# A Comprehensive Set of Tools for Solving Partial Differential Equations; Diffpack*

Are Magnus Bruaset  Hans Petter Langtangen

**Abstract**

This chapter presents an overview of the functionality in Diffpack, which is a software environment for the numerical solution of partial differential equations. Examples on how object-oriented programming techniques are applied for software design and implementation are provided. In addition, we present a collection of sample Diffpack applications.

## 1 Introduction

The increase in computer power has enabled investigations of complicated mathematical models, thus leading to a demand for comprehensive numerical software. In the field of partial differential equations (PDEs) the software must deal with a large number of topics, including discretization schemes, numerical linear algebra, user interfaces, visualization and computer architecture. Moreover, the numerical solution of PDEs requires extensive computer power, and the software is subjected to extreme efficiency requirements. As a consequence, software development for PDEs is usually a comprehensive and time-consuming process.

To increase the productivity of simulator development based on numerical solution of PDEs, we believe that reuse of code is a central issue. The mathematics and the numerical methods may differ relatively little from one PDE application to another[1], and the computer implementations should reflect this fact. Moreover, having established software for individual PDEs it should be easy to combine the modules to solve a system of PDEs. Such reuse of code does not only save programming effort, but also increases the reliability since new software will access well-tested components. Another requirement of modern scientific computing is the possibility of coupling a PDE solver to reporting facilities, parameter analysis and optimization modules. This requires the PDE solvers for different applications to have the same interface. The realization of these desirable properties makes strong demands on the software design and the generalization of the implementation of numerical algorithms. This calls for

*This paper was originally published in *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pages 63–92. Birkhäuser, 1997.

[1]See [14] for some examples from solid and fluid mechanics.

better programming techniques than those currently supported by traditional languages like FORTRAN 77 and C.

Object-oriented programming (OOP) represents techniques for reuse of code, enhancing the productivity of software development and maintenance, as well as increasing the reliability and robustness. The Diffpack project [11] is an initiative for exploring abstract data types (ADT) and OOP applied to numerical methods, see [3] for an introduction to numerical applications of the techniques. The objective of Diffpack is to substantially decrease the human efforts of programming without a significant sacrifice with respect to the computational efficiency, as compared to conventional FORTRAN codes. Diffpack is coded in C++, and a critical investigation of the efficiency of Diffpack code relative to highly tuned, special purpose programs in FORTRAN is discussed in [2].

In recent years, the application of ADT and OOP to PDE codes has emerged as an active field. Numerics with challenging data structures, e.g., multilevel methods and (adaptive) mesh generation, as well as numerics with parallel and distributed computing, have received particular interest [22, 23, 26]. Examples of novel implementations of more conventional numerics for PDEs, which is the topic of the present chapter, appear in the references [9, 16, 18, 20, 31, 25, 28, 29, 30, 32, 33, 34].

The Diffpack software is essentially a collection of C++ classes organized in libraries. The application area is mainly the numerical solution of PDEs with a particular focus on finite element methods. The libraries are especially constructed for rapid prototyping of simulators for new problems using classes at a high abstraction level. A layered design also offers classes at lower levels, thus allowing the programmer to control the details of data structures and numerics if this is desired. The CPU-intensive numerics are implemented in low level classes while a typical high level class organizes low level classes and offers a convenient programming interface.

The basic philosophy has been to provide a development environment where the programmer can concentrate on the essential and critical numerics and leave much of the coding work related to, e.g., input, output and more established numerics, to classes in the libraries. When creating a simulator, this principle is reflected in the fact that most of the application code relates to the equation being solved, while generic numerics, program management, input data menus and coupling to visualization, require little effort. The resulting program will then be flexible with respect to run-time combinations of the various numerical ingredients in the problem, e.g., linear systems solvers and discretization schemes.

By providing a set of building blocks at different abstraction levels for established numerical methods, research projects applying Diffpack can spend most man hours on the implementation of new, non-trivial algorithms.

Diffpack consists of four main libraries: BasicTools, LaTools, DpKernel and DpUtil, plus a set of applications. In addition, there is a programming envi-

ronment based on C++ and Unix utilities[2]. First we present an overview of the contents of the libraries. Thereafter, some of the classes related to finite element programming are discussed in more detail with an emphasis on OOP. A concrete example is given to demonstrate the principles for a simple model problem. Finally, we discuss how OOP can be utilized to structure simulators and their mutual dependencies. Diffpack is currently available on the Internet [11].

## 2 Current applications

Diffpack is a generic tool in the sense that if one grasps the main ideas in a simple problem, a similar implementational approach can be used to study much more complicated problems. To convince the reader that the Diffpack libraries are not restricted to trivial problems, a list of some Diffpack application that have been coded and verified is given below. Most of the codes work in 1D, 2D and 3D, leaving the spatial dimension to be a run-time choice made by the user.

- *Convective-diffusive transport equations*, e.g.,

$$\frac{\partial u}{\partial t} + \vec{v} \cdot \nabla u = \nabla \cdot [k \nabla u] + f \tag{1}$$

  where $u$ is the primary unknown (temperature or specie concentration), $\vec{v}$ is a fluid velocity field, $k$ is a prescribed diffusion field and $f$ is a known source term. The boundary conditions can be of general Dirichlet, Neumann or Robin (mixed) type. A general solver for (1), with documentation, is a part of the Diffpack distribution.

- *Convective-diffusive-reactive equations* on the form (1) with $f$ as a function of $u$. Such nonlinear problems have been handled by various methods like operator splitting, Newton-Raphson iteration or successive substitutions (Picard iteration). The efficiency of such Diffpack solvers, compared to special purpose FORTRAN codes, is investigated in [2].

- *A simulator for the electrical activity in the human heart and body*, consisting of a Laplace equation in a domain $\Omega$ and a diffusion-reaction equation in a domain $H$ ($\Omega \cap H = \emptyset$, and $H$ is a "hole" in $\Omega$):

$$\begin{aligned} c_1 \frac{\partial V}{\partial t} + G(V) &= c_2 \nabla^2 V \quad \text{in } H, \\ \nabla^2 V_0 &= 0 \quad \text{in } \Omega, \end{aligned}$$

  where $c_1$ and $c_2$ are constants, $V$ and $V_0$ are the primary unknowns, and $V = V_0$ on the common boundary of $H$ and $\Omega$. The Laplace equation is solved by finite element methods, whereas an explicit finite difference scheme is used for the diffusion-reaction equation.

[2]The BasicTools and LaTools libraries, as well as the Unix-based framework, are heavily used also in other projects not related to PDEs, see for instance [4].

- *The incompressible Navier-Stokes equations*,

$$\begin{aligned} \nabla \cdot \vec{v} &= 0, \\ \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} &= -\nabla p + \frac{1}{\mathrm{Re}} \nabla^2 \vec{v}, \end{aligned}$$

  where $\vec{v}$ is the velocity, $p$ is the pressure and Re is the Reynolds number. Boundary conditions are of two types: prescribed $\vec{v}$ or prescribed $\partial \vec{v}/\partial n$. The problem is solved by penalty and augmented Lagrangian methods (the solver based on the penalty method comes with the Diffpack distribution). Other solution methods, e.g. based on mixed finite elements, are also implemented.

- *Coupled heat and fluid flow*, provided by combining the Navier-Stokes solver (with an additional gravity term) with the convective-diffusive transport solver. This module can deal with free thermal convection in fluids.

- *The equations of linear thermo-elasticity*,

$$\nabla \left[ (\lambda + \mu) \nabla \cdot \vec{u} \right] + \nabla \cdot \left[ \mu \nabla \vec{u} \right] = \alpha \nabla \left( (3\lambda + 2\mu) T \right).$$

  Here $\vec{u}$ is the displacement field of the elastic medium, $\lambda$ and $\mu$ are elastic "constants", $\alpha$ is a thermal expansion coefficient and $T$ is the temperature change from a configuration where zero stress is equivalent to zero strain. The boundary conditions can be of two types: Either $\vec{u}$ is specified or the normal stress is prescribed. The code and its documentation are parts of the Diffpack distribution. Physical applications cover structural analysis with possible temperature effects and hydrocarbon or groundwater reservoir deformation due to porous media flow (then $T$ is interpreted as the fluid pressure).

- *The linear wave equation*,

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot \left[ k \nabla u \right]$$

  where $\sqrt{k}$ is the local wave velocity. This equation describes shallow water waves, membrane vibrations, electromagnetic waves, and many other wave phenomena. The code and the documentation come with Diffpack.

- *Nonlinear weakly dispersive water wave models*, described by a set of Boussinesq equations,

$$\begin{aligned} \frac{\partial \eta}{\partial t} + \frac{\partial r}{\partial t} + \nabla \cdot \vec{Q} &= 0, \\ \frac{\partial \phi}{\partial t} + \frac{\alpha}{2} (\nabla \phi)^2 + \eta - \epsilon \left\{ \frac{h}{2} \frac{\partial^2 r}{\partial t^2} + \frac{h}{2} \nabla \cdot \left( h \nabla \frac{\partial \phi}{\partial t} \right) - \frac{h^2}{6} \nabla^2 \frac{\partial \phi}{\partial t} \right\} &= 0. \end{aligned}$$

  Here $\vec{Q}$ is given by

$$\vec{Q} = (h + \alpha\eta + \alpha r) \nabla \phi + \epsilon h \left( \frac{1}{6} \frac{\partial \eta}{\partial t} - \frac{\partial r}{\partial t} - \frac{1}{3} \nabla h \cdot \nabla \phi \right) \nabla h.$$

The primary unknowns are the surface elevation $\eta(x,y,t)$ and the velocity potential $\phi(x,y,t)$. The function $h(x,y)$ represents the stationary sea bottom, while $r(x,y,t)$ models movement of the bottom due to earthquakes or faulting. The parameters $\alpha$ and $\epsilon$ are prescribed constants, measuring the effects of nonlinearity and dispersion, respectively. The equations are solved by a staggered finite difference method in time and by a Galerkin finite element method in space.

- *Fully nonlinear water waves*, described by

$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{in } \Omega, \\ \frac{\partial\eta}{\partial t} + \nabla\phi\cdot\nabla\eta &= \frac{\partial\phi}{\partial z} \quad \text{on } z=\eta(x,y,t), \\ \frac{\partial\phi}{\partial t} + \frac{1}{2}||\nabla\phi||^2 + g\eta &= 0 \quad \text{on } z=\eta(x,y,t). \end{aligned}$$

  Here $\phi$ is the velocity potential in the water, $\eta$ is the water surface elevation, and $g$ is the acceleration due to gravity. The system of equations is solved by a finite element method where the unknown surface $z=\eta(x,y,t)$ is equivalently modeled by a variable coefficient in the Laplace equation for $\phi$. Linear systems are solved by the conjugate gradient method with an FFT based direct method for $\nabla^2\phi=0$ as preconditioner.

- *Two-phase porous media flow*, described by the equations

$$\begin{aligned} \nabla\cdot[\lambda(S)\nabla P] &= 0, \\ \frac{\partial S}{\partial t} - \nabla\cdot(f(S)\lambda(S)\nabla P) &= 0, \end{aligned}$$

  where $S$ and $P$ are the primary unknowns (pressure and saturation), and $\lambda$ and $f$ are prescribed functions. The boundary conditions cover impermeable boundaries, known pressure and saturation values as well as known well rates. One Diffpack solver for this problem applies Petrov-Galerkin methods in combination with fully implicit time integration and simultaneous solution of the pressure and saturation [13]. Other solvers [12] have also been developed.

- *Hele-Shaw polymer flow*, where the fluid is modeled as generalized Newtonian,

$$\nabla\cdot[\mu||\nabla p||^m\nabla p] = 0$$

  with $m$ being a known constant and $\mu$ is a known function ($m$ and $\mu$ usually involve the temperature). The pressure $p$ or its normal derivative (the normal velocity) are prescribed as boundary conditions. This model has important applications to forming of plastic materials.

- *Stochastic groundwater flow and specie transport*,

$$\begin{aligned} \nabla \cdot [K\nabla H] &= 0 \\ \frac{\partial u}{\partial t} - K\nabla H \cdot \nabla u &= k\nabla^2 u \end{aligned}$$

where $K$ is a stochastic permeability field, $H$ is the hydraulic head (primary unknown), $u$ is the specie concentration (primary unknown) and $k$ is a diffusion constant. The boundary conditions are of Dirichlet or Neumann type for $u$ and $p$. This stochastic initial-boundary value problem is solved by Monte Carlo simulation as well as by approximate analytical techniques that lead to simplified deterministic PDEs which are solved by standard Diffpack software, see [24].

- *A model for metal casting*, consisting of a sophisticated coupled system of nonlinear PDEs. The system arises from a two-phase (liquid/solid) model with phase changes. The equations reflect mass conservation, momentum and energy balance as well as mass conservation of a solute [15].

$$\begin{aligned} \frac{\partial}{\partial t}(\rho h) + \rho_l \nabla\cdot(h\mathbf{v}) + \rho_l \nabla\cdot(f_s H\mathbf{v}) - \nabla\cdot(\lambda\nabla T) &= 0, \\ \frac{\partial}{\partial t}(\rho c) + \rho_l \nabla\cdot(c_l\mathbf{v}) &= 0, \\ \rho_l \frac{\partial \mathbf{v}}{\partial t} + \rho_l \nabla\cdot(\frac{1}{g_l}\mathbf{v}\mathbf{v}^t) - 2\mu\nabla\cdot\varepsilon(\mathbf{v}) + \frac{2}{3}\mu\nabla(\nabla\cdot\mathbf{v}) + g_l\nabla p + \frac{\mu g_l}{K}\mathbf{v} &= 0, \\ \frac{\partial \rho}{\partial t} + \rho_l \nabla\cdot\mathbf{v} &= 0. \end{aligned}$$

Here, $\rho = g_l\rho_l + g_s\rho_s$ is the density of the two-phase averaging volume and $g_l, \rho_l, \rho_s$, and $g_s = 1 - g_l$ are the volume fraction liquid, the liquid density, the solid density, and the volume fraction solid, respectively. Moreover, $\mathbf{v}$ is a superficial velocity vector and

$$\varepsilon(\mathbf{v}) = \frac{1}{2}(\nabla\mathbf{v} + (\nabla\mathbf{v})^T).$$

Furthermore, $p, \mu, K, h, f_s, H, \lambda, T, c$, and $c_l$ are the liquid pressure, the dynamic viscosity, the permeability of the mushy zone (partially solidified metal), the specific enthalpy in the averaging volume, the mass fraction solid, the specific latent heat, the effective heat conductivity, the temperature, the solute concentration in the averaging volume, and the liquid solute concentration, respectively. Note that since the solid velocity is zero the momentum conservation equation is valid only in the liquid and mushy zone. In the solid zone we require that $\mathbf{v} = 0$. This rather complicated system of nonlinear PDEs has been given a fully object-oriented implementation in Diffpack [21].

- *The equations governing the spinning of long and thin polymer fibers.* The model includes equations for quasi one-dimensional free surface non-Newtonian fluid flow coupled with an axi-symmetric heat transfer equation

[1].

$$
\begin{aligned}
R^2\rho w &= Q, \\
Q\frac{dw}{dz} &= -2\sigma_T R + \sigma_s\frac{dR}{dz} + \frac{d}{dz}(3\mu\frac{dw}{dz}R^2) + \rho g R^2, \\
\mu &= \eta_0 \exp(b_1(\frac{X_\infty}{X})^{b_2})\left[\frac{1}{1+c_1(\eta_0\frac{dw}{dz})^{m-1}}\right], \\
\frac{dY}{dz} &= \frac{nK}{w}Y^{(n-1)/n}, \quad Y = -\ln(1-\frac{X}{X_\infty}), \\
C_p w\frac{\partial T}{\partial z} &= -\frac{C_p}{2}\frac{dw}{dz}r\frac{\partial T}{\partial r} + \frac{k}{\rho}\frac{1}{r}\frac{\partial}{\partial r}(r\frac{\partial T}{\partial r}) + \frac{c}{\rho}.
\end{aligned}
$$

The primary unknowns are the radius $R(z)$ of the fiber, the velocity $w(z)$, the temperature field $T(r,z)$ and the degree of crystallinity $X(z)$ which governs heat production during the solidification of the flowing polymer. The other parameters in the model are taken as prescribed functions or constants. The solution methods include finite element, finite difference and Runge-Kutta methods.

# 3 The need for abstractions

Traditional programming in FORTRAN expresses numerical methods in terms of manipulations of arrays, reals and integers. Looking at a PDE, e.g.,

$$\frac{\partial u}{\partial t} = \nabla\cdot[k\nabla u]$$

the mathematical quantities used in scientific communication are much more abstract. We refer to the primary unknown $u$ and the prescribed coefficient $k$ as scalar fields. These fields are defined over domains, which in the software appear in a discretized form called grid. Moreover, the PDE contains spatial and temporal operators as well as associated boundary and initial conditions. OOP makes it possible to program in terms of such mathematical abstractions instead of directly manipulating primitive array structures. Of course, operations on abstract quantities must be realized as array operations in "do-loops" in the compiled code to obtain maximum efficiency, but the programmer is not involved with such low-level code.

The obvious attempt to apply OOP to PDE solvers is to create ADTs for the common mathematical and numerical quantities like fields, grids, operators, boundary conditions, linear systems etc. It is the authors' experience that mathematics provide a set of *possible* abstract software components, but in practice many of the mathematical abstractions have limited applications in their original form. For example, fields and grids are natural ADTs as will be explained in section 5, while operators and boundary conditions need a refined meaning which in Diffpack appears as special cases of more general concepts.

These concepts will hardly have any mathematical significance, but are instead aimed at obtaining more flexible and efficient implementations. Our message is hence that the construction of useful abstract data types can get some obvious ideas from mathematics itself, but a lot of additional work is needed to design tools that have the required flexibility for dealing with a wide range of real world applications.

# 4 Overview

In this section we briefly list some functionality that is found in Diffpack.

**BasicTools.** The BasicTools library contains basic C++ classes for strings, smart pointers, arrays, mathematical and statistical functions, menus for input data, and I/O. The smart pointer is a template class that offers reference counting and simplified dynamic memory management. The array classes are particularly suited for the large vectors and matrices encountered when solving PDEs.

For a programmer it is convenient to always read from and write to a single I/O source instead of having to distinguish between files, strings, shared memory, networks and ASCII/binary formats. By creating a base class for input and one for output, a unified syntax for input and output operations is enabled. Subclasses in this hierarchy implement specific I/O sources such as strings and files, including standard input and output. All I/O sources can by the programmer be set to either ASCII or binary mode, without affecting the syntax of the I/O statements. There is also an I/O subclass dedicated to a hardware independent binary format based on the standard RPC/XDR library in C.

**LaTools.** The LaTools library contains tools for linear algebra, especially for the representation and iterative solution of large sparse matrix systems. A thorough documentation of the basic ideas in LaTools is available in [6], see also [7]. Introducing a parameterized abstract matrix class for numerical computations, the programmer may work with a general matrix interface without dealing with, e.g., details of sparse matrix storage or special efficiency considerations. At present, there are classes implementing rectangular dense matrices, banded matrices, general sparse matrices, structured sparse matrices, diagonal matrices and point operators, the latter being convenient when programming finite difference methods.

Methods for solving matrix systems are also organized in a class hierarchy. Solution of the system is then accomplished by having a base class pointer to this hierarchy and calling a virtual function `solve` that takes a linear system ($Ax = b$) object as input, where $x$ is a possible starting vector for iterative methods. At the return from this function, $x$ holds the computed solution. Preconditioning is a part of the linear system class representation. This approach makes it easy to avoid the many details of matrix data structures and iterative algorithms that commonly complicate C and FORTRAN code for PDEs. LaTools

contains, at the time of this writing, these iterative methods: Conjugate Gradients, GMRES($k$), Orthomin($k$), Generalized Conjugate Residuals (restarted Orthomin), SYMMLQ, Transpose-free QMR, BiCGStab, SSOR, SOR, Gauss-Seidel and Jacobi iterations. For PDE problems there are also sophisticated iterations like multigrid and domain decomposition methods, see page 10. Using one of the solvers as template, it is straightforward to implement new methods.

The available preconditioners include the ILU/MILU/RILU family for sparse matrices, SSOR, SOR and Jacobi methods, user defined procedures (e.g. efficient direct solvers), as well as multilevel and domain decomposition preconditioners. For further discussions of preconditioned iterative methods we refer to [5] and references therein. The basic iterative methods can be combined with the desired preconditioner at run time. Several stopping criteria and tools for monitoring the convergence history of iterative solvers are available. In addition, there is a class hierarchy for nonlinear solvers such as Newton's method and successive substitution (Picard iteration). LaTools offers a framework for iterative solvers that makes it very easy to implement new methods and storage formats [6].

LaTools demonstrates a basic design strategy in Diffpack in that various methods are realized as subclasses in a hierarchy. The user can then at runtime choose methods and combine them. For example, the user can interactively choose a matrix format from the matrix hierarchy, a solver from the solver hierarchy and a preconditioner from the preconditioner hierarchy. This reflects a main application of Diffpack, namely easy and flexible experimentation with various numerical methods and parameters.

**DpKernel.** Classes directly related to differential equations are collected in the DpKernel and DpUtil libraries. In DpKernel one finds classes for the most basic abstractions, such as fields and grids. We have developed an abstract base class for grids, with lattices, scattered points and finite element grids as typical subclasses. A field contains a grid, a set of numerical data and a rule for calculating field values from the grid and the data. Scalar fields are organized in a class hierarchy. Particular subclass implementations include finite element fields, finite difference fields (on uniform lattices) and piecewise constant fields. Vector fields are simply an array of scalar fields, but are represented as a separate class hierarchy. In a Diffpack simulator, the primary unknowns, the derived quantities (flux, stress etc.) and the coefficients in the PDEs are represented as fields. It is common, but not necessary, that all the fields are defined over the same grid. Using smart pointers to the grid object in the field classes makes it easy and safe for the fields to share the same grid data structure. We will later present the field classes in somewhat more detail.

In DpKernel there are several classes for finite element programming. These are independent of the equations to be solved and aim at a wide range of applications, including heat transfer, fluid flow, and structural analysis. The finite element classes cover numerical integration rules over elements, an element hierarchy defining the basis functions and the geometric mapping of various elements, and representation of elemental matrices and vectors.

The present version of Diffpack supports elements of the multilinear (linear, bilinear, trilinear) and multiquadratic (quadratic, biquadratic and triquadratic) type, the eight node 2D quadrilateral and the 20 node 3D box element as well as triangles and tetrahedras. Several mixed finite elements are also available. There are two types of numerical integration rules; Gauss quadrature up to order 10 and nodal point rules up to order 3. For triangles and tetrahedras only 1, 2 and 3 point Gauss quadrature is available. The addition of new rules is of course a trivial task.

**DpUtil.** This library is an extension of DpKernel and offers, for example, some finite element preprocessors, a toolbox for storage and retrieval of fields, interface to various visualization tools, interface to random fields, a collection of finite element algorithms, and a high-level interface to LaTools aimed at PDE applications. For time dependent problems there is a class that administers the time stepping parameters.

At present Diffpack supports file formats for the following visualization programs: `plotmtv`, `gnuplot`, `xmgr`, `xgraph`, `AVS`, `IRIS Explorer` (AVS ucd format) and `Isvas 3.1`. In addition, an interface to `Vtk` is under development. For grid generation there is a preprocessor for rectangular and box shaped domains with an efficient interface aimed at academic test problems and a super element based preprocessor that requires the user to provide a coarse mesh describing the geometry. More complicated geometries require automatic mesh generation software, and for this purpose Diffpack has an interface to the well-known `GEOMPACK` package by Barry Joe [17].

**Multilevel and domain decomposition methods.** PDEs can be efficiently solved by multilevel and domain decomposition methods, see [27] for further discussion. In particular, for wide classes of problems such methods can compute the numerical solution in $O(n)$ arithmetic operations, where $n$ denotes the total number of degrees of freedom. This behaviour is optimal in the sense that the order of such work estimates can not be improved upon. Besides being efficient numerical algorithms, it is also convenient that the methods in question usually are well suited for parallel implementation. In general, this is due to their implicit decoupling of a large-scale problem into smaller parts.

In the case of multilevel methods, of which multigrid is the best known example, the problem is split into subproblems by introducing a sequence of computational grids $\{\mathcal{T}_j\}_{j=0}^{J}$ with different levels of spatial resolution $h_0 > h_1 > \cdots > h_J$. In connection to finite element methods, this approach leads to a sequence of finite element spaces $\{\mathcal{V}_j\}_{j=0}^{J}$ such that $\mathcal{V}_j \subset \mathcal{V}_{|+\infty}$. The general approach is to smooth the rapidly changing (high frequency) part of the error at a fine grid level, typically by applying a stationary iteration based on a matrix splitting (SOR, Jacobi, etc.) or an incomplete factorization. The corresponding residual is then restricted to a coarser grid level where the smooth (low frequency) error components are dealt with by solving a corresponding coarse-grid formulation of the problem at hand. The coarse level solution can

then be interpolated back to the fine level grid for a correction step, possibly involving further smoothing steps. The two-level V-cycle just outlined can be extended to a general number of grid levels, and to using other patterns (*cycles*) for communication of data between the different levels.

A similar strategy is also used in domain decomposition methods, with the exception that the division into subproblems is based on a splitting of the computational domain $\Omega$ into subdomains $\{\Omega_j\}_{j=0}^{J}$. The discretized differential operator is then restricted to each subdomain, thus leading to a collection of smaller problems that can be solved independently. If the subdomains are non-overlapping, we also have to solve special equations for the unknowns located at the inner boundaries between neighbouring subdomains. Finally, the solutions of all the local subproblems are joined to form a global solution. The approach using such non-overlapping domains is often referred to as a substructure method. Another formulation of domain decomposition is based on the use of overlapping domains. In this case there is no need to solve for the inner boundaries, since these nodes are duplicated in both neighbouring domains. Instead, we have to pay special attention to the overlapping parts when forming the global solution. This approach, which is known as Schwarz methods, leads essentially to two different strategies: (*i*) the multiplicative variant where the solution of each subproblem uses values from each neighbouring domain to specify local boundary conditions, and (*ii*) the additive methods where each subproblem is solved independently of all other domains.

Careful analysis of the multilevel and domain decomposition concepts shows that most methods of this type can be formulated in a unified framework, where the basic steps involve transfer operators for restriction and interpolation, exact or approximate subspace solvers, and the evaluation of residuals on a subdomain. This framework can also be used as a guideline for very flexible implementations of the corresponding algorithms, see [8, 35] for a description of the design and implementation of such methods in Diffpack. This particular implementation currently incorporates multilevel iterations (additive, multiplicative and nonlinear types), overlapping Schwarz methods (additive, multiplicative and nonlinear types, with or without a coarse grid), Schur complement iterations (exact and approximate versions), and Schur complement preconditioners (Neumann-Neumann and wire basket types, with or without a coarse grid).

In Diffpack, the multilevel and domain decomposition strategies are realized as a layer on top of the linear algebra and finite element functionality described above, thus taking full advantage of already existing high-level building blocks. For instance, such software components have been used to create linear and nonlinear operators, smoothers, transfer operators and residuals. Using the same software philosophy as in other parts of Diffpack, it is trivial for the user to run a multigrid solver and experiment with various pre- and post-smoothers (choice of algorithm, number of sweeps, order of unknowns), coarse grid solvers (iterative and direct, grid size), cycle-types, nested iteration, non-matching grids, semi-coarsening, multigrid used as a preconditioner or as a stand alone solver, different nonlinear versions, grid types and special procedures to initialize operators. For domain decomposition, the type and precision of sub-domain solvers,

the decomposition of the domain, the type of a coarse grid and coarse-grid solver and the scaling of transfer operators are of main interest.

**A development environment.** Comprehensive packages such as Diffpack involve a large number of files, libraries, subdirectories and dependencies. Different users have different demands when using a comprehensive package. For example, a novice user will link his new application to well tested, non-optimized libraries where array index checks and null pointer checks are provided. More experienced users may contribute to further development of the libraries and need to link their applications to the most recent, possible unstable, non-optimized beta version of the package. Other users having stable, but computationally intensive applications, need to link their codes to optimized versions of Diffpack. Most networks today are heterogeneous and different users need the software on different hardware platforms using different compilers. Another central problem is to verify that a new version of the libraries or an application produce the same results as before with a particular compiler option on a given platform. Finally, various degrees of recursive directory clean-up are needed.

The complexity in software development and usage as we have described above, have been seriously addressed in the Diffpack project. A `gnumake`-based system has been developed that hides all the details associated with the complexity described above and simplifies compilation and verification tasks.

# 5 General representation of fields

Scalar and vector fields are basic quantities that enter PDEs. Finite element fields are fundamental for representing the primary unknowns in finite element applications. Coefficients in PDEs are also natural field abstractions, and these may include explicit formulae, constants, finite difference fields imported from other computations, as well as finite element fields. Using ADTs and OOP one can easily create an environment that allows a flexible representation of fields in a simulator. For the purpose of demonstrating how OOP and class hierarchies are applied in a system like Diffpack, we will outline some details concerning the representation of fields.

The basic idea is that one programs in terms of a base class `Field` for scalar fields or `Fields` for vector fields. Class `Field` will typically be a class without data, but with a definition of some functions, e.g., for evaluating the field:

```
class Field {
public:
  virtual real valuePt  (const Ptv(real)& x);
  virtual real valueFEM (FiniteElement& fe);
};
```

Here, the type `real` is simply a macro which makes it easy to switch between single precision (`float`) and double precision (`double`) arithmetics. The function `valuePt` is the obvious interface for evaluating the field at a space-time point.

In finite element methods, the process of locating the element that contains a given point `x` may be very time consuming, and since one often knows the details of the relevant element in terms of a particular object (of class `FiniteElement`) an additional function `valueFEM` is included for efficiency in finite element computations.

Subclasses of `Field` can now implement various types of fields. For example, the constant field is trivial,

```
class FieldConst : public Field {
  real constant;
public:
  real valuePt  (const Ptv(real)& x) { return constant; }
  real valueFEM (FiniteElement& fe)  { return constant; }
};
```

Explicit functions, like $f(x) = A\sin(\omega||x||)$, are implemented as subclasses of `FieldFunc`:

```
class FieldFunc : public Field {    // general class for explicit functions
public:
  real valuePt  (const Ptv(real)& x) =0;
  real valueFEM (FiniteElement& fe) // find the global point implied by fe
    { Ptv(real) x; fe.getGlobalEvalPt(x); return valuePt (x); }
};

class MySineFunc : public FieldFunc {
  real A, omega;
public:
  real valuePt  (const Ptv(real)& x)
    { return A*sin(omega*x.norm()); }
};
```

Representation of a function in terms of a class like this is often referred to as a *functor* [10]. We see how elegant the `MySineFunc` functor is: Parameters to the functions like $A$ and $\omega$ are stored as data members while the parameters that are a part of all functions representing fields (`x` and `t`) are explicitly given as argument to the evaluation function. It is now easy to pass `MySineFunc` to modules that take general `Field` or `FieldFunc` arguments and perform evaluations in terms of `valuePt` or `valueFEM`. Traditional implementations in FORTRAN and C would need to have $A$ and $\omega$ as global variables.

Fields over grids, like `FieldFE` for finite element fields and `FieldFD` for "finite difference fields" over uniform lattices, are implemented as subclasses of a general interface class `FieldWithPointValues`:

```
class FieldWithPointValues : public Field {
public:
  int       getNoPoints() =0;      // total no of grid points
  Ptv(real) getPt(int i) =0;       // get coordinates of grid point no i
  real      valuePoint (int i) =0; // value at grid point no i
  // ...
};
```

Before presenting classes `FieldFE` and `FieldFE` we briefly mention that `GridFE` is a class for finite element grids, `GridLattice` is a class for uniform box grids commonly used in finite difference methods and `Vec(real)` is a vector class[3] with real entries.

```
class FieldFE : public FieldWithPointValues {
  Handle(GridFE)    grid;          // associated finite element grid
  Handle(Vec(real)) nodal_values;  // field values at the nodal points
public:
  real valuePt  (const Ptv(real)& x);                  // complicated!
  real valueFEM (FiniteElement& fe);                   // simple
  int  getNoPoints () { return grid->getNoNodes(); }
  real valuePoint (int i) { return nodal_values(i); }
  // ...
};

class FieldFD : public FieldWithPointValues {
  Handle(GridLattice) grid;     // simple uniform box grid
  Handle(Vec(real))   ptvalues; // field values at the grid points
public:
  real valuePt  (const Ptv(real)& x);                  // simple
  real valueFEM (FiniteElement& fe);                   // just call valuePt
  int  valuePoint (int i) { return ptvalues(i); }
  // ...
};
```

We have here introduced a *handle* which is a smart pointer, that is, class `Handle(X)` is a pointer to class `X` that can perform reference counting and automatically delete `X` when there are no more references to this object. For fields with grids this is important since several fields may share the same grid. This grid cannot be deleted before all fields have finished their use. Representing the field values as a `Handle(Vec(real))` enables the possibility of sharing the field values between several fields. This is convenient when the computations are performed in a geometrically transformed domain. One can then have a computational grid and a physical grid, with two corresponding `FieldFE` objects. These two field objects can share the same data structure for holding the nodal point values, but point to different grids.

Corresponding to each of the field classes above, there are similar classes for representing vector fields (`Fields`, `FieldsWithPointValues`, `FieldsFE`, `FieldsFD` etc). The important advantage of having field hierarchies can be demonstrated by considering the PDE

$$\frac{\partial u}{\partial t} + \vec{v} \cdot \nabla u = \nabla \cdot (k \nabla u) \tag{2}$$

where $u$ is the unknown scalar field, while $\vec{v} = (v_1, v_2)$ is a known 2D vector field and $k$ is a known scalar field. Suppose $u$ is represented by a finite element field, $v_1$ and $v_2$ are represented by scalar finite difference fields, while $k$ is given

[3]The expression `Vec(real)` corresponds to the template construction `Vec<real>`, but we use C preprocessor macros instead of templates simply because it gives greater flexibility and control for the program developer.

by an explicit function. It would be convenient to code the PDE without paying attention to the *type* of field we are working with. In the solution method we only need to evaluate $\vec{v}$ and $k$ at specific spatial points. This information hiding is now trivially accomplished by applying the field hierarchy. We represent $u$ as a `FieldFE` object, $k$ as a functor `kFunc` derived from `FieldFunc`, and $(v_1, v_2)$ as a `FieldsFD` object. In the simulator class for equation (2) full flexibility is enabled by using a `Handle(Field)` pointer to $k$ and a `Handle(Fields)` pointer to $(v_1, v_2)$. Information on the particular type of diffusion or velocity field is only needed at construction time of the fields. Afterwards we can evaluate the coefficients in the PDE by just calling the virtual `valuePt` function, that is, the details of whether $k$ is a function, constant or a finite element field are completely hidden in the parts of the code where the only functionality we need of $k$ is to evaluate the field at a space-time point.

# 6 Some details on finite element methods

This section outlines the finite element toolboxes. It is assumed that the reader has basic knowledge about finite element methods and conventional finite element programming.

**Boundary conditions.** Flexible assignment of boundary conditions is accomplished by introducing a set of boundary indicators $b_1, \ldots, b_q$. Each node in a finite element mesh can be marked by one or more boundary indicators. This functionality enables the user to divide the boundary into various (possibly overlapping) segments. Internal boundaries can also be modeled. In application programs one introduces a convention for associating boundary indicators with boundary conditions. For example, in a heat conduction program one could let $b_1$ and $b_2$ mark two parts of the boundary where the temperature is known while $b_3$ could make the part of the boundary where the heat flux is prescribed. When generating the grid, the user must know this interpretation of the first three boundary indicators. Additional indicators can be added by the user for marking parts of the boundary that are to be plotted. Essential boundary conditions are easily coded by checking whether a node is subjected to a particular boundary indicator, while Neumann type conditions are implemented by checking whether a side in an element is subjected to a certain indicator. A side is marked with an indicator if all the nodes on the side are marked with that particular indicator.

Since every simulator has its own convention of interpreting the boundary indicators as boundary conditions, it is easy to include a consistency check on the given conditions in a particular execution. Consider the heat conduction simulator again. By running through all nodes one can easily detect the inconsistency of marking a node with, e.g., $b_1$, $b_2$ and $b_3$.

**Finite elements.** A finite element is here said to consist of two parts, one reflecting the geometry of the element and one reflecting the interpolation prop-

erties. Physical ("material") properties or numerical integration rules are not a part of an object defining a finite element. Various elements are organized in a hierarchy with class `ElmDef` as base. The virtual functions in the hierarchy perform, for example, geometric mapping from local to physical coordinates, evaluation of the interpolation (basis) functions, access to nodes on the various sides in the element, and drawing of the element. For isoparametric elements the geometric mapping applies the basis functions, and the nodes used in the mapping coincides with the basis function nodes. By class derivation and redefinition of the basis functions and their nodes it is straightforward to derive special elements that appear in mixed finite element methods.

A separate class `ElmItgRules` has a collection of numerical integration rules that can be combined with a finite element. Hence, it is easy to use different integration rules on the same element. Unfortunately, much of the finite element programming literature tends to mix geometric properties, interpolation properties, material properties (coefficients in the PDEs) and integration rules in the code. It is important that each of these quantities must be represented as separate building blocks when developing generic code intended for a wide range of applications.

**Finite element grids.** Finite element grids are represented by the class `GridFE`. The grid contains the nodal coordinates and the element connectivity as well as information on boundary indicators and finite element types. Different element types can of course be used in the same grid. The grid is viewed as a purely geometric object with no direct relation to particular differential equations. For example, the grid contains only geometric information, and no element-wise "material" properties (these are instead represented separately by a piecewise constant field over the grid). The `BasisFuncGrid` class contains additional information on the type of basis functions for each element. For isoparametric elements, class `BasisFuncGrid` can simply reproduce `GridFE` information, while for mixed finite element methods the class has additional data and must modify many of the `GridFE` functions to account for the nodes used to specify the basis functions.

**Finite element fields.** A very simple finite element field class was outlined in section 5. In the real Diffpack implementation the `FieldFE` class represents its grid information in terms of a `BasisFuncGrid` object rather than a `GridFE` object. The nodal values are, nevertheless, collected in a vector as outlined in section 5. Functionality of a `FieldFE` object includes, among other things, input and assignment of nodal field values, grid access, and interpolation of the field and its derivatives at arbitrary points. If the grid is simply a uniform lattice, the field is automatically available in a lattice or "finite difference" form as well. This enables field indexing like $f(i,j,k)$ and easy access to grid parameters like the spacings $\Delta x$, $\Delta y$ and $\Delta z$.

Vector and tensor fields over finite element grids are offered as class `FieldsFE`, containing just an array of smart pointers to scalar fields. Hence, all the `FieldFE`

member functions are available for each component of the vector or tensor field. However, several of these functions are reimplemented in class `FieldsFE` to increase the computational efficiency.

**Fields and linear systems.** Finite element methods will usually give rise to linear systems. We represent the primary unknowns in a simulation by field objects, while in the linear system, the unknowns are collected in a single vector $x$. A mapping between the numbering of the degrees of freedom in a collection of fields and in a linear system is therefore required. Class `DegFreeFE` keeps track of this information. When solving a scalar PDE with unknown scalar field $u$, $x$ can simply be the vector of nodal values of $u$. A `DegFreeFE` object is then the identity mapping. For a vector PDE where $\vec{v}$ is the unknown, the nodal values of each component in $\vec{v}$ must be combined to form $x$. More complicated situations arise in mixed finite element methods where there are multiple fields and the degrees of freedom differ for each field. Based on the mapping between degrees of freedom in the fields and the vector $x$, the class `DegFreeFE` can compute the matrix bandwidth or sparsity pattern. The essential boundary conditions that affect the linear system are also administered by class `DegFreeFE`. When solving a set of PDEs sequentially, one will typically associate a `DegFreeFE` object with each linear system. The linear system itself is represented by a class `LinEqAdm` which acts as a simplified interface to LaTools when using finite element methods.

**Evaluation of finite element equations.** Each elemental matrix and vector are stored in an `ElmMatVec` object. This object has a smart pointer to a `DegFreeFE` object that can modify the elemental matrix and vector to account for essential boundary conditions. If the `ElmMatVec` object is computed by numerical integration, the programmer needs to implement the integrand of the weak form of the boundary value problem at an integration point. To accomplish this, the programmer must have access to the basis functions, their derivatives, the Jacobian, and the integration weight at the current integration point. In addition, the global coordinates of the integration point may be needed. All this information is offered by the class `FiniteElement`. This class also offers administration of the numerical integration over the elements. Of course, such information is gathered from its accompanying `GridFE`, `ElmDef`, `BasisFuncGrid` and `ElmItgRules` objects, reflecting the layered design principle.

**Implementing a simulator.** The most common finite element algorithms, such as the complete element assembly process, are collected in a class `FEM`. A simulator solving PDEs is usually implemented as a subclass of `FEM`. The various parts of these algorithms that may be problem dependent are collected in virtual functions. Evaluation of the integrand of the weak form and assignment of initial and boundary conditions cannot be given sensible default implementations and are hence represented by pure virtual functions. Other functions, performing elemental matrix and vector computation or numerical integration

over the element are given default implementations which can be overridden in the simulator subclass if necessary. This design allows rapid prototyping of a simulator applying general inherited algorithms. At a later stage, parts of the inherited algorithms can be rewritten for the particular problem at hand to increase the computational efficiency.

# 7 An example

**The mathematical model.** To demonstrate a typical finite element code in Diffpack, we present a very simple example where the Poisson equation

$$\nabla \cdot (k\nabla u) = -f$$

is solved in a domain $\Omega \in \mathbf{R}^d$. The boundary condition is for simplicity $u = 0$. The functions $k$ and $f$ are prescribed, but their format may vary. As in section 5 the functions may be explicit formulae, or discrete fields defined on a grid which may not necessarily be identical to the grid used for the primary unknown $u$. Employing a Galerkin finite element method with $N_i$ as test and trial functions, the elemental matrix $E_{ij}$ becomes

$$E_{ij} = \int_{\Omega_e^{\text{ref}}} k\nabla N_i \cdot \nabla N_j \det J d\Omega \tag{3}$$

where $\Omega_e^{\text{ref}}$ denotes the reference element and $\det J$ is the Jacobian of the mapping between local (reference) and global coordinates. Correspondingly, the elemental vector $b_i$ reads

$$b_i = \int_{\Omega_e^{\text{ref}}} f N_i d\Omega. \tag{4}$$

**Finite element class interfaces.** We will first give a brief presentation of the relevant parts of some classes for finite element programming.

```
class FiniteElement {
public:
  int     getNoBasisFunc () const;    // number of basis functions
  int     getNoSpaceDim () const;     // number of space dimensions
  real     N (int i) const;           // basis (interpolation) function #i
  real    dN (int i, int dir) const;  // derivative of N(i) in direction dir
  real    detJxW () const;            // Jacobian times num.itg. weight
  real    getGlobalEvalPt (Ptv(real)& x); // global coord. of itg. point
};
```

The `Ptv(real)` object represents a spatial point in $\mathbf{R}^d$. In the finite element grid and the degrees of freedom handler we need a few functions:

```
class GridFE : public Grid {
public:
  int getNoNodes () const;             // the no of nodes in the grid
  int getNoSpaceDim () const;          // the no of space dimensions
  BooLean BoNode (int n, int i) const; // is node n subjected to
                                       // boundary indicator number i?
};


class DegFreeFE {
public:
  void fillEssBC (int dof, real v);    // set essential BC at a d.o.f.
  void initEssBC ();                   // init for calls to fillEssBC
};
```

Here `BooLean` is an `enum` for boolean types. The finite element algorithms needed in our model problem are offered by class `FEM`:

```
class FEM {
public:
  virtual void fillEssBC () =0;
  virtual void makeSystem (DegFreeFE& dof, LinEqAdm& lineq);
  virtual void calcElmMatVec (int e, ElmMatVec& elmat, FiniteElement& fe);
  virtual void numItgOverElm (ElmMatVec& elmat, FiniteElement& fe);
  virtual void integrands (ElmMatVec& elmat, FiniteElement& fe) =0;
};
```

Here `makeSystem` runs the loop over all elements, `calcElmMatVec` computes the elemental matrix and vector, normally by numerical integration which is carried out by `numItgOverElm`. This function calls `integrands` which in our case samples the contribution to (3) and (4) at a numerical integration point (the `fe` argument has information about the basis functions, the Jacobian etc. at the current integration point). In addition, we need to set the essential boundary conditions in `fillEssBC`.

**The simulator class.** The simulator for solving our simple boundary value problem is implemented as a subclass `PoissonEq` derived from `FEM`. The local data will be a grid, a field representation of $u$, $f$ and $k$, a linear system and a degrees of freedom handler.

```
class PoissonEq : public FEM {
  GridFE          grid;  // finite element grid
  Handle(Field)   k,f;   // coefficients in the PDE
  Handle(FieldFE) u;     // finite element field over the grid
  LinEqAdm        lineq; // representation and solution of linear systems
  DegFreeFE       dof;   // mapping between u values and unknowns in lineq
public:
  PoissonEq ();
  virtual void fillEssBC ();   // set essential boundary conditions
  virtual void integrands (ElmMatVec& elmv,
                           FiniteElement& fe);   //E_ij and b_i
  void scan ();       // read input data and initialize the data members
  void driver ();     // main algorithm
};
```

Class `FEM` requires us to implement a `fillEssBC` function that tells the `DegFreeFE` object about our essential boundary conditions in the linear system. In this context we need to relate boundary indicators to the boundary conditions. A natural choice is to let indicator number 1 indicate nodes where $u = 0$. The contribution of the discrete finite element equations at an integration point is evaluated in `integrands`. There are two coefficients in the PDE, $f$ and $k$, and these are conveniently represented as `Field` abstractions, see Section 5. Reading of input data (from a menu) and initialization of the data structures are performed in `scan`. Grid generation by means of a preprocessor and choice of storage scheme and solvers for the linear system will typically be carried out here. Finally, we have a function `driver` for administering the computational tasks. Here is the code:

```
void PoissonEq:: fillEssBC () {
  dof.initEssBC();
  const int nno = grid.getNoNodes();  // no of nodes in the grid = # d.o.f.
  for (int i = 1; i <= nno; i++) {
    if (grid.BoNode(i,1))    // is boundary indicator #1 marked at node i?
      dof.fillEssBC (i,0.0); // set value of essential BC at d.o.f. #i
}

void PoissonEq:: integrands (ElmMatVec& elmv, FiniteElement& fe) {
  int i,j,k; real s;                    // counters and help variables
  const int  nbf = fe.getNoBasisFunc(); // range for i and j
  const real detJxW = fe.detJxW();      // Jacobian times the itg. weight
  const int  d = fe.getNoSpaceDim();    // no of space dimensions
  real f_pt = f->valueFEM(fe);          // interpolate f at curr. itg. point
  real k_pt = k->valueFEM(fe);          // interpolate k at curr. itg. point

  for (i = 1; i <= nbf; i++) {
    for (j = 1; j <= i; j++) {
      for (s=0, k=1; k <= d; k++)       // scalar product in integrand
         s += fe.dN(i,k)*fe.dN(j,k);
      s *= k_pt*detJxW;
      elmv.A(i,j) += s;                 // add element matrix contribution
      if (i!=j) elmv.A(j,i) += s;       // take advantage of symmetry
    }
    elmv.b(i) += f_pt*detJxW;           // add element vector contribution
  }
}

void PoissonEq:: scan () {
  // make grid and perform other initializations
  // bind the f and k handles to proper subclass objects of Field, f.ex.
  // let k be a FieldFE read from file and let f be an explicit function
}

void PoissonEq:: driver () {
  makeSystem (dof, lineq);     // inherited function, makes lin. system
  lineq.solve ();              // solve linear system
  // print results
}
```

The program for solving our model problem will typically have the form

```
main () {
  PoissonEq simulator;
  simulator.scan();      // read input data and initialize
  simulator.driver();    // compute solution
}
```

# 8 Code extension based on OOP

Suppose we want solve a problem which is slightly different from the one in the previous section, e.g., let $f = 0$, $k = 1$, with $u = g$ on the boundary. The traditional way to include this version of the problem is to extend the original code with if-else tests and possibly additional functions and data items. A more elegant extension based on the OOP concept consists of deriving a new class `LaplaceEq` from `PoissonEq` and re-implementing `fillEssBC` and `scan`. Since both these functions are virtual, the general algorithms will automatically call the new functions. In `LaplaceEq::scan` we set $f$ and $k$ to constant fields and in `LaplaceEq::fillEssBC` we evaluate the $g$ function. The data items in `LaplaceEq` will only be a `Field` handle to represent $g$. Most of the computations in `LaplaceEq` will take place in the already tested `PoissonEq` and in the library classes. When the new code works, one can increase the efficiency by re-writing `integrands` such that the operations involving $f$ and $k$ are avoided. If one applies a grid with e.g. triangle or tetrahedra elements, one can also increase the efficiency considerably by integrating the elemental contributions analytically. The inherited `calcElmMatVec` function (in `FEM`) can then be rewritten to simply evaluate the analytical expressions, hence there is no need for numerical integration and the `integrands` routine.

Another possible extension is to consider a PDE with an extra term, e.g., a convection term $\vec{v} \cdot \nabla u$, where $\vec{v}$ is some prescribed vector field. A simulator for this problem is conveniently implemented as a subclass `ConvDiffEq` of class `PoissonEq`. If the boundary conditions are the same as in class `PoissonEq`, one needs to re-write `integrands` only, otherwise a new `fillEssBC` must be provided. The main advantage of implementing various versions of a common PDE problem in different classes is that the original classes remain unaltered. The strategy is particularly well suited when different people work on the same basic PDE but need different extensions of the code.

OOP and the concept of inheritance allow us to establish a rough prototype simulator for a PDE by inheriting default versions for most of the virtual functions. The virtual functions that *must* be specified in the simulator class (typically initial and boundary conditions and the specification of the PDE) are given simple implementations. Enhanced implementations of the simulator may arise from further use of inheritance where some of the virtual functions are improved, with respect to either generality or efficiency.

**Programming at a lower level.** The efficiency can sometimes be increased by exploiting special features of the problem in question. As previously mentioned, the Diffpack libraries have a layered design in order to enable the pro-

grammer to develop code at a lower level where there is more control of the numerical details. As an example, consider the wave equation $\partial^2 u/\partial t^2 = \nabla^2 u$ discretized by finite elements in space and by an explicit finite difference scheme in time. The resulting equations are of the form

$$Mu^{n+1} = 2Mu^n - Mu^{n-1} + Ku^n$$

where $M$ and $K$ are matrices and the superscript $n$ denotes the time level of the unknown field values $u$ of $u$. Of course, we could easily use the explained `FEM` features to assemble a linear system object at each time level

and solve the system. However, this is unnecessarily time consuming since $M$ and $K$ are constant in time. By constructing $M$ and $K$ initially and lumping $M$, the solution at each time level is enabled by simple matrix-vector multiplications. Instead of a `LinEqAdm` object, we could use two matrices $M$ (diagonal) and $K$ (sparse). The mass matrix $M$ is computed by a special `FEM` function, while $K$ can be calculated in the ordinary way using another overloaded `FEM::makeSystem` function. The procedures are documented in [19].

# 9 Controllers

The simulator can be treated as a concrete ADT containing the complete implementation, or it can be used as an abstract simulator where only the general functionality, not the numerical details, is visible. For such an abstract simulator it is easy to develop general code for various types of program control. By a controller we mean an ADT that manages several runs of a simulator. Usually, the simulator is derived from the controller ADT. Often the controller represents the implementation of a numerical method where the simulator acts as a "black-box" functional relationship. Examples of controllers include ADTs for optimization, stochastic simulation and parameter sensitivity analysis. The design of the latter controller type is outlined below.

In Diffpack there is a class `MenuUDC` that offers functionality for running through a large number of simulations. By proper treatment of the results one can use this feature to perform a sensitivity analysis of a simulator.

Consider for example a simulator with the input parameters $a$, $b$ and $c$, all of them being real numbers. If we specify $a \in \{1, 2\}$, $b = 3$ and $c \in \{3.5, 8, 100\}$, we would like the program to enter a loop over the six combinations of these input data and execute the simulator for each combination. We then need the following functionality: (1) an interpreter of the special input syntax, (2) an administering loop over all combinations, (3) a user defined function that executes the simulator for one particular combination of the input parameters, and (4) a user defined function that post-processes the data from each simulation. Points 1 and 2 are offered as functions in a menu system (class `MenuSystem`), while points 3 and 4 are represented as virtual functions in a base class `MenuUDC` from which the user's simulation class must be derived. The functions for points 1 and 2 can then operate on a general `MenuUDC` pointer or reference. If we have a simulator, we can equip it with a parameter sensitivity analysis module

by deriving the simulator from class `MenuUDC` and implementing a function that performs one single execution and one function that collects the interesting data from the current execution and stores this information in, e.g., tabular form. For the latter purpose, classes for automatic report generation have been created in Diffpack. With these tools, the user can run a large set of problems and browse through the results using Unix tools on ASCII files, `Mosaic` or `netscape` for HTML files, or PostScript viewers for LaTeX files.

Tools for the numerical solution of stochastic PDEs by Monte Carlo simulation can be developed as a general `StochasticPDE` controller that acts as a base class for the simulator. This class contains the data structures and code for managing the Monte Carlo loop. A set of virtual functions that must be implemented in the user's class provides all the problem dependent actions in the problem. These functions include realization of the random input data, execution of a deterministic problem, updating of statistical estimators etc. In other words, the controller does all the general things while virtual functions take care of the specialities in the user's problem.

# 10 Combining simulators

When simulators for PDEs are represented as independent ADTs, there is an attractive method for combining such ADTs in order to solve a system of PDEs. We will now outline this method. Suppose we have a simulator ADT for solving the incompressible Navier-Stokes equations

$$\nabla \cdot \vec{v} = 0, \tag{5}$$

$$\varrho \left( \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} \right) = -\nabla p + \mu \nabla^2 \vec{v} + (\varrho + \alpha T)\vec{g}, \tag{6}$$

where $\vec{v}$ is the velocity, $p$ is the pressure, $\varrho$ is the density, $\mu$ is the dynamic coefficient of viscosity, $\alpha$ is a thermal expansion coefficient, $T$ is the temperature deviation and $\vec{g}$ is the acceleration of gravity. Furthermore, we have another ADT for solving the heat transport equation:

$$\frac{\partial T}{\partial t} + \vec{v} \cdot \nabla T = k \nabla^2 T \tag{7}$$

where $T$ is the temperature (deviation), $\vec{v}$ is the velocity field and $k$ is a thermal diffusion coefficient. The structure of the two ADTs will typically be

```
class NavierStokes : public ... {
protected:
  Handle(Fields) v;    // primary unknown: velocity
  Handle(Field)  p;    // primary unknown: pressure
  Handle(Field)  T;    // known coefficient in the gravity term (here =0)
  real rho, alpha, mu; // coefficients in the PDE
  void solveAtThisTimeLevel ();  // solve for v and p at one time level
  void scan (MenuSystem&);       // read input and initialize
  void integrands (ElmMatVec& elmat, FiniteElement& fe);  // def. weak form
```

```
};

class HeatTrans : public ... {
protected:
  Handle(Field)  T;    // primary unknown
  Handle(Fields) v;    // known fluid velocity
  solveAtThisTimeLevel ();
  void scan (MenuSystem&);
  void integrands (ElmMatVec& elmat, FiniteElement& fe);
};
```

The `solveAtThisTimeLevel` function typically computes the primary unknowns as spatial fields at a particular time level. (This function will be like `driver` in class `PoissonEq`.)

Suppose we now want to create a simulator for free convection in fluids using the common Boussinesq approximation as indicated in (6) when $T \neq 0$. If we choose a sequential approach for the coupled system of PDEs we can, at each time level, first solve the incompressible Navier-Stokes equations with a prescribed temperature in the buoyancy term and then solve the heat equation for the temperature with the velocity field as known. The merging of the two simulators can easily be accomplished in C++ by creating a new class `FreeConv` that has `HeatTrans` and `NavierStokes` as data members. In `FreeConv` we let `T` in `NavierStokes` point to `T` in `HeatTrans`, and `v` in `HeatTrans` point to `v` in `NavierStokes`. The class `FreeConv` should implement a function `solveAtThisTimeLevel` that first calls the similar function in `NavierStokes` to compute the velocity field and then calls `solveAtThisTimeLevel` in `HeatTrans` to compute the temperature field. Note that the communication of data is accomplished by letting fields in a parent class point to the corresponding fields in the other class. It only takes a few lines of C++ code in class `FreeConv` to combine the numerics of the two simulators. More importantly, the new simulator introduces no modifications of the numerics in the already tested parent simulators. Hence, the combination of ADTs in this way increases the reliability of the new code and reduces the implementation effort substantially. C++' promise of code reusability is here demonstrated to be a reality in a numerical context. Below is an outline of the basic steps in the implementation.

```
class FreeConv {
  HeatTrans    heat;
  NavierStokes flow;
  void solveAtThisTimeLevel ();
  void scan (MenuSystem&);
};

void FreeConv::scan (MenuSystem& menu) {
  menu.setCommandPrefix("heat"); heat.scan (menu);
  menu.setCommandPrefix("flow"); flow.scan (menu);
  menu.unsetCommandPrefix();
  heat.v.rebind (flow.v);  // coupling of coefficients/unknowns
  flow.T.rebind (heat.T);
}
```

```
void FreeConv::solveAtThisTimeLevel () {
  flow.solveAtThisTimeLevel ();  // compute NavierStokes::v and p
  heat.solveAtThisTimeLevel ();  // compute HeatTrans::T
}
```

The `rebind` function sets the handle (pointer) to point to the argument. Two simulators like `NavierStokes` and `HeatTrans` will typically generate menu items with the same names (e.g., for linear solver parameters), thus requiring the menu prefixes "heat" and "flow" to provide unique menu items. Moreover, `FreeConv` must be a friend of the two other classes if `v` and `T` are not public variables. If `NavierStokes` and `HeatTrans` are library classes which cannot be modified, one simply derives new subclasses of these with a suitable interface such that the class `FreeConv` can rebind the handles to each other. Similarly, if the `T` is missing in class `NavierStokes`, one can derive a subclass containing `T` and a new version of `integrands` where the additional term is included. Deriving a subclass instead of extending an existing class has the advantage that the old code is still intact for purposes of testing. We believe that the ideas of OOP in this section may play an important in implementation and verification procedures for simulators solving systems of PDEs.

# 11 Conclusions

The Diffpack software for PDEs has been outlined. A particular feature of this software, compared to some other similar packages, is its flexibility with respect to numerical methods and applications. This flexibility is closely related to the object-oriented implementation of the numerical methods, where mathematical abstractions are heavily exploited, while hiding the technical details. Finding the proper abstractions for OOP often demands considerable effort to obtain a unified view on a set of numerical techniques.

The BasicTools and LaTools libraries may be useful for numerical programming in general. The other libraries in Diffpack support the numerical solution of PDEs. Through an example we have presented a high-level design of simulators for PDEs and demonstrated that a general finite element solver for a linear scalar stationary PDE requires only a few lines of code. Extensions to time dependency and non-linearities demand a few additional lines.

Using OOP, the data and code can naturally be shared among different versions of a simulator. More precisely, the simulator is a data collection with a corresponding set of actions. Modifications of a simulator can be performed by deriving a new simulator from an existing one, thus inheriting all the data and actions from the parent simulator. Additional data can be included, and the actions that differ are redefined. A more advanced simulator can therefore be created by programming only the *differences* from an existing simulator. The code of the parent simulator is not modified. A complicated simulator may hence consist of a sequence of separate modules, where each module is an executable simulator for the problem. This gives important flexibility, efficiency and reliability in combining different implementations.

Moreover, it is easy to combine several existing simulators to solve a system of PDEs in sequence at each time level. The important gain, besides negligible implementation effort, is that each PDE in the system is solved by an externally tested module which is not modified when it is incorporated into the new code. At least this is an important advantage when establishing a rough prototype simulator for a system of PDEs.

The very simplified examples presented in this chapter have been extended with success to more realistic cases involving PDEs from advanced mathematical models in continuum mechanics. Examples of this can be found in [11].

# References


[1] E. Andreassen, E. Gundersen, E. L. Hinrichsen and H. P. Langtangen: Numerical simulation of melt spinning of polymer fibers. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 195–212, Birkhäuser, 1997.

[2] E. Arge, A. M. Bruaset, P. B. Calvin, J. F. Kanney, H. P. Langtangen and C. T. Miller. On the computational efficiency of C++ programs. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 91–118, Birkhäuser, 1997.

[3] E. Arge, A. M. Bruaset, and H. P. Langtangen. Object-oriented numerics. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 7–26, Birkhäuser, 1997.

[4] E. Arge and Ø. Hjelle. Software tools for modeling scattered data. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 45–60, Birkhäuser, 1997.

[5] A. M. Bruaset. Krylov subspace iterations for sparse linear systems. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 255–280, Birkhäuser, 1997.

[6] A. M. Bruaset and H. P. Langtangen: Object-Oriented Design of Preconditioned Iterative Methods in Diffpack. To appear in *ACM Transactions on Mathematical Software*, 1996.

[7] A. M. Bruaset, and H. P. Langtangen. Basic tools for linear algebra. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 27–44, Birkhäuser, 1997.

[8] A. M. Bruaset, H. P. Langtangen and G. Zumbusch. Domain decomposition and multilevel methods in Diffpack. Report STF42 A96017, SINTEF Applied Mathematics, 1996. (Submitted for publication, URL: `http://www.oslo.sintef.no/diffpack/reports`).

[9] D. M. Butler, W. E. Mason and C. H. Tong: An Object-Oriented Domain Analysis of Partial Differential Equations. In [23], 1994.

[10] J. O. Coplien: *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[11] The Diffpack WWW home page.
(URL: `http://www.oslo.sintef.no/avd/33/3340/diffpack`).

[12] K. G. Frøysa. *Modelling and Simulation of 3D Flow with Gravity Forces in Porous Media.* PhD Thesis, Department of Mathematics, University of Bergen, 1995.

[13] E. Gundersen and H. P. Langtangen: Evaluation of some finite element methods for two-phase porous media flow. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 213–234, Birkhäuser, 1997.

[14] E. Haug and H. P. Langtangen. Basic equations in Eulerian continuum mechanics. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 121–156, Birkhäuser, 1997.

[15] E. Haug, T. Rusten and H. Thevik: A Mathematical Model of Macrosegregation Formation in Binary Alloy Solidification. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. , Birkhäuser, 1997.

[16] M. Haveraaen, V. Madsen and H. Munthe-Kaas: Algebraic Programming Technology for Partial Differential Equations, In A. Maus et al. (ed.): *Norsk Informatikk Konferanse '92, Tapir, Trondheim Norway*, pp. 55-68, 1992.

[17] B. Joe. GEOMPACK – a software package for the generation of meshes using geometric algorithms, *Adv. Eng. Software*, 13, pp. 325-331, 1991.

[18] D. S. Kershaw, M. K. Prasad and M. J. Shaw: Object-Oriented Development of a Three Dimensional, Unstructured Grid Shock Hydrodynamics Model. In [23], 1994.

[19] H. P. Langtangen: Efficient Finite Element Solution of the Linear Wave Equation in Diffpack. Report STF33 A94056, SINTEF Applied Mathematics, Oslo, 1994.
(URL: `http://www.oslo.sintef.no/diffpack/reports`).

[20] R. I. Mackie: Object Oriented Programming of the Finite Element Method, *Int. J. Num. Meth. Engng.*, (425-436), 1992.

[21] E. Mo, T. Rusten and H. Thevik: Computation of Macrosegregation due to Solidification Shrinkage. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 177–194, Birkhäuser, 1997.

[22] OON-SKI '93, *Proceedings of the First Annual Object-Oriented Numerics Conference at Sunriver, Oregon*, 1993.

[23] OON-SKI '94, *Proceedings of the Second Annual Object-Oriented Numerics Conference at Sunriver, Oregon*, 1994.

[24] H. Osnes. *Stochastic Analysis of Groundwater Flow*. PhD Thesis, Department of Mathematics, University of Oslo, 1996.

[25] The PETSc Package WWW home page.
(URL: `http://www.mcs.anl.gov/petsc/petsc.html`).

[26] U. Rüde: Mathematical and Computational Techniques for Multilevel Adaptive Methods. *Frontiers in Applied Mathematics, vol. 13*, SIAM, 1993.

[27] B. Smith, P. Bjørstad and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[28] P. Vankeirsbilck and G. Nelissen. ELEMD: An Object-Oriented Software System for Grid Elements with Multiple Discretizations for Solving PDEs. In [23], 1994.

[29] P. Vankeirsbilck and D. Vandevoorde: Object-Oriented Development of Unstructured Finite Volume Solvers for Hyperbolic Conservation Laws. In [22], 1993.

[30] J. C. Wiley: An Object-Oriented Decomposition of the Adaptive-hp Finite Element Method. In [23], 1994.

[31] D. W. Yergeau, R. W. Dutton and R. J. G. Goossens: A General OO-PDE Solver for TCAD Applications. In [23], 1994.

[32] G. W. Zeglinski and R. S. Han. Object oriented matrix classes for use in a finite element code using C++. *Int. J. Numer. Meth. Engrg.*, 37:3921–3937, 1994.

[33] T. Zimmermann, Y. Dubois-P'elerin and P. Bomme: Object-oriented finite element programming: I. Governing principles, *Comp. Meth. Appl. Mech. Eng.*, 98:291-303, 1992.

[34] Y. Dubois-P'elerin, T. Zimmermann and P. Bomme: Object-oriented finite element programming: II. A prototype program in Smalltalk, *Comp. Meth. Appl. Mech. Eng.*, 98:361-397, 1992.

[35] G. Zumbusch. Multigrid Methods in Diffpack. Report STF42 A96016, SINTEF Applied Mathematics, Oslo, 1996.
(URL: `http://www.oslo.sintef.no/diffpack/reports`).