Composable Difference Operators for Coordinate Invariant Partial Differential Equations

Malin Ljungberg

March 16, 2007

Abstract

Computer simulations are a cost efficient complement to laboratory experiments. Software for the solution of partial differential equations is part of the basic infrastructure for the evolving field of Computational Science and Engineering. Numerical analysts are involved in the development of suitable methods and algorithms for the solution of a wide range of partial differential equations.

Based on the needs of a numerical analyst, we here identify requirements on software for the solution of partial differential equations. In particular, we look at support for flexible formulations of finite difference methods on curvilinear structured grids.

We present FlexOp, a software solution that meets the requirements that have been identified. The semantics of the FlexOp is highly mathematical, and includes coordinate invariant operators. The discretization strategy is specified using parameterized classes, and can be chosen independently for different instances of the same operator. The use of static polymorphism allows for a flexible and efficient implementation of the application of the discretized operator.

In order to assess the FlexOp, we use them to implement a compact fourth-order Numerov method, using the TENGOME infrastructure as a basis and C++ as the implementation language.

We find that the FlexOp are easy to use, because of the high agreement between mathematical derivation and implementation of the method. Compared with a special purpose implementation for a particular discretization scheme, we find that FlexOp offer a significant reduction in the number of lines of code, together with an associated enhanced maintainability. The increased flexibility comes without a cost in the form of an increase of execution time, when compared with the special purpose application.

1 Introduction

Computer simulations are a cost efficient complement to laboratory experiments. Software for the solution of partial differential equations is part of the basic infrastructure for the evolving field of Computational Science and Engineering. Numerical analysts are involved in the development of suitable methods and algorithms for the solution of a wide range of partial differential equations.

The needs of a numerical analyst are quite different than those of a computational scientist. Instead of tweaking parameters which relate to a particular application area, the numerical analyst needs to vary aspects of the numerical methods themselves. For the numerical analyst, the application problem is secondary. The relevant outcome of an experiment is not the solution to the problem, but rather how well the selected algorithm performed for that particular problem. One of the challenges for the numerical analyst is to find ways to easily tailor numerical methods for a range of physical problems.

The laws of Physics do not depend on the choice of coordinate system, which is why coordinate invariant differential operators are preferred when describing physical problems. When such a problem is to be solved, the geometry of the problem sometimes makes curvilinear coordinates a natural choice [10]. The metric coefficients involved in the formulation, for a specific coordinate system, introduce an additional level of complexity for the discretization of such a problem.

The goal of the present work is the design of software representations of composable, general differential operators, to be used in the numerical solution of Partial Differential Equations (PDEs) on curvilinear structured grids. We call these software representations the FlexOp, since they need to be flexible with respect to composition as well as discretization. The FlexOp are intended, as a tool for the numerical analyst, to be used when developing and enhancing algorithms for the solution of PDEs. We are in particular looking at Finite Difference Methods (FDM), since they constitute an efficient approach to the numerical solution of PDEs on structured grids.

The FlexOp are intended to be used as a part of a problem solving environment. Aspects which are not covered by the FlexOp, such as for example parallelism or preconditioning, should be handled by other parts of the environment. This modular approach to the construction of problem solving environments provides flexibility on one hand, and user-friendliness on the other. Separating modules with different functionality makes it easier to improve and optimize them independently.

There is a tradeoff between flexibility and efficiency. In order to meet these conflicting interests we explore the use of generative programming. The FlexOp are implemented using template based, static polymorphism, as well as fixed size arrays. By choosing compile time flexibility over run

time flexibility, we increase the opportunities for compiler optimizations, and enable a vastly increased flexibility without a loss of run time efficiency.

The outline of the present article is as follows. In Section 2 we present a motivating example; the development of a fourth-order compact Numerov scheme for the Poisson equation. In Section 3 we identify requirements on the FlexOp, based on the motivating example. In Section 4 we discuss software solutions satisfying the requirements. In Section 5 we present a pilot implementation of the FlexOp. In Section 6 we show how the motivating example is treated using the pilot implementation. In Section 7 we evaluate the FlexOp. Section 8 contains our conclusions.

2 Motivating Example

When finding a numerical solution to a PDE, using a gridded method such as a FDM, the grid, on which the problem is discretized, provides a basis for the software representation of the problem. For some geometries, a curvilinear structured grid is a natural choice.

In FDM partial derivatives are approximated by difference operators, involving data on a limited number of neighboring grid points. Retrieval of these data can be made efficient through the use of data structures based on the structure of the grid.

The shape of the problem domain is a determining factor for the selection of discretization grid. As an example, Figure 1 shows the cross-section of a solid plate of varying thickness. The width of the plate, which is the direction not shown, is considered to be infinite. The geometry has been discretized along the coordinate lines of orthogonal curvilinear coordinates. These coordinates are body fitted, which simplifies the formulation of boundary conditions.

A PDE formulated in terms of curvilinear coordinates will in general contain metric coefficients. As a motivating example, we will study the temperature distribution in the plate in Figure 1 at thermal equilibrium. This is described by the Poisson equation

$$-\nabla \cdot (\kappa \nabla u) = f, \tag{1}$$

where u is the temperature, κ the thermal conductivity, and f is a source term.

Written in coordinate form, using the curvilinear coordinates ξ_1 and ξ_2, and assuming no changes in the ξ_3 direction, the Poisson equation becomes

$$-\frac{\partial}{\partial \xi_1}\left(a_1 \frac{\partial u}{\partial \xi_1}\right) - \frac{\partial}{\partial \xi_2}\left(a_2 \frac{\partial u}{\partial \xi_2}\right) = \gamma f(\xi_1, \xi_2) \equiv g(\xi_1, \xi_2) \tag{2}$$

where the coefficients are

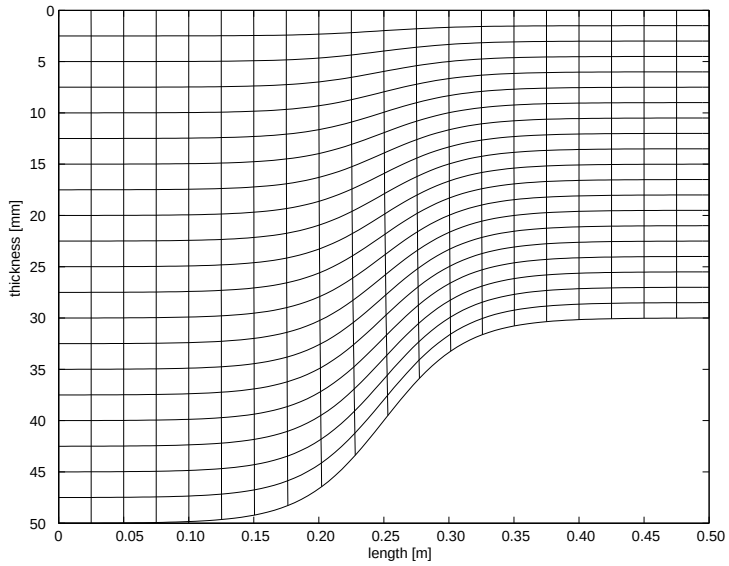

Figure 1: Cross-sectional grid for a solid plate of variable thickness. The orthogonality of the grid lines is less clear in the figure, due to the compression of the length axis.

$$a_1 = \frac{\gamma\kappa}{\eta_1^2}, \text{ and } a_2 = \frac{\gamma\kappa}{\eta_2^2},$$

where γ and η_i are metric coefficients. Note that γ and η_i depend on the values of ξ_1 and ξ_2. The treatment of the boundary conditions is deferred to the appendix, in order to keep the presentation short.

The use of curvilinear coordinates thus results in an equation involving variable coefficients.

One of the tasks for a numerical analyst is to experiment with different discretizations of differential operators, particularly in combination with different methods for solving the discretized system. When the numerical analyst has decided which discretization she wants to use, she must figure out at which offsets the various terms and coefficients should be evaluated. A second-order accurate discretization of (2) is shown here as an illustration.

$$\begin{aligned} &-\frac{1}{h_1^2}(a_1^{i+\frac{1}{2},j}(u^{i+1,j}-u^{i,j}) - a_1^{i-\frac{1}{2},j}(u^{i,j}-u^{i-1,j})) \\ &-\frac{1}{h_2^2}(a_2^{i,j+\frac{1}{2}}(u^{i,j+1}-u^{i,j}) - a_2^{i,j-\frac{1}{2}}(u^{i,j}-u^{i,j-1})) = g^{i,j}, \end{aligned} \tag{3}$$

where $u^{i,j}$ denotes the approximate solution at the point (ξ_1^i, ξ_2^j).

This second-order discretization results in eight terms involving metric coefficients evaluated on the base grid as well as on two different staggered grids.

The next step is to translate the discretized operator into program code. A common approach to finding a solution to a system of equations, such as (3), is the use of an iterative method. The system is then transformed into a matrix vector equation, $Bu = g$, where the vector u holds the unknowns. The operator B could be represented as a banded matrix with entries that depend on i and j, because the metric coefficients, and possibly also the physical parameters of the problem, are variable. We introduce the tensor notation of Otto and Åhlander [15], [2] to express the nonzero elements of B:

$$\begin{aligned}
B^{i,j}_{i,j} &= \frac{1}{h_1^2}\left(a_1^{i-\frac{1}{2},j} + a_1^{i+\frac{1}{2},j}\right) + \frac{1}{h_2^2}\left(a_2^{i,j-\frac{1}{2}} + a_2^{i,j+\frac{1}{2}})\right), \\
B^{i,j}_{i-1,j} &= -\frac{1}{h_1^2}a_1^{i-\frac{1}{2},j}, \\
B^{i,j}_{i+1,j} &= -\frac{1}{h_1^2}a_1^{i+\frac{1}{2},j}, \\
B^{i,j}_{i,j-1} &= -\frac{1}{h_2^2}a_2^{i,j-\frac{1}{2}}, \\
B^{i,j}_{i,j+1} &= -\frac{1}{h_2^2}a_2^{i,j+\frac{1}{2}}.
\end{aligned} \tag{4}$$

For stability reasons implicit time-marching methods (using matrix operators) are sometimes applied to time dependent problems, even though explicit methods may be the more obvious choice for these, because of their simplicity. In an explicit method, the estimated values of $u^{i,j}(t_k)$ for a certain time level are computed as a difference expression in the values at the previous time level, $u^{i,j}(t_{k-1})$. This may be done by constructing the matrix operator and performing a matrix-vector multiplication, but this only makes sense in the case that B is time invariant, and does not need to be updated for each time step. For the case of time dependent coefficients, it is better to evaluate the difference expression directly, as it stands.

The second-order discretization of the problem, that we have presented, shows some of the intricacies involved when writing software for the numerical solution of PDEs, which have been discretized on curvilinear structured grids. We will continue by describing a compact fourth-order discretization, which is derived using a Numerov technique [4, 5]. For this case the difficulties that arise will be even more obvious.

By introducing the operator notation

$$D_0^1 u^{i,j} = \frac{1}{2h_1}(u^{i+1,j} - u^{i-1,j}), \tag{5}$$

and

$$D^1_{1/2}u^{i,j} = \frac{1}{h_1}(u^{i+\frac{1}{2},j} - u^{i-\frac{1}{2},j}). \tag{6}$$

we can write a Taylor series expansion for the discretization of the first term of the differential equation (2) as

$$\begin{aligned}
&-\frac{\partial}{\partial\xi_1}\left(a_1\frac{\partial u}{\partial\xi_1}\right)^{i,j} = \\
&-D^1_{1/2}(a_1^{i,j}D^1_{1/2}u^{i,j}) \\
&+h_1^2\left(\frac{1}{12}a_1\frac{\partial^4 u}{\partial\xi_1^4} + \frac{1}{6}\frac{\partial a_1}{\partial\xi_1}\frac{\partial^3 u}{\partial\xi_1^3} + \frac{1}{8}\frac{\partial^2 a_1}{\partial\xi_1^2}\frac{\partial^2 u}{\partial\xi_1^2} + \frac{1}{24}\frac{\partial^3 a_1}{\partial\xi_1^3}\frac{\partial u}{\partial\xi_1}\right)^{i,j} + \mathcal{O}(h_1^4).
\end{aligned} \tag{7}$$

This can be reformulated as

$$\begin{aligned}
&-\frac{\partial}{\partial\xi_1}\left(a_1\frac{\partial u}{\partial\xi_1}\right)^{i,j} = \\
&-D^1_{1/2}(a_1^{i,j}D^1_{1/2}u^{i,j}) \\
&+\frac{h_1^2}{12}\frac{\partial^3}{\partial\xi_1^3}\left(a_1\frac{\partial u}{\partial\xi_1}\right)^{i,j} \\
&-\frac{h_1^2}{12}\frac{\partial}{\partial\xi_1}\left(a_1^{-1}\frac{\partial a_1}{\partial\xi_1}\frac{\partial}{\partial\xi_1}\left(a_1\frac{\partial u}{\partial\xi_1}\right)\right)^{i,j} \\
&+\frac{h_1^2}{12}\frac{\partial}{\partial\xi_1}\left(a_1^{-1}\left(\frac{\partial a_1}{\partial\xi_1}\right)^2\frac{\partial u}{\partial\xi_1}\right)^{i,j} \\
&-\frac{h_1^2}{24}\frac{\partial}{\partial\xi_1}\left(\frac{\partial^2 a_1}{\partial\xi_1^2}\frac{\partial u}{\partial\xi_1}\right)^{i,j} + \mathcal{O}(h_1^4).
\end{aligned} \tag{8}$$

The second term of the differential equation can be expanded in an analogous fashion.

Now we replace higher order derivatives with expressions from the differential equation, in order to obtain a compact scheme. We have that

$$\begin{aligned}
\frac{\partial}{\partial\xi_1}\left(a_1\frac{\partial u}{\partial\xi_1}\right) &= -\frac{\partial}{\partial\xi_2}\left(a_2\frac{\partial u}{\partial\xi_2}\right) - g, \text{and} \\
\frac{\partial}{\partial\xi_2}\left(a_2\frac{\partial u}{\partial\xi_2}\right) &= -\frac{\partial}{\partial\xi_1}\left(a_1\frac{\partial u}{\partial\xi_1}\right) - g,
\end{aligned}$$

Using these substitutions and the Taylor expansions, we get

$$
\begin{aligned}
0 = & -\frac{\partial}{\partial \xi_1}\left(a_1 \frac{\partial u}{\partial \xi_1}\right)^{i,j} - \frac{\partial}{\partial \xi_2}\left(a_2 \frac{\partial u}{\partial \xi_2}\right)^{i,j} - g^{i,j} = \\
& -D^1_{1/2}(a_1^{i,j} D^1_{1/2} u^{i,j}) - D^2_{1/2}(a_2^{i,j} D^2_{1/2} u^{i,j}) - g^{i,j} \\
& -\frac{h_1^2}{12}\frac{\partial^2}{\partial \xi_1^2}\left(\frac{\partial}{\partial \xi_2}\left(a_2 \frac{\partial u}{\partial \xi_2}\right)\right)^{i,j} - \frac{h_2^2}{12}\frac{\partial^2}{\partial \xi_2^2}\left(\frac{\partial}{\partial \xi_1}\left(a_1 \frac{\partial u}{\partial \xi_1}\right)\right)^{i,j} \\
& -\frac{h_1^2}{12}\left(\frac{\partial^2 g}{\partial \xi_1^2}\right)^{i,j} - \frac{h_2^2}{12}\left(\frac{\partial^2 g}{\partial \xi_2^2}\right)^{i,j} \\
& +\frac{h_1^2}{12}\frac{\partial}{\partial \xi_1}\left(a_1^{-1}\frac{\partial a_1}{\partial \xi_1}\frac{\partial}{\partial \xi_2}\left(a_2 \frac{\partial u}{\partial \xi_2}\right)\right)^{i,j} + \frac{h_2^2}{12}\frac{\partial}{\partial \xi_2}\left(a_2^{-1}\frac{\partial a_2}{\partial \xi_2}\frac{\partial}{\partial \xi_1}\left(a_1 \frac{\partial u}{\partial \xi_1}\right)\right)^{i,j} \\
& +\frac{h_1^2}{12}\frac{\partial}{\partial \xi_1}\left(a_1^{-1}\frac{\partial a_1}{\partial \xi_1} g\right)^{i,j} + \frac{h_2^2}{12}\frac{\partial}{\partial \xi_2}\left(a_2^{-1}\frac{\partial a_2}{\partial \xi_2} g\right)^{i,j} \\
& +\frac{h_1^2}{12}\frac{\partial}{\partial \xi_1}\left(a_1^{-1}\left(\frac{\partial a_1}{\partial \xi_1}\right)^2 \frac{\partial u}{\partial \xi_1}\right)^{i,j} + \frac{h_2^2}{12}\frac{\partial}{\partial \xi_2}\left(a_2^{-1}\left(\frac{\partial a_2}{\partial \xi_2}\right)^2 \frac{\partial u}{\partial \xi_2}\right)^{i,j} \\
& -\frac{h_1^2}{24}\frac{\partial}{\partial \xi_1}\left(\frac{\partial^2 a_1}{\partial \xi_1^2}\frac{\partial u}{\partial \xi_1}\right)^{i,j} - \frac{h_2^2}{24}\frac{\partial}{\partial \xi_2}\left(\frac{\partial^2 a_2}{\partial \xi_2^2}\frac{\partial u}{\partial \xi_2}\right)^{i,j} + \mathcal{O}(h_1^4) + \mathcal{O}(h_2^4).
\end{aligned}
\tag{9}
$$

Replacing derivatives on the right-hand side by second-order approximations gives the following fourth-order discretization of the differential equation (2).

$$
\begin{aligned}
&-D^1_{1/2}(a_1^{i,j}D^1_{1/2}u^{i,j}) - D^2_{1/2}(a_2^{i,j}D^2_{1/2}u^{i,j}) \\
&-\frac{h_1^2}{12}D^1_{1/2}D^1_{1/2}D^2_{1/2}\left(a_2^{i,j}D^2_{1/2}u^{i,j}\right) \\
&-\frac{h_2^2}{12}D^2_{1/2}D^2_{1/2}D^1_{1/2}\left(a_1^{i,j}D^1_{1/2}u^{i,j}\right) \\
&+\frac{h_1^2}{12}D^1_0\left(\left(a_1^{-1}\frac{\partial a_1}{\partial \xi_1}\right)^{i,j} D^2_{1/2}\left(a_2^{i,j}D^2_{1/2}u^{i,j}\right)\right) \\
&+\frac{h_2^2}{12}D^2_0\left(\left(a_2^{-1}\frac{\partial a_2}{\partial \xi_2}\right)^{i,j} D^1_{1/2}\left(a_1^{i,j}D^1_{1/2}u^{i,j}\right)\right) \\
&+\frac{h_1^2}{12}D^1_{1/2}\left(\left(a_1^{-1}\left(\frac{\partial a_1}{\partial \xi_1}\right)^2\right)^{i,j} D^1_{1/2}u^{i,j}\right) \\
&+\frac{h_2^2}{12}D^2_{1/2}\left(\left(a_2^{-1}\left(\frac{\partial a_2}{\partial \xi_2}\right)^2\right)^{i,j} D^2_{1/2}u^{i,j}\right) \\
&-\frac{h_1^2}{24}D^1_{1/2}\left(\left(\frac{\partial^2 a_1}{\partial \xi_1^2}\right)^{i,j} D^1_{1/2}u^{i,j}\right) - \frac{h_2^2}{24}D^2_{1/2}\left(\left(\frac{\partial^2 a_2}{\partial \xi_2^2}\right)^{i,j} D^2_{1/2}u^{i,j}\right) = \\
&= g^{i,j} + \frac{h_1^2}{12}\left(\frac{\partial^2 g}{\partial \xi_1^2}\right)^{i,j} + \frac{h_2^2}{12}\left(\frac{\partial^2 g}{\partial \xi_2^2}\right)^{i,j} \\
&-\frac{h_1^2}{12}D^1_0\left(\left(a_1^{-1}\frac{\partial a_1}{\partial \xi_1}\right)^{i,j} g^{i,j}\right) - \frac{h_2^2}{12}D^2_0\left(\left(a_2^{-1}\frac{\partial a_2}{\partial \xi_2}\right)^{i,j} g^{i,j}\right)
\end{aligned}
\tag{10}
$$

where the derivatives of the metric coefficient a_1 are approximated as

$$
\begin{aligned}
\left(\frac{\partial a_1}{\partial \xi_1}\right)^{i,j} &\approx D^1_0 a_1^{i,j}, \text{and} \\
\left(\frac{\partial^2 a_1}{\partial \xi_1^2}\right)^{i,j} &\approx D^1_{1/2}D^1_{1/2}a_1^{i,j},
\end{aligned}
$$

and similarly for a_2. The source term, g, and its second derivatives are known.

Equation (10) is written on a compact form through the use of the difference operators D^i_0 and $D^i_{1/2}$. When the expression is to be evaluated in the code, it is usually necessary to write out all the terms explicitly. Here we show what this would look like. We consider *the third term only*, in order to save space. The third term

$$-\frac{h_1^2}{12}D^1_{1/2}D^1_{1/2}D^2_{1/2}\left(a_2^{i,j}D^2_{1/2}u^{i,j}\right) \tag{11}$$

expands to

$$\begin{aligned}-\frac{h_2^{-2}}{12}\Big(&a_2^{i+1,j+\frac{1}{2}}u^{i+1,j+1}-\left(a_2^{i+1,j+\frac{1}{2}}+a_2^{i+1,j-\frac{1}{2}}\right)u^{i+1,j}+a_2^{i+1,j-\frac{1}{2}}u^{i+1,j-1}\\ &-2\left(a_2^{i,j+\frac{1}{2}}u^{i,j+1}-\left(a_2^{i,j+\frac{1}{2}}+a_2^{i,j-\frac{1}{2}}\right)u^{i,j}+a_2^{i,j-\frac{1}{2}}u^{i,j-1}\right)\\ &+a_2^{i-1,j+\frac{1}{2}}u^{i-1,j+1}-\left(a_2^{i-1,j+\frac{1}{2}}+a_2^{i-1,j-\frac{1}{2}}\right)u^{i-1,j}+a_2^{i-1,j-\frac{1}{2}}u^{i-1,j-1}\Big)\end{aligned} \tag{12}$$

In the matrix-vector formulation of the problem, $Bu = g$, this term contributes to nine of the entries of the matrix operator, B, for each i, j-pair. Since each of these contributions contains metric coefficients, which are variable, they will, in general, be different for different values of i and j. The other nine terms of the left-hand side of (10) will contribute in a similar way. The full discretization of the left-hand side contains 68 terms.

We can see that the implementation of a numerical method, such as this Numerov method, can contain many terms with varying coefficients and indices. Typing these terms into the code by hand is cumbersome and error prone.

3 Aims of the Software

The application example illustrates the pitfalls that the numerical analyst may experience when implementing a numerical method. An intricate numerical approximation is found based on a detailed mathematical derivation. The numerical analyst then has to translate this approximation into program code. This can be

a) as entries in a matrix operator to be applied to the operand, or

b) in the form of algebraic expressions involving the operand, as in (12).

Going from the mathematical basis for the numerical method, such as (9), to either of these is a process which is highly error prone, due to the detail involved. Every factor and every index of every factor must be entered correctly.

Furthermore, because of the difficulties involved in coding such a numerical method, it is also hard to experiment with variations in the implementation of a numerical method. Picking a different discretization for one term in

(10) would result in changes in many of the elements of the matrix operator, as in a), or in a large number of the terms in the discretized equation, as in b).

Our goal is to supply a set of classes, which we will call Flexible Operators (FlexOp) to support the numerical analyst in the implementation of FDM on curvilinear structured grids. In view of the discussion above, we conclude that FlexOp should in particular meet the following requirements:

Requirement 1: a high level language of description for numerical approximations,

Requirement 2: support for selection of different discretization schemes for different parts of the expression,

Requirement 3: automatic inclusion of metric coefficients for coordinate invariant operators, and

Requirement 4: efficient evaluation of algebraic and difference expressions on large data sets, both

i. when using a matrix-vector formulation, and

ii. for direct evaluation.

When we investigate how others have handled these requirements, we find that the area of software for FDM on curvilinear structured grids is not very well covered in the literature, in spite of its potential for producing highly efficient PDE solvers. We believe that this is due to the technical difficulties involved with including the metric coefficients in the software. Two efforts that should be mentioned in this context are Sophus [7] and Overture [6].

Sophus uses finite element methods, rather than FDM, but is still relevant here, because of the similarity in approach. Sophus contains high level abstractions, such as coordinate invariant operators. The developer is thus relieved of the responsibility for the metric coefficients, which are included automatically by the software. Sophus uses a special purpose C++ preprocessor [3] to convert the code written, using higher level abstractions, into code that is more easily optimized by the compiler. This is the way in which Requirements 1 and 3 are combined with Requirement 4 in Sophus.

Overture also satisfies many, but not all, of the requirements above. Requirement 2 is not fulfilled to our satisfaction. Overture supports second and fourth-order accurate discretizations of derivatives. These discretizations are selected in the **MappedGridOperators** class. Physical fields are represented by the class **MappedGridFunction**. The way difference operators are discretized on a particular field can not be varied independently for each particular instance of a difference operator in an expression. Instead they are

all determined by the particular instance of the **MappedGridOperators** class, that is associated with a field. Requirement 4(ii) is also not fulfilled by Overture, since the components of a difference expression are evaluated consecutively, which means that a complicated expression will induce several loops over all the grid points. A more efficient implementation would be to evaluate all components within the same loop over the grid points.

4 Solutions

In the previous section, we specified requirements that the FlexOp software is supposed to fulfill. Here we will describe our software solution, and also discuss other options considered.

4.1 Modeling of Difference Approximations

According to Requirements 1 and 2, FlexOp should provide a high level language of description for numerical approximations, which includes support for selection of different discretization schemes for different parts of the expression. This means that we want a software representation that reflects the differential operator that is being discretized, and where the actual discretization of the operator is selectable. One option could be to give the discretization scheme, etc. as parameters in a function call. Listing 1 shows what a representation of (11) could look like, using this approach.

```
-h1*h1/12*partial(Xdir,Dhalf,
                  partial(Xdir,Dhalf,
                          partial(Ydir,Dhalf,
                                  a2*partial(Ydir,
                                             Dhalf)(u))));
```

Listing 1: A possible software representation of a discretization of the term in (11)

In Listing 1, the method `partial` represents the continuous differential operator. The parameters `Xdir` and `Ydir` are used to indicate along which coordinate line the differentiation is to be applied. The discretization of the `partial` operator is prescribed by another parameter, which in this example is given as `Dhalf`.

The representation of Listing 1 supplies the desired flexibility, provided that such expressions can be combined into larger algebraic expressions, e.g., a discretization such as (10).

A disadvantage of selecting the discretization through the use of a function parameter is that the selection will not be known until the code is executed, and the exact action of the code will be determined at that point, using a selection mechanism, such as `if`.

In order to increase the performance of the code, we choose to make the selection of the discretization already at compile time, using the template mechanism. The software representation of the term in question in FlexOp is

```
-h1*h1/12*partial<Dhalf,Xdir>(partial<Dhalf,Xdir>(
          partial<Dhalf,Ydir>(a2*
                              partial<Dhalf,Ydir>(u))));
```

Listing 2: FlexOp representation of a discretization of the term in (11).

where the template parameters `Xdir`, `Ydir`, and `Dhalf` are types. Using template parameters instead of ordinary function parameters allows for compile time resolution of the actual action of the `partial` method.

The FlexOp representation in Listing 2 has a strong resemblance to the continuous difference operator that it is a discretization of. Discretization is determined through the use of a set of types, such as `Dhalf`. These types are called the `DiffOp`, and are recursively composable. A `DiffOp` is used to specify the discretization each time an operator is used, which means that the same operator can be discretized in several different ways in the same expression.

We conclude that the FlexOp discretized operators provide the high level language of Requirement 1 as well as the flexibility of discretization of Requirement 2.

4.2 Coordinate Invariant Operators

The third requirement that was identified was that metric coefficients should be handled automatically for the case of differential operators that have a coordinate invariant form. Listing 3 shows how one possible discretization of the Laplacian is represented using FlexOp. The discretization selection is done by providing a parameter, in this case `Dhalf` to the `divergence` and `grad` methods, which are representations of the coordinate invariant continuous operators.

```
divergence<Dhalf>(grad<Dhalf>(u));
```

Listing 3: FlexOp representation of a second-order accurate discretization of $\nabla \cdot \nabla u$.

When the expression in Listing 3 is evaluated, the result is computed using the specified discretization, as well as all the correct metric coefficients. The FlexOp representation of coordinate invariant operators provides a great service to the numerical analyst, who is relieved of the burden of handling discretization indices as well as metric coefficients.

4.3 The Stencil in FlexOp

Requirement 4(i) above stated the need for efficient evaluation of algebraic and difference expressions for the case when a matrix operator is used. The `Stencil` class provides a way of transforming a difference expression, described in a high level language, as in (10), into a matrix operator, in a format that lends itself to efficient matrix-vector multiplication.

A `Stencil` is constructed from a difference expression, which has the same form as when it is to be directly evaluated. The only restriction is that the expression must be linear in the operand. A dummy argument, supplied by the `Stencil` itself, is used in place of the operand, as illustrated in Listing 4, which shows how a `Stencil` is formed and applied for the Helmholtz operator; $\nabla \cdot \nabla u + \kappa u$.

```
  // op is the dummy operand
Stencil<Mut>::OpType
  op(Stencil<Mut>::op());

Stencil<Mut>
  Helmholtz(divergence<Dhalf>(grad<Dhalf>(op))
  + kappa*op);

// create a MonoTensor
// for efficient matrix operator application
MonoTensor<m,n> mt;
mt.setFromStencil(Helmholtz);
mt.apply(result,u);
```

Listing 4: Constructing a `Stencil` for the Helmholtz operator, and applying to the operand u, using the `MonoTensor` matrix format for efficiency.

The `Stencil` class is parameterized on the data type of the operand. Scalar and vector valued `Stencils` are supported.

A `Stencil` is composed of `Entries`, each of which represents an offset and a coefficient. When a `Stencil` is applied to grid data, the result is also of grid data type. In Listing 4, both the operand `u`, on which the `Helmholtz` operator is applied, and the result, `result`, are of the data type `Mut`, which is a variable of scalar type.

In order to get a matrix-vector formulation, it is possible to construct the corresponding matrix operator, B, from a `Stencil`. Applying the matrix operator is more efficient than applying the `Stencil`, because of superior data layout. When a `Stencil` is formed from a difference expression, the expression is traversed recursively, and each term is added to the `Stencil`, either as a new `Entry`, or as a modification of an existing one.

The `Stencil` and `Entry` class solve the issue of entering the terms of a difference expression, such as (3), into their correct position in a matrix operator. Each entry of the matrix operator is represented by an `Entry` in the `Stencil`.

A `Stencil` may be directly applied, or it may be used as input when constructing a matrix operator, as shown in Listing 4. The `MonoTensor` in this example is in a format, which is trimmed for efficient matrix-vector multiplication. The `MonoTensor` is further described in Section 5.3.

There are two points in particular on which the FlexOp `Stencil` adds value, as compared with previous efforts, such as those of Ødegård [14] and Karpovich et al. [9]. Firstly the FlexOp `Stencil` can be constructed from coordinated invariant operators on curvilinear coordinates. This means that the developer is released of the burden of including the correct metric coefficients in the discretization. Secondly the discretizations on which a FlexOp `Stencil` is based are given as parameters in the difference expression. This separation of the difference operators on one hand, and their discretization on the other, offers a simple interface for experimenting with different discretizations of one and the same difference operator.

4.4 Efficient Evaluation Using Expression Templates

Requirement 4(ii) listed above concerns efficient direct evaluation of algebraic and difference expressions on large data sets. We have chosen to use expression templates for achieving this goal, as described by e.g. Vandevoorde [16]. The key advantage of expression templates is that they allow for efficient evaluation of arbitrary algebraic expressions on large sets of data.

Expression templates use parameterized classes to represent these data sets, as well as algebraic operations on them. These parameterized classes can be combined into arbitrarily complicated algebraic expressions, using operator overloading. Listing 5 shows how the large data sets `data1` and

data2 are being added and the result is assigned to data3. All three data sets are defined as Expressions based on the large data type Scalar<1000>.

```
Expression<Scalar<1000> > data1, data2, data3;

....
// data1 and data2 are assigned values, not shown here

data3 = data1 + data2;
```

Listing 5: Using expression template techniques to add data1 and data2.

The combination of parameterized classes with operator overloading, makes it possible for the compiler preprocessor to generate a one-loop evaluation for arbitrarily complicated algebraic expressions. The two-term addition of Listing 5 expands to the loop shown in Listing 6. The compiler works recursively, which means that it can handle algebraic expressions of arbitrary complexity, up to its recursion depth limit.

```
for(int ii = 0; ii < 1000; ii++) {
  data3[ii] = data1[ii] + data2[ii];
}
```

Listing 6: The code generated by the compiler preprocessor from the example in Listing 5.

We see that expression templates use generics, which gives static polymorphism and fixed size arrays, to provide efficient evaluation of arbitrary algebraic expressions on large data sets.

5 Pilot Implementation

We have identified the requirements that the numerical analyst has on FlexOp, and found software solutions that meet these requirements.

The next step is to make a pilot implementation of the FlexOp to verify that the parts can act together as a whole.

For the pilot implementation we need to pick

- infrastructure, and
- implementation language.

5.1 Selection of Infrastructure

The FlexOp are designed to be used in conjunction with an existing infrastructure, which contains the appropriate underlying data structures. These include data types representing discretized scalar and vector fields, a grid generator which supplies coordinate values and metric coefficients on the grid, a matrix operator, and so on.

FlexOp is part of a larger effort, with the long term goal of designing flexible software for numerical problems defined on structured grids. Among our achievements are Cogito, which provides flexible partitioning for parallelization, EINSUM, which provides tensor notation for the manipulation of multi-dimensional arrays, and TENGOME [12], which provides an infrastructure for constructing PDE solvers on curvilinear structured grids.

TENGOME is implemented in C++ and Fortran 95, and contains the data types needed by FlexOp, and more, including a library of composable preconditioners.

5.2 Selection of Implementation Language

When selecting the implementation language for the FlexOp, the following factors were taken into account

- modular code structure based on classes,
- overloading of arithmetic operators,
- parameterized classes and functions, and
- interface to preexisting efficient solvers.

The formulation and discretization of a PDE problem involves a wide range of concepts, mostly mathematical. Classes and overloaded operators were deemed important, since an agreement between the software structure and familiar mathematical concepts greatly enhances the understandability of the code. For a further discussion on this subject see also Åhlander et al. [1].

Inheritance and parameterized classes can both be used for implementing variability. In both cases, a set of similar classes is implemented, and a different version of the class is selected in order to get a specific behavior. When inheritance is used, the selection will be resolved at run time, through the use of virtual function calls. Using parameterized classes instead will allow the selection to be resolved at compile time, and thus increase the opportunities for compile time optimizations. This is the basis for the requirement that the implementation language supports parameterized classes and functions.

C++ was selected as the language of implementations based on the requirements listed above, and also because it widely supported, and well know by the implementers.

In a recent proposal, Gregor et al. [8], describe the **ConceptC++** language, which provides a way to drastically enhance the generative powers of C++. The proposal is accompanied by a compiler implementation; **ConceptGCC**. Parts of the code for this paper have been implemented in ConceptC++, with a satisfactory result.

Unfortunately our experiments with **ConceptGCC** led us to the conclusion that the compiler at that time was not yet stable enough to base our pilot implementation on. In spite of the fact that **ConceptGCC** could not be used, we will give a brief description of it here, as it constitutes a promising approach to generic programming.

In ordinary C++ template programming, parameters of class type are used in the meta-code, to specify the types and methods that should be used in the ordinary C++ code that the compiler preprocessor generates. When the preprocessor translates the meta-code, it makes the substitutions as it encounters these parameters. The error messages that are generated, when the substituted code does not conform, are often not extremely helpful. We are told that some method returns the wrong type, or a class does not contain a certain method. It may be hard to understand how such an error message relates to the intention of the code.

`ConceptC++` introduces the *concept*, which is a way to specify the interface a class must have. Such interfaces are defined in terms of one or several *concepts*. When a parameterized class is defined in `ConceptC++`, its parameters may be constrained by requiring it to conform to a particular *concept*. This makes it possible for the preprocessor to check the interface of the actual parameters supplied when the parameterized class is used. If the interface does not conform, useful error messages are produced.

The generative powers of `ConceptC++` come from the fact that it is easy for a developer to specify how a class should *model* a specific *concept*. The methods and types of a *concept* may be defined to have default implementation. If, for example, it is required that the *concept* `Field` contains the `operator[](int)`, it is natural to define this operator to be any predefined `operator[](int)` for classes which models the *concept*. For the cases when no natural default implementation exists, or when the class does not contain the method specified by the default, a *concept_map* is used for specifying how a specific class models a particular method or type which is required by the *concept* it is modeling.

The *concept_map* supplies a very convenient way of defining how a specific class should implement a certain interface requirement. Since the *concept_map* is written independently of the class that it maps, it is possible to use classes that were constructed with no particular *concept* in mind. It is

no longer necessary to have a complete correspondence between the way a class is defined and the way it is used as a template parameter.

Expression templates are conveniently defined using *concepts* and their related *concept_maps*.

As mentioned earlier, the **ConceptGCC** compiler unfortunately existed only in a beta version, and could not support an implementation of the full code. In the future, we are expecting to switch our implementation back to **ConceptC++**. This is expected to lead to a significant reduction in the number of lines of code in the implementation.

5.3 The Pilot Implementation

Figure 2 shows how FlexOp has been combined with the TENGOME infrastructure. An *equation solver* layer is included in Figure 2, to indicate that any complete PDE solver must also be able to find the solution to the system of equations that the matrix operator represents. For time dependent problems the solver will instead, or in addition, contain a time stepping layer.

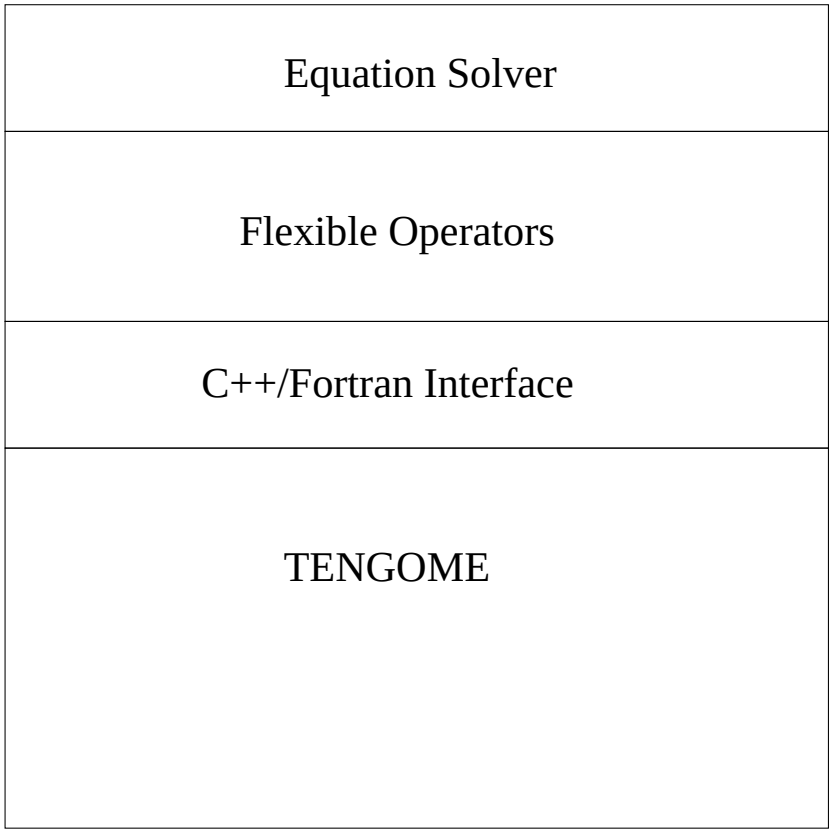

Figure 2: FlexOp as part of a complete PDE solver, based on TENGOME.

TENGOME contains a number of classes that are used by FlexOp. These

include basic infrastructure, such as the `GridScalar` and `GridVector` classes that represent discretized fields. Other important classes include the `Metric`, which provides the metric coefficients for the curvilinear coordinates used in the discretization of the problem, as well as the `MonoTensor`, which is the representation of the matrix operator in TENGOME.

TENGOME is written in Fortran and C++. The interface between FlexOp and the Fortran 95 modules of TENGOME is written using a wrapper technique [13], which involves defining a C++ interface class for those modules which are used by FlexOp.

Listing 7 shows how some of the TENGOME data types are used by FlexOp. The `LogicalGrid` is a TENGOME data type that represents a structured grid in two dimensions. The `GridDiffeomorphism` is a grid generator. As the name indicates, it supplies the values of the curvilinear coordinates and their derivatives on any given `LogicalGrid`. These values are used to initialize the `Metric`, which is the class that keeps track of the coordinate values and their derivatives at the grid points. In FlexOp, a `GridGeometry` is defined based on a `Grid` and a `Metric` defined on this grid. Once the `GridGeometry` is in place, it is possible to define variables of scalar or vector types, such as `scalar` and `vector` on this grid. These can then be used in difference expressions, as discussed in the previous section.

```
// define the inputs to the GridDiffeomorphism
const int mCoarse = 21;
const int nCoarse = 101;

typedef Geometry2d<mCoarse,nCoarse> CoarseGrid;

LogicalGrid<CoarseGrid> lg;

double r1,d0,d1;
r1 = 500;        // length
d0 = 50;         // thickness on left side
d1 = 40;         // thickness on right side
GridDiffeomorphism gd(r1,d0,d1,lg);

// define the logical grid for the problem at hand
const int m = 101;
const int n = 501;

typedef Geometry2d<m,n> Grid;

// define and couple to the metric
typedef GridGeometry<Grid,OrthoMetric<Grid> > Geom;
OrthoMetric<Grid>* met =  new OrthoMetric<Grid>(gd);

Geom::coupleMetric(met);

// define the datatypes for describing the problem
typedef Scalar<Geom> ScalarType;
typedef MutableComposable<ScalarType> GridScalar;
GridScalar scalar;

typedef GridVector<Geom> GridVectorType;
typedef
  MutableComposableVector<GridVectorType> GridVector;
GridVector vector;
```

Listing 7: TENGOME classes are used to define the `Metric`, and the `GridDiffeomorphism` and `LogicalGrid` on which it is based. The data arrays named `scalar` and `vector` are also represented by classes from TENGOME.

FlexOp provides support for the gradient of a scalar, the divergence of a vector as well as the curl. The gradient and divergence are supported for arbitrary dimensionalities, whereas the curl is only supported in two and

three dimensions.

6 Case Study: A Numerov Method

In Section 2 we derived a fourth-order compact Numerov method for the solution of the Poisson equation. In order to assess the benefits of FlexOp, we proceeded to implement this method, using the pilot implementation of FlexOp, as described in the previous section.

The purpose of this test application is to verify the expressiveness of the FlexOp, and its interoperability with TENGOME. The functionality studied is the evaluation of difference expressions and the formation of matrix operators. The actual solution of the PDE problem is not performed by the test application.

FlexOp provides a high level language, which closely resembles the mathematical description. Listing 8 shows how the FlexOp are used to construct the matrix operator for the Numerov problem. Note how FlexOp automatically includes the metric coefficients in the discretizations of the coordinate invariant operators `divergence` and `grad`. The construction of a matrix operator is a two step process. The `Stencil` is first formed from the difference expression, and the entries are then transferred into the matrix operator, in the form of a TENGOME `MonoTensor` object.

```
Stencil<Mut>
poisson(divergence<Dhalf>(kappa*grad<Dhalf>(op))+
 - h1*h1/12*partial<Dhalf,Xdir>(partial<Dhalf,Xdir>(
       partial<Dhalf,Ydir>(a2*partial<Dhalf,Ydir>(op))))+
 - h2*h2/12*partial<Dhalf,Ydir>(partial<Dhalf,Ydir>(
       partial<Dhalf,Xdir>(a1*partial<Dhalf,Xdir>(op))))+
 + h1*h1/12*partial<D0,Xdir>(Inv(a1)*da1dx*
       partial<Dhalf,Ydir>(a2*partial<Dhalf,Ydir>(op)))
 + h2*h2/12*partial<D0,Ydir>(Inv(a2)*da2dy*
       partial<Dhalf,Xdir>(a1*partial<Dhalf,Xdir>(op)))
 + h1*h1/12*partial<Dhalf,Xdir>(Inv(a1)*da1dx*da1dx*
                                partial<Dhalf,Xdir>(op))
 + h2*h2/12*partial<Dhalf,Ydir>(Inv(a2)*da2dy*da2dy*
                                partial<Dhalf,Ydir>(op))+
 - h1*h1/24*partial<Dhalf,Xdir>(d2a1dx2*
                                partial<Dhalf,Xdir>(op))+
 - h2*h2/24*partial<Dhalf,Ydir>(d2a2dy2*
                                partial<Dhalf,Ydir>(op)));

  MonoTensor<m,n> mt;
  mt.setFromStencil(poisson);
```

Listing 8: Constructing a matrix operator for the compact fourth-order Numerov scheme in (10) using FlexOp.

Listing 9 shows the computation of the metric coefficients `da1dx`, `da2dy`, `d2a1dx2`, and `d2a2dy2`, that are used in Listing 8.

```
  da1dx   = partial<D0,Xdir>(a1);
  da2dy   = partial<D0,Ydir>(a2);
  d2a1dx2 = partial<Dhalf,Xdir>(partial<Dhalf,Xdir>(a1));
  d2a2dy2 = partial<Dhalf,Ydir>(partial<Dhalf,Ydir>(a2));
```

Listing 9: Computation of the metric coefficients `da1dx`, `da2dy`, `d2a1dx2`, and `d2a2dy2` using FlexOp

The discretization of the boundary conditions is described in Appendix A, together with the mathematical derivation that it is based on.

The expression in Listing 8 is to be compared with the corresponding difference equation, which is what we would get if all the terms of (10) were written out explicitly. We have not included this equation in this paper, since it is huge. Instead we indicated its complexity by showing the discretization

of one of its fifteen terms in (12). The corresponding discretization of all fifteen terms would contain 68 terms.

As we can see, the readability of the code in Listing 8 is vastly superior to that of the equivalent discretization. By relieving the programmer of the responsibility for describing shifts and shifted metric coefficients, the risk of introducing typing errors during the discretization process is significantly reduced.

7 Assessment

The FlexOp provide several different types of benefits. As far as implementation is concerned, we see that the high level language of FlexOp significantly reduces the number of lines of code that the developer needs to write. For the direct evaluation, we can compare the difference expression of Listing 8 with the 68 terms of the huge equation described above.

In Listing 8, we see how FlexOp is used to construct an matrix operator from a difference expression. Without the support of FlexOp, the numerical analyst would have to construct each entry of the matrix operator by hand, based on the 68 terms of the expanded version of (10). By automatizing this process FlexOp significantly reduces development time.

The time spent on testing is also greatly reduced through the use of FlexOp, since the risk for coding errors is much smaller when using the FlexOp high level language, compared to manually entering all the terms and offsets.

A further benefit of the FlexOp is that they generate code that is optimized for the way it is to be used. Since FlexOp is used in combination with an existing infrastructure, it will be the choice of infrastructure that determines the format of the matrix operator. This opens up for the possibility to select the infrastructure based on how efficient its matrix operator format is for the problem at hand.

Expression templates are used for the case of direct evaluation of the difference expression, which ensures that the complete difference expression is evaluated in one and the same loop.

We have previously shown [11] that there is no increased computational cost associated with the addition of FlexOp and a C++/Fortran interface layer, to the TENGO infrastructure, which is a subset of TENGOME. For the case when a matrix operator is used, the execution time is unchanged when the inter-language interface is introduced. For the case of direct evaluation, the execution time is unchanged when the grid data are represented using the TENGO `GridScalar` instead of ordinary C++ arrays.

The fact that FlexOp can be integrated with code that has been optimized, saves the developer from having to consider re-coding for optimization

purposes. The skipping of this optimization step leads to lower development time and a code closely adhering to the mathematics, on which it is based.

8 Conclusions and Future Work

Based on the needs of a numerical analyst, we have identified requirements on software for the formulation of FDM on curvilinear structured grids. The requirements are that such software should contain

Requirement 1: a high level language of description for numerical approximations,

Requirement 2: support for selection of different discretization schemes for different parts of the expression,

Requirement 3: automatic inclusion of metric coefficients for coordinate invariant operators, and

Requirement 4: efficient evaluation of algebraic and difference expressions on large data sets, both

i. when using a matrix-vector formulation, and

ii. for direct evaluation.

We have presented FlexOp, a software solution that meets these requirements. FlexOp is a set of flexible generic classes for the implementation of finite difference methods on curvilinear structured grids.

In order to assess the FlexOp, we have used them to implement a compact fourth-order Numerov method, using the TENGOME infrastructure as a basis, and C++ as the implementation language.

We find that the benefits of using FlexOp are

- high agreement between mathematical derivation and implementation of the method,
- a significant reduction of the number of lines of code,
- a decreased need for testing, and
- a decreased need for optimization.

In the future, we plan to expand the FlexOp by introducing general interfaces to solvers and time stepping schemes. When the **ConceptGCC** compiler matures, we will also convert FlexOp to **ConceptC++**, which will reduce the number of lines of code, and thus increase the maintainability.

A Boundary conditions for the application example.

We impose the following boundary conditions

$$u(0,\xi_2) = T_0(\xi_2), u(1,\xi_2) = T_1(\xi_2), \quad 0 \leq \xi_2 \leq 1, \tag{13}$$

$$\frac{\partial u}{\partial \xi_2}(\xi_1,0) = \frac{\partial u}{\partial \xi_2}(\xi_1,1) = 0, \quad 0 \leq \xi_1 \leq 1, \tag{14}$$

where $T_0'(0) = T_0'(1) = 0$ and $T_1'(0) = T_1'(1) = 0$ for compatability reasons.

We use a grid in the ξ_1-direction, which is *on-boundary* at both ends, which yields

$$\begin{aligned} u^{1,j} =& T_0^j \equiv T_0(\xi_2^j), \\ u^{N,j} =& T_1^j \equiv T_1(\xi_2^j). \end{aligned}$$

In the ξ_2-direction, we choose a grid that is *boundary-centered* at both boundaries. A boundary-centered discretization is shown in Figure 3.

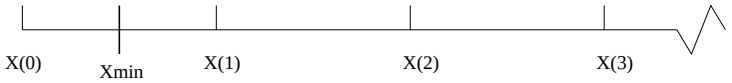

Figure 3: Boundary centered discretization around the boundary at X_{min}.

The centering of the grid facilitates the construction of a compact and sufficiently accurate discretization of the Neumann boundary conditions.

Taylor expanding $u(\xi_1,\xi_2)$ around $\xi_2 = 0$, for the points $\xi_2 = \frac{h_2}{2}$ and $\xi_2 = -\frac{h_2}{2}$, gives

$$\begin{aligned} \frac{1}{h_2}\left(u(\xi_1,\frac{h_2}{2}) - u(\xi_1,-\frac{h_2}{2})\right) =& \frac{\partial u}{\partial \xi_2}(\xi_1,0) \\ &+ \frac{h_2^2}{24}\frac{\partial^3 u}{\partial \xi_2^3}(\xi_1,0) + \mathcal{O}(h_2^4) \end{aligned} \tag{15}$$

Combining the PDE (2) and the boundary condition (14) yields

$$\begin{aligned} -a_2(\xi_1,0)\frac{\partial^3 u}{\partial \xi_2^3}(\xi_1,0) =& \frac{\partial}{\partial \xi_1}\left(\frac{\partial a_1}{\partial \xi_2}\frac{\partial u}{\partial \xi_1}\right) - \frac{2}{a_2}\frac{\partial a_2}{\partial \xi_2}\frac{\partial}{\partial \xi_1}\left(a_1\frac{\partial u}{\partial \xi_1}\right) \\ &+ \frac{\partial g}{\partial \xi_2} - \frac{2}{a_2}\frac{\partial a_2}{\partial \xi_2}g \end{aligned}$$

Substituting this into (15) gives

$$
\begin{aligned}
&a_2(\xi_1,0)\frac{\partial u}{\partial \xi_2}(\xi_1,0) \\
&= \frac{a_2(\xi_1,0)}{h_2}\left(u(\xi_1,\frac{h_2}{2}) - u(\xi_1,-\frac{h_2}{2})\right) \\
&+ \frac{h_2^2}{24}\left(\frac{\partial}{\partial \xi_1}\left(\frac{\partial a_1}{\partial \xi_2}\frac{\partial u}{\partial \xi_1}\right) - \frac{2}{a_2}\frac{\partial a_2}{\partial \xi_2}\frac{\partial}{\partial \xi_1}\left(a_1\frac{\partial u}{\partial \xi_1}\right) + \frac{\partial g}{\partial \xi_2} - \frac{2}{a_2}\frac{\partial a_2}{\partial \xi_2}g\right) \\
&+ \mathcal{O}(h_2^4)
\end{aligned}
$$

where the left hand side should be zero, because of the boundary condition.

At the $\xi_2 = 0$ boundary, we have the discretization index $j = \frac{1}{2}$. Using a second-order accurate discretization of the derivatives in the h_2^2 term, gives us the following fourth-order accurate discretization:

$$
\begin{aligned}
&\frac{a_2^{i,\frac{1}{2}}}{h_2}(u^{i,1} - u^{i,0}) + \\
&\frac{h_2^2}{24}\left(\frac{1}{2}D^1_{1/2}\left(D^2_{1/2}(a_1^{i,0})D^1_{1/2}u^{i,0} + D^2_{1/2}(a_1^{i,1})D^1_{1/2}u^{i,1}\right)\right. \\
&\left. - \frac{1}{a_2^{i,\frac{1}{2}}}D^2_{1/2}(a_2^{i,\frac{1}{2}})D^1_{1/2}\left(a_1^{i,0}D^1_{1/2}u^{i,0} + a_1^{i,1}D^1_{1/2}u^{i,1}\right)\right) = \\
&- \frac{h_2^2}{24}\left(\left(\frac{\partial g}{\partial \xi_2}\right)^{i,\frac{1}{2}} - \frac{2}{a_2^{i,\frac{1}{2}}}D^2_{1/2}(a_2^{i,\frac{1}{2}})g^{i,\frac{1}{2}}\right).
\end{aligned}
\tag{16}
$$

The `SubDomain` parameterized class in FlexOp defines a slice of a gridded geometry. A boundary is such slice. A `Stencil` may be defined on a boundary, by parameterizing the `Stencil` on a `SubDomain`. Such a `Stencil` is defined only for points in the `SubDomain`, but its `Entries` may still refer to points outside of the `SubDomain`.

Listing 10 shows how a `Stencil` is formed, based on the left-hand side of (16).

```
const int zeroGridLineYdir =
     thetaDir.lowerBoundaryPoint;

typedef
   SubDomain<Geom,zeroGridLineYdir,Ydir>
       LeftBoundaryYdir;

typedef
   MutableComposable<Mutable<double,LeftBoundaryYdir> >
     LeftBoundaryDataYdir;

// Build a Stencil for the boundary point, i.e. the
// point which is  h/2 away from the domain boundary.
Stencil<LeftBoundaryDataYdir>
 bcLeftYdir(stagger<Ydir>(a2)/h2*(shiftData<Ydir>(op,1)
     - op)
    + h2*h2/24*(0.5*partial<Dhalf,Xdir>(
       partial<Dhalf,Ydir>(a1)*partial<Dhalf,Xdir>(op)+
       partial<Dhalf,Ydir>(shiftData<Ydir>(a1,1))*
           partial<Dhalf,Xdir>(shiftData<Ydir>(op,1)))+
       -Inv(stagger<Ydir>(a2))*partial<Dhalf,Ydir>(
          stagger<Ydir>(a2))*partial<Dhalf,Xdir>(
             a1*partial<Dhalf,Xdir>(op) +
             shiftData<Ydir>(a1,1))*
             partial<Dhalf,Xdir>(
                       shiftData<Ydir>(op,1))));
```

Listing 10: Constructing a `Stencil` for a boundary, using FlexOp.

The border at $\xi_2 = 1$ is treated similarly.

References

[1] K. Åhlander, M. Haveraaen, and H. Munthe-Kaas. On the role of mathematical abstractions for scientific computing. In R. F. Boisvert and P. T. P. Tang, editors, *The Architecture of Scientific Software*, pages 145–158, Boston, 2001. Kluwer Academic Publishers.

[2] K. Åhlander and K. Otto. Software design for finite difference schemes based on index notation. *Future Generation Computer Systems*, 22:102–109, 2006.

[3] O. S. Bagge. CodeBoost: A framework for transforming C++ programs. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[4] R. Boisvert. Families of high order accurate discretizations of some elliptic problems. *SIAM J. Sci. Statist. Comput.*, 2:268–284, 1981.

[5] L. Collatz. *The Numerical Treatment of Differential Equations*. Springer-Verlag, Berlin, 3rd edition, 1960.

[6] W. Henshaw D. Brown and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In M. Henderson, C. Anderson, and S. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Philadelphia, 1999. SIAM.

[7] P. Grant, M. Haveraaen, and M. Webster. Coordinate free programming of computational fluid dynamics problems. *Scientific Programming*, 8:211–230, 2000.

[8] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '06)*. ACM Press, October 2006.

[9] J. Karpovich, M. Judd, W. Strayer, and A. Grimshaw. A parallel object-oriented framework for stencil algorithms. Technical Report CS-93-13, University of Virginia, Charlottesville, VA, USA, 1993.

[10] D. Knoll, J. Morel, L. Magonlin, and M. Shashkov. Physically motivated discretization methods; A strategy for increased predictiveness. *Los Alamos Science*, 29:188–212, 2005.

[11] M. Ljungberg. High performance generative programming with a fortran 95 application. In *Proceedings of the workshop Parallel/High-Performance OO Scientific Computing*. ECOOP, July 2005. Glasgow, Scotland, submitted to Scientific Programming.

[12] M. Ljungberg, K. Otto, and M. Thuné. Design and usability of a PDE solver framework for curvilinear coordinates. *Advances in Engineering Software*, 37(12):814–825, 2006.

[13] M. Ljungberg and M. Thuné. Mixed C++/Fortran 90 implementation of parallel flow solvers. In C. B. Jenssen et al., editor, *Parallel CFD 2000, Trends and Applications*, pages 233–240. North–Holland, 2001.

[14] Å. Ødegård. *Applications of high level software for parallel solution of Partial Differential Equations*. PhD thesis, University of Oslo, Norway, 2006. ISSN 1501-7710, No 507.

[15] K. Otto. A unifying framework for preconditioners based on fast transforms. Technical Report 187 (revised), Uppsala University, Department of Scientific Computing, Box 337, 751 05 Uppsala, Sweden, 1999. `http://user.it.uu.se/\~kurt/rep187.ps`.

[16] D. Vandevoorde and N. Josuttis. *C++ Templates: The Complete Guide*. Addison Wesley, 2002. ISBN 0201734842.