Delta r^2)
\end{aligned}$$

to give the approximation

$$u(-1,-1) = u(1,1) - (u(1,0) - u(-1,0)) - (u(0,1) - u(0,-1))$$

The `taylor2ndOrderEvenCorner` boundary condition will reduce to an even symmetry boundary condition if the neighbouring points also satisfy the symmetry condition (i.e. if the function is even about the boundary).

The `taylor4thOrderEvenCorner` boundary condition is a fourth-order accurate boundary condition for the ghost points that reduces to an an even symmetry boundary condition if the neighbouring points also satisfy the symmetry condition. See the ogmg documentation for a further details on the `taylor2ndOrderEvenCorner` and `taylor4thOrderEvenCorner` boundary conditions.

4.5 BoundaryConditionParameters : passing optional parameters for boundary conditions

Use this class to pass optional parameters to the boundary condition routines. See section (4.1) for an example code that demonstrates the use of this class.

4.5.1 Applying a boundary condition to a portion of a boundary

Normally a boundary condition is applied to the whole side (or face). To apply a given boundary condition to only some part of a side one can use the `mask` array that lives in the BoundaryConditionParameters object.

4.5.2 constructor

BoundaryConditionParameters()

Description: This class is used to pass optional parameters to the boundary condition routines.

Optional parameters: The following parameters are public members of this class:

- **int lineToAssign:** apply Dirichlet BC on this line.
- **int orderOfExtrapolation:** order of extrapolation for various BC's. A value ¡ 0 means use orderOfExtrapolation=3 for 2nd-order accuracy and orderOfExtrapolation=5 for fourth order
- **int orderOfInterpolation:** not used yet(?)
- **int ghostLineToAssign:** assign this ghost line (various bc's)
- **extraInTangentialDirections:** extend the set of pointts assigned by this many points in the tangential directions
- **numberOfCornerGhostLinesToAssign:** assign at most this many lines at edges and corners, by default do all. For a second order method that only uses one ghost line one could set this value to 1 to avoid assigning any unused ghost points. NOTE: Some BC's may still assign all ghost points, this is only used as a recommendation.

cornerExtrapolationOption: by default (=0) corner points are extrapolated along diagonals. Setting this parameter to 1,2 or 3 means corner points are not extrapolated along direction 1,2, or 3. This option was introduced to keep some symmetries in 3d computations.

IntegerArray components: holds components to assign for various BC's

IntegerArray uComponents,fComponents: holds components to assign for various BC's

RealArray a,b0,b1,b2,b3: hold parameters for various BC's

int useMask : if TRUE use the mask (below) to determine where boundary conditions should be applied.

IntegerArray mask : supply a mask array to indicate where the BC's should be applied. This array is only used if useMask=TRUE.

Example: This example shows how to extrapolate to order 4:

```
BoundaryConditionParameters bcParams;
bcParams.orderOfExtrapolation=4;
...
int wall=3;
real value=0., time=0.;
u.applyBoundaryCondition(0,BCTypes::extrapolate,wall,value,time,bcParams);
....
```

4.5.3 setCornerBoundaryCondition

int
setCornerBoundaryCondition(CornerBoundaryConditionEnum bc)

Description: Specify the boundary conditions for the corners and edges.

bc (input) : use this boundary condition on all corners and edges.

Notes: For a vectorSymmetryCorner, use setVectorSymmetryCornerComponent(component) to indicate which components form the "vector" for the vector symmetry corner BC (e.g. where the velocity components start in the list of components) In 3D for example the vector symmetry will be applied to the set of components: [component,component+1,component+2] with all other components set by even symmetry

4.5.4 setCornerBoundaryCondition

int
setCornerBoundaryCondition(CornerBoundaryConditionEnum bc, int side1, int side2, int side3 = -1)

Description: Specify the boundary conditions for the corners and edges.

bc (input) : use this boundary condition on the specified corner or edge.

side1,side2,side3 (input): To indicate a corner, each of side1,side2, and side3 should be either 0 or 1; the corner will then be $(r_1 = side1, r_2 = side2, r_3 = side3)$. To indicate an edge set one of side1,side2,side3 to be -1 and the others to be 0 or 1. If side1==-1 then the edge will be parallel to axis1 : $(r_1 = [0,1], r_2 = side2, r_3 = side3)$. if side2==-1 then the edge will be parallel to axis2 : $(r_1 = side, r_2 = [0,1], r_3 = side3)$ etc.

Notes: For a vectorSymmetryCorner, use setVectorSymmetryCornerComponent(component) to indicate which components form the "vector" for the vector symmetry corner BC (e.g. where the velocity components start in the list of components) In 3D for example the vector symmetry will be applied to the set of components: [component,component+1,component+2] with all other components set by even symmetry

4.5.5 cornerBoundaryCondition

CornerBoundaryConditionEnum
getCornerBoundaryCondition(int side1, int side2, int side3 = -1) const

Description: Return the boundary condition that applies to a corner or edge.

side1,side2,side3 : Values of (0,0,0) would be a corner, (0,0,-1) would be an edge

4.5.6 setVectorSymmetryCornerComponent

int
setVectorSymmetryCornerComponent(int component)

Description: Indicate which components form the "vector" for the vector symmetry corner BC (e.g. where the velocity components start in the list of components) In 3D for example the vector symmetry will be applied to the set of components: [component,component+1,component+2] with all other components set by even symmetry

4.5.7 getVectorSymmetryCornerComponent

int
getVectorSymmetryCornerComponent() const

Description: Return the component that indicates the first component of the "vector" for the vector symmetry corner BC

4.5.8 setUseMask

int
setUseMask(int trueOrFalse =TRUE)

Description: Turn on (or off) the use of the mask array for selectively applying boundary conditions at certain points.

4.5.9 getUseMask

int
getUseMask() const

Description: Return the current value of the useMask flag.

4.5.10 mask()

intArray &
mask()

Description: Return a reference to the boundary condition mask array. It is up to the user to dimension this array to be the correct size.

If setUseMask(true) has been called then any boundary condition will only be applied where the mask array has non-zero values.

The applyBoundaryCondition routine will evaluate the mask on a given side according to the value of bcParameters.lineToAssign, by default this will be the boundary itself.

```
getGhostIndex( c.indexRange(),side,axis,I1,I2,I3,bcParameters.lineToAssign);
where( mask(I1,I2,I3) )
    apply the boundary condition
```

4.5.11 useMixedBoundaryMask

int
assignAllPointsOnMixedBoundaries(bool trueOrFalse =true)

Description: Boundary conditions on mixed boundaries are normally NOT assigned at interior boundary points, unless you call this function with "true"

trueOrFalse : set to true if you want all points to be assigned on mixed boundaries (boundaries that are partially boundary points and partially interpolation points). The default is false.

4.5.12 getVariableCoefficients

RealMappedGridFunction*
getVariableCoefficients() const

Description: Return a pointer to the grid function that was previously supplied through a call to `setVariableCoefficients( RealMappedGridFunction & var )`. Do not use this version if you initially passed a grid collection function.

4.5.13 getVariableCoefficients

RealMappedGridFunction*
getVariableCoefficients(const int & grid) const

Description: Return a pointer to the grid function that was previously supplied through a call to `setVariableCoefficients( RealGridCollectionFunction & var )`.

grid (input) : return the mappedGridFunction for this component grid.

4.5.14 setVariableCoefficients

void
setVariableCoefficients(RealMappedGridFunction & var)

Description: Supply a grid function for variable coefficients. The meaning of the grid function depends on the boundary condition to which it is applied. A reference to 'var' will be kept.

var (input) : coefficient values for a boundary condition that requires variable coefficients. This grid function could only live on a single boundary if there is only one boundary where the values are needed.

4.5.15 setVariableCoefficients

void
setVariableCoefficients(RealGridCollectionFunction & var)

Description: Supply a grid function for variable coefficients. The meaning of the grid function depends on the boundary condition to which it is applied. A reference to 'var' will be kept. **NOTE:** This grid function will take precedence over any variable coefficients specified through the `setVariableCoefficients( RealMappedGridFunction & var )`, i.e. A `GridCollectionFunction` will be used before a `MappedGridFunction`.

var (input) : coefficient values for a boundary condition that requires variable coefficients. This grid function could only live on a single boundary if there is only one boundary where the values are needed.

4.5.16 setRefinementLevelToSolveFor

void
setRefinementLevelToSolveFor(int level)

Description:

level (input) : indicate that a particular refinement level is being solved for.

4.5.17 setBoundaryConditionForcingOption

int
setBoundaryConditionForcingOption(BoundaryConditionForcingOption option)

Description:

option (input) : specify the form of the right-hand-sde for the boundary condition.

4.5.18 getBoundaryConditionForcingOption

BoundaryConditionForcingOption
getBoundaryConditionForcingOption() const

Description:

Return value: the form of the right-hand-sde for the boundary condition.

4.6 How to write your own boundary conditions

If you need to assign a boundary condition that is not of the form of one of the implemented elementary boundary conditions then you can write a loop something like the following

```
Index Ib1,Ib2,Ib3, I1g,I2g,I3g, I1p,I2p,I3p;
int myBoundaryCondition = ...;

// apply Boundary conditions
for( int axis=0; axis<mg.numberOfDimensions; axis++ )
  for( int side=Start; side<=End; side++ )
  { // apply a BC :
    if( mg.boundaryCondition(side,axis) == myBoundaryCondition )
    { // Index's for boundary values:
      getBoundaryIndex(mg.gridIndexRange,side,axis,Ib1,Ib2,Ib3);
      // Index's for first ghost line
      getGhostIndex(mg.gridIndexRange,side,axis,Ig1,Ig2,Ig3,1);
      // Index's for first interior line
      getGhostIndex(mg.gridIndexRange,side,axis,Ip1,Ip2,Ip3,-1);

      u(Ib1,Ib2,Ib3)=...;             // set boundary values
      u(Ig1,Ig2,Ig3)=u(Ip1,Ip2,Ip3);  // set ghost values to first line in
    }
  }
```

5 Implicit operators and Coefficient Matrices

The `MappedGridOperator` functions such as `laplacianCoefficient`, `xCoefficient` etc. generate a "coefficient-matrix" (sparse matrix representation) for the indicated operator. In this section we describe how coefficient-matrices can be created to define a system of equations for a PDE boundary-value problem.

To create a coefficient matrix you should create a grid function in the following way

```
  MappedGrid mg; // from somewhere
  int stencilSize=9;       // number of points in the stencil, 9 points assuming 2D
  realMappedGridFunction coeff(mg,stencilSize,all,all,all);
  coeff.setIsACoefficientMatrix(TRUE,stencilSize);
```

From this declaration we see the the elements of the stencil are stored as

```
  coeff(m,I1,I2,I3)  m=0,1,...,stencilSize-1
where
  Index I1,I2,I3   : Index's for the grid function coordinate dimensions
```

Thus all the coefficients of the stencil are stored in the first component. For example, a nine point approximation to the Laplace operator might be stored as

```
  coeff(6,I1,I2,I3)=0    coeff(7,I1,I2,I3)=1    coeff(8,I1,I2,I3)=0
  coeff(3,I1,I2,I3)=1    coeff(4,I1,I2,I3)=-4   coeff(5,I1,I2,I3)=1
  coeff(0,I1,I2,I3)=0    coeff(1,I1,I2,I3)=1    coeff(2,I1,I2,I3)=0
```

The typical user will not need to know exactly how the coefficients are stored (indeed, there is more than one storage format). This *representation* of the sparse matrix should really be hidden. It is useful, however, to have an idea of the format of the matrix coefficient array. The actual representation is stored in an object of type `SparseRep`. See section 5.7 for more details.

Once a coefficient-matrix grid-function has been declared, the sparse matrix representing a PDE boundary value problem can be formed as follows

```
MappedGridOperators op(mg);                              // create some differential operators
op.setStencilSize(stencilSize);
coeff.setOperators(op);

coeff=op.laplacianCoefficients();                        // get the coefficients for the Laplace operator
// fill in the coefficients for the boundary conditions
coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,allBoundaries);      // equations on boundary
coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries);    // equations on the ghost line
coeff.finishBoundaryConditions();
```

In this example we form the Laplace operator with Dirichlet boundary conditions. By default one ghost-line is used so we must supply equations there. (See the description of the `MappedGridFunction` member function `setIsACoefficientMatrix` for details on how to change the number of ghostlines that are used.) The `coeff` grid function can be give to a sparse matrix solver, such as `Oges`. See the examples for more details.

Let us consider, in a bit more detail, what happens in the above example. Let us suppose that the we are dealing with a simple one-dimensional grid corresponding to a line on the unit interval and that we have one ghost line value. After the line `coeff=op.laplacianCoefficients();` is executed the sparse matrix will be filled in (at all interior points and boundary points) with a discrete approximation to the Laplacian, resulting in a (sparse) representation for the following matrix

$$\begin{array}{l l}
\begin{bmatrix}
0 & 0 & 0 & \dots & & & \\
\frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & & \\
0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & \\
0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \\
\vdots & \vdots & & \ddots & \ddots & \ddots & \\
 & & & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\
 & & & \dots & 0 & 0 & 0
\end{bmatrix}
&
\begin{array}{l}
i=-1 \quad \text{(ghostline)} \\
i=0 \\
i=1 \\
i=2 \\
\vdots \\
i=N \\
i=N+1 \quad \text{(ghostline)}
\end{array}
\end{array}$$

So far no equation is applied at the ghost lines (first and last rows). Internally this matrix is stored in a sparse fashion with only 3 values stored per row (actually we need 4 values per row in 1D since the extrapolation equations below use 4 points by default). After the dirichlet boundary condition is applied with `coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,allBoundaries);` the equation on the boundary will be replaced with the identity operator. The resulting matrix is

$$\begin{array}{l l}
\begin{bmatrix}
0 & 0 & 0 & \dots & & & \\
0 & 1 & 0 & 0 & \dots & & \\
0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & \\
0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \\
\vdots & \vdots & & \ddots & \ddots & \ddots & \\
 & & & 0 & 0 & 1 & 0 \\
 & & & \dots & 0 & 0 & 0
\end{bmatrix}
&
\begin{array}{l}
i=-1 \quad \text{(ghostline)} \\
i=0 \\
i=1 \\
i=2 \\
\vdots \\
i=N \\
i=N+1 \quad \text{(ghostline)}
\end{array}
\end{array}$$

Finally the values at the ghost points are assigned using extrapolation,

$$\begin{array}{l l}
\begin{bmatrix}
1 & -3 & 3 & -1 & & & \\
0 & 1 & 0 & 0 & \dots & & \\
0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & \\
0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \\
\vdots & \vdots & & \ddots & \ddots & \ddots & \\
 & & 0 & 0 & 0 & 1 & 0 \\
 & & 0 & -1 & 3 & -3 & 1
\end{bmatrix}
&
\begin{array}{l}
i=-1 \quad \text{(ghostline)} \\
i=0 \\
i=1 \\
i=2 \\
\vdots \\
i=N \\
i=N+1 \quad \text{(ghostline)}
\end{array}
\end{array}$$

If we wanted to apply a Neumann boundary condition we could have said

```
coeff=op.laplacianCoefficients();                          // get the coefficients for the Laplace operator
coeff.applyBoundaryConditionCoefficients(0,0,neumann,allBoundaries);   // equations on the ghost line
coeff.finishBoundaryConditions();
```

which would result in the following matrix:

$$\begin{array}{l l}
\begin{bmatrix}
\frac{1}{2h} & 0 & -\frac{1}{2h} & 0 & \dots & & \\
\frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & & \\
0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & \\
0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \\
\vdots & \vdots & & \ddots & \ddots & \ddots & \\
 & & & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\
 & & \dots & 0 & -\frac{1}{2h} & 0 & \frac{1}{2h}
\end{bmatrix}
&
\begin{array}{l}
i=-1 \quad \text{(ghostline)} \\
i=0 \\
i=1 \\
i=2 \\
\vdots \\
i=N \\
i=N+1 \quad \text{(ghostline)}
\end{array}
\end{array}$$

Note that the equation is applied on the boundary and the Neumann condition is the equation that sits a the ghost line.

Given one of the above matrices it is now apparent how we must fill-in the right-hand-side function when we are going to solve a problem. In the dirichlet boundary condition case we should give the RHS for the Laplace operator, $u_{xx} = f(x)$ at all interior points and the dirichlet BC values, $u = g(x)$, on the boundary (by default the Oges solver will fill in zero values at all extrapolation equations, otherwise we would have to set the ghost line values to zero).

$$\begin{bmatrix} 1 & -3 & 3 & -1 & & & \\ 0 & 1 & 0 & 0 & \dots & & \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & \\ 0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \\ \vdots & \vdots & & \ddots & \ddots & \ddots & \\ & & 0 & 0 & 0 & 1 & 0 \\ & & 0 & -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} u_{-1} \\ u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_N \\ u_{N+1} \end{bmatrix} = \begin{bmatrix} 0 \\ g(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ g(x_N) \\ 0 \end{bmatrix}$$

In the neumann case we should give the RHS for the Laplace operator at the interior **and** the boundary and we should give the RHS for the neumann condition, $\partial u/\partial n = k(x)$, at the ghost line. In this case the RHS vector would look like

$$\begin{bmatrix} \frac{1}{2h} & 0 & -\frac{1}{2h} & 0 & & & \\ 0 & 1 & 0 & 0 & \dots & & \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & \\ 0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \\ \vdots & \vdots & & \ddots & \ddots & \ddots & \\ & & 0 & 0 & 0 & 1 & 0 \\ & & 0 & 0 & -\frac{1}{2h} & 0 & \frac{1}{2h} \end{bmatrix} \begin{bmatrix} u_{-1} \\ u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_N \\ u_{N+1} \end{bmatrix} = \begin{bmatrix} k(x_0) \\ f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_N) \\ k(x_N) \end{bmatrix}$$

For a system of equations the situation is a bit more complicated but as for a single equation all coefficients of the stencil appear in the first component.

```
MappedGridOperators op(mg);                         // create some operators
op.setStencilSize(stencilSize);
op.setNumberOfComponentsForCoefficients(numberOfComponentsForCoefficients);
coeff.setOperators(op);

// Form a system of equations for (u,v)
//     a1(  u_xx + u_yy ) + a2*v_x = f_0
//     a3(  v_xx + v_yy ) + a4*u_y = f_1
//  BC's:   u=given   on all boundaries
//          v=given   on inflow
//          v.n=given on walls
const int a1=1., a2=2., a3=3., a4=4.;
//  const int a1=1., a2=0., a3=1., a4=0.;

coeff=a1*op.laplacianCoefficients(all,all,all,0,0)+a2*op.xCoefficients(all,all,all,0,1)
     +a3*op.laplacianCoefficients(all,all,all,1,1)+a4*op.yCoefficients(all,all,all,1,0);

coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,  allBoundaries);
coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries);
// coeff.display("Here is coeff after dirichlet/extrapolate BC's for (0) ");
```

```
coeff.applyBoundaryConditionCoefficients(1,1,dirichlet,  inflow);
coeff.applyBoundaryConditionCoefficients(1,1,extrapolate,inflow);
coeff.applyBoundaryConditionCoefficients(1,1,neumann,     wall);
// coeff.display("Here is coeff with dirichlet (0) and neumann BC's on wall (1)");

coeff.finishBoundaryConditions();
```

See example 2 for more details.

5.1 Poisson's equation on a MappedGrid

In this example we solve Poisson's equation on a MappedGrid (file `Overture/examples/tcm.C`)

```
//================================================================================
//  Coefficient Matrix Example
//    Solve Poisson's equation on a MappedGrid
//      o first solve with Dirichlet BC's
//      o secondly solve with Dirichlet on some sides and Neumann on others
//===============================================================================
#include "Overture.h"
#include "MappedGridOperators.h"
#include "Oges.h"
#include "SquareMapping.h"
#include "OGPolyFunction.h"

#define ForBoundary(side,axis)   for( axis=0; axis<mg.numberOfDimensions(); axis++ ) \
                                 for( side=0; side<=1; side++ )
int
main(int argc, char *argv[])
{
  Overture::start(argc,argv);  // initialize Overture

  int n=11;
  // cout << "Enter Oges::debug, n (number of grid lines)\n";
  // cin >> Oges::debug >> n;

  // make some shorter names for readability
  BCTypes::BCNames dirichlet             = BCTypes::dirichlet,
                   neumann               = BCTypes::neumann,
                   extrapolate           = BCTypes::extrapolate,
                   allBoundaries         = BCTypes::allBoundaries;

  SquareMapping map;
  int numberOfGridLines=n;
  map.setGridDimensions(axis1,numberOfGridLines);
  map.setGridDimensions(axis2,numberOfGridLines);

  MappedGrid mg(map);
  int side;
  for( side=Start; side<=End; side++ )
  {
    mg.setNumberOfGhostPoints(side,axis1,2);
  }
  mg.update(MappedGrid::THEvertex | MappedGrid::THEcenter | MappedGrid::THEvertexBoundaryNormal);
  // label boundary conditions
  const int inflow=1, outflow=2, wall=3;
  mg.setBoundaryCondition(Start,axis1,inflow);
  mg.setBoundaryCondition(End  ,axis1,outflow);
  mg.setBoundaryCondition(Start,axis2,wall);
  mg.setBoundaryCondition(End  ,axis2,wall);

  // create a twilight-zone function for checking errors
  int degreeOfSpacePolynomial = 2;
  int degreeOfTimePolynomial = 1;
  int numberOfComponents = mg.numberOfDimensions();
  OGPolyFunction exact(degreeOfSpacePolynomial,mg.numberOfDimensions(),numberOfComponents,
                       degreeOfTimePolynomial);



  // make a grid function to hold the coefficients
  Range all;
  int stencilSize=int( pow(3,mg.numberOfDimensions()) );
  realMappedGridFunction coeff(mg,stencilSize,all,all,all);
  coeff.setIsACoefficientMatrix(TRUE,stencilSize);

  // create grid functions:
  realMappedGridFunction u(mg),f(mg);

  MappedGridOperators op(mg);                              // create some differential operators
  op.setStencilSize(stencilSize);
  coeff.setOperators(op);

  coeff=op.laplacianCoefficients();       // get the coefficients for the Laplace operator
  if( Oges::debug & 64 )
    coeff.display("Here is coeff=laplacianCoefficients");

  // fill in the coefficients for the boundary conditions
  coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,allBoundaries);
  coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries);
  coeff.finishBoundaryConditions();

  Oges solver( mg );                      // create a solver
  solver.setCoefficientArray( coeff );   // supply coefficients

  // assign the rhs: u.xx+u.yy=f, u=exact on the boundary
  Index I1,I2,I3, Ia1,Ia2,Ia3;
  getIndex(mg.indexRange(),I1,I2,I3);

  f(I1,I2,I3)=exact.xx(mg,I1,I2,I3,0)+exact.yy(mg,I1,I2,I3,0);
  int axis;
  Index Ib1,Ib2,Ib3;
  ForBoundary(side,axis)
  {
    if( mg.boundaryCondition(side,axis) > 0 )
    {
      getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);
      f(Ib1,Ib2,Ib3)=exact(mg,Ib1,Ib2,Ib3,0);
    }
  }

  solver.solve( u,f );   // solve the equations

  // u.display("Here is the solution to u.xx+u.yy=f");
  real error=0.;
  error=max(error,max(abs(u(I1,I2,I3)-exact(mg,I1,I2,I3,0))));
  printf("Maximum error with dirichlet bc's= %e\n",error);


  // -----------------------
  // ----- Neumann BC's ----
  // -----------------------

  mg.setBoundaryCondition(Start,axis1,wall);
  mg.setBoundaryCondition(End  ,axis1,wall);
  mg.setBoundaryCondition(Start,axis2,wall);
  mg.setBoundaryCondition(End  ,axis2,wall);

  coeff=op.laplacianCoefficients();       // get the coefficients for the Laplace operator
  // fill in the coefficients for the boundary conditions
  coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,  inflow);

  coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,inflow);

  coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,  outflow);
  coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,outflow);

  coeff.applyBoundaryConditionCoefficients(0,0,neumann,    wall);
  coeff.finishBoundaryConditions();

  f(I1,I2,I3)=exact.xx(mg,I1,I2,I3,0)+exact.yy(mg,I1,I2,I3,0);

  Index Ig1,Ig2,Ig3;
  bool singularProblem=TRUE;
  ForBoundary(side,axis)
  {
    if( mg.boundaryCondition()(side,axis) ==wall )
    { // for Neumann BC's -- fill in f on first ghostline
      getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);
      getGhostIndex(mg.gridIndexRange(),side,axis,Ig1,Ig2,Ig3);
      realArray & normal = mg.vertexBoundaryNormal(side,axis);
      if( mg.numberOfDimensions()==2 )
        f(Ig1,Ig2,Ig3)=
            normal(Ib1,Ib2,Ib3,0)*exact.x(mg,Ib1,Ib2,Ib3,0)
           +normal(Ib1,Ib2,Ib3,1)*exact.y(mg,Ib1,Ib2,Ib3,0);
      else
        f(Ig1,Ig2,Ig3)=
            normal(Ib1,Ib2,Ib3,0)*exact.x(mg,Ib1,Ib2,Ib3,0)
           +normal(Ib1,Ib2,Ib3,1)*exact.y(mg,Ib1,Ib2,Ib3,0)
           +normal(Ib1,Ib2,Ib3,2)*exact.z(mg,Ib1,Ib2,Ib3,0);
    }
    else if( mg.boundaryCondition()(side,axis) ==inflow ||  mg.boundaryCondition()(side,axis) ==outflow
    {
      singularProblem=FALSE;
      getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);
      f(Ib1,Ib2,Ib3)=exact(mg,Ib1,Ib2,Ib3,0);
    }
  }

  // if the problem is singular Oges will add an extra constraint equation to make the system nonsingula
  if( singularProblem )
    solver.set(OgesParameters::THEcompatibilityConstraint,TRUE);
  // Tell the solver to refactor the matrix since the coefficients have changed
  solver.setRefactor(TRUE);
  // we need to reorder too because the matrix changes a lot for the singular case
  solver.setReorder(TRUE);

  if( singularProblem )
  {
    // we need to first initialize the solver before we can fill in the rhs for the compatbility equatic
    solver.initialize();
    int ne,i1e,i2e,i3e,gride;
    solver.equationToIndex( solver.extraEquationNumber(0),ne,i1e,i2e,i3e,gride);
    getIndex(mg.dimension(),I1,I2,I3);
    f(i1e,i2e,i3e)=sum(solver.rightNullVector[0](I1,I2,I3)*exact(mg,I1,I2,I3,0,0.));
  }

  solver.solve( u,f );   // solve the equations
  getIndex(mg.indexRange(),Ia1,Ia2,Ia3,1);  // include ghost points
  // mg.indexRange().display("Here is mg.indexRange()");
  // Ia1.display("Here is Ia1");

  error=max(error,max(abs(u(Ia1,Ia2,Ia3)-exact(mg,Ia1,Ia2,Ia3,0))));
  // abs(u(Ia1,Ia2,Ia3)-exact(mg,Ia1,Ia2,Ia3,0)).display("abs(error)");
  printf("Maximum error with neumann bc's= %e\n",error);


  Overture::finish();
  return(0);
}

```

5.2 Systems of Equations on a MappedGrid

In the general case one can define a matrix for a boundary-value problem for a system of equations....

In this example we generate the matrix corresponding to the following system of equations

$$\begin{aligned} a_1\Delta u + a_2 v_x - u &= g_0 \\ a_3\Delta v + a_4 u_y &= g_1 \\ u = g_0 \quad , \quad v_n &= g_1 \quad \text{on the boundary} \end{aligned}$$

Note the use of the `identityCoefficents` operator.

(file `Overture/examples/tcm2.C`)

```
//===============================================================================
//  Coefficient Matrix Example
//    Solve a system of equations on a MappedGrid
//==============================================================================
#include "Overture.h"
#include "MappedGridOperators.h"
#include "Oges.h"
#include "SquareMapping.h"
#include "AnnulusMapping.h"
#include "OGPolyFunction.h"
#include "display.h"

#define ForBoundary(side,axis)   for( axis=0; axis<mg.numberOfDimensions(); axis++ ) \
                                for( side=0; side<=1; side++ )
int
main(int argc, char *argv[])
{
  Overture::start(argc,argv);  // initialize Overture
  // cout << "Enter Oges::debug\n";   cin >> Oges::debug;

  // make some shorter names for readability
  BCTypes::BCNames dirichlet             = BCTypes::dirichlet,
                   neumann               = BCTypes::neumann,
                   extrapolate           = BCTypes::extrapolate,
                   allBoundaries         = BCTypes::allBoundaries;

  //  AnnulusMapping map;   // switch this with the line below to get an Annulus
  SquareMapping map;
  map.setGridDimensions(axis1,5);
  map.setGridDimensions(axis2,5);

  MappedGrid mg(map);
  mg.update(MappedGrid::THEvertex | MappedGrid::THEcenter | MappedGrid::THEvertexBoundaryNormal);

  // label boundary conditions
  const int inflow=1, wall=2;
  mg.boundaryCondition()(Start,axis1)=inflow;
  mg.boundaryCondition()(End  ,axis1)=inflow;
  mg.boundaryCondition()(Start,axis2)=wall;
  mg.boundaryCondition()(End  ,axis2)=wall;

  // create a twilight-zone function for checking errors
  int degreeOfSpacePolynomial = 2;
  int degreeOfTimePolynomial = 1;
  int numberOfComponents = mg.numberOfDimensions();
  OGPolyFunction exact(degreeOfSpacePolynomial,mg.numberOfDimensions(),numberOfComponents,
                       degreeOfTimePolynomial);

  // make a grid function to hold the coefficients
  Range all;
  int stencilSize=int( pow(3,mg.numberOfDimensions()) );
  int numberOfComponentsForCoefficients=2;
  int stencilDimension=stencilSize*SQR(numberOfComponentsForCoefficients);
  realMappedGridFunction coeff(mg,stencilDimension,all,all,all);
  // make this grid function a coefficient matrix:
  int numberOfGhostLines=1;  // we will solve for values including the first ghostline
  coeff.setIsACoefficientMatrix(TRUE,stencilSize,numberOfGhostLines,numberOfComponentsForCoefficients);
  coeff=0.;

  MappedGridOperators op(mg);                            // create some operators
  op.setStencilSize(stencilSize);
  op.setNumberOfComponentsForCoefficients(numberOfComponentsForCoefficients);
  coeff.setOperators(op);

  // Form a system of equations for (u,v)
  //     a1(  u_xx + u_yy ) + a2*v_x - u = f_0
  //     a3(  v_xx + v_yy ) + a4*u_y     = f_1
  //  BC's:   u=given   on all boundaries
  //          v=given   on inflow
  //          v.n=given on walls
  const real a1=1., a2=2., a3=3., a4=4.;
  // const real a1=1., a2=0., a3=1., a4=0.;

  const int eqn0=0;    // labels equation 0
  const int eqn1=1;    // labels equation 1
  const int uc=0, vc=1;  // labels for the u and v components
  coeff=a1*op.laplacianCoefficients(all,all,all,eqn0,uc)+a2*op.xCoefficients(all,all,all,eqn0,vc)
         -op.identityCoefficients(all,all,all,eqn0,uc)
      +a3*op.laplacianCoefficients(all,all,all,eqn1,vc)+a4*op.yCoefficients(all,all,all,eqn1,uc);
  if( Oges:: debug & 4 )
    display(coeff,"Here is coeff after assigning interior equations ","%5.2f ");

  coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,  allBoundaries);
  coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries);
  if( Oges:: debug & 4 )
    display(coeff,"Here is coeff after dirichlet/extrapolate BC's for (0) ","%5.2f ");

  coeff.applyBoundaryConditionCoefficients(1,1,dirichlet,  inflow);
  coeff.applyBoundaryConditionCoefficients(1,1,extrapolate,inflow);
  coeff.applyBoundaryConditionCoefficients(1,1,neumann,     wall);

  if( Oges:: debug & 4 )
    display(coeff,"Here is coeff with dirichlet (0) and neumann BC's on wall (1)","%5.2f ");

  coeff.finishBoundaryConditions();

  realMappedGridFunction u(mg,all,all,all,2),f(mg,all,all,all,2);

  Oges solver( mg );                      // create a solver
  solver.setCoefficientArray( coeff );   // supply coefficients to solver

  // assign the right-hand-side
  Index I1,I2,I3;
  getIndex(mg.indexRange(),I1,I2,I3);
  f(I1,I2,I3,0)=a1*(exact.xx(mg,I1,I2,I3,0)+exact.yy(mg,I1,I2,I3,0))+a2*exact.x(mg,I1,I2,I3,1)-exact(mg,
  f(I1,I2,I3,1)=a3*(exact.xx(mg,I1,I2,I3,1)+exact.yy(mg,I1,I2,I3,1))+a4*exact.y(mg,I1,I2,I3,0);

  int side,axis;
  Index Ib1,Ib2,Ib3;
  Index Ig1,Ig2,Ig3;
  ForBoundary(side,axis)
  {
    if( mg.boundaryCondition()(side,axis) > 0  )
    {
      getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);
      f(Ib1,Ib2,Ib3,0)=exact(mg,Ib1,Ib2,Ib3,0);
      if( mg.boundaryCondition()(side,axis)==inflow )
      {
        f(Ib1,Ib2,Ib3,1)=exact(mg,Ib1,Ib2,Ib3,1);
      }
      else
      {
        // for Neumann BC's -- fill in f on first ghostline
        getGhostIndex(mg.gridIndexRange(),side,axis,Ig1,Ig2,Ig3);
        realArray & normal = mg.vertexBoundaryNormal(side,axis);
        if( mg.numberOfDimensions()==2 )
          f(Ig1,Ig2,Ig3,1)=
            normal(Ib1,Ib2,Ib3,0)*exact.x(mg,Ib1,Ib2,Ib3,1)
              +normal(Ib1,Ib2,Ib3,1)*exact.y(mg,Ib1,Ib2,Ib3,1);
        else
          f(Ig1,Ig2,Ig3,1)=
            normal(Ib1,Ib2,Ib3,0)*exact.x(mg,Ib1,Ib2,Ib3,1)
              +normal(Ib1,Ib2,Ib3,1)*exact.y(mg,Ib1,Ib2,Ib3,1)
                +normal(Ib1,Ib2,Ib3,2)*exact.z(mg,Ib1,Ib2,Ib3,1);
      }
    }
  }

  if( Oges:: debug & 4 )
    display(f,"Here is the rhs");

  solver.solve( u,f );   // solve the equations

  getIndex(mg.gridIndexRange(),I1,I2,I3,1);

  display(u,"Here is the solution u","%5.2f ");

  if( Oges:: debug & 4 )
    display(exact(mg,I1,I2,I3,Range(0,1)),"Here is the exact solution");

  for( int n=0; n<numberOfComponentsForCoefficients; n++ )
  {

    real error=0.;
    display(evaluate(abs(u(I1,I2,I3,n)-exact(mg,I1,I2,I3,n))),"Error including ghost points","%6.2e ");

    error=max(error,max( abs(u(I1,I2,I3,n)-exact(mg,I1,I2,I3,n))));
    printf("Maximum error for component %i is = %e\n",n,error);
  }


  Overture::finish();
  return(0);
}
```

5.3 Poisson's equation on a CompositeGrid

In this example we solve Poisson's eqution on a CompositeGrid (file `Overture/examples/tcm3.C`)

```
//===============================================================================
//  Coefficient Matrix Example
//    Using Oges to solve Poisson's equation on a CompositeGrid
//
// Usage: `tcm3 [<gridName>] [-solver=[yale][harwell][slap][petsc][mg]] [-debug=<value>] [-outputMatrix]
//               [-noTiming] [-check] [-plot] [-trig] [-tol=<value>] [-freq=<value>] [-dirichlet] [-neuma
//
//   The -check option is used for regression testing -- it will test various solvers on a few grids
//
// NOTE:
// To get PETSc log info, compile PETScEquationSolver with the destructor calling PetscFinalize()
//   and use the command line arg -log_summary
//   memory usage: -trmalloc_log
//
// Parallel examples:
```

```
//   mpirun -np 2 tcm3 square20.hdf -solver=petsc
//   mpirun -np 2 tcm3 cic.hdf -solver=petsc
//   srun -N1 -n1 -ppdebug tcm3 square20.hdf -solver=petsc
//   srun -N1 -n2 -ppdebug tcm3 sibe2.order2.hdf -solver=petsc
//==================================================================================
#include "Overture.h"
#include "MappedGridOperators.h"
#include "Oges.h"
#include "CompositeGridOperators.h"
#include "SquareMapping.h"
#include "AnnulusMapping.h"
#include "OGTrigFunction.h"
#include "OGPolyFunction.h"
#include "SparseRep.h"
#include "display.h"
#include "Ogmg.h"
#include "Checker.h"
#include "PlotStuff.h"
#include "ParallelUtility.h"
#include "LoadBalancer.h"

#define ForBoundary(side,axis)   for( int axis=0; axis<mg.numberOfDimensions(); axis++ ) \
                                 for( int side=0; side<=1; side++ )

bool measureCPU=TRUE;
real
CPU()
// In this version of getCPU we can turn off the timing
{
  if( measureCPU )
    return getCPU();
  else
    return 0;
}

void
plotResults( PlotStuff & ps, Oges & solver, realCompositeGridFunction & u, realCompositeGridFunction & e
// ========================================================================================
// Plot results from Oges
// ========================================================================================
{

  GraphicsParameters psp;

  aString answer;
  aString menu[]=
  {
    "solution",
    "error",
    "grid",
    "exit",
    ""
  };

  for( ;; )
  {
    ps.getMenuItem(menu,answer,"choose an option");
```

```
      if( answer=="exit" )
      {
        break;
      }
      else if( answer=="solution" )
      {
        psp.set(GI_TOP_LABEL,"Solution u");
        PlotIt::contour(ps,u,psp);
      }
      else if( answer=="error" )
      {
        psp.set(GI_TOP_LABEL,"error");
        PlotIt::contour(ps,err,psp);
      }
      else if( answer=="grid" )
      {
        psp.set(GI_TOP_LABEL,"grid");
        PlotIt::plot(ps,*u.getCompositeGrid(),psp);
      }

    }

  }


  int
  assignForcing(int option, CompositeGrid & cg, realCompositeGridFunction & f, OGFunction & exact )
  // ==============================================================================================
  //
  //  Assign the right-hand-side.
  //
  // /option (input) : 0 = dirichlet, 1=neumann
  //
  // ==============================================================================================
  {
    const int numberOfDimensions = cg.numberOfDimensions();

    Index I1,I2,I3;
    Index Ib1,Ib2,Ib3;
    Index Ig1,Ig2,Ig3;


    for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
    {
      MappedGrid & mg = cg[grid];
      // mg.mapping().getMapping().getGrid();
      // printF(" signForJacobian=%e\n",mg.mapping().getMapping().getSignForJacobian());
  #ifdef USE_PPP
      realSerialArray fLocal; getLocalArrayWithGhostBoundaries(f[grid],fLocal);
  #else
      realSerialArray & fLocal = f[grid];
  #endif

      getIndex(mg.indexRange(),I1,I2,I3);
      int includeGhost=1; // include parallel ghost pts in fLocal:
      bool ok = ParallelUtility::getLocalArrayBounds(f[grid],fLocal,I1,I2,I3,includeGhost);
      if( !ok ) continue; // there are no points on this processor.

      realArray & x= mg.center();
  #ifdef USE_PPP
      realSerialArray xLocal; getLocalArrayWithGhostBoundaries(x,xLocal);
  #else
      const realSerialArray & xLocal = x;
  #endif

      // Assign the forcing : f = e.xx + e.yy + e.zz (e=exact solution)
      RealArray ed(I1,I2,I3);
      const int rectangularForTZ=0;
      fLocal=0.;
      for( int axis=0; axis<cg.numberOfDimensions(); axis++ )
      {
        int ntd=0, nxd[3]={0,0,0}; //
        nxd[axis]=2;  // compute e.xx (axis=0), e.yy (axis=1), ...
        exact.gd( ed,xLocal,mg.numberOfDimensions(),rectangularForTZ,ntd,nxd[0],nxd[1],nxd[2],I1,I2,I3,0,0
        fLocal(I1,I2,I3)+=ed(I1,I2,I3);
      }


      ForBoundary(side,axis)
      {
        if( mg.boundaryCondition(side,axis) > 0 )
        {
          #ifdef USE_PPP
            const realSerialArray & normal  = mg.vertexBoundaryNormalArray(side,axis);
          #else
            const realSerialArray & normal  = mg.vertexBoundaryNormal(side,axis);
          #endif

          getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);

          bool ok = ParallelUtility::getLocalArrayBounds(f[grid],fLocal,Ib1,Ib2,Ib3,includeGhost);
          if( !ok ) continue; // there are no points on this processor.
          if( option==0 )
          {
            // Dirichlet BC's : assign the value of f on the boundary:
            RealArray ue(Ib1,Ib2,Ib3);
            exact.gd( ue,xLocal,mg.numberOfDimensions(),rectangularForTZ,0,0,0,0,Ib1,Ib2,Ib3,0,0.);
            fLocal(Ib1,Ib2,Ib3)=ue;
          }
          else
          {
            // Neumann BC's : assign the value of f on the ghost line:

            getGhostIndex(mg.gridIndexRange(),side,axis,Ig1,Ig2,Ig3);
            bool ok = ParallelUtility::getLocalArrayBounds(f[grid],fLocal,Ig1,Ig2,Ig3,includeGhost);
            if( !ok ) continue; // there are no points on this processor.

            realSerialArray uex(Ib1,Ib2,Ib3), uey(Ib1,Ib2,Ib3);
            exact.gd( uex,xLocal,numberOfDimensions,rectangularForTZ,0,1,0,0,Ib1,Ib2,Ib3,0,0.);
            exact.gd( uey,xLocal,numberOfDimensions,rectangularForTZ,0,0,1,0,Ib1,Ib2,Ib3,0,0.);

            fLocal(Ig1,Ig2,Ig3) = normal(Ib1,Ib2,Ib3,0)*uex + normal(Ib1,Ib2,Ib3,1)*uey;
            if( numberOfDimensions==3 )
            {
              exact.gd( uex,xLocal,numberOfDimensions,rectangularForTZ,0,0,0,1,Ib1,Ib2,Ib3,0,0.);  // uex
              fLocal(Ig1,Ig2,Ig3) +=normal(Ib1,Ib2,Ib3,2)*uex;
            }

          }

        }
      }
    }
    // f.applyBoundaryCondition(0,BCTypes::dirichlet,BCTypes::allBoundaries,0.);
    // f.display("Here is f");

    return 0;
  }

  int
  computeTheError( int option, CompositeGrid & cg, realCompositeGridFunction & u,
                   realCompositeGridFunction & err, OGFunction & exact, real & error )
  // ==========================================================================================
  //
  //  Compute the error in the solution.
  //
  // /option (input) : 0 = dirichlet, 1=neumann
  //
  // ==========================================================================================
  {

    err=0.;
    error=0.;
    real errorWithGhostPoints=0;
    Index I1,I2,I3;
    for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
    {
      MappedGrid & mg = cg[grid];
      realArray & x= mg.center();
  #ifdef USE_PPP
      realSerialArray xLocal; getLocalArrayWithGhostBoundaries(x,xLocal);
      realSerialArray uLocal; getLocalArrayWithGhostBoundaries(u[grid],uLocal);
      realSerialArray errLocal; getLocalArrayWithGhostBoundaries(err[grid],errLocal);
      intSerialArray maskLocal; getLocalArrayWithGhostBoundaries(cg[grid].mask(),maskLocal);
  #else
      const realSerialArray & xLocal = x;
      realSerialArray & uLocal = u[grid];
      realSerialArray & errLocal = err[grid];
      const intSerialArray & maskLocal = cg[grid].mask();
  #endif

      getIndex(cg[grid].indexRange(),I1,I2,I3,1);
      int includeGhost=1; // include parallel ghost pts in uLocal
      bool ok = ParallelUtility::getLocalArrayBounds(u[grid],uLocal,I1,I2,I3,includeGhost);

      real ueMax=0.; // holds the max value of the exact soln on this grid
      RealArray ue;
      if( ok )
      { // evaluate the exact solution
        ue.redim(I1,I2,I3);
        const int rectangularForTZ=0;
        exact.gd( ue,xLocal,mg.numberOfDimensions(),rectangularForTZ,0,0,0,0,I1,I2,I3,0,0.);
        ueMax=max(fabs(ue));
      }
      ParallelUtility::getMaxValue(ueMax); // max value over all procs

      real gridErrWithGhost=0., gridErr=0.;
      if( ok )
      {
        where( maskLocal(I1,I2,I3)!=0 )
          errLocal(I1,I2,I3)=abs(uLocal(I1,I2,I3)-ue);

        gridErrWithGhost=max(errLocal(I1,I2,I3))/ueMax;

        getIndex(cg[grid].indexRange(),I1,I2,I3);
        bool ok = ParallelUtility::getLocalArrayBounds(u[grid],uLocal,I1,I2,I3,includeGhost);
        if( !ok ) continue; // there are no points on this processor.

        where( maskLocal(I1,I2,I3)!=0 )
          errLocal(I1,I2,I3)=abs(uLocal(I1,I2,I3)-ue(I1,I2,I3));

        gridErr=max(errLocal(I1,I2,I3))/ueMax;
      }
      ParallelUtility::getMaxValue(gridErr); // max value over all procs
      ParallelUtility::getMaxValue(gridErrWithGhost); // max value over all procs

      error=max(error, gridErr );
      errorWithGhostPoints=max(errorWithGhostPoints, gridErrWithGhost);

      printF(" grid=%i (%s) max. rel. err=%e (%e with ghost)\n",grid,(const char*)cg[grid].getName(),
             gridErr,gridErrWithGhost);

      if( Oges::debug & 8 )
      {
        display(u[grid],"solution u");
        display(err[grid],"abs(error on indexRange +1)");
        // abs(u[grid](I1,I2,I3)-exact(cg[grid],I1,I2,I3,0)).display("abs(error)");
      }
    }
    if( option==0 )
      printF("Maximum relative error with dirichlet bc's= %e (%e with ghost)\n",error,errorWithGhostPoints
    else
      printF("Maximum relative error with neumann bc's= %e\n",error);

    return 0;
  }


  int
  main(int argc, char *argv[])
  {
    Overture::start(argc,argv);  // initialize Overture

    printF("Usage: tcm3 [<gridName>] [-solver=[yale][harwell][slap][petsc][mg]] [-debug=<value>][-outputMa
                       "[-noTiming] [-check] [-trig] [-tol=<value>] [-order=<value>] [-plot] [-ilu=] [-gmr
                       "[-freq=<value>] [-dirichlet] [-neumann]\n");

    const int maxNumberOfGridsToTest=3;
    int numberOfGridsToTest=maxNumberOfGridsToTest;
    aString gridName[maxNumberOfGridsToTest] =   { "square5", "cic", "sib" };
    // here are upper bounds on the errors we expect for each grid. This seems the only reliable
    // way to compare results from different machines, especially for iterative solvers.
    const real errorBound[maxNumberOfGridsToTest][2][2]=
      { 5.e-8,4.e-8,    5.e-7,9.e-7,  // square, dirichlet/neuman(DP) dir/neu(SP)
        7.e-4,2.e-3,    7.e-4,2.e-3, // cic
        6.e-3,7.e-3,    6.e-3,7.e-3  // sib
      };
    const int precision = REAL_EPSILON==DBL_EPSILON ? 0 : 1;
    int twilightZoneOption=0;

    int solverType=OgesParameters::yale;
    aString solverName="yale";
    aString iterativeSolverType="bi-cg";
    bool check=false;
    real tol=1.e-8;
    int orderOfAccuracy=2;
    int plot=0;
    int iluLevels=-1; // -1 : use default
    int problemsToSolve=1+2;  // solve dirichlet=1 and neumann=2
    bool outputMatrix=false;

    real fx=2., fy=2., fz=2.; // frequencies for trig TZ

    int len=0;
    if( argc >= 1 )
    {
      for( int i=1; i<argc; i++ )
      {
        aString arg = argv[i];
        if( arg=="-noTiming" )
          measureCPU=FALSE;
        else if( len=arg.matches("-debug=") )
        {
          sScanF(arg(len,arg.length()-1),"%i",&Oges::debug);
          printF("Setting Oges::debug=%i\n",Oges::debug);
        }
        else if( len=arg.matches("-tol=") )
        {
          sScanF(arg(len,arg.length()-1),"%e",&tol);
          printF("Setting tol=%e\n",tol);
        }
        else if( len=arg.matches("-freq=") )
        {
          sScanF(arg(len,arg.length()-1),"%e",&fx);
          fy=fx; fz=fx;
          printF("Setting fx=fy=fz=%e\n",fx);
        }
        else if( len=arg.matches("-ilu=") )
        {
          sScanF(arg(len,arg.length()-1),"%i",&iluLevels);
          printF("Setting ilu levels =%i\n",iluLevels);
        }
        else if( len=arg.matches("-gmres") )
        {
          iterativeSolverType="gmres";
        }
        else if( len=arg.matches("-outputMatrix") )
        {
          outputMatrix=true;
        }
        else if( len=arg.matches("-dirichlet") )
        {
          problemsToSolve=1; // just solve dirichlet problem
        }
        else if( len=arg.matches("-neumann") )
        {
          problemsToSolve=2; // just solve neumann problem
        }
        else if( len=arg.matches("-order=") )
        {
          sScanF(arg(len,arg.length()-1),"%i",&orderOfAccuracy);
          if( orderOfAccuracy!=2 && orderOfAccuracy!=4 )
          {
            printF("ERROR: orderOfAccuracy should be 2 or 4!\n");
            Overture::abort();
          }
          printF("Setting orderOfAccuracy=%i\n",orderOfAccuracy);
        }
        else if( arg(0,7)=="-solver=" )
        {
          solverName=arg(8,arg.length()-1);
          if( solverName=="yale" )
            solverType=OgesParameters::yale;
          else if( solverName=="harwell" )
            solverType=OgesParameters::harwell;
          else if( solverName=="petsc" || solverName=="PETSc" )
  #ifdef USE_PPP
            solverType=OgesParameters::PETScNew;
  #else
          solverType=OgesParameters::PETSc;
  #endif
          else if( solverName=="slap" || solverName=="SLAP" )
            solverType=OgesParameters::SLAP;
          else if( solverName=="mg" || solverName=="multigrid" )
            solverType=OgesParameters::multigrid;
          else
          {
            printF("Unknown solver=%s \n",(const char*)solverName);
            throw "error";
          }

          printF("Setting solverType=%i\n",solverType);
        }
        else if( arg=="-plot" )
        {
          plot=true;
        }
        else if( arg=="-check" )
        {
          check=true;
        }
        else if( arg=="-trig" )
```

```
        {
          twilightZoneOption=1;
        }
        else
        {
          numberOfGridsToTest=1;
          gridName[0]=argv[1];
        }
      }
    }

    if( Oges::debug > 3 )
      SparseRepForMGF::debug=3;

    aString checkFileName;
    if( REAL_EPSILON == DBL_EPSILON )
      checkFileName="tcm3.dp.check.new";  // double precision
    else
      checkFileName="tcm3.sp.check.new";
    Checker checker(checkFileName);  // for saving a check file.

    PlotStuff ps(false,"tcm3");

    // make some shorter names for readability
    BCTypes::BCNames dirichlet             = BCTypes::dirichlet,
      neumann               = BCTypes::neumann,
      extrapolate           = BCTypes::extrapolate,
      allBoundaries         = BCTypes::allBoundaries;

    int numberOfSolvers = check ? 2 : 1;
    real worstError=0.;
    for( int sparseSolver=0; sparseSolver<numberOfSolvers; sparseSolver++ )
    {
      if( check )
      {
        if( sparseSolver==0 )
        {
          solverName="yale";
          solverType=OgesParameters::yale;
        }
        else
        {
          solverName="slap";
          solverType=OgesParameters::SLAP;
        }
      }

      checker.setLabel(solverName,0);

      for( int it=0; it<numberOfGridsToTest; it++ )
      {
        aString nameOfOGFile=gridName[it];
        checker.setLabel(nameOfOGFile,1);

        printF("\n ****************************************************************\n"
               " ******** Checking grid: %s                 ************ \n"
               " ****************************************************************\n\n",(const char*)name

        CompositeGrid cg;
        if( false )
        {
          LoadBalancer loadBalancer;
          loadBalancer.setLoadBalancer(LoadBalancer::allToAll);
          getFromADataBase(cg,nameOfOGFile,loadBalancer);
        }
        else
        {
          getFromADataBase(cg,nameOfOGFile);
        }
        cg.displayDistribution("cg after reading.");

        cg.update(MappedGrid::THEmask | MappedGrid::THEvertex | MappedGrid::THEcenter | MappedGrid::THEver

        if( Oges::debug >3 )
        {
          for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
            displayMask(cg[grid].mask(),"mask");
        }

        const int inflow=1, outflow=2, wall=3;

        // create a twilight-zone function for checking the errors
        OGFunction *exactPointer;
        if( twilightZoneOption==1 ||
            min(abs(cg[0].isPeriodic()(Range(0,cg.numberOfDimensions()-1))-Mapping::derivativePeriodic))==
        {
          // this grid is probably periodic in space, use a trig function
          printF("TwilightZone: trigonometric polynomial, fx=%9.3e, fy=%9.3e, fz=%9.3e\n",fx,fy,fz);
          exactPointer = new OGTrigFunction(fx,fy,fz);
        }
        else
        {
          printF("TwilightZone: algebraic polynomial\n");
          // cg.changeInterpolationWidth(2);

          int degreeOfSpacePolynomial = orderOfAccuracy;
          int degreeOfTimePolynomial = 1;
          int numberOfComponents = cg.numberOfDimensions();
          exactPointer = new OGPolyFunction(degreeOfSpacePolynomial,cg.numberOfDimensions(),numberOfCompon
                                            degreeOfTimePolynomial);


        }
        OGFunction & exact = *exactPointer;

        // make a grid function to hold the coefficients
        Range all;
        Index I1,I2,I3, Ia1,Ia2,Ia3;

        const int width=orderOfAccuracy+1;
        int stencilSize=int(pow(width,cg.numberOfDimensions())+1);  // add 1 for interpolation equations

        realCompositeGridFunction coeff(cg,stencilSize,all,all,all);

```

```
    const int numberOfGhostLines=orderOfAccuracy/2;
    coeff.setIsACoefficientMatrix(TRUE,stencilSize,numberOfGhostLines);
    coeff=0.;

    // create grid functions:
    realCompositeGridFunction u(cg),f(cg);
    realCompositeGridFunction err(cg);

    real error;

    CompositeGridOperators op(cg);                            // create some differential operators
    op.setStencilSize(stencilSize);
    op.setOrderOfAccuracy(orderOfAccuracy);
    //   op.setTwilightZoneFlow(TRUE);
    // op.setTwilightZoneFlowFunction(exact);

    f.setOperators(op); // for apply the BC
    coeff.setOperators(op);

    // cout << "op.laplacianCoefficients().className: " << (op.laplacianCoefficients()).getClassName()
    // cout << "-op.laplacianCoefficients().className: " << (-op.laplacianCoefficients()).getClassName

    if( false )
    {
      coeff=op.laplacianCoefficients();       // get the coefficients for the Laplace operator
    }
    else
    { // new way for parallel -- this avoids all communication
      for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
      {
        getIndex(cg[grid].gridIndexRange(),I1,I2,I3);
        op[grid].coefficients(MappedGridOperators::laplacianOperator,coeff[grid],I1,I2,I3);
      }
    }

    // fill in the coefficients for the boundary conditions
    coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,  allBoundaries);
    coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries);
    BoundaryConditionParameters bcParams;
    if( orderOfAccuracy==4 )
    {
      bcParams.ghostLineToAssign=2;
      coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries,bcParams); // extrap 2nd
    }
    coeff.finishBoundaryConditions();
    // coeff.display("Here is coeff after finishBoundaryConditions");

    if( false )
    {
      for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
      {
        displayCoeff(coeff[grid],sPrintF("Coeff matrix for grid %i",grid));

        // coeff[grid].sparse->classify.display("the classify matrix after applying finishBoundaryCond
        //      coeff[grid].display("this is the coefficient matrix");
      }
    }
```

```
      Oges solver( cg );                     // create a solver
      solver.set(OgesParameters::THEsolverType,solverType);

      if( outputMatrix )
        solver.set(OgesParameters::THEkeepSparseMatrix,true);

      if( solver.isSolverIterative() )
      {
        solver.setCommandLineArguments( argc,argv );
        solver.set(OgesParameters::THEpreconditioner,OgesParameters::incompleteLUPreconditioner);

        if( iterativeSolverType=="gmres" )
          solver.set(OgesParameters::THEsolverMethod,OgesParameters::generalizedMinimalResidual);
        else
        {
          if( solverType==OgesParameters::PETSc )
            solver.set(OgesParameters::THEsolverMethod,OgesParameters::biConjugateGradientStabilized);
          else if( solverType==OgesParameters::PETScNew )
          { // parallel: -- NOTE: in parallel the solveMethod should be preonly and the parallelSolverMe
            solver.set(OgesParameters::THEbestIterativeSolver);
            // solver.set(OgesParameters::THEparallelSolverMethod,OgesParameters::biConjugateGradient);
            // solver.set(OgesParameters::THEparallelSolverMethod,OgesParameters::gmres);
            // Use an LU solver on each processor:
            // solver.set(OgesParameters::THEpreconditioner,OgesParameters::luPreconditioner);
            // This also works: Use an LU on each processor:
            // solver.parameters.setPetscOption("-sub_pc_type","lu");
          }
          else
            solver.set(OgesParameters::THEsolverMethod,OgesParameters::biConjugateGradient);
        }

        solver.set(OgesParameters::THErelativeTolerance,max(tol,REAL_EPSILON*10.));
        solver.set(OgesParameters::THEmaximumNumberOfIterations,10000);
        if( iluLevels>=0 )
          solver.set(OgesParameters::THEnumberOfIncompleteLULevels,iluLevels);
      }

      printF("\n === Solver:\n %s\n =====\n",(const char*)solver.parameters.getSolverName());


      // ---------------------------------
      // --------- Dirichlet BC's --------
      // ---------------------------------
      if( problemsToSolve % 2 ==1 )
      {

        solver.setCoefficientArray( coeff );   // supply coefficients
        // Assign the right-hand-side f
        assignForcing( 0,cg,f,exact );


        u=0.;  // initial guess for iterative solvers
        real time0=CPU();
        solver.solve( u,f );   // solve the equations
        real time=CPU()-time0;
        printF("\n*** max residual=%8.2e, time for 1st solve of the Dirichlet problem = %8.2e (iteration
               solver.getMaximumResidual(),time,solver.getNumberOfIterations());

        // solve again
        if( true )
        {
          // u=0.;
          time0=CPU();
          solver.solve( u,f );   // solve the equations
          time=CPU()-time0;
          printF("\n*** max residual=%8.2e, time for 2nd solve of the Dirichlet problem = %8.2e (iterati
                 solver.getMaximumResidual(),time,solver.getNumberOfIterations());
        }
        if( outputMatrix )
        {
          printF("tcm3:INFO: save the matrix to file tcm3Matrix.out (using writeMatrixToFile). \n");
          solver.writeMatrixToFile("tcm3Matrix.out");

          aString fileName = "sparseMatrix.dat";
          printF("tcm3:INFO: save the matrix to file %s (using outputSparseMatrix)\n",(const char*)fileN
          solver.outputSparseMatrix( fileName );
        }


        // ---- check the errors in the solution ---

        const int numberOfGridsPoints=max(1,cg.numberOfGridPoints());
        const real solverSize=solver.sizeOf();
        printF("\n.....solver: size = %8.2e (bytes), grid-pts=%i, reals/grid-pt=%5.2f \n",
               solverSize,numberOfGridsPoints,solverSize/(numberOfGridsPoints*sizeof(real)));

        // u.display("Here is the solution to Laplacian(u)=f");
        computeTheError( 0,cg,u,err,exact, error );
        worstError=max(worstError,error);

        checker.setCutOff(errorBound[it][precision][0]); checker.printMessage("dirichlet: error",error,t

        if( plot )
        {
          ps.createWindow("tcm3");
          plotResults( ps,solver,u,err );
        }

      }


      // --------------------------------
      // --------- Neumann BC's ----------
      // --------------------------------
      if( (problemsToSolve/2) % 2 ==1 )
      {
        coeff=0.;
        for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
        {
          getIndex(cg[grid].gridIndexRange(),I1,I2,I3);
          op[grid].coefficients(MappedGridOperators::laplacianOperator,coeff[grid],I1,I2,I3);
        }

        // fill in the coefficients for the boundary conditions
        coeff.applyBoundaryConditionCoefficients(0,0,neumann,allBoundaries);
        if( orderOfAccuracy==4 )
          coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries,bcParams); // extrap 2n

        coeff.finishBoundaryConditions();

        solver.setCoefficientArray( coeff );   // supply coefficients

        bool singularProblem=true;
        for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
        { // this loop does nothing for now
          MappedGrid & mg = cg[grid];
          ForBoundary(side,axis)
          {
            if( mg.boundaryCondition(side,axis) > 0  )
            {
            }
            else if( mg.boundaryCondition(side,axis) ==inflow ||  mg.boundaryCondition(side,axis) ==outf
            {
              singularProblem=false;
            }
          }
        }

        // Assign the right-hand-side f
        assignForcing( 1,cg,f,exact );


        // if the problem is singular Oges will add an extra constraint equation to make the system nons
        if( singularProblem )
          solver.set(OgesParameters::THEcompatibilityConstraint,TRUE);

        // Tell the solver to refactor the matrix since the coefficients have changed
        solver.setRefactor(TRUE);
        // we need to reorder too because the matrix changes a lot for the singular case
        solver.setReorder(TRUE);

        if( singularProblem )
        {
          // we need to first initialize the solver before we can fill in the rhs for the compatibility
          solver.initialize();
          realCompositeGridFunction ue(cg);
          exact.assignGridFunction(ue,0.);
          real value=0.;
          solver.evaluateExtraEquation(ue,value);

          solver.setExtraEquationValues(f,&value );
        }

        u=0.;  // initial guess for iterative solvers
        real time0=CPU();
        solver.solve( u,f );   // solve the equations
        real time=CPU()-time0;
        printF("residual=%8.2e, time for 1st solve of the Neumann problem = %8.2e (iterations=%i)\n",
               solver.getMaximumResidual(),time,solver.getNumberOfIterations());

        // turn off refactor for the 2nd solve
        solver.setRefactor(FALSE);
        solver.setReorder(FALSE);
        // u=0.;  // initial guess for iterative solvers
        time0=CPU();
        solver.solve( u,f );   // solve the equations
        time=CPU()-time0;
        printF("residual=%8.2e, time for 2nd solve of the Neumann problem = %8.2e (iterations=%i)\n",
               solver.getMaximumResidual(),time,solver.getNumberOfIterations());

        computeTheError( 1,cg,u,err,exact, error );

        worstError=max(worstError,error);

        checker.setCutOff(errorBound[it][precision][1]); checker.printMessage("neumann: error",error,tim

      }

      if( plot )
        plotResults( ps,solver,u,err );

      delete exactPointer; exactPointer=0;// kkc 090902, this was a memory leak making new OGFunction's

    }  // end it (number of grids)


  }  // end sparseSolver


  fflush(0);
  printF("\n\n ****************************************************************************************
  if( worstError > .025 )
    printF(" ************** Warning, there is a large error somewhere, worst error =%e ****************
           worstError);
  else
    printF(" ************** Test apparently successful, worst error =%e ******************\n",worstError
  printF(" *******************************************************************************************

  fflush(0);

  Overture::finish();

  return(0);
}


```

5.4 Systems of equations on a CompositeGrid

(file `Overture/examples/tcm4.C`)

```
//===============================================================================
//  Coefficient Matrix Example
//    Solve a System of Equations on a CompositeGrid
//
//Usage: tcm4 [<gridName>] [-solver=<yale|harwell|slap|petsc>] [-debug=<value>] [-noTiming] [-trig]
```

```
//            [-dirichlet] [-neumann] [-freq=<value>][-outputMatrix]
//================================================================================
#include "Overture.h"
#include "Oges.h"
#include "CompositeGridOperators.h"
#include "OGPolyFunction.h"
#include "OGTrigFunction.h"
#include "Checker.h"
#include "ParallelUtility.h"
#include "SparseRep.h"
#include "PlotIt.h"

#define ForBoundary(side,axis)   for( axis=0; axis<mg.numberOfDimensions(); axis++ ) \
                                 for( side=0; side<=1; side++ )

namespace {

  bool measureCPU=TRUE;

  enum TWType {
    TWPoly,
    TWTrig
  };


  // make some shorter names for readability
  BCTypes::BCNames
                   dirichlet             = BCTypes::dirichlet,
                   neumann               = BCTypes::neumann,
                   extrapolate           = BCTypes::extrapolate,
                   normalComponent       = BCTypes::normalComponent,
                   aDotU                 = BCTypes::aDotU,
                   generalizedDivergence = BCTypes::generalizedDivergence,
                   generalMixedDerivative= BCTypes::generalMixedDerivative,
                   aDotGradU             = BCTypes::aDotGradU,
                   vectorSymmetry        = BCTypes::vectorSymmetry,
                   allBoundaries         = BCTypes::allBoundaries;


  enum ProblemFlags {
    dirichletFlag = 0x1,
    neumannFlag   = dirichletFlag<<1,
    neumannDirichletFlag = neumannFlag<<1
  };

  int numberOfComponents = 2;
  bool outputMatrix = false;
  const real a1=1., a2=2., a3=3., a4=4.;
  int includeGhost = 1;
  //  const real a1=1., a2=0., a3=1., a4=0.;

  void buildMatrix(ProblemFlags problem, realCompositeGridFunction &coeff)
  {
    coeff = 0.;

    // Solve a system of equations for (u_0,u_1) = (u,v)
    //     a1(  u_xx + u_yy ) + a2*v_x = f_0
    //     a3(  v_xx + v_yy ) + a4*u_y = f_1

    CompositeGridOperators &op = *coeff.getOperators();

    Range e0(0,0), e1(1,1);  // e0 = first equation, e1=second equation
    Range c0(0,0), c1(1,1);  // c0 = first component, c1 = second component
    coeff=a1*op.laplacianCoefficients(e0,c0)+a2*op.xCoefficients(e0,c1)
      +a3*op.laplacianCoefficients(e1,c1)+a4*op.yCoefficients(e1,c0);

    if ( problem==dirichletFlag )
      {
        coeff.applyBoundaryConditionCoefficients(0,0,dirichlet,  allBoundaries);
        coeff.applyBoundaryConditionCoefficients(0,0,extrapolate,allBoundaries);

        coeff.applyBoundaryConditionCoefficients(1,1,dirichlet,  allBoundaries);
        coeff.applyBoundaryConditionCoefficients(1,1,extrapolate,allBoundaries);
      }
    else if ( problem==neumannFlag )
      {
        coeff.applyBoundaryConditionCoefficients(0,0,neumann,  allBoundaries);
        coeff.applyBoundaryConditionCoefficients(1,1,neumann,  allBoundaries);
      }
    else if ( problem==neumannDirichletFlag )
      {
        coeff.applyBoundaryConditionCoefficients(0,0,neumann,  allBoundaries);

        coeff.applyBoundaryConditionCoefficients(1,1,dirichlet,  allBoundaries);
        coeff.applyBoundaryConditionCoefficients(1,1,extrapolate,allBoundaries);
      }

    coeff.finishBoundaryConditions();
    if( Oges::debug & 16 )
      coeff.display("Here is coeff after finishBoundaryConditions");

  }

  void buildForcing(ProblemFlags problem, realCompositeGridFunction &f, OGFunction &exact)
  {
    f=0.;
    // assign the rhs:  u=exact on the boundary
    CompositeGrid &cg = *f.getCompositeGrid();
    int numberOfDimensions = cg.numberOfDimensions();
    Index I1,I2,I3, Ia1,Ia2,Ia3;
    int side,axis;
    Index Ib1,Ib2,Ib3;
    Index Ig1,Ig2,Ig3;
    for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
      {
        MappedGrid & mg = cg[grid];
        getIndex(mg.indexRange(),I1,I2,I3);
        realArray & x= mg.center();
#ifdef USE_PPP
        realSerialArray xLocal; getLocalArrayWithGhostBoundaries(x,xLocal);
#else
        const realSerialArray & xLocal = x;
#endif
        f[grid](I1,I2,I3,0)=a1*(exact.xx(mg,I1,I2,I3,0)+exact.yy(mg,I1,I2,I3,0))+a2*exact.x(mg,I1,I2,I3,
        f[grid](I1,I2,I3,1)=a3*(exact.xx(mg,I1,I2,I3,1)+exact.yy(mg,I1,I2,I3,1))+a4*exact.y(mg,I1,I2,I3,
        if( cg.numberOfDimensions()==3 )
          {
            f[grid](I1,I2,I3,0)+=a1*exact.zz(mg,I1,I2,I3,0);
            f[grid](I1,I2,I3,1)+=a3*exact.zz(mg,I1,I2,I3,1);
          }

        if ( problem==dirichletFlag )
          {
            ForBoundary(side,axis)
              {
                if( mg.boundaryCondition()(side,axis) > 0 )
                  {
                    getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);
                    f[grid](Ib1,Ib2,Ib3,0)=exact(mg,Ib1,Ib2,Ib3,0);
                    f[grid](Ib1,Ib2,Ib3,1)=exact(mg,Ib1,Ib2,Ib3,1);
                  }
              }
          }
        else if ( problem==neumannFlag )
          {
            ForBoundary(side,axis)
              {
#ifdef USE_PPP
                const realSerialArray & normal  = mg.vertexBoundaryNormalArray(side,axis);
#else
                const realSerialArray & normal  = mg.vertexBoundaryNormal(side,axis);
#endif

                getGhostIndex(mg.gridIndexRange(),side,axis,Ig1,Ig2,Ig3);
                getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);
                //              realSerialArray fLocal;
                //              bool ok = ParallelUtility::getLocalArrayBounds(f[grid],fLocal,Ig1,Ig2,Ig
                //              if( !ok ) continue; // there are no points on this processor.
                const int rectangularForTZ=0;
                realSerialArray uex(Ib1,Ib2,Ib3), uey(Ib1,Ib2,Ib3);
                if( mg.boundaryCondition()(side,axis) > 0 )
                  {
                    for ( int n=0; n<numberOfComponents; n++ )
                      {
                        exact.gd( uex,xLocal,numberOfDimensions,rectangularForTZ,0,1,0,0,Ib1,Ib2,Ib3,n,0
                        exact.gd( uey,xLocal,numberOfDimensions,rectangularForTZ,0,0,1,0,Ib1,Ib2,Ib3,n,0
                        f[grid](Ig1,Ig2,Ig3,n) = normal(Ib1,Ib2,Ib3,0)*uex + normal(Ib1,Ib2,Ib3,1)*uey;
                        if( numberOfDimensions==3 )
                          {
                            exact.gd( uex,xLocal,numberOfDimensions,rectangularForTZ,0,0,0,1,Ib1,Ib2,Ib3
                            f[grid](Ig1,Ig2,Ig3,n) +=normal(Ib1,Ib2,Ib3,2)*uex;
                          }
                      } // for each component
                  } // if a real boundary
              } // for boundary
          } // if neumann
        else if ( problem==neumannDirichletFlag )
          {
            ForBoundary(side,axis)
              {
#ifdef USE_PPP
```

```
                const realSerialArray & normal  = mg.vertexBoundaryNormalArray(side,axis);
#else
                const realSerialArray & normal  = mg.vertexBoundaryNormal(side,axis);
#endif
                if( mg.boundaryCondition()(side,axis) > 0 )
                  {
                    getBoundaryIndex(mg.gridIndexRange(),side,axis,Ib1,Ib2,Ib3);
                    getGhostIndex(mg.gridIndexRange(),side,axis,Ig1,Ig2,Ig3);

                    // neumann condition on the first variable
                    const int rectangularForTZ=0;
                    realSerialArray uex(Ib1,Ib2,Ib3), uey(Ib1,Ib2,Ib3);
                    exact.gd( uex,xLocal,numberOfDimensions,rectangularForTZ,0,1,0,0,Ib1,Ib2,Ib3,0,0.);
                    exact.gd( uey,xLocal,numberOfDimensions,rectangularForTZ,0,0,1,0,Ib1,Ib2,Ib3,0,0.);
                    f[grid](Ig1,Ig2,Ig3,0) = normal(Ib1,Ib2,Ib3,0)*uex + normal(Ib1,Ib2,Ib3,1)*uey;
                    if( numberOfDimensions==3 )
                      {
                        exact.gd( uex,xLocal,numberOfDimensions,rectangularForTZ,0,0,0,1,Ib1,Ib2,Ib3,0,0
                        f[grid](Ig1,Ig2,Ig3,0) +=normal(Ib1,Ib2,Ib3,2)*uex;
                      }

                    // dirichlet condition on the second variable
                    f[grid](Ib1,Ib2,Ib3,1)=exact(mg,Ib1,Ib2,Ib3,1);
                  }
              }
          } // if neumann+dirichlet

      } // makeForcing

  } // anonymous namespace

  void solveSystem(int solverType, int numberOfConstraints, OGFunction &exact,
                   realCompositeGridFunction &coeff, realCompositeGridFunction &f, realCompositeGridFunc
  {
    CompositeGrid &cg = *coeff.getCompositeGrid();
    Oges solver( cg );                      // create a solver
    int numberOfDimensions = cg.numberOfDimensions();
    int numberOfGrids = cg.numberOfComponentGrids();
    solver.setCoefficientArray( coeff );   // supply coefficients
    solver.set(OgesParameters::THEsolverType,solverType);
    if( solverType==OgesParameters::SLAP ||  solverType==OgesParameters::PETSc )
      {
        solver.set(OgesParameters::THEpreconditioner,OgesParameters::incompleteLUPreconditioner);
        solver.set(OgesParameters::THEtolerance,max(tol,REAL_EPSILON*10.));
      }

    if ( numberOfConstraints==1 )
      {
        // this should cause the automatic creation of the compatibility constraint on the first equatio
        solver.set(OgesParameters::THEcompatibilityConstraint,true);
        solver.initialize(); // this will form the right null vector
        realCompositeGridFunction ue(cg);
        exact.assignGridFunction(ue,0.);
        real value=0.;
        solver.evaluateExtraEquation(ue,value);

        solver.setExtraEquationValues(f,&value );
      }
    else if ( numberOfConstraints==2 )
      {
        // kkc 090903
        // Because we have a system of two equations we cannot use the built-in compatibility constraint
        // The following code sets up a constraint for each equation (total of two extra equations) and
        solver.initialize();
        Range all;
        real nullVectorScaling = 0;
        solver.get(OgesParameters::THEnullVectorScaling, nullVectorScaling);
        realCompositeGridFunction &constraint = solver.rightNullVector;
        constraint.updateToMatchGrid(cg,all,all,all,numberOfComponents);
        constraint=0.;
        Index I1,I2,I3;
        for ( int n=0; n<numberOfComponents; n++ )
          {
            for( int grid=numberOfGrids-1; grid>=0; grid-- ) // why is this loop backwards? I took it di
              {
                MappedGrid & c = cg[grid];
                IntegerDistributedArray & classifyX = coeff[grid].sparse->classify;

                getIndex(c.dimension(),I1,I2,I3);
                real scale = (n+1)*nullVectorScaling; // multiply by n+1 so that each equation gets a di
                // do not include the ghost line so that we retain u.n=0 as a BC!
                where( classifyX(I1,I2,I3,n)==SparseRepForMGF::interior || classifyX(I1,I2,I3,n)==Sparse
                  constraint[grid](I1,I2,I3,n)=scale;
              }
          }
        solver.numberOfExtraEquations = 2; // this should probably be in OgesParameters...
        solver.set(OgesParameters::THEcompatibilityConstraint,true);
        solver.set(OgesParameters::THEuserSuppliedCompatibilityConstraint,true);
        solver.updateToMatchGrid( cg ); // why do we need this? it will call initialize...

        realCompositeGridFunction ue(cg,all,all,all,numberOfComponents);
        exact.assignGridFunction(ue,0.);
        ArraySimple<real> value(numberOfComponents);
        value = 0.;
        for ( int n=0; n<numberOfComponents; n++ )
          {
            solver.evaluateExtraEquation(ue,value[n],n);
          }
        solver.setExtraEquationValues(f,value.ptr());
        if( Oges::debug & 16 )
          {
            constraint.display("here are the constraint coefficients");
            cout<<"here are the extra equation values"<<value<<endl;
          }
      }

    if( outputMatrix )
      solver.set(OgesParameters::THEkeepSparseMatrix,true);

    u=0.;  // for interative solvers.
    real time0=getCPU();
    solver.solve( u,f );   // solve the equations

    printf("residual=%8.2e, time for solve = %8.2e (iterations=%i)\n",
           solver.getMaximumResidual(),getCPU()-time0,solver.getNumberOfIterations());

    if( false ) // kkc 090903, changed from true to false, why was solve being called twice??
      {
        solver.solve( u,f );   // solve the equations

        printf("residual=%8.2e, time for 2nd solve = %8.2e (iterations=%i)\n",
               solver.getMaximumResidual(),getCPU()-time0,solver.getNumberOfIterations());

      }
  }

  real computeMaxError(int n, realCompositeGridFunction &u, OGFunction &exact)
  {
    CompositeGrid &cg = *u.getCompositeGrid();
    Index I1,I2,I3;
    //    for( int n=0; n<numberOfComponents; n++ )
    //      {
    real error=0.;
    for( int grid=0; grid<cg.numberOfComponentGrids(); grid++ )
      {
        getIndex(cg[grid].indexRange(),I1,I2,I3);
        realArray err = (u[grid](I1,I2,I3,n)-exact(cg[grid],I1,I2,I3,n))/max(abs(exact(cg[grid],I1,I2,I3
        where( cg[grid].mask()(I1,I2,I3)!=0 )
          {
            error=max(error,max(abs(err)));
          }
        if( Oges::debug & 4 )
          {
            abs(u[grid](I1,I2,I3,n)-exact(cg[grid],I1,I2,I3,n)).display("abs(error)");
            u.display("u");

          }
      }
    //  PlotIt::contour(*Overture::getGraphicsInterface(),u);
    //  printf("Maximum relative error in component %i with dirichlet bc's= %e\n",n,error);
    //  checker.printMessage(msg,error,time);
    return error;
  }
  //  }
}

real
CPU()
// In this version of getCPU we can turn off the timing
{
  if( measureCPU )
    return getCPU();
  else
    return 0;
}
int
main(int argc, char **argv)
{
  Overture::start(argc,argv);  // initialize Overture

  printF("Usage: tcm4 [<gridName>] [-solver=<yale|harwell|slap|petsc>] [-debug=<value>] [-noTiming] [-tr
      "           [-dirichlet] [-neumann] [-neumann+dirichlet] [-freq=<value>][-outputMatrix] [-chec

  const int maxNumberOfGridsToTest=3;
  int numberOfGridsToTest=maxNumberOfGridsToTest;
  aString gridName[maxNumberOfGridsToTest] =   { "square5", "cic", "sib" };
  const int maxNumberOfSolversToTest = 4;
  int numberOfSolversToTest = maxNumberOfSolversToTest;
  aString solverName[maxNumberOfSolversToTest] = {"yale","harwell","slap","petsc"};
  int solverType[maxNumberOfSolversToTest] = {OgesParameters::yale,
                                              OgesParameters::harwell,
                                              OgesParameters::SLAP,
                                              OgesParameters::PETSc};

  real tol=1.e-8;
  BCTypes::BCNames bcTest = dirichlet;
  int problemsToSolve = dirichletFlag|neumannFlag|neumannDirichletFlag;
  TWType twType = TWPoly;
  real fx=2., fy=2., fz=2.; // frequencies for trig TZ
  int degreeOfSpacePolynomial = 2;
  int degreeOfTimePolynomial = 1;
  int len=0;
  if( argc >= 1 )
  {
    for( int i=1; i<argc; i++ )
    {
      aString arg = argv[i];
      if( arg=="-noTiming" )
        measureCPU=FALSE;
      else if( arg(0,6)=="-debug=" )
      {
        sScanF(arg(7,arg.length()-1),"%i",&Oges::debug);
        printf("Setting Oges::debug=%i\n",Oges::debug);
      }
      else if (arg=="outputMatrix" )
      {
        outputMatrix=true;
      }
      else if ( arg=="-dirichlet" )
      {
        problemsToSolve = dirichletFlag;
      }
      else if ( arg=="-neumann" )
      {
        problemsToSolve = neumannFlag;
      }
      else if ( arg=="-neumann+dirichlet" )
      {
        problemsToSolve = neumannDirichletFlag;
      }
      else if ( arg=="-trig" )
      {
        twType = TWTrig;
      }
      else if( arg(0,7)=="-solver=" )
      {
        aString solver=arg(8,arg.length()-1);
        if( solver=="yale" )
```

```
          solverType[0]=OgesParameters::yale;
        else if( solver=="harwell" )
          solverType[0]=OgesParameters::harwell;
        else if( solver=="slap" )
          solverType[0]=OgesParameters::SLAP;
        else if( solver=="petsc" )
          solverType[0]=OgesParameters::PETSc;
        else
        {
          printf("Unknown solver=%s \n",(const char*)solver);
          throw "error";
        }

        numberOfSolversToTest = 1;
        solverName[0] = solver;

        //      printf("Setting solverType=%i\n",solver);
      }
      else if ( arg=="-check" )
      {
        //numberOfSolversToTest=2;
        //solverName[0] = "yale"; solverType[0] = OgesParameters::yale;
        //solverName[1] = "slap"; solverType[1] = OgesParameters::SLAP;
        // *wdh* 100608 -- only check slap to make the test faster
        numberOfSolversToTest=1;
        solverName[0] = "slap"; solverType[0] = OgesParameters::SLAP;
        tol=1.e-3;
      }
      else if( (len=arg.matches("-freq=")) )
      {
        sScanF(arg(len,arg.length()-1),"%e",&fx);
        fy=fx; fz=fx;
        printF("Setting fx=fy=fz=%e\n",fx);
      }
      else
      {
        numberOfGridsToTest=1;
        gridName[0]=arg;
      }
    }
  }
  //kkc 090902  else
  //kkc 090902    cout << "Usage: ‘tcm4 [<gridName>] [-solver=[yale][harwell][slap][petsc]] [-debug=<val

  aString checkFileName;
  if( REAL_EPSILON == DBL_EPSILON )
    checkFileName="tcm4.dp.check.new";  // double precision
  else
    checkFileName="tcm4.sp.check.new";
  Checker checker(checkFileName);  // for saving a check file.
  checker.setCutOff(0.); // for initial testing of the modified code...

  real worstError=0.;

  for ( int ls=0; ls<numberOfSolversToTest; ls++ )
    {
      checker.setLabel(solverName[ls],0);
      for( int it=0; it<numberOfGridsToTest; it++ )
        {
          aString nameOfOGFile=gridName[it];
          checker.setLabel(nameOfOGFile,1);

          cout << "\n ****************************************************************\n";
          cout << " ******** Checking grid: " << nameOfOGFile << " ************ \n";
          cout << " ****************************************************************\n\n";

          CompositeGrid cg;
          getFromADataBase(cg,nameOfOGFile);
          cg.update(MappedGrid::THEvertex | MappedGrid::THEcenter | MappedGrid::THEvertexBoundaryNormal)

          const int inflow=1, outflow=2, wall=3;
          int grid;
          for( grid=0; grid<cg.numberOfComponentGrids(); grid++ )
            {
              if( cg[grid].boundaryCondition()(Start,axis1) > 0 )
                cg[grid].boundaryCondition()(Start,axis1)=inflow;
              if( cg[grid].boundaryCondition()(End  ,axis1) > 0 )
                cg[grid].boundaryCondition()(End  ,axis1)=inflow;
              if( cg[grid].boundaryCondition()(Start,axis2) > 0 )
                cg[grid].boundaryCondition()(Start,axis2)=wall;
              if( cg[grid].boundaryCondition()(End  ,axis2) > 0 )
                cg[grid].boundaryCondition()(End  ,axis2)=wall;
            }

          // create a twilight-zone function
          OGFunction *exactP = (twType==TWPoly) ? (OGFunction *)new OGPolyFunction(degreeOfSpacePolynomi
                                                                                   cg.numberOfDimensions
                                                                                   numberOfComponents,
                                                                                   degreeOfTimePolynomia
                                                   (OGFunction *)new OGTrigFunction(fx,fy,fz);
          OGFunction &exact = *exactP;

          Range all;
          // make a grid function to hold the coefficients
          int stencilSize=int( pow(3,cg.numberOfDimensions())+1 );  // add 1 for interpolation equations
          int stencilDimension=stencilSize*SQR(numberOfComponents);
          realCompositeGridFunction coeff(cg,stencilDimension,all,all,all);
          // make this grid function a coefficient matrix:
          int numberOfGhostLines=1;
          coeff.setIsACoefficientMatrix(TRUE,stencilSize,numberOfGhostLines,numberOfComponents);
          coeff=0.;

          // create grid functions:
          realCompositeGridFunction u(cg,all,all,all,numberOfComponents),
            f(cg,all,all,all,numberOfComponents);
          //      PlotIt::contour(*Overture::getGraphicsInterface(),u);

          CompositeGridOperators op(cg);                            // create some differential operator
          op.setNumberOfComponentsForCoefficients(numberOfComponents);
          u.setOperators(op);                                 // associate differential operators with u
          coeff.setOperators(op);
          aString msg;
          real time0 = getCPU();
          if ( problemsToSolve & dirichletFlag )
            {
              buildMatrix(dirichletFlag, coeff);
              buildForcing(dirichletFlag, f, exact);
              solveSystem(solverType[ls],0,exact,coeff,f,u,tol);
              real time = getCPU()-time0;

              for ( int n=0; n<numberOfComponents; n++ )
                {
                  real error = computeMaxError(n,u,exact);
                  sPrintF(msg,"dirichlet: error (n=%i)",n);
                  checker.printMessage(msg,error,time);
                  worstError=max(worstError,error);
                }
            }

          if ( problemsToSolve & neumannDirichletFlag )
            { // we put this problem here so that we don't confuse the sparse rep classify with two extr
              buildMatrix(neumannDirichletFlag, coeff);
              buildForcing(neumannDirichletFlag, f, exact);
              solveSystem(solverType[ls],1,exact,coeff,f,u,tol);
              real time = getCPU()-time0;

              for ( int n=0; n<numberOfComponents; n++ )
                {
                  real error = computeMaxError(n,u,exact);
                  sPrintF(msg,"neumann-dirichlet: error (n=%i)",n);
                  checker.printMessage(msg,error,time);
                  worstError=max(worstError,error);
                }
            }

          if ( problemsToSolve & neumannFlag )
            {
              buildMatrix(neumannFlag, coeff);
              buildForcing(neumannFlag, f, exact);
              solveSystem(solverType[ls],2,exact,coeff,f,u,tol);
              real time = getCPU()-time0;

              for ( int n=0; n<numberOfComponents; n++ )
                {
                  real error = computeMaxError(n,u,exact);
                  sPrintF(msg,"neumann: error (n=%i)",n);
                  checker.printMessage(msg,error,time);
                  worstError=max(worstError,error);
                }
            }


          // u.display("Here is the solution to u.xx+u.yy=f");
          delete exactP; exactP=0;

        } // end loop over grids
    } // end loop over solvers to test

  printf("\n\n ***************************************************************************************
  if( worstError > .025 )
    printf(" ************** Warning, there is a large error somewhere, worst error =%e ****************
          worstError);
  else
    printf(" ************** Test apparently successful, worst error =%e ******************\n",worstError
  printf(" ************************************************************************************************

  Overture::finish();
  return(0);
}

```

5.5 Solving Poisson's equation to fourth-order accuracy

The file `Overture/examples/tcmOrder4.C` shows how to solve an elliptic problem to fourth-order accuracy. In this case we use two ghost lines (really only needed for Neumann boundary conditions). We need to tell the operators to use fourth order and we need to build the coefficient matrix using 2 ghost lines. In order to extrapolate the second ghost line we use a `BoundaryConditionParameters` object. The order of extrapolation will be set to the order of accuracy plus one, by default. This example will only work with version 15 or later.

5.6 Multiplying a grid function or array times a coefficient matrix

Suppose one wants to form the variable coefficient operator such as

$$L = x\frac{\partial}{\partial x} + y\frac{\partial}{\partial y}$$

In order to multiply a grid function (or array of the correct shape) times a coefficient matrix one can use the `multiply` function as illustrated in the next example

```
MappedGrid mg(...);
Range all;
MappedGridOperators op(mg);
realMappedGridFunction coeff(mg,9,all,all,all);
...

Index I1,I2,I3;
getIndex(mg.dimension,I1,I2,I3);   // define Index's for the entire grid
// form the operator  x d/dx + y d/dy

RealArray x,y;
x=mg.vertex(I1,I2,I3,0);  // make a copy since we cannot pass a view to multiply
y=mg.vertex(I1,I2,I3,1);
coeff= multiply(x,op.xCoefficients()) + multiply(y,op.yCoefficents());
```

One cannot use the normal multiplication operator, $\star$, because the array operations would not be conformable. Since the `multiply` reshapes it's first argument in order to multiply it times the second argument **one cannot pass a view of an array as the first argument of the multiply function**. Passing a view will result in an A++ errror.

A `multiply` function is also defined for multiplying a scalar `realCompositeGridFunction` times a `realCompositeGridFunction` coefficient matrix.

5.7 SparseRep: Define a Storage Format for a Sparse Matrix

This section is primarily for the use of people who are writing new Operator classes. Normal beings may not want to read this.

The class `SparseRepForMGF` defines the sparse representation for coefficient-matrix MappedGridFunction. A coefficient matrix contains a pointer to a `SparseRepForMGF`. This object holds all the information that defines how the stencil is stored in the first component. For example this object will know that the value `coeff(m,i1,i2,i3)` is the coefficient that multiples the grid function value at the point `(i1',i2',i3')`. To save this information in a compact form, each point on the grid is given an equation number, (stored in the `intMappedGridFunction equationNumber`), so that instead of saving the three numbers `(i1',i2',i3')` only a single equationNumber need be saved.

The `SparseRepForMGF` object also contains an `intMappedGridFunction classify`. The `classify` array holds a value for each point on the grid to indicate the kind of equation (`interior`, `boundary`, `extrapolation`, `interpolation`...) that is being applied at that point. This information can be used by a sparse solver (such as `Oges`) to automatically zero out the right-hand-side for certain equations, such as extrapolation.

Here is how `SparseRep` is used by the grid function classes.

If we have a `realMappedGridFunction coeff` or a `realCompositeGridFunction coeff` then the statement

```
coeff.setIsACoefficientMatrix(TRUE);
```

will cause a `SparseRep` object to be created (and coeff will keep a pointer to it). The `SparseRep` object will be initialized with a call to `SparseRep::updateToMatchGrid`. This will give initial values to the `classify` and `equationNumber` arrays assuming a standard stencil.

When `coeff` is filled in with values for the interior with a statement like

```
coeff=op.laplacianCoefficients();
```

then normally the default values for the `classify` and `equationNumber` arrays will be correct.

However, when the boundary conditions are filled in with a statment like

```
coeff.applyBoundaryConditionCoefficinets(0,neumann,...);
```

then the default values for the `classify` and `equationNumber` arrays will have to be changed. (On a vertex grid the neumann boundary condition is the equation for the ghost-line but it is centred on the boundary and thus the default equation numbers are wrong.) For some examples look at the implementation of the `applyBoundaryConditionCoefficients` in the file `BoundaryOperators.C`.

If `coeff` is a `realMappedGridFunction` then the statement

```
coeff.finishBoundaryConditions();
```

will add extrapolation equations at corners and insert equations for periodic boundary conditions.

If `coeff` is a `realCompositeGridFunction` then the statement

```
coeff.finishBoundaryConditions();
```

will call `coeff[grid].finishBoundaryConditions()` for each component grid function and in addition it will add in the interpolation equations to `coeff`.

5.7.1 Public enumerators

Here are the public enumerators:

classifyTypes: This enumerator contains a list of classify types.. Any non-negative value indicates a used point. Negative values are equations with zero for the rhs

```
enum classifyTypes
{
  interior=1,
  boundary=2,
  ghost1=3,
  ghost2=4,
  ghost3=5,
  ghost4=6,
  interpolation=-1,
  periodic=-2,
  extrapolation=-3,
  unused=0
};
```

5.7.2 Constructors

SparseRepForMGF()

5.7.3 indexToEquation

int
indexToEquation(int n, int i1, int i2, int i3)

Description: Return the equation number for given indices

n (input): component number (n=0,1,..,numberOfComponents-1)

i1,i2,i3 (input): grid indices

Return value: The equation number.

5.7.4 setCoefficientIndex

int
setCoefficientIndex(const int & m,
const int & na, const Index & I1a, const Index & I2a, const Index & I3a,
const int & nb, const Index & I1b, const Index & I2b, const Index & I3b)

Description: Assign row and column numbers to entries in a sparse matrix. Rows and columns in the sparse matrix are numbered according to the values of (n,I1,I2,I3) where n is the component number and (I1,I2,I3) are the coordinate indicies on the grid. The component number n runs from 0 to the numberOfComponentsForCoefficients and is used when solving a system of equations.

m (input): assign row/column values for the m'th entry in the sparse matrix

na,I1a,I2a,I3a (input): defines the row(s)

Nb,I1b,I2b,I3b (input): defines the column(s)

5.7.5 setCoefficientIndex

int
setCoefficientIndex(const int & m,
const int & na, const Index & I1a, const Index & I2a, const Index & I3a,
const int & equationNumber0)

Description: Assign row and column numbers to entries in a sparse matrix. This routine is normally only used for assign equation numbers on CompositeGrid's when the equationNumber belongs to a point on a different MappedGrid. Rows and columns in the sparse matrix are numbered according to the values of (n,I1,I2,I3) where n is the component number and (I1,I2,I3) are the coordinate indicies on the grid. The component number n runs from 0 to the numberOfComponentsForCoefficients and is used when solving a system of equations.

m (input): assign row/column values for the m'th entry in the sparse matrix

na,I1a,I2a,I3a (input): defines the row(s)

equationNumber (input): defines an equation number

5.7.6 sizeOf

real
sizeOf(FILE *file = NULL) const

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

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

Return value: the number of bytes.

5.7.7 updateToMatchGrid

int
updateToMatchGrid(MappedGrid & mg,
int stencilSize0 = unchanged,
int numberOfGhostLines0 = unchanged,
int numberOfComponents0 = unchanged,
int offset0 = unchanged)

Description: Initialize the equationNumber and classify arrays. The equation number array is initialized according to value of stencilSize. The stencil width will be chosen to be pow(stencilSize,1/d) where d is the number of space dimensions. Thus

- If $3^d \leq$ stencilSize $< 5^d$ (d=space dimension) then the stencil is assumed to be a standard 3^d stencil and the first 3^d entries are initialized in the standard form. Any excess entries are given an equation number of 0 (unused).
- If $5^d \leq$ stencilSize $< 7^d$ then the stencil is assumed to be a standard 5^d setncil and initialized in the standard form. Any excess entries are given an equation number of 0 (unused).
- etc.
- If stencilSize is less than 3^d then equationNumber array is set to zero.

mg (input): update to match this grid.

stencilSize0 (input): maximum size for the stencil (for each component). By default (i.e. if no value is specified then stencilSize0 remains unchanged from its current value. (It is initially set to 9).

numberOfComponents0 (input): number of components. By default (i.e. if no value is specified then numberOfComponents0 remains unchanged from its current value. (It is initially set to 1).

offset0 (input): offset equation numbers by this amount. By default (i.e. if no value is specified then offset0 remains unchanged from its current value. (It is initially set to 0).

5.7.8 setParameters

void
setParameters(int stencilSize0 = unchanged,
int numberOfGhostLines0 = unchanged,
int numberOfComponents0 = unchanged,
int offset0 = unchanged)

Description: Set various parameters. Use this routine if you want to set the properties of the SparseRep object before you have a MappedGrid. You must call updateToMatchGrid for these values to take effect.

stencilSize0 (input): maximum size for the stencil (for each component). By default (i.e. if no value is specified then stencilSize0 remains unchanged from its current value. (It is initially set to 9).

numberOfComponents0 (input): number of components. By default (i.e. if no value is specified then numberOfComponents0 remains unchanged from its current value. (It is initially set to 1).

offset0 (input): offset equation numbers by this amount. By default (i.e. if no value is specified then offset0 remains unchanged from its current value. (It is initially set to 0).

5.7.9 setClassify

int
setClassify(const classifyTypes & type,
const Index & I1_, const Index & I2_, const Index & I3_, const Index & N
)

Description: Specify the classification for a set of Index values

5.7.10 equationToIndex

int
equationToIndex(const int eqnNo, int & n, int & i1, int & i2, int & i3)

Description: Convert an Equation Number to a point on a grid (Inverse of indexToEquation)

eqnNo0 (input): equation number

n (output): component number (n=0,1,..,numberOfComponents-1)

i1,i2,i3 (output): grid indices

5.7.11 fixUpClassify

int
fixUpClassify(realMappedGridFunction & coeff)

Description: Fixup up the classify array to take into account the mask array and periodicity

coeff (input): The coefficient matrix

6 Fourier Operators

6.1 General Info

Use this class to perform various operations on the Fourier transform of a real valued function such as

- forward and reverse transforms
- derivatives and integrals in fourier space

This class can be used to implement (pseudo) spectral appoximations to PDEs. The Overture class MappedGridOperators uses this class to compute spectral derivatives.

The fourier transform is represented as a real transform (sine-cosine).

By default the elements of the arrays that we operate on are u(0:nx-1,0:ny-1,0:nz-1,C) where C is a Range that species which components to operate on. (The array dimensions can be different from 0:nx-1, etc.). This can be changed to the form $u(R1, R2, R3, C)$ where $R1$ has length nx, $R2$ has length ny and $R3$ has length nz.

In practice you may keep a duplicate point in the array. You may declare an array to be u(0:nx,0:ny,0:nz) where u(0,all,all) == u(nx,all,all) These routines only change the values u(0:nx-1,0:ny-1,0:nz-1,C).

6.2 Constructors

```
FourierOperators(const int & numberOfDimensions_,
                 const int & nx_,
                 const int & ny_ =1,
                 const int & nz_ =1)
```

Description: Define the number of space dimensions and the number of grid points.

numberOfDimensions_: The number of space dimensions (1,2, or 3)

nx_, ny_, nz_: The number of grid points (minus one) in each dimension (nx,ny,nz should be a power of two or a product of small primes for efficiency).

Author: WDH

6.3 fourierLaplacian

```
void
fourierLaplacian(const RealArray & uHat,
                 RealArray & uLaplacianHat,
                 const int & power =1,
                 const Range & Components0 =nullRange)
```

Description: Apply the Laplacian operator (or powers of the Laplacian operator) in fourier space. The power can be positive or negative.

uHat (input) : the fourier transform

uLaplacianHat (output) : uHat multiplied by "$[-(k_x^2+k_y^2+k_z^2)]^{\texttt{power}}$". Note that the mean (i.e. the constant mode) is set to zero in all cases.

power (input): The power of the operator to apply.

Components0 (input) : optional components to operate on (default is all components)

Author: WDH

6.4 fourierDerivative

void
fourierDerivative(const RealArray & uHat,
RealArray & uHatDerivative,
const int & xDerivative =1,
const int & yDerivative =0,
const int & zDerivative =0,
const Range & Components0 =nullRange)

Description: Compute a derivative in Fourier space. The order of the derivative can be be positive or negative.

uHat (input) : the fourier transform

uHatDerivative (output) : The derivative in fourier space.

xDerivative (input): The order of the x-derivative

yDerivative (input): The order of the y-derivative

zDerivative (input): The order of the z-derivative

Components0 (input) : optional components to operate on (default is all components)

Author: WDH

6.5 fourierToReal

void
fourierToReal(const RealArray & uHat,
RealArray & u,
const Range & Components0 =nullRange)

Description: Perform a transform from fourier space to real space (backward transform)

uHat (input) : the fourier transform

u (output) : The array to be assigned the backward fourier transform.

Components0 (input) : optional components to operate on (default is all components)

Author: WDH

6.6 realToFourier

void
realToFourier(const RealArray & u,
RealArray & uHat,
const Range & Components0 =nullRange)

Description: Real space to fourier space (forward transform)

u (input) : The array to fourier transform.

uHat (output) : the fourier transform

Components0 (input) : optional components to operate on (default is all components)

Author: WDH

6.7 setDefaultRanges

void
setDefaultRanges(const Range & R1_,
const Range & R2_ =nullRange,
const Range & R3_ =nullRange)

Description: Change the Ranges over which the transforms are performed. This may also change the number of points. The operations will then be applied to u(R1_,R2_,R3_,C)

R1_,R2_,R3_ : new ranges

6.8 setDimensions

void
setDimensions(const int & numberOfDimensions_,
const int & nx_,
const int & ny_ =1,
const int & nz_ =1)

Description: Define the number of space dimensions and the number of grid points.

numberOfDimensions_: The number of space dimensions (1,2, or 3)

nx_, ny_, nz_: The number of grid points (minus one) in each dimension (nx,ny,nz should be a power of two or a product of small primes for efficiency).

Author: WDH

6.9 setPeriod

void
setPeriod(const real & xPeriod_,
const real & yPeriod_ = twoPi,
const real & zPeriod_ = twoPi)

Description: Set the period, default is 2*pi.

xPeriod_, yPeriod_, zPeriod_ (input) : The length of the periodic interval in each direction

Author: WDH

6.10 transform

void
transform(const int & forwardOrBackward,
const RealArray & u,
RealArray & uHat,
const Range & Components0)

Description: Perform a forward or backward fourier transform. (This routine is called by `realToFourier` and `fourierToReal`.

forwardOrBackward (input): 0=forward, 1=backward

u (input) : The array to fourier transform.

uHat (output) : the fourier transform

Components0 (input) : optional components to operate on (default is all components)

Author: WDH

6.11 Examples

6.11.1 Example using A++ arrays

Here is an example code demonstrating the use of this class with A++ arrays (file `Overture/examplesps.C`)

```
#include "FourierOperators.h"
//================================================================================
// Test out the FourierOperators Class
//================================================================================

// define a function and derivatives
#define U(x,y)   sin(px*x)*cos(py*y)

#define UX(x,y)  px*cos(px*x)*cos(py*y)
#define UY(x,y) -py*sin(px*x)*sin(py*y)
#define U_LAPLACIAN(x,y) -(px*px+py*py)*sin(px*x)*cos(py*y)
#define U_INVERSE_LAPLACIAN(x,y) sin(px*x)*cos(py*y)*(-1./(px*px+py*py))

```

```
#define X(i)  xPeriod*i/nx
#define Y(j)  yPeriod*j/ny

int
main(int argc, char *argv[])
{
  Overture::start(argc,argv);  // initialize Overture

  int nd=8,nx=8,ny=8;
  realArray u(nd,nd),uHat(nd,nd),u2(nd,nd),uHatX(nd,nd),ux(nd,nd);
  realArray x(nd,nd),y(nd,nd);
  Range R1(0,nx-1),R2(0,ny-1);

  // x is periodic with period xPeriod, y is periodic with period yPeriod
  real xPeriod=1., yPeriod=2., px=twoPi/xPeriod, py=twoPi/yPeriod;
  // assign values to x,y, and u
  int i,j;
  for( j=0; j<ny; j++ )
    for( i=0; i<nx; i++ )
    {
      x(i,j)=X(i);
      y(i,j)=Y(j);
    }
  u(R1,R2)=U(x(R1,R2),y(R1,R2));

  int numberOfDimensions=2;
  FourierOperators fourier(numberOfDimensions,nx,ny);
  fourier.setPeriod(xPeriod,yPeriod);

  u.display("Here is u");
  fourier.realToFourier( u,uHat );
  uHat.display("Here is uHat");
  fourier.fourierToReal( uHat,u2 );

  real maxError=max(fabs(u2-U(x,y)));
  cout << "Maximum error in F^-1(Fu) = " << maxError << endl;
  // u2.display("Here is F^-1(Fu back again ");

  fourier.fourierDerivative(uHat,uHatX,1);   // x derivative
  // wHatX.display("Here is wHatX");
  fourier.fourierToReal( uHatX,ux );
  maxError=max(fabs(ux-UX(x,y)));
  cout << "Maximum error in u.x = " << maxError << endl;

  fourier.fourierDerivative(uHat,uHatX,0,1);   // y derivative
  fourier.fourierToReal( uHatX,ux );
  maxError=max(fabs(ux-UY(x,y)));
  cout << "Maximum error in u.y = " << maxError << endl;

  fourier.fourierLaplacian(uHat,uHatX,1);   // xx+yy derivative
  fourier.fourierToReal( uHatX,ux );
  maxError=max(fabs(ux-U_LAPLACIAN(x,y)));
  cout << "Maximum error in u.xx+u.yy = " << maxError << endl;

  fourier.fourierLaplacian(uHat,uHatX,-1);   // (xx+yy)^-1 operator
  fourier.fourierToReal( uHatX,ux );
  maxError=max(fabs(ux-U_INVERSE_LAPLACIAN(x,y)));
  cout << "Maximum error in inverse laplacian = " << maxError << endl;

  Overture::finish();
  return 0;
}
```

6.11.2 Example using mappedGridFunctions and MappedGridOperators

Here is an example code demonstrating the use of the MappedGridOperators to compute pseudo-spectral derivatives. The `MappedGridOperators` contain a pointer to a `FourierOperators` object which can be obtained with the `getFourierOperators()` member function. You may want to access this pointer in order to write more efficient code or to access the extra functionality that is found in the `FourierOperators` class.
(file `Overture/examplestestSpectral.C`)

```
#include "Overture.h"
#include "OGTrigFunction.h"  // Trigonometric function
#include "MappedGridOperators.h"
#include "LineMapping.h"
#include "Square.h"
#include "BoxMapping.h"
#include "NameList.h"
#include "FourierOperators.h"

//================================================================================
//  Test out the MappedGridOperators pseudo-spectral derivatives
//================================================================================
int
main(int argc, char *argv[])
{
  Overture::start(argc,argv);  // initialize Overture

  int debug=0, numberOfDimensions=2;

  int nx[3] = { 8,8,1};   // number of grid points (minus 1) in each direction
  // frequencies for exact solution, cos(fx[0]*pi*x)*cos(fx[1]*pi*y)*cos(fx[2]*pi*z)
  int fx[3] = { 2,2,0 };
  real period[3] = {1.,1.,1.};

  NameList nl;                 // The NameList object allows one to read in values by name
  aString name(80),answer(80);
  printf(
   " Parameters for Example 3: \n"
   " ------------------------ \n"
   "   name                                              type    default  \n"
   "numberOfDimensions (nd=) (assign first)             (int)      %i     \n"
   "nx,ny,nx                                            (int)   %i %i %i \n"
   "fx,fy,fz (fx*xPeriod=even)                          (int)   %i %i %i       \n"
   "xPeriod,yPeriod,zPeriod                             (real)  %e %e %e       \n",
     numberOfDimensions,nx[0],nx[1],nx[2],fx[0],fx[1],fx[2],period[0],period[1],period[2]);

  // ==========Loop for changing parameters=======================
  for( ;; )
  {
    cout << "Enter changes to variables, exit to continue" << endl;
    cin >> answer;
```

```
      if( answer=="exit" ) break;
      nl.getVariableName( answer, name );   // parse the answer
      if( name== "numberOfDimensions" || name=="nd" )
      {
        numberOfDimensions=nl.intValue(answer);
        if( numberOfDimensions==1 )
        {
          nx[1]=nx[2]=1;  fx[1]=fx[2]=0;
        }
        else if( numberOfDimensions==2 )
        {
          nx[1]=8, nx[2]=1; fx[1]=2, fx[2]=0;
        }
        else
        {
          nx[1]=8, nx[2]=8; fx[1]=2, fx[2]=2;
        }
      }
      else if( name== "nx" )
        nx[0]=nl.realValue(answer);
      else if( name== "ny" )
        nx[1]=nl.realValue(answer);
      else if( name== "nz" )
        nx[2]=nl.realValue(answer);
      else if( name== "fx" )
        fx[0]=nl.realValue(answer);
      else if( name== "fy" )
        fx[1]=nl.realValue(answer);
      else if( name== "fz" )
        fx[2]=nl.realValue(answer);
      else if( name== "xPeriod" )
        period[0]=nl.realValue(answer);
      else if( name== "yPeriod" )
        period[1]=nl.realValue(answer);
      else if( name== "zPeriod" )
        period[2]=nl.realValue(answer);
      else
        cout << "unknown response: [" << name << "]" << endl;
    }

    LineMapping line;
    SquareMapping square(0.,period[0],0.,period[1]);                  // Make a mapping, unit square
    BoxMapping box(0.,period[0],0.,period[1],0.,period[2]);;
    // choose a line, square or box depending on the number of dimensions
    Mapping & map = numberOfDimensions==1 ? (Mapping&)line :
                  ( numberOfDimensions==2 ? (Mapping&)square : (Mapping&)box );

    for( int axis=0; axis<numberOfDimensions; axis++ )
    {
      map.setGridDimensions(axis,nx[axis]+1);               // number of grid points
      map.setIsPeriodic(axis,Mapping::functionPeriodic);
    }
    MappedGrid mg(map);                              // MappedGrid for a square
    mg.update();

    Range all;
    realMappedGridFunction u(mg);
```

```

  MappedGridOperators op(mg);                      // define some differential operators
  u.setOperators(op);                          // Tell u which operators to use
  // ---- compute all derivatives with the pseudo-spectral method ----
  u.getOperators()->setOrderOfAccuracy(MappedGridOperators::spectral);

  OGTrigFunction true(fx[0],fx[1],fx[2]);  // create an exact solution (Twilight-Zone solution)

  real error;
  int n=0;      // only test first component

  Index I1,I2,I3,N;
  getIndex(mg.dimension(),I1,I2,I3);             // assign I1,I2,I3, all grid points including ghost
  u(I1,I2,I3)=true(mg,I1,I2,I3,n,0.);          // assign true solution

  error = max(fabs(u.x()(I1,I2,I3)-true.x(mg,I1,I2,I3,n)));
  cout << "u.x : Maximum error (spectral) = " << error << endl;
  if( debug & 4 )
  {
    fabs( u.x()(I1,I2,I3)-true.x(mg,I1,I2,I3,n)).display("Error in u.x");
    true.x(mg,I1,I2,I3,n).display(" true u.x");
    u.x()(I1,I2,I3).display("computed u.x");
    true(mg,I1,I2,I3,n).display(" true u");
    u(I1,I2,I3).display("discrete u");
  }

  error = max(fabs(u.y()(I1,I2,I3)-true.y(mg,I1,I2,I3,n)));
  cout << "u.y : Maximum error (spectral) = " << error << endl;
  if( debug & 4 )
  {
    fabs(u.y()(I1,I2,I3)-true.y(mg,I1,I2,I3,n)).display("Error in u.y");
    u.y()(I1,I2,I3).display("u.y");
    true.y(mg,I1,I2,I3,n).display("true.y");
  }

  error = max(fabs(u.xx()(I1,I2,I3)-true.xx(mg,I1,I2,I3,n)));
  cout << "u.xx : Maximum error (spectral) = " << error << endl;

  error = max(fabs(u.xy()(I1,I2,I3)-true.xy(mg,I1,I2,I3,n)));
  cout << "u.xy : Maximum error (spectral) = " << error << endl;

  error = max(fabs(u.yy()(I1,I2,I3)-true.yy(mg,I1,I2,I3,n)));
  cout << "u.yy : Maximum error (spectral) = " << error << endl;

  error = max(fabs(u.laplacian()(I1,I2,I3)-(true.xx(mg,I1,I2,I3,n)+true.yy(mg,I1,I2,I3,n)
                                          +true.zz(mg,I1,I2,I3,n))));
  cout << "u.laplacian : Maximum error (spectral) = " << error << endl;

  error = max(fabs(u.z()(I1,I2,I3)-true.z(mg,I1,I2,I3,n)));
  cout << "u.z : Maximum error (spectral) = " << error << endl;

  error = max(fabs(u.xz()(I1,I2,I3)-true.xz(mg,I1,I2,I3,n)));
  cout << "u.xz : Maximum error (spectral) = " << error << endl;

  error = max(fabs(u.yz()(I1,I2,I3)-true.yz(mg,I1,I2,I3,n)));
  cout << "u.yz : Maximum error (spectral) = " << error << endl;

```

```
    error = max(fabs(u.zz()(I1,I2,I3)-true.zz(mg,I1,I2,I3,n)));
    cout << "u.zz : Maximum error (spectral) = " << error << endl;

    // ****************************************************************************
    // Now get the FourierOperators (this must be done only after at least one
    // derivative has been computed)
    // ****************************************************************************
    FourierOperators & fourier = *op.getFourierOperators();

    // compute the transform directly
    realMappedGridFunction uHat(mg);
    fourier.realToFourier( u,uHat );
    uHat.display("Here is uHat");


    Overture::finish();
    cout << "Program Terminated Normally! \n";
    return 0;
  }
```

References

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

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

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

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

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

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

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

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

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

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

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

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

Index