CHALMERS

FINITE ELEMENT CENTER

PREPRINT 2002–06

DOLFIN: Dynamic Object oriented Library for FINite element computation

Johan Hoffman and Anders Logg

Chalmers Finite Element Center
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg Sweden 2002

CHALMERS FINITE ELEMENT CENTER

Preprint 2002–06

DOLFIN: Dynamic Object oriented Library for FINite element computation

Johan Hoffman and Anders Logg

CHALMERS

Chalmers Finite Element Center
Chalmers University of Technology
SE–412 96 Göteborg Sweden
Göteborg, April 2002

DOLFIN: Dynamic Object oriented Library for FINite element computation
Johan Hoffman and Anders Logg
NO 2002–06
ISSN 1404–4382

Chalmers Finite Element Center
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone: +46 (0)31 772 1000
Fax: +46 (0)31 772 3595
`www.phi.chalmers.se`

Printed in Sweden
Chalmers University of Technology
Göteborg, Sweden 2002

DOLFIN: DYNAMIC OBJECT ORIENTED LIBRARY FOR FINITE ELEMENT COMPUTATION

JOHAN HOFFMAN AND ANDERS LOGG

Abstract. In this report, we present a new software project intended to provide a platform for adaptive finite element computation in both research and education. The software DOLFIN is implemented in C++ and provides an object-oriented powerful and flexible environment in which finite element methods for differential equations can be easily implemented. The current version of DOLFIN already includes a number of modules for special differential equations, including the incompressible Navier–Stokes equations. DOLFIN is *free* and *open-source*, which we believe is a key point in creating truly useful software.

1. Introduction

Solving differential equations is one of the most important tasks in science and engineering. Given a mathematical model of the form

$$A(u) = f, \tag{1.1}$$

where A is a differential operator on some domain and f is given data, the task is to find the solution u. For non-trivial problems we are forced to seek an approximate solution U, and the task is now to compute U such that the error $e = U - u$ is smaller than a given tolerance $\mathrm{TOL} > 0$. *Adaptive finite element methods* provide a general framework for computing approximate solutions of differential equations such as (1.1), including techniques for estimating the size of the error e and techniques for adapting the computation to meet the criterion $\|e\| \leq \mathrm{TOL}$, see [7] for a general survey. In principle the task of computing an approximate solution U to eq. (1.1) can thus be handled *automatically*, by software implementing the framework of [7]. This is where DOLFIN comes into the picture.

A good code should be *flexible* enough to handle many different equations and algorithms, and *efficient* enough to handle large-scale three-dimensional problems on complex geometries. We believe a good compromise between flexibility and efficiency is reached by implementing the software as an object-oriented environment in C++. A good code should also be *easy to use*. Usability does not have to conflict with flexibility as long as the software is accessible at different levels. We explain this in more detail below. The last,

Date: April 10, 2002.

Key words and phrases. DOLFIN, finite element method, implementation, automatic assembling, variational formulation, adaptivity, solver, GNU GPL.

Department of Computational Mathematics, Chalmers University of Technology, S–412 96 Göteborg, Sweden, *email*: hoffman@math.chalmers.se, logg@math.chalmers.se.

but not the least important, cornerstone of a software project is that it is *available*, i.e. free and open-source; this makes it easier to write flexible, efficient and usable software.

Flexibility is achieved by providing a set of standard tools which are commonly used in finite element codes. These tools are implemented as C++ classes and can in principle be combined to enable the implementation of any given algorithm. These tools or classes include e.g. finite elements, equations, grids, vectors, sparse matrices and solvers. The perhaps most important tool is the *discretiser*, which can automatically assemble sparse matrices from the variational formulation of the differential equation. This approach makes it easy to implement methods for simple problems such as the Poisson equation, as well as more complex algorithms for solving e.g. the Navier–Stokes equations. To solve Poisson's equation all we have to do is to take the variational formulation and assemble into a sparse matrix using the discretiser (which is one line of code). We then take a multi-grid solver and solve the equations (another line of code). To solve the Navier–Stokes equations, we have to do a little bit more, but the basic building blocks are the same.

With a good compiler at hand, we can write *efficient* code in C++, provided we are careful and implement good algorithms. Since for large-scale three-dimensional problems memory usage is often of importance, we have to be careful and cheap with memory at all levels. Care is thus taken to ensure that all basic tools such as e.g. grids and sparse matrices are stored and accessed as efficiently as possible. To further stress the importance of being cheap with memory and cpu resources, constructs such as

```
Vector x(1000) = 1;
Vector y;
...
y = x;
```

are avoided since `y = x` involves allocating memory for `y` as well as copying all elements of `x`. Instead the user has to be more explicit and do

```
y.CopyFrom(&x);
```

which makes it more obvious that we are performing an operation that may consume both cpu and memory resources.

Usability means different things to different persons. A novice user who is not at all familiar with finite element methods, or perhaps not even with differential equations, should still be able to use the program. Ideally the true novice should only have to press the solve button to make something happen. On the other hand, just pressing the solve button is not enough for all users. Most users want to access the program at a lower level. This level will be different for different users and thus the program has to be accessible at many different levels. In this way the key to usability is *accessibility*. DOLFIN is accessible at three different levels: *user space*, *module space* and *kernel space*, as is explained in more detail below.

To ensure the *availability* of the program, DOLFIN is licensed under the GNU GPL, see [2]. This means that anyone can download and install DOLFIN free of charge. This also means that the source code is available and that anyone can modify the code to suit their needs (and possibly redistribute the code).

In the following sections we give an account for the features and the structure of DOLFIN. Part of these features are already implemented. For a more detailed account on the progress of DOLFIN visit the home page [9]. Comments and contributions are welcome.

2. Basic concepts and organisation

To achieve the goals stated in the introduction, we need to have a clear structure and this is simplified by a modular and object-oriented approach. Different parts or components of DOLFIN are thus implemented as C++ classes, which are organised in a modular way according to Figure 1. The most fundamental of these classes is the `FiniteElement`. Following Ciarlet [6] and Brenner-Scott [5], a *finite element* $(K, \mathcal{P}, \mathcal{N})$ is defined by

(1) A domain $K \subseteq \mathbb{R}^N$ with piecewise smooth boundary, typically a triangle or a tetrahedron;
(2) A finite-dimensional space $\mathcal{P}$ of functions on K together with a set of basis functions (the *shape functions*);
(3) A basis $\mathcal{N} = \{N_1, N_2, \ldots, N_k\}$ for the dual space $\mathcal{P}^*$ (the *degrees of freedom*).

A finite element thus contains geometric information about the domain K, a function space $\mathcal{P}$ with certain properties and a set of degrees of freedom $\mathcal{N}$ connected to the function space. This does not mean that we construct a large array of finite elements containing this information, since that would consume large amounts of memory. Instead the information is stored elsewhere and the `FiniteElement` class and its related classes, such as e.g. `FunctionSpace`, `ShapeFunction` and `LocalField` are used only as temporary containers or interfaces. When assembling a linear system from a given variational formulation (which may be part of a loop for solving a nonlinear system) the finite element, including the domain and the local function space, is then for every element updated to represent the current finite element: geometric information is fetched from the grid and data for the local function space, e.g. derivatives of the shape functions, is computed. In Table 1 we list the most important classes involved in this process.

Other important parts of DOLFIN are the `Grid` class which handles geometry information and grid refinement, and algebraic solvers such as the `KrylovSolver`. Closely related to the discussion above on the `FiniteElement` class are the two classes `Discretiser` and `Equation`. We discuss the assembly routines contained in `Discretiser` in more detail below in Section 3 and we also give an overview of the algebraic solvers in Section 6.

3. Automatic assembling with overloaded operators

An important step in automating the process of computational modeling is the automatic assembling of a linear system from a given variational formulation. DOLFIN handles this by *operator overloading*. In C++ binary operators can be defined for two objects of a class or for two objects from different classes. This makes it possible to define the operator '`*`' between two objects `v` and `w` of type `ShapeFunction`, representing two shape functions v and w on the domain K of a finite element. We define the operator '`*`' in the following

`FiniteElement`	The *finite element*, containing geometric information and information about the local function space.
`FunctionSpace`	A finite-dimensional space of functions on the domain of a finite element.
`TriLinSpace, TetLinSpace, ...`	Sub-classes derived from `FunctionSpace`.
`ShapeFunction`	A member of the basis for a local function space on the domain of a finite element.
`TriLinFunction, TetLinFunction, ...`	Sub-classes derived from `ShapeFunction`.
`LocalField`	A member of the local function space on the domain of a finite element, i.e. a linear combination of the shape functions. A `LocalField` can also be viewed as the restriction of a `GlobalField` to the domain of a finite element.
`GlobalField`	A function (possibly vector-valued) defined on the whole of the computational domain.

TABLE 1. A list of classes closely related to our implementation of a *finite element*.

way:

$$\texttt{v * w} = \frac{1}{|K|}\int_K v \cdot w \; dx, \tag{3.1}$$

where $|K|$ is the volume (or area) of the domain K. We also define

$$\texttt{a * v} = \frac{1}{|K|}\int_K \alpha \cdot v \; dx, \tag{3.2}$$

where α is a real number represented in the program by the variable `a` of type `real`. We also define similar operations between two objects of the class `LocalField`, representing linear combinations of shape functions, and also between two objects of class `LocalField` and `ShapeFunction` respectively. These operations are defined as linear combinations using the basic operators (3.1) and (3.2).

Using these operators we can assemble directly from a given variational formulation. As an example, consider the *heat equation*,

$$\dot{u} - \nabla \cdot (c \nabla u) = f, \tag{3.3}$$

on $\Omega \times (0, T]$ together with a suitable set of boundary and initial conditions, where $c = c(x,t)$ is a given heat conductivity and $f = f(x,t)$ is a given source term. Assuming that V is the space of continuous piecewise linear functions on a partition $\mathcal{T} = \{K_i\}$ of $\Omega = \cup_i K_i$ and $0 = t_0 < t_1 < \ldots < t_M = T$ is a partition of $[0, T]$, the cG(1)dG(0) method for (3.3) reads: Find $U^n \in V$ for $n = 1, \ldots, M$ with U^0 given such that

$$\int_\Omega \left(\frac{U^n - U^{n-1}}{k_n} \right) v \; dx + \int_\Omega c(\cdot, t_n) \nabla U^n \cdot \nabla v \; dx = \int_\Omega f(\cdot, t_n) v \; dx, \tag{3.4}$$

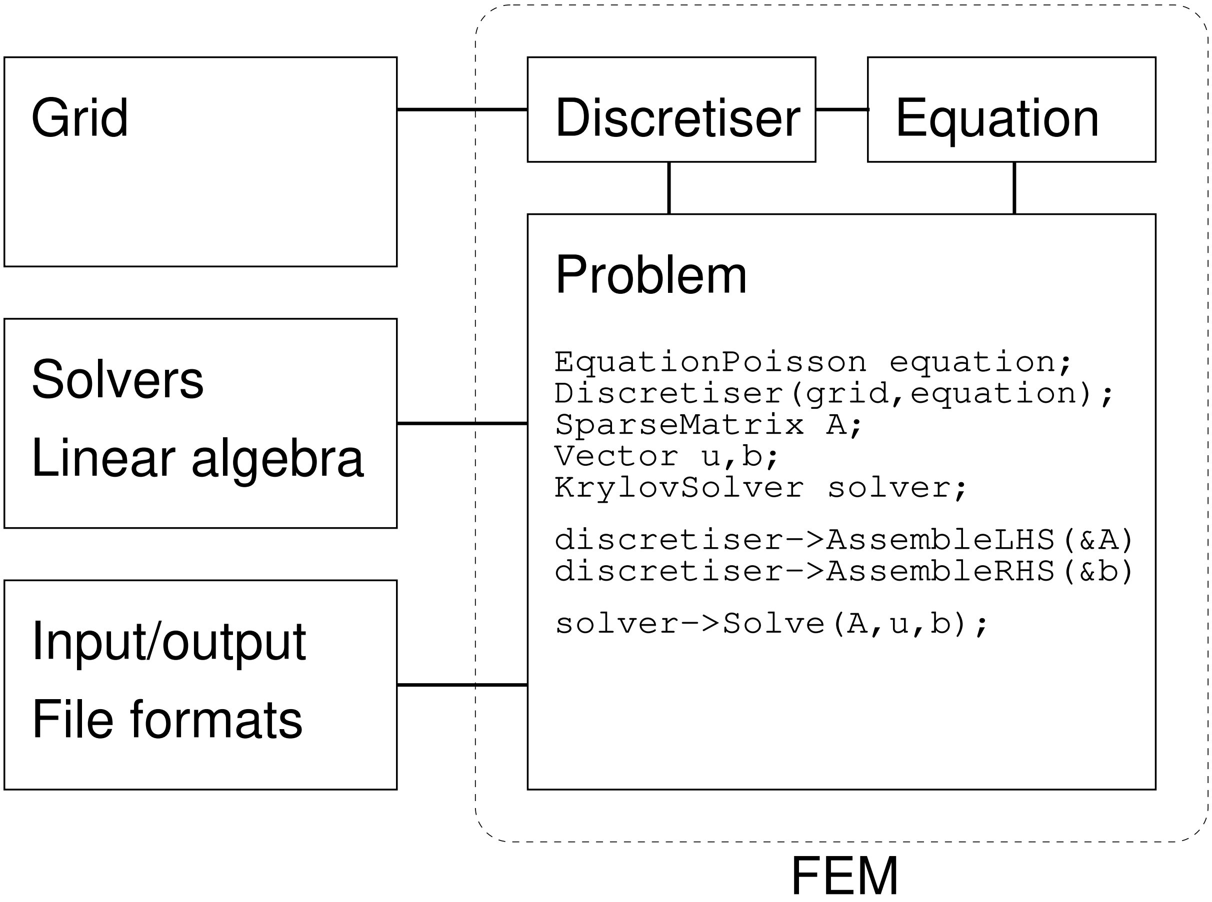

FIGURE 1. Conceptual organisation of DOLFIN.

for all $v \in V$, where $\{k_n\} = \{t_n - t_{n-1}\}$ is a sequence of time steps. Solving for U^n in every time step, we assemble a linear system from the variational formulation (3.4). In the notation used by DOLFIN we specify the left hand side as

$$A \sim \texttt{(u*v - up*v)/k + c*(u.dx*v.dx + u.dy*v.dy + u.dz*v.dz)},$$

and the right hand side as

$$b \sim \texttt{f*v},$$

where `u` and `v` are objects of class `ShapeFunction` and `up`, `c` and `f` are given objects of the class `LocalField`. We then solve $Ax = b$ for x, where the vector x contains the degrees of freedom for U^n, using the algebraic solver of choice. For piecewise linear test and trial functions, derivatives (represented by $(\cdot)$`.dx`, $(\cdot)$`.dy` and $(\cdot)$`.dz`) are constant on every element. For higher order elements the basic operators (3.1) and (3.2) have to be defined also for the derivatives of the shape functions.

The actual evaluation of the integrals corresponding to the multiplication operators can be implemented in various ways, depending on the type of elements and function spaces. As an example consider again piecewise linear elements on triangles or tetrahedrons, equipped with the standard Lagrange shape functions: $\{\lambda_1, \lambda_2, \lambda_3\}$ and $\{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}$ respectively. For these simple elements we have the exact formulas

$$\frac{1}{|K|}\int_K \lambda_1^{m_1}\lambda_2^{m_2}\lambda_3^{m_3}\ dx = \frac{2\cdot m_1!m_2!m_3!}{(m_1+m_2+m_3+2)!},$$

and

$$\frac{1}{|K|}\int_K \lambda_1^{m_1}\lambda_2^{m_2}\lambda_3^{m_3}\lambda_4^{m_4}\ dx = \frac{3!\cdot m_1!m_2!m_3!m_4!}{(m_1+m_2+m_3+m_4+3)!}.$$

From this it follows that if `v` represents a shape function λ_i, we have

$$\texttt{a * v} = \frac{1}{|K|}\int_K \alpha\lambda_i\ dx = \alpha/3$$

for triangles and

$$\texttt{a * v} = \frac{1}{|K|}\int_K \alpha\lambda_i\ dx = \alpha/4$$

for tetrahedrons.

For other types of elements, especially higher-order elements, we may want to use quadrature. The only difference is then the actual implementation of the operators, not the specification of the variational formulation.

4. Access levels

As discussed above, DOLFIN is accessible at a number of different levels. Currently the three main levels are *user space*, *module space* and *kernel space*. In simple terms, user space is for the person who just wants to solve a particular problem (for which there already exists a module). Module space is for the person who wants to implement a new algorithm for a specific equation, or perhaps improve an existing module. Kernel space is for the person who needs to add core features or improve the existing basic tools.

4.1. **User space.** In user space, DOLFIN is accessed through a simple C/C++ interface providing a basic set of commands. Using these basic commands, a user only has to specify the geometry and the equation that should be solved, together with boundary conditions (including initial conditions for time-dependent problems). The behaviour of the solver for the specified equation can be controlled by changing the values of different parameters (for which default values are supplied). In the future, an extra layer in the form of a graphical user interface may be added on top of the basic C/C++ interface. In Table 2 we show a typical example of a simple program that solves Poisson's equation using DOLFIN.

4.2. **Writing modules.** If the simple interface discussed above is not enough or no module exists for a particular equation, a new module has to be added. On module level the user has access to a set of basic tools implemented as C++ classes. The most important of these are `Equation`, `Discretiser`, `GlobalField`, `SparseMatrix` and `Vector`. An `Equation` contains the variational formulation for a specific equation. Using the `Discretiser` we can

```
#include <dolfin.h>

int main(int argc, char **argv)
{
    dolfin_set_problem("poisson");

    dolfin_set_parameter("output file",     "poisson.dx");
    dolfin_set_parameter("grid file",       "tetgrid.inp");
    dolfin_set_parameter("space dimension", 2)

    dolfin_set_boundary_conditions(my_bc);
    dolfin_set_function("source",f);

    dolfin_init(argc,argv);
    dolfin_solve();
    dolfin_end();

    return 0;
}
```

TABLE 2. An implementation of a solver for Poisson's equation at user level.

automatically assemble a `SparseMatrix` and a load `Vector` from the variational formulation, and then solve using one of the algebraic solvers which are also provided as basic tools. These basic classes or tools can be combined in many ways. As an example we can have many equations and the solution of one equation may be fed into another equation, for example in a fixed point iteration between the momentum and the pressure equations when solving the incompressible Navier–Stokes equations in an operator splitting approach.

A `GlobalField` represents a function, possibly vector-valued, and can be used to couple the degrees of freedom contained in a `Vector` to a real-valued function, or to represent a user supplied function for the right-hand side or a coefficient in an equation. This class is important, since it allows the user to make reference to functions in the variational formulation independent of their actual representation. We give an example of this below in Tables 3 and 4 where we present a simple implementation of a module for Poisson's equation.

A DOLFIN module in its simplest form is composed of two main classes: an `Equation` sub-class and a `Problem` sub-class. The `Equation` implements the variational formulation of the problem and the `Problem` class implements the algorithm to be used. In simple cases, such as the Poisson problem, all that has to be done is to assemble a sparse matrix from the variational formulation and then solve a linear system using one of the algebraic solvers provided by DOLFIN. For other problems the algorithm may be more complex.

```
class EquationPoisson: public Equation{
public:

    EquationPoisson():Equation(3){
        AllocateFields(1);
        field[0] = &f;
    }

    real IntegrateLHS(ShapeFunction &u, ShapeFunction &v){
        return ( u.dx*v.dx + u.dy*v.dy + u.dz*v.dz );
    }

    real IntegrateRHS(ShapeFunction &v){
        return ( f * v );
    }

private:
    LocalField f;
};
```

TABLE 3. An implementation of a module for Poisson's equation: the class `EquationPoisson`.

4.3. **Kernel.** At kernel level the user has access to the actual implementation of all the basic tools that are available at module level. This is where the real action takes place, and this is where the user (or the developer) can add new functionality to DOLFIN, including the addition of new algebraic solvers, improvement of existing methods, addition of new input and output formats, or perhaps addition of completely new functionality. As an example 5 shows part of the `Equation` base class, from which `EquationPoisson` and other equations are derived.

5. INPUT AND OUTPUT

DOLFIN is a solver. It does not have have any pre-processing tools (mesh generation), nor any post-processing tools (visualisation). This specialisation makes it possible to focus entirely on the solution process. This also means that we have to rely on other codes in order to have a complete system containing also pre- and post-processing. There already exist a number of such tools, both free and non-free, that can be used with DOLFIN, and the idea is not to focus and adapt DOLFIN to work well with only one of these tools, but instead provide interfaces to as many pre- and post-processing tools as possible. The current version can import data from GiD [1] and MATLAB [3] (which are both commercial software) and export data to OpenDX (which is a free open-source package, [4]), GiD and MATLAB. The code is organised in a way that should allow for easy addition of new formats for input and output.

```
void ProblemPoisson::Solve()
{
    EquationPoisson equation;
    SparseMatrix A;
    Vector x,b;
    KrylovSolver solver;
    GlobalField u(grid,&x);
    GlobalField f(grid,"source");
    Discretiser discretiser(grid,equation);

    equation.AttachField(0,&f);
    discretiser.Assemble(&A,&b);
    solver.Solve(&A,&x,&b);

    u.SetLabel("u","temperature");
    u.Save();
}
```

TABLE 4. An implementation of a module for Poisson's equation: the class `ProblemPoisson`.

```
class Equation{
public:

    Equation(int nsd);
    ~Equation();

    virtual real IntegrateLHS(ShapeFunction &u, ShapeFunction &v) = 0;
    virtual real IntegrateRHS(ShapeFunction &v) = 0;

    void UpdateLHS(FiniteElement *element);
    void UpdateRHS(FiniteElement *element);
    void AttachField(int i, GlobalField *globalfield, int component = 0);

    ...
```

TABLE 5. Part of the base class `Equation` at kernel level from which `EquationPoisson` and other equations are derived.

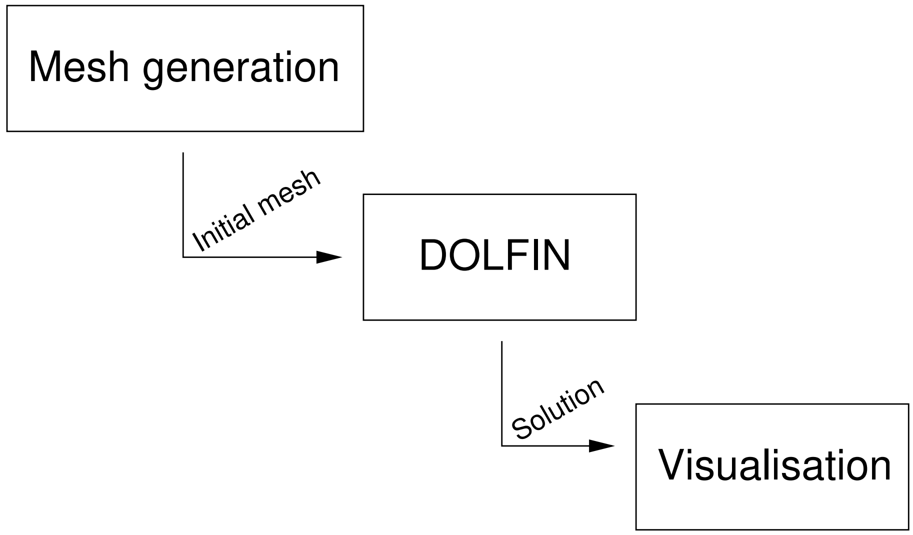

FIGURE 2. The solver (DOLFIN) is the key part in the larger system containing also mesh generation (pre-processing) and visualisation (post-processing).

6. Algebraic solvers

In the current version of DOLFIN several algebraic solvers are implemented to handle the large sparse linear systems of equations that arise in the discretisation of differential equations. Apart from a direct solver and the standard classical stationary iterative methods, such as the Jacobi and the Gauss-Seidel methods, Krylov methods, such as the conjugate gradient method and GMRES, are implemented. Future versions of DOLFIN will include also preconditioners, multigrid solvers, and solvers for eigenvalue problems.

7. Examples

In this section, we present a couple of basic examples of equations we have solved using DOLFIN: *Poisson's equation*, *convection–diffusion*, and the *incompressible Navier–Stokes* equations. Solvers for these equations are implemented as modules in DOLFIN, and more modules for other equations will be added in the future.

7.1. Poisson's equation.

We solve Poisson's equation:

$$-\Delta u(x) = f(x), \quad x \in \Omega, \tag{7.1}$$

on the unit square $\Omega = (0,1) \times (0,1)$ with homogeneous Dirichlet boundary conditions on two opposing edges and homogeneous Neumann boundary conditions on the other two edges. The source term $f = f(x)$ is given by

$$f(x) = \begin{cases} 100, & (x-0.5)^2 + (y-0.5)^2 < 0.3^2, \\ 0, & (x-0.5)^2 + (y-0.5)^2 \geq 0.3^2. \end{cases}$$

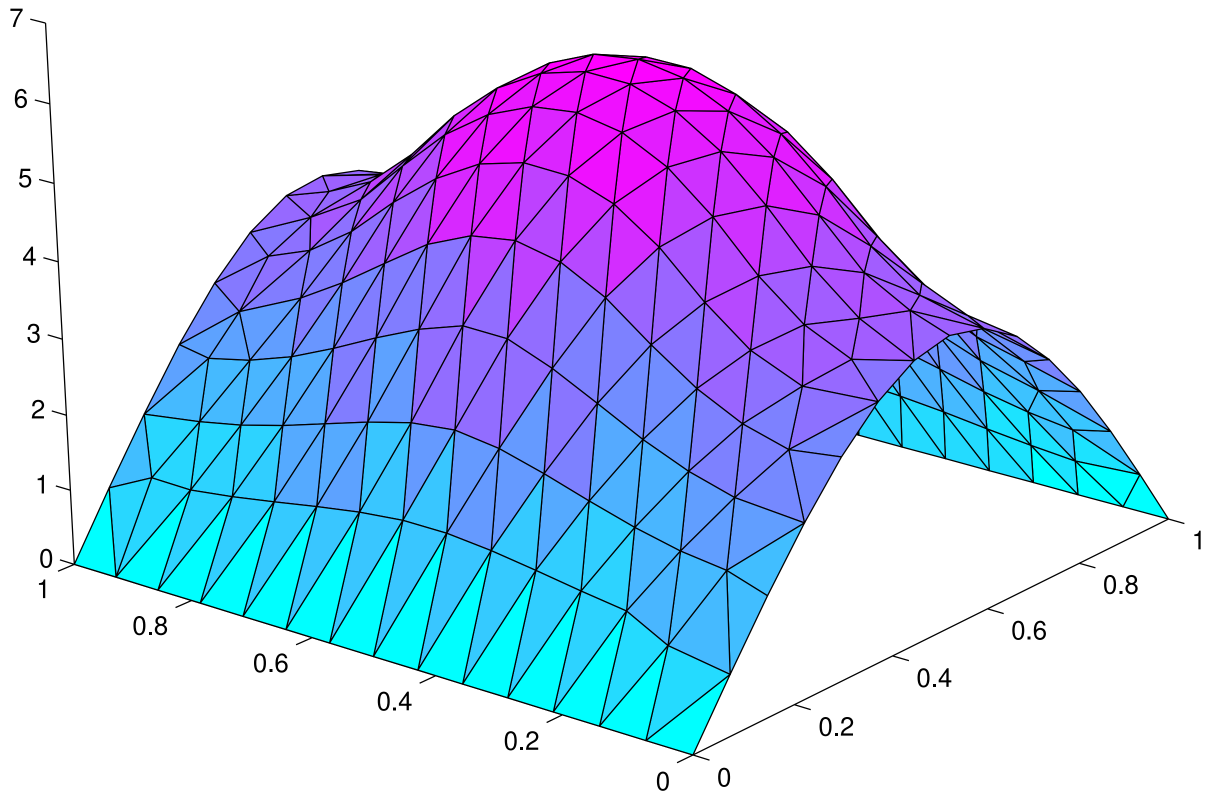

FIGURE 3. Poisson's equation on the unit square. The grid was generated with GiD and the solution is visualised using the `pdesurf` command in MATLAB.

7.2. **Convection–Diffusion.** Another basic example is the time-dependent convection–diffusion equation:

$$\dot{u} + b \cdot \nabla u - \nabla \cdot (\epsilon \nabla u) = f, \tag{7.2}$$

together with a set of boundary and initial conditions, where b is a given vector field for the convection, ϵ is a given diffusivity and f is a given source term. The solution $u = u(x,t)$ represents the temperature (or the concentration of a substance) at (x,t). We compute the solution for a given velocity field $b = (-10, 0)$, with $f = 0$ and $\epsilon = 0.1$ on the domain presented in Figure 4. The temperature u is taken to be $u = 1$ on the dolphin and $u = 0$ on the right inflow boundary. On the remaining three edges of the boundary we choose homogeneous Neumann boundary conditions: $\partial_n u = 0$.

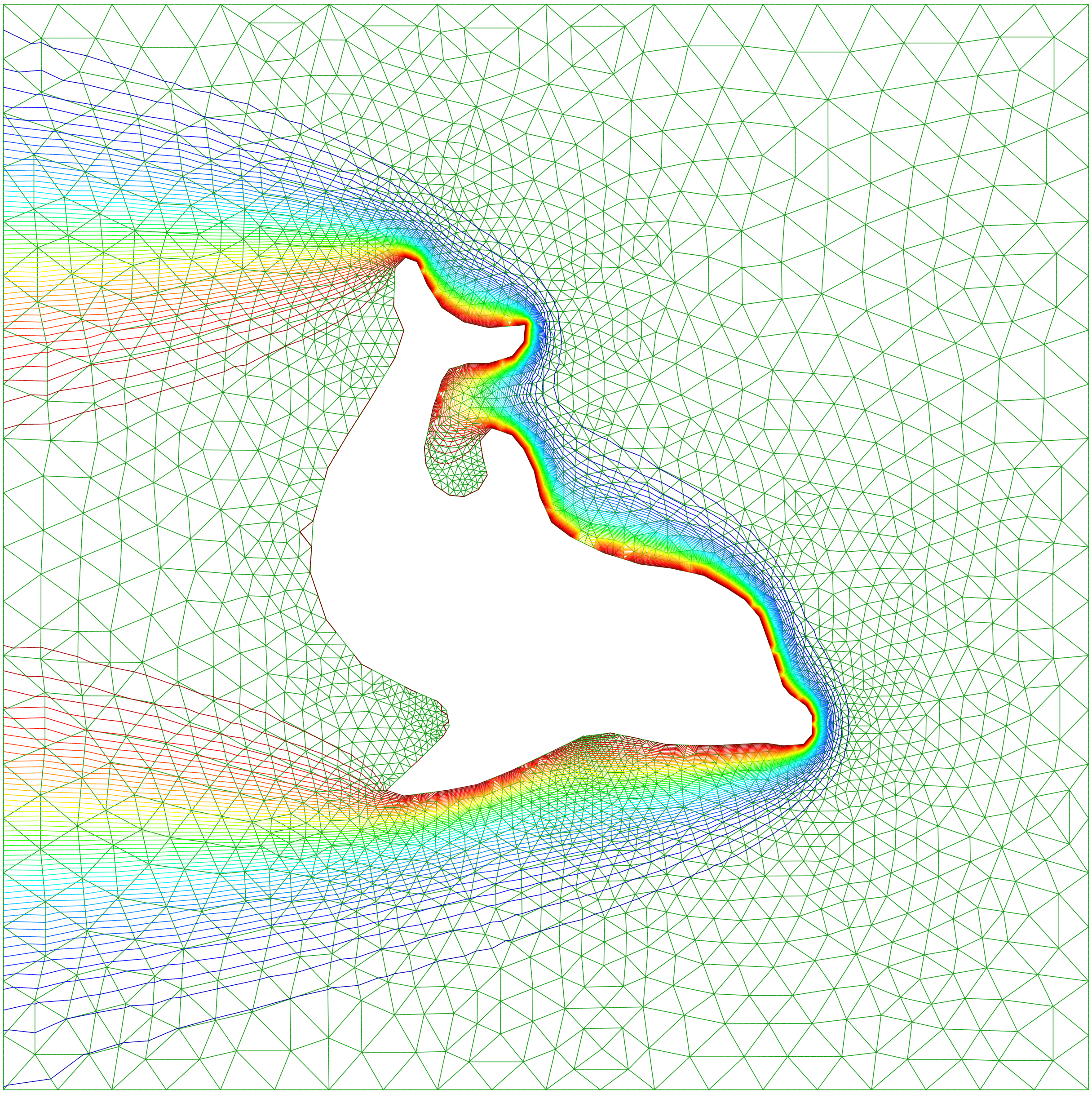

FIGURE 4. Convection–diffusion around a dolphin. The grid was generated with MATLAB:s PDE toolbox and the solution is visualised using *contour lines* in GiD.

7.3. **The incompressible Navier–Stokes equations.** The third example consists of the incompressible Navier–Stokes equations:

$$\begin{aligned} \dot{u} + u \cdot \nabla u - \nu \Delta u + \nabla p &= f, \\ \nabla \cdot u &= 0, \end{aligned} \tag{7.3}$$

from which we want to compute the velocity field u and the pressure p. The viscosity is given by ν and f is a given force term. In Figure 5 we plot isosurfaces for the stream-wise velocity in a computation of transition to turbulence for a jet flow with periodic boundary conditions, on a mesh consisting of 390,000 tetrahedral elements as described in [8].

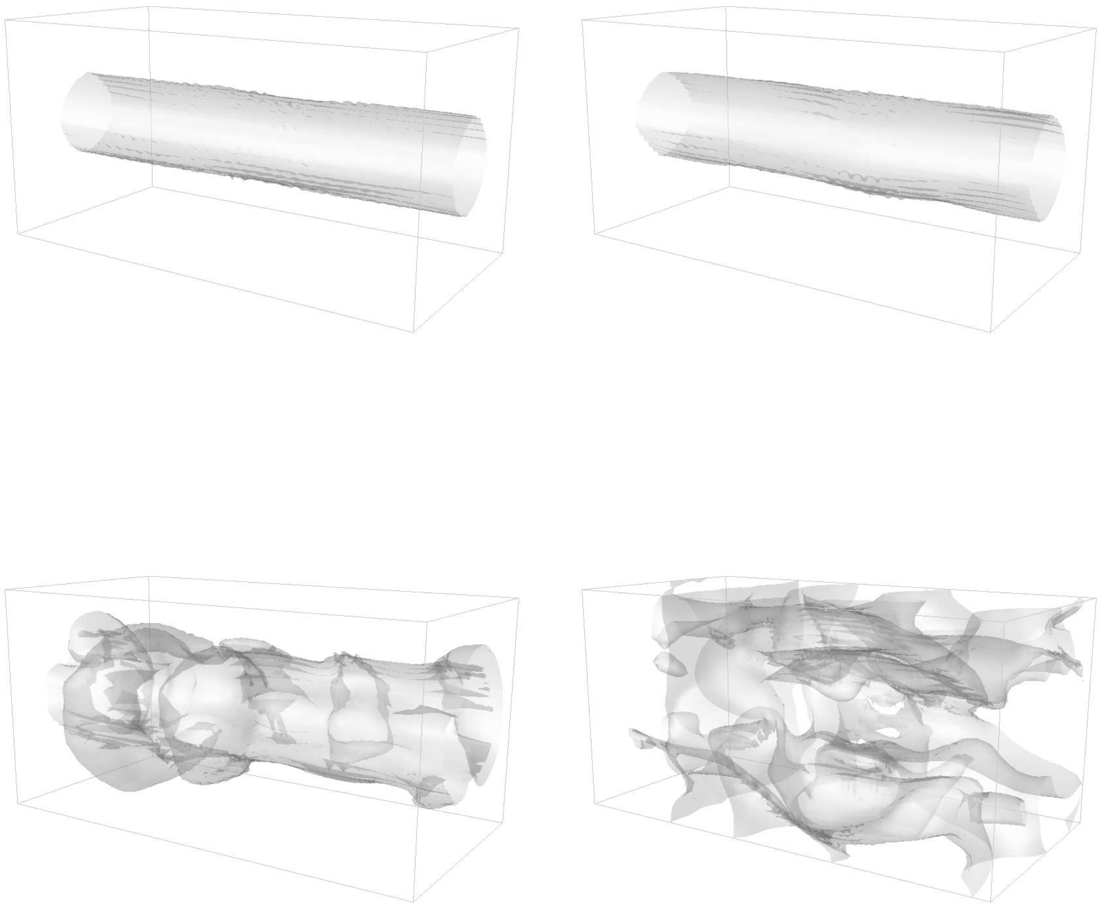

FIGURE 5. Transition to turbulence in a jet: isosurfaces for the stream-wise velocity. Visualisation using OpenDX.

References

[1] *Gid – the personal pre and postprocessor*, `http://gid.cimne.upc.es/`, (2002).
[2] *Gnu gpl*, `http://www.gnu.org/`, (2002).
[3] *The mathworks*, `http://www.mathworks.com/`, (2002).
[4] *Open visualisation data explorer*, `http://www.opendx.org/`, (2002).
[5] S. Brenner and L. Scott, *The Mathematical Theory of Finite Element Methods*, Springer-Verlag, New York, 1994.
[6] P. Ciarlet, *The Finite Element Method for Elliptic Problems*, North-Holland, 1978.
[7] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson, *Introduction to adaptive methods for differential equations*, in Acta Numerica 1995, Cambridge University Press, 1995, pp. 105–158.
[8] J. Hoffman and C. Johnson, *A computational study of transition to turbulence in shear flow*, Chalmers Finite Element Center Preprint 2002–04, (2002).
[9] J. Hoffman and A. Logg, *Dolfin, version 0.2.9*, `http://www.phi.chalmers.se/dolfin/`, (2002).

Chalmers Finite Element Center Preprints

2001–01 *A simple nonconforming bilinear element for the elasticity problem*
Peter Hansbo and Mats G. Larson

2001–02 *The $\mathcal{LL}^*$ finite element method and multigrid for the magnetostatic problem*
Rickard Bergström, Mats G. Larson, and Klas Samuelsson

2001–03 *The Fokker-Planck operator as an asymptotic limit in anisotropic media*
Mohammad Asadzadeh

2001–04 *A posteriori error estimation of functionals in elliptic problems: experiments*
Mats G. Larson and A. Jonas Niklasson

2001–05 *A note on energy conservation for Hamiltonian systems using continuous time finite elements*
Peter Hansbo

2001–06 *Stationary level set method for modelling sharp interfaces in groundwater flow*
Nahidh Sharif and Nils-Erik Wiberg

2001–07 *Integration methods for the calculation of the magnetostatic field due to coils*
Marzia Fontana

2001–08 *Adaptive finite element computation of 3D magnetostatic problems in potential formulation*
Marzia Fontana

2001–09 *Multi-adaptive galerkin methods for ODEs I: theory & algorithms*
Anders Logg

2001–10 *Multi-adaptive galerkin methods for ODEs II: applications*
Anders Logg

2001–11 *Energy norm a posteriori error estimation for discontinuous Galerkin methods*
Roland Becker, Peter Hansbo, and Mats G. Larson

2001–12 *Analysis of a family of discontinuous Galerkin methods for elliptic problems: the one dimensional case*
Mats G. Larson and A. Jonas Niklasson

2001–13 *Analysis of a nonsymmetric discontinuous Galerkin method for elliptic problems: stability and energy error estimates*
Mats G. Larson and A. Jonas Niklasson

2001–14 *A hybrid method for the wave equation*
Larisa Beilina, Klas Samuelsson and Krister Åhlander

2001–15 *A finite element method for domain decomposition with non-matching grids*
Roland Becker, Peter Hansbo and Rolf Stenberg

2001–16 *Application of stable FEM-FDTD hybrid to scattering problems*
Thomas Rylander and Anders Bondeson

2001–17 *Eddy current computations using adaptive grids and edge elements*
Y. Q. Liu, A. Bondeson, R. Bergström, C. Johnson, M. G. Larson, and K. Samuelsson

2001–18 *Adaptive finite element methods for incompressible fluid flow*
Johan Hoffman and Claes Johnson

2001–19 *Dynamic subgrid modeling for time dependent convection–diffusion–reaction equations with fractal solutions*
Johan Hoffman

These preprints can be obtained from

`www.phi.chalmers.se/preprints`