Designing software for simulating incompressible fluid flow

Roxana E. Diaconescu
Norwegian University of Science and Technology (NTNU)
IDI/NTNU 7491 Trondheim, Norway
roxana@idi.ntnu.no
Tel: +47 73 59 44 83, Fax: +47 73 59 44 66

Abstract

We report on our experiences with designing and implementing software for simulating incompressible fluid flow. This work is part of a larger project in Computational Science and Engineering within the Norwegian University of Science and Technology. We discuss the implementation of numerical algorithms for solving the incompressible Navier-Stokes equations using the C++ library Diffpack. Emphasis will be on various aspects related to software design, object-oriented methods and software engineering.

The paper discusses reusability issues from different perspectives. It shows how reuse is determined by the components to be reused, their organization and the quality of the design.

1 Introduction

The Computational Science and Engineering (CSE) project at Norwegian University of Science and technology (NTNU) seeks to challenge the one-discipline approach to the engineering problems by employing people with complementary expertise to solve problems of industrial importance. The success of the engineering projects relies on tools of engineering sciences, together with those of numerical analysis, applied mathematics and computer science. An example of such problem of an industrial importance is the Navier Stokes equations for incompressible flows. These problems occur in many practical situations, as for instance the study of the behavior of the fluid around a flexible cylinder hooking an oil platform to the sea bed. The general idea with such problems is to reduce the costs involved in experimentation through simulation and rapid prototyping.

The contribution of this article is to offer insight in CSE projects, from the computer science perspective. The goal was to develop the Navier Stokes simulation software in less time than writing it from scratch, to serve as a basic software platform for research within the project. From the computer science point of view we were interested in benefits of reuse, especially in the decrease of development time and effort for the software.

Reuse is one of the benefits of encapsulation, in object-oriented design. The general opinion is that reuse has proved its benefits, and the number of success stories confirms it. Moreover, software engineering literature gives considerable insight into reuse problems and their solutions. The reuse library problem, the quality of the components, domain analysis methods, metrics problem and organization for reuse (management structure of an organization) seem to be solved [Pou99].
Numerical software is lagging behind the current status of software engineering methods. This is proved by the reluctance in adopting object-oriented techniques in a field committed to the functional paradigm and Fortran (lately C) programming. There are important, pioneering efforts, though, towards switching to component based design and Diffpack library is one of them.

In our paper we analyze the factors which affected our reuse effort. For doing this we consider several issues. We show, considering a case study consisting of actual development and reuse, that much of the time and effort are spent in the design and debugging phase because of the difficulties in using the components. Moreover, organizing the components in few large packages impedes efficient reuse and fails in giving a clear logical view of the library.

The level of abstractions in components can affect reuse, but there is no clear guide-line showing the proper level of information hiding or reuse mechanisms, such as inheritance or composition. We believe it is more relevant to discuss the coupling between architectural components and we show that strong dependency affects reuse. Clustering the related components is one way of loosening the design coupling.

The rest of the paper is organized as follows. Section 2 presents the numerical equations, the numerical techniques to solve them, and the software implementation. Section 3 presents the results of the Diffpack case study and discusses the reuse library problem. Section 4 concludes and summarizes the paper.

2 The design of the application

In this section we will describe the case study, consisting of the actual reuse and implementation. We will present the equations, the numerical algorithm used to solve them and its implementation using Diffpack library.

The Navier Stokes equations describing the fluid flow are:

$$
\rho \cdot \left( \frac{\partial v}{\partial t} + v \cdot \nabla v \right) = - \nabla p + \mu \cdot \nabla^2 v + \rho \cdot b
$$

(1)

$$
\nabla \cdot v = 0
$$

(2)

The former equation is referred to as the momentum equation, while the latter is referred to as the mass conservation equation. The parameters and unknowns are as follows: \( \rho \) is the fluid density, \( v \) is the fluid velocity, \( p \) is the fluid pressure, \( \mu \) is a constant coefficient viscosity, and \( b \) denotes external body forces.

The finite element method to solve the Navier Stokes equations is based on an operator-splitting approach described in [Lan99]. The main idea is to split the computation of the velocity field in two, computing an intermediary value. The system of equations is decoupled, and the velocity field is first computed, followed by the computation of the pressure.

The Galerkin finite element formulation for the above equations is given next. The same basis functions are used for velocity and pressure. The symbols \( r,s \) iterate over space dimensions, while \( l \) indicates the time level.

$$
M_k^{(1)} = - \Delta t a_r(v_1, \ldots, v_d) - \nu \Delta t K v_r
$$

(3)

This equation corresponds to the calculation of the coefficients for the intermediate velocity. The matrix \( M \) is called the mass matrix and its entries are given by the form:

$$
M_{i,j} = \int_\Omega N_i N_j d\Omega
$$

(4)
Here $N_i$ represents the shape or basis functions used in the finite element approximation for the velocity and the pressure fields:

$$\mathbf{v}_i^t = \sum_{j=1}^{n} v_i^T N_j, \quad \mathbf{p}^t = \sum_{j=1}^{n} p_j^T N_j$$

(5)

The $K$ matrix arises from the Laplace operator and has the entries

$$K_{i,j} = \sum_s \int_{\Omega} \frac{\partial N_i}{\partial x_s} \frac{\partial N_j}{\partial x_s} d\Omega$$

(6)

The term $a_r$ represents the nonlinear terms and it has the entries

$$a_{r_i}^t = \sum_s \int_{\Omega} N_i N_k v_k \frac{\partial N_j}{\partial x_s} v_s^t d\Omega$$

(7)

At the end of the above computations and after solving the linear equation (3) we have the coefficient fields computed and can proceed with the next step of the algorithm computing the velocity fields as vector updates.

Knowing intermediary velocity fields, we can solve the Poisson equation for the pressure:

$$K \mathbf{p}^{t+1} = \frac{\rho}{\Delta t} \mathbf{B}^* \mathbf{v}_s^t$$

(8)

where the $K$ matrix has the same form as above, and

$$\mathbf{B}_{i,j}^* = \int_{\Omega} N_i \frac{\partial N_j}{\partial x_s} d\Omega$$

(9)

Knowing the pressure we can update the velocity field:

$$M \mathbf{v}^{t+1} = M \mathbf{v}_s^* - (\mathbf{B}_r^T \mathbf{p}^{t+1} - \rho \mathbf{c}_r) \frac{\Delta t}{\rho}$$

(10)

Matrices $M, B$ have the same entries as above. The $c_r$ contains the body forces and has the entries:

$$c_r = \int_{\Omega} N_i b_r d\Omega$$

(11)

The vector updates are not taking a linear system form, so they will be computed as one-dimensional array operations.

The above equations are solved and their solution is saved for each time step.

The way we have designed our application is quite simple and natural, given the numerical algorithm. The figure 1 on page 4 depicts the design of the application in terms of a class diagram in UML (Unified Modeling Language). The grey classes are the design classes that we have implemented, while the other classes (transparent in the picture) are the original Diffpack classes the simulator uses/interacts with to solve the PDE by FEM. The main design idea is that besides the $a_r$ vector, all the other matrices in the equations, $M, K, B$s are time independent and we will not need to compute them at each time step. The matrix $M$ is computed directly by using Diffpack's Mass class:

```javascript
Mass.make(*this, *grid);
M = Mass.M();
```

The computation of the time dependent vector $a_r$ is little more complicated. From the expression (7) we isolate a time independent part and compute it separately, in a vector $C$:

$$C = \int_{\Omega} N_i N_j \frac{\partial N_j}{\partial x_s} d\Omega$$

(12)
This vector will be computed by a redefined version of the `makeSystem` class `FEM`'s virtual function, only once, outside the time loop.

The $a_x$ vector will be then computed by a function invoked within the time loop:

```c
void NsSplit::ComputeA (Vec(real) C)  // The C vector.
    Handle(FieldsFE) ux,  // The velocity field.
    int rx,  // The given space dimension.
    Vec(real)& Ax)  // The output parameter ar.
```

For the $K$ matrix we will use the `integrandK`'s class to assemble it. Similarly, we will use the `integrandBs` supplier class for computing the $B_s$ matrices. A supplier class has granted rights to a client class to access its non-public parts. In our case the supplier classes are `integrandK` and `integrandBs`, and the client is the simulator class to which they “supply” the computation of the integrands for the two matrices, $K$ and $B_s$. The simulator uses the two classes mentioned above to assemble the element matrices for the equations:

```c
void IntegrandK::integrands(ElmMatVec& elmat, const FiniteElement& fe),
void IntegrandBs::integrands(ElmMatVec& elmat, const FiniteElement& fe).
```

Then the coefficients from the formula (3) are computed in a function:

```c
void NsSplit::ComputeCoef( Handle(FieldsFE) ux,  // The velocity field.
    int rx,  // The space direction.
    Handle(FieldsFE) &Kx)  // The resulting coefficient field.
```

The Poisson equation is solved in the function:
void NsSplit::solvePoisson(),

where Diffpack object lineq (the abstract form of a linear equation) is constructed by storing the $K, B_x, v_x^*$ in its right hand side, respectively left hand side. After constructing the linear system, by invoking its solve method, the solution is computed by Diffpack’s linear algebra package.

The update of the intermediary velocity field, equation (10), is straightforward matrix vector operations (also provided by Diffpack).

The algorithm repeats for each time step, having the time independent elements computed just once, outside the time loop.

To summarize, we give the mapping between the numerical algorithm and the software implementation in Table 1 on page 5:

Table 1: The mapping between the numerical algorithm and the computer implementation

<table>
<thead>
<tr>
<th>The numerical algorithm</th>
<th>The implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4) $M_{i,j} = \int_{\Omega} N_i N_j d\Omega$</td>
<td>Mass.make(*this, *grid); M = Mass.M();</td>
</tr>
<tr>
<td>(6) $K_{i,j} = \int_{\Omega} \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} d\Omega$</td>
<td>void integrandK::integrands(ElmMatVec&amp; elmat, const FiniteElement&amp; fe)</td>
</tr>
<tr>
<td>(9) $B_{i,j} = \int_{\Omega} N_i \frac{\partial v_x}{\partial x} d\Omega$</td>
<td>void integrandBs::integrands(ElmMatVec&amp; elmat, const FiniteElement&amp; fe)</td>
</tr>
<tr>
<td>(12) $C = \int_{\Omega} N_i N_j \frac{\partial \nu}{\partial x} d\Omega$</td>
<td>void NsSplit::makeSystem(Vec(real)&amp; X)</td>
</tr>
<tr>
<td>(7) $a_i^t = \int_{\Omega} N_i N_k \frac{\partial v_x}{\partial x} v_x^* d\Omega$</td>
<td>for all time steps</td>
</tr>
<tr>
<td>(3) $M_k^{(1)} = -\Delta + \nu \Delta K v_x$</td>
<td>void NsSplit::ComputeCoeff(Handle(FieldsFE) ux, int rx, Vec(real)&amp; Ax)</td>
</tr>
<tr>
<td>(8) $K^{t+1} = \frac{\partial B_x}{\partial \rho}$</td>
<td>void NsSplit::solvePoisson()</td>
</tr>
<tr>
<td>(10) $M_v^{t+1} = M v_x^* - (B_x^{t+1} - \rho_0) \frac{\partial \rho}{\partial \rho}$</td>
<td>Simple vector matrix operations.</td>
</tr>
</tbody>
</table>

3 Results and Discussion

Before reporting the results of our case study, we must emphasize the goal and the criteria for judging the results. The goal was to establish a working platform as a basis for research in different fields in shorter time and using less effort than if developing it from scratch. For this purpose we used the C++ library Diffpack. Different people in the project had different criteria for judging the result. From the computer science point of view we were interested in the benefits of applying object-oriented technology, especially reuse, to decrease the time and effort for writing numerical applications. For numerical analysts one of the criteria was the accuracy of the results (numerical solution). We concentrate on the former point of view in this paper. For all the parties in the project the effort used to design, implement and debug the code was important.

In order to evaluate the reusability factors, we have considered the following issues:

1. A case study, consisting of actual development and reuse.
2. The components to be reused.

3. Specific factors concerning code characteristics such as organization of components and design issues.

3.1 The case study

The case study consisted of the implementation of a FEM solver for the Navier Stokes equations. Reuse was achieved by using the C++ library Diffpack. The results were:

- Lots of time and effort were spent in the design phase, in order to understand the components and to deploy the design of the solver from the components.

- Less time and effort were spent in the actual development phase, given that Diffpack’s components accounted for most of abstractions needed by the implementation of the numerical algorithm.

- Lots of time and effort were spent in the debugging phase, given that the inner working and interactions of the components, as well as the lateral effects were not well understood due to the poor documentation and components’ organization, mostly.

We will prove in the following subsections that the time and effort spent in the design and debugging phase can be decreased, and reuse made easier.

3.2 The components to be reused

These are Diffpack classes, which offer abstractions for most of the objects in the application domain: grid, finite elements, degree of freedom, fields, matrix, vectors, linear equation, linear solver, and so on.

The components must be manually retrieved and inspected. Diffpack’s command dpman can be used to retrieve a component interface and its description. The result of the command consists of the class header for the component, which has a textual description appended at the end. Most of the data members of the interface and the operations are not commented. The most “important” functions are described at the end of the header file. The interfaces are too long, and they lack complete comments describing each data member, each operation and its parameters. As a consequence it is hard to understand the functionality and the usage for a given component. Moreover, each component refers to other Diffpack components, documented in the same manner. All these are illustrated in table 2.

The numbers in the table are approximate, but accurate enough to give a correct idea about the component description.

The number of Diffpack components we have used in our solver is 14. The header file consists of 98 lines, while the implementation was 546 lines of code (including comments). About the same size we found for another implementation of Navier Stokes equations, given by the authors of Diffpack. Therefore the coding effort was not significant, but the effort to analyze and identify the components we needed was considerable.

The text description for each component is far from complete or comprehensible. The following quotation from such a description illustrates what we mean:

One can apply class FEM without knowing the details of the implementation. However it is strongly recommended that serious finite element programmers study the bodies of the class FEM functions to understand how the building blocks of the finite element tool box in Diffpack work.
Table 2: Component description

<table>
<thead>
<tr>
<th>Component</th>
<th>Number of lines for the interface</th>
<th>Number of lines for the description</th>
<th>Number of references to other Diffpack components</th>
</tr>
</thead>
<tbody>
<tr>
<td>FieldFE</td>
<td>170</td>
<td>208</td>
<td>15</td>
</tr>
<tr>
<td>GridFE</td>
<td>250</td>
<td>652</td>
<td>15</td>
</tr>
<tr>
<td>FEM</td>
<td>290</td>
<td>310</td>
<td>10</td>
</tr>
<tr>
<td>DegFreeFE</td>
<td>200</td>
<td>420</td>
<td>10</td>
</tr>
<tr>
<td>FieldsFE</td>
<td>99</td>
<td>200</td>
<td>7</td>
</tr>
<tr>
<td>Vec</td>
<td>113</td>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>LinEqAdmFE</td>
<td>50</td>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>

Start with makeSystem and follow the program flow until the element loop is finished. The understanding of this class requires the reader to be familiar with finite element programming, C++ and object-oriented programming.

The only problem is that the class implementations are not available, although they were meant to serve as documentation.

Our conclusion is that the components in Diffpack are therefore hard to be “reused”. Standard, shorter interfaces, with comments for every class member would decrease the learning time for the library.

3.3 Design and organization of the components

The library consists of a directory organized in three main packages.

- Basic tool package, providing mainly for data structures, such as arrays, sets, and so on. It consists of approximately 235 classes.

- Linear algebra package for solving linear equations by using direct and iterative methods. The package has a linkage to external Fortran functions from BLAS and Lapack libraries. It has approximately 165 classes.

- The package for solving PDEs, covering abstractions for both FEM and FD. This has approximately 168 classes.

Figure 2 depicts the reversed engineered Unified Modeling Language (UML) diagram for Diffpack using Rational Rose for Solaris. Given the organization of the components in only three main packages with very high number of components each, we regrouped the logically related components in smaller packages. We intended to increase Diffpack’s logical view understandability. Even after restructuring, the result was illegible. The point is that a user needs a clear logical view of the organization of the components in order to be able to efficiently reuse them. We do not necessarily impose the view that “the best libraries range from only 30 components to, in rare cases, as many as 250 components” [Fot99]. It may be easy to understand why this is a sensible point. It is clear that the bigger the library, more care must be taken in modularization and documentation at both the component level and conceptual level (architecture of the library).

Besides the components and their organization, reuse can also be investigated from “design for reuse” perspective. By this we mean how the object-oriented
technology is applied, so reuse is made easier. We will focus on the “proper” level of abstraction/information hiding enabling reuse. In terms of visibility, our discussion can be synthesized “white-box v.s. black-box”. In the literature we found two views regarding white-box v.s. black-box.

One view [GJHV99] is reuse by inheritance vs. reuse by composition. The main disadvantage of reuse by inheritance according to this view is that “inheritance breaks encapsulation” [Str86]. Consequently is advised to favor composition over class inheritance.

Another view [CG93] describes the degree of abstraction of architectural components in terms of “white box” and “black box”. According to this view, the inner workings and behaviors of white box components are very visible and well understood. The interactions between white boxes are usually predicted with few unexpected side effects. The inner workings and behavior of black box components are hidden from the user. Interactions between black boxes are more difficult to predict, and may result in surprising and undesirable side effects. And this is because over-abstracting an architectural component can significantly increase the effort required to deploy that component in an architecture, as well increase the

---

Figure 2: The UML Rational Rose logical view of Diffpack
risk in reusing that architecture [CG93].

Neither of the above views is exemplified on a concrete example and they give contradictory advice (i.e. first favors black box, second the contrary). We will examine this in Diffpack case.

According to the former view, the main reuse mechanism in Diffpack is inheritance. Newly implemented applications are usually derived from Diffpack’s classes (e.g. FEM, SimCase, etc.). Moreover, the entire Diffpack hierarchy is based on inheritance. The point is, though, that there is not only white, or only black. New functionality is obtained by composing components in order to implement the required algorithms (in Diffpack). On the other hand, it is hard to even indicate the “grey scale”: is it more inheritance, is it more composition? This is hard to answer and even so, it is hard to prove the relevancy of this answer, i.e. if this is a problem or an advantage in Diffpack’s case.

According to the latter view, some of the components have high visibility, because they are based on inheritance. But because all of them refer to many other components, the inner working and the interactions between the components are hard to understand. We found again ourselves in the dilemma of not being able to use these guidelines/experiences in pointing out the problem and the solution in evaluating the reusability factors in this concrete situation.

We find it more relevant to discuss another issue regarding design for reuse, namely the coupling between components. As we have shown, most of Diffpack classes contain a relatively high number of references to other components. Therefore we could say that Diffpack has a tightly coupled architecture. Even though this is the whole point with object-orientation and reuse, i.e. to build from objects/components, we have found from our case study that making the components too tightly coupled to other components is not advisable. One reason is that the components become a part of a context, which has to be very well understood in order to make it usable. Second, some of the flexibility is lost as a consequence of such coupling. As it is impossible not to have some coupling and dependencies between components, a solution would be “clustering”. By this we mean the grouping of related/dependent components in a cluster (e.g. matrix module) with a clear dependency view. This way we will have high dependencies, tight coupling within a cluster, but looser coupling, less dependencies among clusters.

In this chapter we have discussed different reusability factors, starting with the issue of actual reuse and implementation, then the components and their organization and concluding with issues about employing object-oriented methods to provide for reuse. We have shown what can be a problem by using Diffpack as a case study.

4 Conclusions

This paper presents a reuse case study for numerical libraries. The components to be reused are numerical analysis abstractions and it is not expected to be easily learned and used. Therefore it is important to have shorter interfaces, with clear functionality and documentation.

Component organization makes up the logical view of the library and its understandability can improve the learning time. Too large component collections are hard to navigate and use.

Components visibility, reuse by inheritance or composition and coupling of the architectural components determine design quality and understandability. The proper level of visibility, i.e. white box v.s. black box is hard to establish, though. On the other hand, it is easy to see why strong dependencies and tight coupling between components is not advisable.
Diffpack is an important step in introducing modern software engineering technologies to the numerical software community. It proves the feasibility of designing components for reuse in the numerical analysis domain. Diffpack succeeds in letting the user work at a higher level of abstraction, by covering difficult implementation aspects of numerical algorithms, e.g. book-keeping for local and global numbering and transformations, boundary indicators, construction of the global matrix and so on. It may be too ambitious in covering the modern numerical analysis techniques, such as finite element methods, mixed elements, adaptivity, multigrids, domain decomposition. Too large component libraries are hard to be reused. Small libraries of greatly used, well designed, domain-specific, high-quality components libraries partly solve the reuse problems [Pou99].

References


