Increasing the Efficiency and Reliability of Software Development for Systems of PDEs[*]

Are Magnus Bruaset[†]
Erik Jarl Holm[‡]
Hans Petter Langtangen[§]

September 7, 1999

Abstract

In this chapter we address an object-oriented strategy for the development of software solving systems of partial differential equations. The proposed development strategy encourages heavy reuse of modules capable of solving the involved subproblems. Using class inheritance for successive refinement of the involved solvers, the complexity of the overall model is increased stepwise, layer by layer. In addition to the obvious advantage of code reuse and modular testing, this approach allows the developer to pull the pieces apart at any time for individual verification.

1 Introduction

The development of large codes for scientific computing is known to be a comprehensive and time consuming process. Moreover, large stand-alone FORTRAN codes dominate the field of scientific computing. Long-term evolvement of such codes is usually an error-prone and expensive process, unless the original software is carefully designed for future extensions. Turning to the field of computer science, years of experience indicate that human efficiency and software reliability can be significantly improved by a modular design that encourages reuse of code. This is also the basic principle underlying the ongoing development of *Problem Solving Environments* in various branches of scientific computing, see [16] and references therein.

A wide range of phenomena in science and technology is modeled by systems of partial differential equations (PDEs). The numerical solution of such systems

[*]This paper was originally published in *Modern Software Tools for Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (eds.), pages 247–268. Birkhäuser, 1997.

[†]SINTEF Applied Mathematics, P.O. Box 124 Blindern, N-0314 Oslo, Norway. Email: `Are.Magnus.Bruaset@math.sintef.no`.

[‡]Institute for Energy Technology, N-2007 Kjeller, Norway. Email: `erikh@ife.no`.

[§]Dept. of Mathematics, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway. Email: `hpl@math.uio.no`.

is a demanding task and has therefore been one of the major research activities in scientific computing. The purpose of this chapter is to show how modular design and code reuse can be applied to the development of software for the numerical solution of systems of PDEs. Related mathematical models for a class of physical phenomena often exhibit a common basic structure, e.g., the fundamental differential operators in a system of PDEs might be the same. This property advocates the development of general modules that can represent the basic structure of a certain type of equations, preferably in a way that makes it easy to specify the details that differ from application to application. In this type of software environment, one can utilize previously debugged modules and thereby achieve increased software reliability and higher human efficiency in the coding process. Although the basic idea is simple and attractive, the design and implementation are far from straightforward. However, in this chapter we will address a particular approach to such software designs based on experiences with an implementation in Diffpack [5, 7]. It should be mentioned that there are also other software libraries for PDEs available, e.g. Cogito [18], ELEMD [14], Kaskade [4], PETSc[2, 12], and FEMLAB [9].

Initially, the software design discussed in this chapter was motivated by the development of a simulator for a particular type of plastic forming process. In this process, hot polymer is injected in a thin gap between flat plates and cooled down. This is a free boundary value problem involving mass, momentum and energy balance equations, coupled with complicated constitutive relations for modeling phase changes and a generalized viscosity of the polymer. Furthermore, three phases are involved; polymer, air and solid flow obstacles, see Section 2. Writing a stand-alone piece of software that solves such a system of PDEs may easily turn into a programmer's nightmare, due to the complexity of the numerical model and the danger of generating erroneous code. From this pessimistic (and realistic) concern, it seems reasonable that the reliability of the final solver should be based on a series of step-wise refinements that can be verified individually. Furthermore, if the final solver is an assembly of modules that are known to be safe, it should be possible to pull the modules apart again and repeat independent verifications. A general rule of thumb is to avoid the copying of source code and to make extensions without editing files containing already debugged code. Although the quest for reliability has been the driving force for this strategy, practical results show a nice side-effect in a dramatic decrease of the time spent on writing and debugging the software. The proposed strategy is not only applicable to this particular simulator. It can also be successfully applied in other physical problems, which we will point out later.

Modularity and code reuse can be achieved by using traditional implementation techniques in FORTRAN. However, this requires very careful and complicated considerations. Object-oriented design and programming techniques offer a much easier and more efficient methodology for obtaining the goals. The implementation can still be expressed in FORTRAN 77 or C, but it is obvious that this process is much more efficient when using a language that provides genuine support for object-oriented techniques. FORTRAN 90 [13] contains many useful constructs that appear as building blocks in object-oriented designs, but

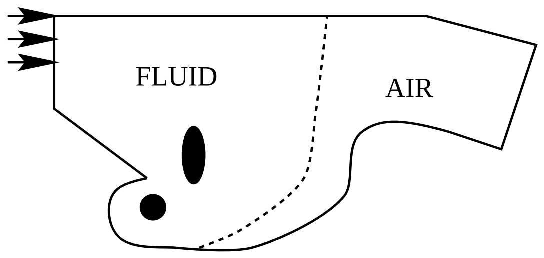

Figure 1: Sketch of the flow domain for a plastic forming process as seen from above. The dashed line is the polymer-air interface, whereas the solid obstacles are displayed as black spots.

unfortunately lacks tools for very important principles such as polymorphism and dynamic binding [17]. Thus, C++ seems to be a reasonable alternative that offers the most important object-oriented constructs along with satisfactory computational efficiency [1]. In the present chapter we will use the C++ terminology and show some C++ code segments. Readers not familiar with C++ should consult standard textbooks [3, 17].

2 A Plastic Forming Process

Slim plastic products, like computer keyboards and plastic bags, are usually formed by injecting hot polymer between two piecewise flat, cold plates, see Figure 1. The fluid is non-Newtonian, that is, the effective viscosity depends on the fluid motion. In the present problem we will use a generalized Newtonian viscosity model, where the dependency of the viscosity on the motion is given by an explicit formula [11]. The polymer displaces air, which results in a free-boundary problem since the fluid-air interface is unknown. Various solid obstacles can be installed between the plates to achieve the desired geometry of the final plastic product. The flow problem is treated as two-dimensional in the xy-plane. Velocities and other quantities are averaged in the z-direction. The normal velocity vanishes on the boundaries, while inflow and outflow are modeled by point injectors. The heat transfer problem is, however, three-dimensional with convection dominating in the xy-directions and conduction dominating in the z-direction. Note that conduction also takes place in the solid obstacles. Since the fluid can solidify, the present application also involves phase changes. Figure 2 shows one snapshot of a particular simulation for a quite complicated problem.

To simplify the implementation, one can have a single grid and perform flow and heat computations in the polymer, the air and the solid obstacles. If the size of the gap between the two plates is $h(\mathbf{x})$, the obstacles are then modeled as by a very small gap $h(\mathbf{x}) = \epsilon \ll 1$, provided that h is scaled properly. The amount of mass transport through the obstacles is proportional to ϵ and can hence be controlled [15]. This approach to generating a simplified computa-

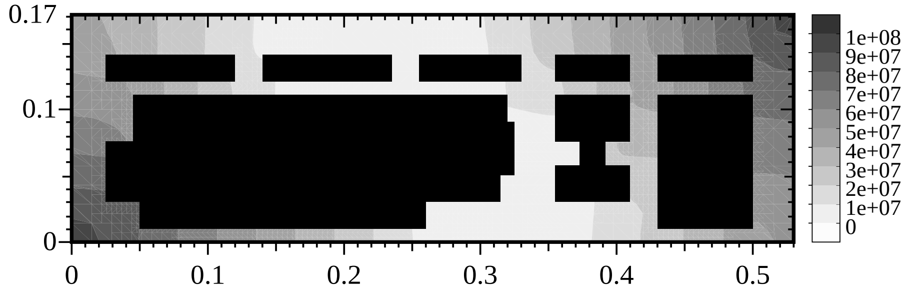

Figure 2: A snapshot of the pressure in a simulation of the injection molding of a computer keyboard. The polymer front can be detected as the isoline $p = 0$.

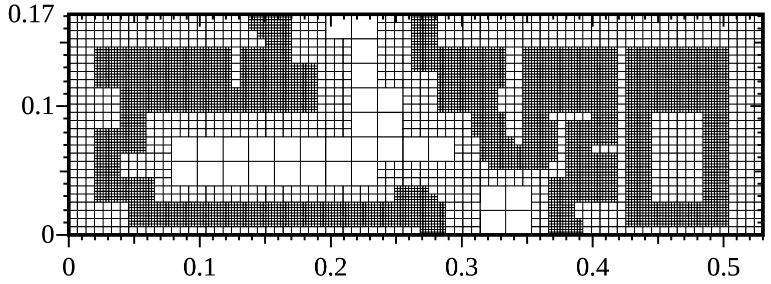

Figure 3: A snapshot of an adaptive grid used in the injection molding computations. The geometry is identical to the one shown in Figure 2.

tional geometry by introducing certain modifications to the PDEs (or to the coefficients contained therein) is often referred to as *domain imbedding* or *the method of fictitious domains*. As shown in [6], the ill-conditioned linear systems arising from this procedure can be efficiently treated by certain preconditioning techniques. In order to increase the accuracy close to the advancing fluid front we have used adaptive grid refinement techniques available through the Diffpack class described in [10]. Due to the object oriented implementation of this adaptive grid class it was included in the simulator by adding just a few statements to the original code. A snapshot of a 2-irregular grid can be seen in Figure 3. As far as the 3D heat transfer problem is concerned, it can under certain circumstances be split into a set of 2D equations, using spectral methods to discretize the equation in the z-direction.

An appropriate mathematical model that allows computations in a single two-dimensional grid, covering polymer, air and obstacles, can be expressed as

$$\nabla \cdot [S(T,p,c,h)\nabla p] = q, \tag{1}$$

$$\frac{\partial c}{\partial t} + \mathbf{v} \cdot \nabla c = 0, \tag{2}$$

$$\rho(c,h)\frac{\partial H^{(i)}(T)}{\partial t} + \rho(c,h)\mathbf{v}\cdot\nabla H^{(i)}(T) = \nabla\cdot\Big(\lambda(c,h)\nabla T(H^{(i)}(T))\Big) + f(\mathbf{v}) \quad (3)$$

for $i = 1, \ldots, m_z$, see [11]. In this system, $p(\mathbf{x},t)$ is the pressure in the polymer or the air, where $\mathbf{x}$ is a 2D spatial point, $c(\mathbf{x},t)$ is an indicator function such that $c > 0$ implies polymer and $c < 0$ implies air. The modeling of the complex shaped front by solving (2) and thereby updating c is called a level set method. The function $h(\mathbf{x})$ reflects the gap between the plates (note that $h = \epsilon \ll 1$ indicates solid obstacles). Moreover, $q(\mathbf{x},t)$ models fluid injection or extraction points (normally through Dirac delta functions), T is the temperature, and $H^{(i)}(\mathbf{x},t)$ is the ith degree of freedom in a spectral expansion of the enthalpy in the third space direction [11]. The number of degrees of freedom, m_z, can usually be small, say 3–5. Furthermore, $\rho(c,h)$ represents the product of the density and heat capacity of polymer. This product varies with the medium (air or solid obstacles). Furthermore, $\lambda(c,h)$ is a heat conduction coefficient. The reason for using the enthalpy as the primary unknown in the energy equation is that phase changes (e.g. the Stefan problem) are then easier to deal with. Note that both c and h are used to identify whether the medium is polymer, air or solid obstacles. The fluid velocity is given by $\mathbf{v} = -\frac{1}{h}S(T,p,c,h)\nabla p$, where $S(T,p,c,h)$ is a known function that models the effective viscosity of the fluids. It is constant in the air ($c < 0$) and a nonlinear function of T, h, and ∇p in the polymer ($c > 0$). The source term $f(\mathbf{v})$ models internal heat generation due to friction (dissipation). The primary unknowns in the PDE system (1)-(3) are the two-dimensional quantities p, c and T. The rest of the quantities are either known or can be directly derived from the primary unknowns.

The core part of the software development techniques to be outlined does not depend on details of the discretization methods. It is therefore sufficient to assume that the initial-boundary value problem is discretized in time by some technique, such that the primary unknowns can be obtained at the same discrete time levels. The discretization of (1)-(3) in space is based on finite elements in our particular implementation.

3 The Basic Ideas

Most PDE systems that arise in solid and fluid mechanics involve only a few differential operators, typically $\partial/\partial t$, or $\partial/\partial t + \mathbf{v}\cdot\nabla$, and ∇^2 or $\nabla\cdot K\nabla$. If each operator could exist as an abstraction, it would be user-friendly to write the code for a system of PDEs as an assembly of various operators. However, the use of operators directly as C++ classes tend to decrease the computational efficiency dramatically. In order to achieve high efficiency, several terms in the equations must be treated simultaneously in the code. This contradicts to some extent our initial requirement of modularity. Furthermore, one might already have an existing high-quality FORTRAN code capable of solving parts of the PDE system. It is difficult to incorporate such software on the operator level.

In our experience, the *solver* for a single PDE (or a system of PDEs) has proved to be a useful abstraction in object-oriented implementations. We will therefore restrict the software design to numerical methods that are based on a particular type of so called *operator splitting* for solving systems of PDEs. Let our system of PDEs be written compactly in the form

$$\begin{aligned}\mathcal{L}_M(p;T,c) &= 0, & (4)\\ \mathcal{L}_E(T;p,c) &= 0, & (5)\\ \mathcal{L}_F(c;p,T) &= 0. & (6)\end{aligned}$$

These three equations correspond to (1), (2) and (3), respectively. The simultaneous action of $\mathcal{L}_M$, $\mathcal{L}_E$ and $\mathcal{L}_F$ is now the total differential operator in the PDE system. The idea is to split this operator and treat each of its components in sequence. For each equation, two of the primary unknowns are treated as "known" such that (4) becomes an equation for p, (6) an equation for finding T and (5) is used to update c. We use the most recent updates for the "known" variables in an equation, that is, the solution algorithm follows a typical Gauss-Seidel strategy. Our solution procedure can be expressed more precisely as follows.

```
Use values from the previous time level as start values $T^{(0)}$,
$c^{(0)}$ and $p^{(0)}$.

For $k = 1, 2, \ldots$ until convergence:
   solve $\mathcal{L}_M(p^{(k)};T^{(k-1)},c^{(k-1)}) = 0$ with respect to $p^{(k)}$
   solve $\mathcal{L}_E(T^{(k)};p^{(k)},c^{(k-1)}) = 0$ with respect to $T^{(k)}$
   solve $\mathcal{L}_F(c^{(k)};p^{(k)},T^{(k)}) = 0$ with respect to $c^{(k)}$
```

When, and if, the loop is terminated, $T^{(k)}$, $c^{(k)}$ and $p^{(k)}$ are the values of T, c and p at the new time level.

Many systems of PDEs have strong couplings and nonlinearities such that this Gauss-Seidel approach may lead to convergence problems. Sometimes improved convergence can be achieved by solving a subset of the equations simultaneously. For example, we could think of replacing the $\mathcal{L}_M = 0$ and $\mathcal{L}_E = 0$ equations by a coupled subsystem $\mathcal{L}_C(p,T;c) = 0$, where p and T are solved for by an approach of implicit nature, e.g., Newton's method. The Gauss-Seidel technique can then be applied to the system of $\mathcal{L}_C(p,T;c) = 0$ and $\mathcal{L}_F(c;p,T) = 0$. If it is required to solve the whole system by a Newton type of algorithm, our basic design idea actually fails. However, it appears that only a few modifications are necessary in order to treat both Gauss-Seidel and Newton solution strategies. These modifications will be presented in Section 6.

The basic idea is now to develop a C++ class for the solution of each PDE (or subsystem of PDEs). The data type used to represent the coefficients should be quite flexible in order to maximize reuse of the PDE solver. Each type of equation will be modeled by a generic base class, whereas a specific equation can be implemented as a subclass. In other words, the common features of the solver are collected in a base class. A subclass inherits all the code and functionality of a base class (cf. [17]), such that the specialization of the subclass consists in programming only the *differences* between the generic and the specialized problem. In C++ this is done by including new data members plus redefining some virtual functions. The associated source code is normally very small.

A solver for the system involving p, T and c can then consist of three specialized C++ objects for the three PDEs, while the implementation of the Gauss-Seidel algorithm is placed on top of these abstractions. In each pass of the Gauss-Seidel algorithm, each PDE object is asked to solve its subproblem at the present time level. Such a design can reuse existing PDE solvers and, by a minimum of coding, tailor generic solvers to specific equations. This is obviously a desired and natural software development strategy. What is new here, is that an object-oriented implementation in C++ makes it very easy, at any time in the development process, to pull the objects apart and verify each PDE solver *without touching the code of these solvers*. This is an important feature that increases the reliability of the development process, and has clearly practical advantages during debugging and verification.

Since each PDE solver is a unit in this set-up, it is easy to use e.g. FORTRAN codes for a particular solver. To ease the integration of such a solver with the suggested C++ design, it may be a good idea to "wrap" a C++ class around the FORTRAN code. In this way the FORTRAN code will appear in the design as if it was a native C++ class. Mixing finite difference and finite element methods for the various equations is also easy when each PDE solver is a stand-alone class. The more C++ oriented details of the design will be presented in Section 5 using typical constructions from the Diffpack libraries.

4 Diffpack

Diffpack is a software system for rapid development of solvers for partial differential equations [7, 5]. Both finite difference and finite element methods can be programmed in Diffpack simulators, but at present only the latter type of methods takes full advantage of the most sophisticated software abstractions. The Diffpack libraries are coded in C++ and are based on object-oriented design and programming techniques. The object-oriented philosophy makes it easy to develop a simulator for a PDE by simply combining building blocks for vectors, matrices, linear systems, linear solvers and preconditioners, nonlinear solvers, finite element/difference grids with corresponding fields etc. The design of Diffpack is layered in the sense that high-level objects are built on top of objects from lower levels. At the most primitive level, the most important data structure is that of efficient C-style arrays, while CPU-intensive code segments employ simple FORTRAN-like constructs that are easily recognized by the optimization modules present in modern compilers. The layered design enables a high level of flexibility that allows the developer to extend and optimize Diffpack objects. An overview of the functionality in Diffpack is provided in [5].

A typical Diffpack simulator for solving a single PDE, e.g. using the finite element method [8], is implemented as a C++ class. Using the equation

$$-\nabla \cdot (K\nabla p) = 0$$

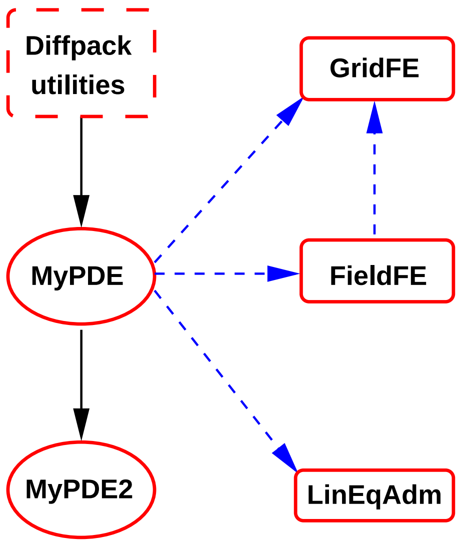

Figure 4: The simulator class `MyPDE` and its main members (finite element grid, associated field and linear system toolkit). Solid arrows indicate class derivation ("is-a" relationship) while dashed arrows represent pointers or references ("has-a" relationship). As indicated, `MyPDE` can itself serve as basis for new simulators, e.g., `MyPDE2`.

as an example, the minimalistic outline of such a class is given below. For our purpose the base class information is irrelevant. Thus, in the examples the specification of the base class(es) is omitted (`public ...`) in order to focus on the main topic. Usually, a simulator like `MyPDE` below inherits some Diffpack utilities to make, e.g., finite element programming easier, see Figure 4.

```
class MyPDE : public ... {
 protected:
  GridFE*   grid;    // finite element grid
  FieldFE*  p;       // finite element field over the grid
  LinEqAdm* lineq;   // interface to linear systems and solvers
 public:
  // definition of integrands in the weak form:
  virtual void integrands (ElmMatVec& elmat, FiniteElement& fe);
  // definition of coeff. in PDE:
  virtual real K (FiniteElement& fe);
  virtual void init ();          // allocate & init grid, p, etc.
  virtual void solveProblem ();  // main driving function
};
```

In an actual Diffpack implementation one applies smart pointers with reference counting (called *handles* in Diffpack terms) instead of primitive C pointers like `GridFE*`. The classes `GridFE`, `FieldFE`, `LinEqAdm`, `ElmMatVec` and `FiniteElement` are available in the Diffpack libraries [5]. Initialization and input to such classes are conveniently handled by the Diffpack menu system.

For example, the user can through the lineq object get a graphical user interface for choosing matrix formats, the linear solver and associated parameters, as well as preconditioners.

The GridFE class is a standard finite element grid containing nodal coordinates, element connectivity and boundary information. The FieldFE class represents a continuous scalar finite element field over a grid and contains a smart pointer to a GridFE object, a vector of nodal values and information on finite elements. The FieldFE object offers of course interpolation of a finite element field at an arbitrary spatial point. For efficiency reasons FieldFE's interpolation function can make use of precomputed basis functions. It is the responsibility of the FiniteElement class to compute and store the basis functions at a point in an element, their derivatives, the Jacobian of the mapping between local and global coordinates, as well as numerical integration points and weights. The real type represents the float or double type in C/C++. Finally, the ElmMatVec class represents the elemental matrix and vector in addition to a mapping from elemental to global degrees of freedom.

It is also worth mentioning that any Diffpack simulator can easily be extended with automatic report generation and other useful features for experimental scientific computing.

Since we use the finite element method when computing p, the p field in class MyPDE is conveniently represented by a FieldFE object. We have not indicated the data structures used to represent the K field, only the computation of K in terms of a virtual function was presented. The field concept naturally leads to object-oriented design and implementation in terms of a class hierarchy. A base class Field can be introduced with (pure) virtual interpolation functions of two types; one that takes a general, global point as argument and one that takes a FiniteElement object as argument to increase the efficiency of finite element solvers. Subclasses can represent constant fields, explicit formulas for functions, finite difference fields over uniform lattice grids, finite element fields, and fields over subdomains in finite element meshes. With the field hierarchy in mind, we can easily suggest a data structure[1] K_ for K in class MyPDE that can be represented in terms of a Field* pointer which is bound to a particular field subclass at run-time. For example, if $K \equiv 1$, K_ points to an object optimized for constant fields. Another common choice for K_ might be a field object where the values are given by an explicit formula for K. Using the field abstraction Field* K_ for K in the protected part of class MyPDE, the virtual function K is conveniently implemented as

```
virtual real K (FiniteElement& fe) { return K_->valueFEM(fe); }
```

The valueFEM function performs evaluation of a field inside a finite element, represented by the fe argument. This implementation works regardless of whether K is constant, a function or a precomputed finite element field.

The integrands procedure for evaluating the integrand of the weak form, in our example $K\nabla N_i \cdot \nabla N_j$, at a numerical integration point defined through the

[1]The identifier K_ uses an underscore character (_) to prevent name conflicts with the member function MyPDE::K.

current status of the `FiniteElement` object, needs to sample the K coefficient. This is done by a call to the virtual function `K`. Hence, the `integrands` function is completely general for all types of variable coefficients, but the `K` function restricts the evaluation of K to interpolation of a Diffpack field object. Other more complicated forms of K, e.g. formulas that involve variables from other PDEs, can be implemented by overriding the `K` function in subclasses. This is a fundamental issue in our design of solvers for systems of PDEs.

Some readers will point out that having a virtual function, like `K` here, in the innermost loop of a computationally intensive code decreases the efficiency. In principle, this is true. However, we only need to evaluate K once for each numerical integration point, whereas several arithmetic operations are required to evaluate the contribution to the elemental matrix and vector from the integration point. The overhead in calling `K` is hence negligible. For physically realistic applications, the expressions for the variable coefficients usually involve many operations, and the overhead of a virtual function call is even smaller.

The `solveProblem` function will compute p and, e.g., store the solution for later visualization. For debugging and verification purposes, it is a good idea first to use a simple K, e.g. $K \equiv 1$. This will only affect the initializing function `init`, which is responsible for allocating and initializing (large) data structures for the grid, various fields, and the linear system. The data structure for `K_` is quite general and the evaluation function `K` is even more general.

5 Systems of PDEs

In this section we will explain how PDE components in the equation system can be treated as individual and independent Diffpack simulators like class `MyPDE`. We will use the system (1)-(3) as a concrete example. Let us now consider the transition of the basic software design into working C++ classes.

We first make a generic implementation of the equation (1). Again it is natural to represent the variable coefficient S as a virtual function that can be redefined in subclasses. In the implementation we assume that the problem is generally nonlinear in p through the dependence of S on p. Therefore we also need a virtual companion function for $\partial S/\partial p$, since this quantity is needed in Newton's method for the nonlinear algebraic equations that arise from (1). Moreover, it will be an advantage to introduce a virtual function for the evaluation of T. Note that only the sign of c is really needed in the flow and energy equations. It is therefore convenient to introduce an indicator function `medium` that returns 1 if a point is inside the polymer, 2 if the point is inside the air, and 3 if the point is inside a solid obstacle. Our interface for handling a general variable coefficient $S(T, p, c, h)$ is then

```
class Flow1 : public ... {
 protected:
  GridFE*     grid;
  FieldFE*    p;
  LinEqAdm*   lineq;
  Field*      h;
```

```
   Fields*    v;  // v=-S*grad(p)
  public:
   virtual void integrands (ElmMatVec& elmat, FiniteElement& fe);
   virtual real S      (FiniteElement& fe);
   virtual real dSdp   (FiniteElement& fe);
   virtual real T      (FiniteElement& fe);
   virtual int  medium (FiniteElement& fe);
   virtual void init   (GridFE* g == NULL);  // init pointers
   virtual void solveProblem ();
 };
```

This class is essentially a collection of scalar fields with shared grid data (one scalar entity for each component of the vector field) and a matrix system. Due to efficiency considerations, the argument passed to the functions evaluating the variable coefficients is of type `FiniteElement`, rather than just a point in space. The `init` function can either generate a grid or set `grid` to point to an external grid object `g`. The latter possibility will be important later. The `medium` function needs to access h and c, but in this base class we assume that we have polymer in all points. Hence, it is not necessary for the base class version of `medium` to have access to the c field. The `S` and `dSdp` functions need to access `medium` and, if the problem is nonlinear, also the `p` field. In addition, we have the velocity field **`v`** that is represented in terms of a `Fields` object. The interface shown above is minimalistic; a real-world example will often equip the class with additional data and functions that are not relevant to the design issues covered in this chapter.

In the base class `Flow1` the `integrands` and most other functions can handle the general case of a nonlinear equation $\nabla \cdot S(T,p,c,h)\nabla p = q$, but in the virtual functions above we implement just a simple choice of S for testing purposes. This could be $S = 1$ in the polymer and $S = \epsilon \ll 1$ (corresponding to $h \ll 1$) in the obstacles. The `T` function has no meaning and should return a constant reference temperature for the isothermal case.

From class `Flow1` we can derive a slightly more advanced class `Flow2`. For testing purposes we can assume, e.g., that there is a prescribed c field, but no temperature coupling. The S function could, e.g., be constant in each of the three media. Hence we could have

```
class Flow2 : public Flow1 {
 protected:
  Field* c_;  // pointer to external or internal front field
 public:
  virtual real S      (FiniteElement& fe); // checks medium(fe)
  virtual real dSdp   (FiniteElement& fe)  { return 0; }
  virtual int  medium (FiniteElement& fe); // checks c_ and h
  virtual void init   (GridFE* g == NULL); // call Flow1::init(g),init c_
};
```

Note that class `Flow2` can, e.g., solve a non-physical test problem for the purpose of making a small step towards a more complicated problem. The next natural extension is to couple the flow solver with a simulator that tracks the polymer/air front by solving equation (2). This is easily accomplished by letting c_ point to an external field in a solver class for c. We can extend Flow2 to handle both an internal and an external c field. This is assumed in the following.

We will refer to any numerical solution method for equation (2) as a *front tracker*. A key point is that class Flow2 does not know anything about the existence of a front tracker object. It only uses general field information through its pointer c_. The front tracker is conveniently implemented as a base class Front1 for solving a generic version of equation (2), using a virtual function for evaluating the velocity field $\mathbf{v}$. This function could simply return a constant value in order to enable easy debugging of class Front1. In a derived class, Front2, we can add a pointer v_ to a vector field, and for example let v_ point to the $-S\nabla p$ vector field in Flow1. The virtual function for the velocity evaluation must then be redefined in class Front2 and call an interpolation function in the vector field class. The main ideas from the flow solver design can be applied directly to the front solver classes, so we omit showing details of class Front1 and Front2.

Since the flow and front solvers communicate only in terms of pointers to fields, these solvers are actually completely independent of each other. We need a manager class to administer the field pointers and the solution procedure. This class will be called HeleShawFill (flow between two flat plates is usually referred to as Hele-Shaw flow and here we also have a filling process). For reasons of convenience the Flow2 and Front2 classes can have references to HeleShawFill. This will enable the solvers to access data in other solvers, through the manager. However, the coupling of Flow2 and Front2 to HeleShawFill should only be visible in these classes, not in their respective base classes. The manager class can then look like this:

```
class HeleShawFill : public ... {
  Flow2*   flow;
  Front2*  front;
  GridFE*  grid;               // common grid
  TimePrm* tip;                // time integration parameters
  void timeLoop();             // time integration algorithm
 public:
  void solveProblem ();        // main driving routine
  void init ();                // init couplings between flow and front
};
```

Class HeleShawFill is in charge of making a common grid and of performing the common time stepping. It is not necessary to use the same time step size in different solvers, but it is convenient that the ratios of the step sizes are integers. The grid is transferred to the init functions of the flow and front solvers. If it is not transferred, the init functions in flow and front can make their own grids. The init function in HeleShawFill must create the common grid and time integration parameters, call the init functions in front and flow, and perform the pointer connections between the simulators. A sketch of the classes and the relations we have discussed so far appears in Figure 5.

The ideas that have been described so far can easily be applied to incorporate a heat transfer simulator as well. Consider for simplicity the equation (3) with

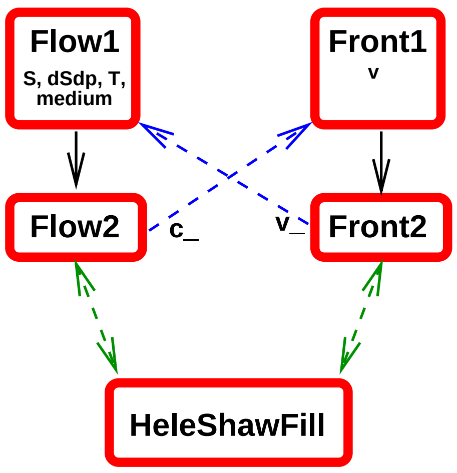

Figure 5: Sketch of the coupling of a flow solver and a front tracker for isothermal Hele-Shaw flow. Solid arrows indicate class derivation ("is-a" relationship) while dashed arrows represent pointers or references ("has-a" relationship). In the base classes `Flow1` and `Front1` we have listed some of the most important virtual functions.

$m_z = 1$. This equation has general variable coefficients ρ, $\mathbf{v}$, λ and f. We represent these by virtual functions. As in class `Flow1`, the coefficients will depend on the medium (polymer, air, obstacle) and it is convenient to have the same indicator function `medium`. The base class `Heat1` for solving (3), for the case $m_z = 1$, implements the equation in a generic way in the `integrands` function, but provides trivial versions of the virtual functions for the variable coefficients. For example, we can let $\mathbf{v} = 0$, $\rho = \lambda = 1$ and assume only one medium for easy verification of the implementation.

A subclass `Heat2` can be derived from `Heat1` where we assume a varying velocity field, represented by a pointer `Fields* v`. We establish the communication with the manager `HeleShawFill` and use this communication line such that the `medium` function in `Heat2` simply calls the `medium` function in `Flow2`. An alternative is to build a local `Heat2::medium` function that uses pointers to c and h in `Flow1` and `Front1`, respectively, but in our opinion this is a less elegant (and less robust) design, since it actually implies a copy of existing (debugged) code in class `Flow1`.

The flow solver must also be extended to handle temperature effects. This is easily accomplished by deriving a class `Flow3` from `Flow2`, where we have a `Field*` pointer `T_` to some temperature field. This pointer will be set to the correct address in `Heat1` by the manager. The virtual `T` function is redefined in `Flow3` and makes use of the `T_` pointer. The manager must then be extended to have three solvers: `Flow3`, `Front2` and `Heat2`. A sketch of the new class structure is displayed in Figure 6.

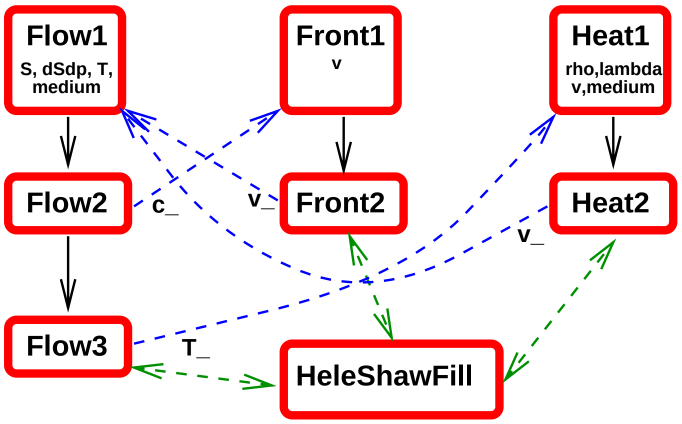

Figure 6: Sketch of the coupling of a flow solver, a front tracker and a heat transfer solver. Solid arrows indicate class derivation ("is-a" relationship) while dashed arrows represent pointers or references ("has-a" relationship). In the base classes we have listed some of the most important virtual functions.

The description of the classes and their relations is perhaps rather technical and C++ oriented. Nevertheless, the technical information will help readers with C++ experience to get a deeper insight into our ideas.

Remarks The proposed design of a solver for a complicated system of PDEs allows the programmer to develop the code in mathematically natural steps, and verify the implementation of each step. At any time in the process, the state of any previous step can easily be recovered for reliability tests. Inheritance is the key needed to avoid any editing of already debugged code; extensions and modifications will always appear in subclasses. Another key point is that the generic base class solvers, `Flow1`, `Front1` and `Heat1`, have no knowledge of each other, or of any other solvers. Hence, they can be reused in a wide range of applications. The base class solvers presented above are tightly connected to the equations (1)-(3), but one can think of more general Poisson, advection and energy equation solvers with a greater potential for reuse. The development steps could then easily be more refined. For example, there could be additional layers, (`Flow3`, `Flow4` and `Flow5`) before a full coupling to other equations and the manager is performed. Based on these ideas, we see the possibilities of creating very flexible solvers, within a specific application area, that can be easily combined. This will be one of the future directions of the Diffpack Project.

6 Extensions of the Concept

A drawback of the previously proposed design of solvers for systems of PDEs is that the equations must be solved in sequence at each time level. As already pointed out, this Gauss-Seidel type of approach may face convergence problems. It is therefore of interest to investigate possible extensions of the design that allow Newton-like methods to be applied.

The basic problem with the Newton iteration and similar methods is that each equation can no longer be responsible for defining its own discrete problem and producing the solution of one of the primary unknowns, given a value of the other unknowns. The manager class must instead build a common linear system, where all the primary unknowns are present. The coupling among p, T and c, e.g. in the flow equation, needs to be considered. In other words, the base class solvers must see more of the interface to the other equations and their corresponding weak forms. The approaches that we discuss below are restricted to finite element solvers. Some of the flexibility of the Gauss-Seidel strategy, where the details of the numerical solution method in each solver is completely hidden, is then unfortunately lost.

The simplest approach will be to consider solution methods for nonlinear systems of equations that only need to evaluate the residual of the equations, not the Jacobian or other matrices. As an alternative to the `integrands` function we can simply provide a similar function that evaluates the integrands of the residual vector of the PDE at an integration point in an element, using the available values from the previous iteration for the unknowns. The elemental vectors from each PDE solver can then be appended to each other to form the composite elemental vector for the whole system in the manager class. The manager must assemble all these composite vectors into the global residual vector for the complete nonlinear system.

Considering full Newton methods, the approach in the previous paragraph can be extended. Besides the vector containing the residuals of the PDEs, we need the Jacobian. The flow solver (`Flow1`) must hence have a function similar to `integrands` where the contributions from (1) to the residual and the Jacobi matrix at the elemental level are computed. This function must compute the coupling between p and itself, p and T, as well as between p and c. Assuming n_e unknowns for p, T and c in an element, the elemental contribution from `Flow1` to the Jacobi matrix for the full system is a rectangular $3n_e \times n_e$ matrix. The `Flow1` solver can still be made quite generic, although the coupling to other equations is now evident even in this base class. For example, derivatives of the PDE with respect to other primary unknowns can be accessed via virtual functions like `dSdp`. That is, we must provide $\partial S/\partial p$, $\partial S/\partial T$ and $\partial S/\partial c$. The latter can in the present physical application be complicated to evaluate. Hence, it may be natural to treat only the flow and heat equations as an implicit system, solved by Newton's method, and use the Gauss-Seidel approach for coupling the heat/flow solver and the front tracker.

The `Flow1` class can of course still be used as a stand-alone solver since the functionality we describe here is only an extension (and no modification) of the previously presented version of the class. A clean way of implementing the functionality for Newton-like methods is to derive a subclass which then can act as base class for what we previously have referred to as class `Flow2`. Another approach would be to use inexact Newton methods, where the partial derivatives needed for the computation of the Jacobian are approximated by finite differences. In this case, the requirements posed on the solver for each PDE would be similar to what we described for nonlinear solvers that only utilize the residual vector of the nonlinear system.

7 Other Applications

The design approach in the previous sections has been tailored to a particular physical application. Nevertheless, it is obvious that this approach and the underlying ideas are of a general nature. Here we will outline some examples of other applications where the methodology can be or has been used.

Many porous media flow problems fit into the framework of equations (1)-(3). The simplest single-phase problems have a linear version of (1) together with (2) for tracking the concentration of, e.g., contaminants. Temperature effects are often of less importance in groundwater flow or hydrocarbon recovery. Seepage flow with a free surface can be solved by a linear version of (1) combined with (2) for tracking the water-air interface. Two-phase porous media flow can be formulated as (1) and a nonlinear version of (2), but without any coupling to a T equation. Finally, more general multi-phase, multi-component porous media flow models usually consist of an equation of type (1), possibly with a time derivative, and a set of nonlinear (almost) hyperbolic equations that are generalizations of equation (2). If temperature effects are important, the equations must be coupled with an energy equation, like (3) with $m_z = 1$.

Within computational fluid dynamics, there are obvious advantages of a flexible simulation environment of the type proposed herein. However, the equations have then a slightly different structure from (1)-(3). If we consider the fairly general problem of free thermal convection in fluids, the equations read

$$\begin{aligned}
\nabla \cdot \mathbf{v} &= 0, &(7)\\
\varrho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) &= -\nabla p + \mu\nabla^2\mathbf{v} + \alpha(T - T_0)\mathbf{g}, &(8)\\
\varrho C_p\left(\frac{\partial T}{\partial t} + \mathbf{v}\cdot\nabla T\right) &= \kappa\nabla^2 T. &(9)
\end{aligned}$$

The quantity $\mathbf{v}(\mathbf{x}, t)$ is the fluid velocity, $T(\mathbf{x}, t)$ is the temperature, ϱ is the fluid density, α is a prescribed coefficient related to density changes due to temperature perturbations $T - T_0$ from a reference temperature T_0, $\mathbf{g}$ denotes the acceleration due to gravity, C_p reflects the heat capacity, and κ denotes the heat conduction coefficient in the fluid. In these equations, the Boussinesq approximation is utilized, that is, the density ϱ is treated as a constant, except in the buoyancy term in the momentum equation (8). It is not possible to eliminate the velocities and thereby obtain a combined mass-momentum equation like (1). Instead, equations (7) and (8) must be solved simultaneously. When formulating a Gauss-Seidel approach for (7)-(9), one should treat (7)-(8) as a single unit. That is, there will typically be one flow class hierarchy for (7)-(8) and one heat class hierarchy for (9). Two-phase problems with sharp interfaces can be treated by level set methods, and then the interface tracking can be based on an auxiliary equation identical to (2). This will add a front solver hierarchy as explained for the system (1)-(3).

From the discussion above, a natural direction of development is to create *application environments*. This term refers to having available a pool of abstractions that can be employed in a certain application regime. These abstractions would take the form of classes representing different solvers, mathematical models, etc., and would adopt a standard for inter-object communication based on the concepts presented in this chapter. In this environment, the user could instantiate a large number of simulators for the coupled problem, just by trying different combinations of solvers and models for the subproblems. The application areas of porous media flow and computational fluid dynamics are attractive candidates for software development along these lines, since they both can take advantage of an experimental approach to the present diversity of computational methods and model formulations.

8 Another Application of the Flexible Design

The representation of a system of PDEs in terms of a class hierarchy for each PDE is obviously advantageous when implementing flexible solution methods. However, the modular design has also other advantages that we will outline in the present section.

Consider a numerical solution method for a single PDE, like equation (1), which is based on the following local-global approach. Initially the domain is partitioned into m coarse grid elements. This coarse grid is denoted by Ω_Δ. We then define a series of locally refined grids, Ω_e, where Ω_e includes Ω_Δ, where coarse mesh element no. e and its adjacent neighbors are refined. Figure 7 shows Ω_e for $e = 1, 2, 3$, in a particular example. To obtain an improved solution p of (1), one can combine computations done on Ω_Δ and Ω_e, $e = 1, \ldots, m$. Note that Ω_e covers the whole domain (contrary to ordinary domain decomposition methods where local refinements are also local problems – the idea of Ω_e is that the problem with local refinements inherits the proper physical boundary conditions). Various methods are possible for obtaining the fine grid solution, and we will focus on a strategy [10] where one essentially solves

$$a(p, \phi) = b(\phi) + q(\phi; p_1, \ldots, p_m). \tag{10}$$

Here, $a(\cdot, \cdot)$ is the bilinear form of the problem, and $b(\cdot)$ and $q(\cdot; \cdots)$ are contributions to the right hand side, ϕ is some test function, and p_e is the solution corresponding to a grid Ω_e.

The solution procedure will now consist of an ordinary finite element assembly process for (10), but where a global problem over Ω_e must be solved when visiting each element in the element-by-element assembly. In other words, there will be m global finite element computations inside the ordinary assembly process. Basically, this is easily accomplished by deriving a solver subclass, say

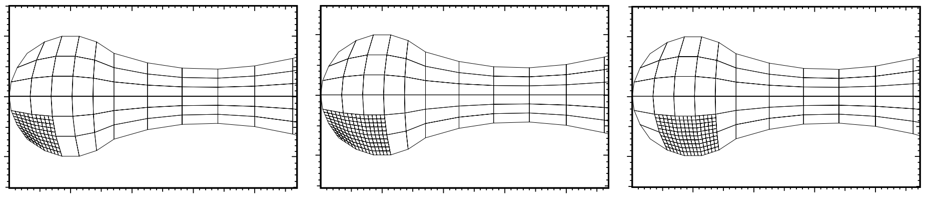

Figure 7: Example on locally refined grids Ω_e in three different cases. Removing the refinements yields a coarse grid Ω_Δ.

`Flow3` from `Flow1` and including a `Flow1` solver as data member. The assembly process is a virtual function in Diffpack PDE solvers, so `Flow3` will need to redefine the default version of that function. In the redefined function, one can for each element e in Ω_Δ call the `Flow1` solver to compute a solution over the refined Ω_e grid, utilize the solution in the assembly process over Ω_Δ, throw away the solution and then proceed with the next coarse grid element. In practice, the data member in class `Flow3` will not be a pure `Flow1` object, but rather a new subclass of `Flow1`, say `Flow4`, where we can make convenient specializations for the problem over Ω_e.

When implementing the `Flow3` and `Flow4` solvers, it turned out that the associated extra code was very small, thus resulting in short development time and high reliability due to maximal reuse of code. The implementation of this numerical algorithm in a FORTRAN 77 program would require substantial modifications and associated debugging. Moreover, the readability of the source code related to the object-oriented C++ approach is much better. We believe that building PDE solvers as modular C++ objects in the way we have sketched above, opens up new possibilities for safe and fast combination of such objects to create novel software for testing new ideas regarding numerical algorithms or physical models. In other words, this modern software approach might play an important role in experimental scientific computing.

9 Concluding Remarks

Today, object-oriented design and implementation are beginning to demonstrate an increase of human efficiency and software reliability also in scientific computing. Examples of numerical libraries for PDEs using this programming paradigm are Cogito [18], ELEMD [14], Kaskade [4], PETSc [2, 12], and Diffpack [5, 7]. All these packages apply the object-oriented concept on rather low level mathematical abstractions, like arrays, linear systems, linear solvers, preconditioners, finite elements etc. A solver can then be built by combining objects from the libraries.

The basic ideas of an object-oriented numerical library can be extended to a higher level where the objects reflect partial differential equations. This has been the topic of the present chapter. We have shown that this is a useful and powerful strategy for solving systems of PDEs. Furthermore, it opens up the possibilities of building repositories of solvers for single PDEs that can be combined with each other in a flexible way. This will dramatically reduce human efforts when developing software for advanced applications. A more detailed implementation in C++ of the design has been suggested to clarify the fundamental ideas.

Many research projects, especially in academia where students frequently enter and leave the activities, have suffered from the lack of programming standards and conventions. The approach suggested in this presentation can successfully be adopted by scientists working within a specific PDE-related application area, thus providing them with a mechanism for better organization of software contributions from different individuals. However, it should be stressed that the success of code reuse and coupling of stand-alone modules to form a complicated software environment relies heavily on a well-documented standard for building interfaces between the involved objects. In particular, there must be clear rules that dictate the behavior of the manager classes, and how the individual modules should be initialized. Such strict standards are needed if different researchers shall be able to develop new modules and reuse existing, already debugged components side-by-side in an explorative software environment.

Acknowledgments

The authors wish to thank the Research Council of Norway for the financial support through the strategic research programs STP.29643 (Toolkits in Industrial Mathematics), 110673/420 (Numerical Computations in Applied Mathematics) and the grant 100640/410. The authors also thank Xing Cai for his many useful suggestions that have spawned several significant improvements in the manuscript.

References

[1] E. Arge, A. M. Bruaset, P. B. Calvin, J. F. Kanney, H. P. Langtangen, and C. T. Miller. On the efficiency of C++ in scientific computing. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 93–119. Birkhäuser, 1997.

[2] S. Balay, W. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 163–202. Birkhäuser, 1997.

[3] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++. An Introduction with Advanced Techniques and Examples.* Addison-Wesley, 1994.

[4] R. Beck, B. Erdman, and R. Roitzsch. An object-oriented adaptive finite element code: Design issues and applications in hyperthermia treatment

planning. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 105–124. Birkhäuser, 1997.

[5] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 63–92. Birkhäuser, 1997.

[6] X. Cai, B. F. Nielsen, and A. Tveito. An analysis of a preconditioner for the discretized pressure equation arising in reservoir simulation. Preprint 1995-4, Department of Informatics, University of Oslo, 1995. (Submitted for publication).

[7] Diffpack World Wide Web home page.
URL: *http://www.oslo.sintef.no/diffpack.*

[8] K. Eriksson, D. Estep, P. Hansbo and C. Johnson. *Computational Differential Equations.* Studentlitteratur (Sweden) and Cambridge University Press (UK), 1996.

[9] FEMLAB World Wide Web home page.
URL: *http://www.math.chalmers.se/Research/Femlab.*

[10] E. J. Holm and H. P. Langtangen. A unified mesh refinement technique with applications to porous media flow, 1997. (Submitted for publication).

[11] E. J. Holm and H. P. Langtangen. A unified model for injection molding, 1997. (Journal paper in preparation).

[12] PETSc World Wide Web home page.
URL: *http://www.mcs.anl.gov/petsc/petsc.html.*

[13] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford Science Publications, 1992.

[14] G. Nelissen and P. F. Vankeirsbilck. Electrochemical modelling and software genericity. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 81–104. Birkhäuser, 1997.

[15] B. F. Nielsen and A. Tveito. On the approximation of the solution of the pressure equation by changing the domain. *SIAM J. Appl. Math.*, 57:15–33, 1997.

[16] J. R. Rice and R. F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Comp. Sci. & Engrg.*, 3:44–53, 1996.

[17] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1992.

[18] M. Thuné, E. Mossberg, P. Olsson, J. Rantakokko, K. Åhlander, and K. Otto. Object-oriented construction of parallel PDE solvers. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 203–226. Birkhäuser, 1997.