DOLFIN User Manual

October 26, 2006

Hoffman, Jansson, Logg, Wells

www.fenics.org

Visit `http://www.fenics.org/` for the latest version of this manual.
Send comments and suggestions to `dolfin-dev@fenics.org`.

Contents

About this manual

This manual is currently being written. As a consequence, some sections may be incomplete or inaccurate.

Intended audience

This manual is written both for the beginning and the advanced user. There is also some useful information for developers. More advanced topics are treated at the end of the manual or in the appendix.

Typographic conventions

- Code is written in monospace (typewriter) `like this`.
- Commands that should be entered in a Unix shell are displayed as follows:

  ```
  # ./configure
  # make
  ```

  Commands are written in the dialect of the `bash` shell. For other shells, such as `tcsh`, appropriate translations may be needed.

Enumeration and list indices

Throughout this manual, elements x_i of sets $\{x_i\}$ of size n are enumerated from $i = 0$ to $i = n - 1$. Derivatives in $\mathbb{R}^n$ are enumerated similarly: $\frac{\partial}{\partial x_0}, \frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_{n-1}}$.

Contact

Comments, corrections and contributions to this manual are most welcome and should be sent to

`dolfin-dev@fenics.org`

Chapter 1

Quickstart

This chapter demonstrates how to get started with **DOLFIN**, including downloading and installing the latest version of **DOLFIN**, and solving Poisson's equation. These topics are discussed in more detail elsewhere in this manual. In particular, see Appendix C for detailed installation instructions and Chapter 6 for a detailed discussion of how to solve partial differential equations with **DOLFIN**.

1.1 Downloading and installing DOLFIN

The latest version of **DOLFIN** can be found on the **FEniCS** web page:

```
http://www.fenics.org/
```

The following commands illustrate the installation process, assuming that you have downloaded release 0.1.0 of **DOLFIN**:

```
# tar zxfv dolfin-0.1.0.tar.gz
# cd dolfin-0.1.0
# ./configure
```

```
# make
# make install
```

Note that you may need to be root on your system to perform the last step.[1] Since **DOLFIN** depends on a number of other packages, you may also need to download and install these packages before you can compile **DOLFIN**. (See Appendix C for detailed instructions.)

1.2 Solving Poisson's equation with DOLFIN

Let's say that we want to solve Poisson's equation on the unit square $\Omega = (0,1) \times (0,1)$ with homogeneous Dirichlet boundary conditions on the boundary $\Gamma_0 = \{(x,y) \in \partial\Omega : x = 0\}$, the Neumann boundary condition $\partial_n u = 1$ on the boundary $\Gamma_1 = \{(x,y) \in \partial\Omega : x = 1\}$, homogeneous Neumann boundary conditions on the remaining part of the boundary and right-hand side given by $f(x,y) = 500\exp(-((x-0.5)^2 + (y-0.5)^2)/0.02)$, corresponding to a source localized at $x = y = 0.5$:

$$\begin{aligned} -\Delta u(x,y) &= f(x,y), \quad x \in \Omega = (0,1) \times (0,1), && (1.1)\\ u(x,y) &= 0, \quad (x,y) \in \Gamma_0 = \{(x,y) \in \partial\Omega : x = 0\}, && (1.2)\\ \partial_n u(x,y) &= 1, \quad (x,y) \in \Gamma_1 = \{(x,y) \in \partial\Omega : x = 1\}, && (1.3)\\ \partial_n u(x,y) &= 0, \quad (x,y) \in \partial\Omega \setminus (\Gamma_0 \cup \Gamma_1). && (1.4) \end{aligned}$$

To solve a partial differential equation with **DOLFIN**, it must first be rewritten in *variational form*. The (discrete) variational formulation of Poisson's equation reads: Find $U \in V_h$ such that

$$a(v, U) = L(v) \quad \forall v \in \hat{V}_h, \tag{1.5}$$

with $(\hat{V}_h, V_h)$ a pair of suitable discrete function spaces (the test and trial

[1] To install **DOLFIN** locally on a normal user account, configure **DOLFIN** to use another installation directory, for example `./configure --prefix=~/local`. Alternatively, you may use the command `./configure.local` to install **DOLFIN** locally in a subdirectory of the source tree.

spaces). The bilinear form $a : \hat{V}_h \times V_h \to \mathbb{R}$ is given by

$$a(v, U) = \int_\Omega \nabla v \cdot \nabla U \, \mathrm{d}x \tag{1.6}$$

and the linear form $L : \hat{V}_h \to \mathbb{R}$ is given by

$$L(v) = \int_\Omega v f \, \mathrm{d}x + \int_{\partial\Omega} v g \, \mathrm{d}s, \tag{1.7}$$

where $g = \partial_n u$ is the Neumann boundary condition.

1.2.1 Setting up the variational formulation

The variational formulation (1.5) must be given to **DOLFIN** as a pair of bilinear and linear forms (a, L) using the form compiler **FFC**. This is done by entering the definition of the forms in a text file with extension `.form`, e.g. `Poisson.form`, as follows:

```
element = FiniteElement(``Lagrange'', ``triangle'', 1)

v = TestFunction(element)
U = TrialFunction(element)
f = Function(element)
g = Function(element)

a = dot(grad(v), grad(U))*dx
L = v*f*dx + v*g*ds
```

The example is given here for piecewise linear finite elements in two dimensions, but other choices are available, including arbitrary order Lagrange elements in two and three dimensions.

To compile the pair of forms (a, L), now call the form compiler on the command-line as follows:

```
# ffc Poisson.form
```

This generates the file `Poisson.h` which implements the forms in C++ for inclusion in your **DOLFIN** program.

1.2.2 Writing the solver

Having compiled the variational formulation (1.5) with **FFC**, it is now easy to implement a solver for Poisson's equation. We first discuss the implementation line by line and then present the complete program. The source code for this example is available in the directory `src/demo/pde/poisson/` of the **DOLFIN** source tree.

At the beginning of our C++ program, which we write in a text file named `main.cpp`, we must first include the header file `dolfin.h`, which gives our program access to the **DOLFIN** class library. In addition, we include the header file `Poisson.h` generated by the form compiler. Since all classes in the **DOLFIN** class library are defined within the namespace `dolfin`, we also specify that we want to work within this namespace:

```
#include <dolfin.h>
#include ``Poisson.h''

using namespace dolfin;
```

Since we are writing a C++ program, we need to create a `main` function. You are free to organize your program any way you like, but in this simple example we just write our program inside the `main` function:

```
int main()
{
  // Write your program here
  return 0;
}
```

We now proceed to specify the right-hand side f of (1.1). This is done by defining a new subclass of `Function` and overloading the `eval()` function to return the value $f(x, y) = 500 \exp(-((x - 0.5)^2 + (y - 0.5)^2)/0.02)$:

```
class Source : public Function
{
  real eval(const Point& p, unsigned int i)
  {
    real dx = p.x - 0.5;
    real dy = p.y - 0.5;
    return 500.0*exp(-(dx*dx + dy*dy)/0.02);
  }
};
```

The Dirichlet boundary condition is specified similarly, by overloading the `eval()` function for a subclass of `BoundaryCondition`:

```
class DirichletBC : public BoundaryCondition
{
  void eval(BoundaryValue& value, const Point& p, unsigned int i)
  {
    if ( std::abs(p.x - 0.0) < DOLFIN_EPS )
      value = 0.0;
  }
};
```

The Neumann boundary conditions is specified as a `Function` that gets integrated over the boundary $\partial\Omega$ of Ω:

```
class NeumannBC : public Function
{
  real eval(const Point& p, unsigned int i)
  {
    if ( std::abs(p.x - 1.0) < DOLFIN_EPS )
      return 1.0;
    else
      return 0.0;
  }
};
```

We may now create the right-hand side, the Dirichlet and the Neumann boundary conditions as follows:

```
Source f;
DirichletBC bc;
NeumannBC g;
```

Next, we need to create a mesh. **DOLFIN** relies on external programs for mesh generation, and imports meshes in **DOLFIN** XML format. However, for simple domains like the unit square or unit cube, **DOLFIN** provides a built-in mesh generator. To generate a uniform mesh of the unit square with mesh size 1/16 (with a total of $2 \cdot 16^2 = 512$ triangles), we can just type

```
UnitSquare mesh(16, 16);
```

Next, we initialize the pair of bilinear and linear forms that we have previously compiled with **FFC**:

```
Poisson::BilinearForm a;
Poisson::LinearForm L(f, g);
```

Note that the right-hand side `f` and the Neumann boundary condition `g` need to be given as arguments to the constructor of the linear form, since the linear form depends on these two functions.

We may now define a `PDE` from the pair of forms, the mesh and the Dirichlet boundary condition:

```
PDE pde(a, L, mesh, bc);
```

To solve the PDE, we now just need to call the `solve` function as follows:

```
Function U = pde.solve();
```

Finally, we export the solution `U` to a file for visualization. Here, we choose to save the solution in VTK format for visualization in ParaView or MayaVi, which we do by specifying a file name with extension `.pvd`:

```
File file(‘‘poisson.pvd’’);
file << u;
```

The complete program for Poisson’s equation now looks as follows:

```
#include <dolfin.h>
#include "Poisson.h"

using namespace dolfin;

int main()
{
  // Right-hand side
  class Source : public Function
  {
    real eval(const Point& p, unsigned int i)
    {
      real dx = p.x - 0.5;
      real dy = p.y - 0.5;
      return 500.0*exp(-(dx*dx + dy*dy)/0.02);
    }
  };

  // Dirichlet boundary condition
  class DirichletBC : public BoundaryCondition
  {
    void eval(BoundaryValue& value, const Point& p, unsigned int i)
    {
      if ( std::abs(p.x - 0.0) < DOLFIN_EPS )
        value = 0.0;
    }
  };

  // Neumann boundary condition
  class NeumannBC : public Function
  {
    real eval(const Point& p, unsigned int i)
    {
      if ( std::abs(p.x - 1.0) < DOLFIN_EPS )
```

```
        return 1.0;
      else
        return 0.0;
    }
  };

  // Set up problem
  Source f;
  DirichletBC bc;
  NeumannBC g;
  UnitSquare mesh(16, 16);
  Poisson::BilinearForm a;
  Poisson::LinearForm L(f, g);
  PDE pde(a, L, mesh, bc);

  // Compute solution
  Function U = pde.solve();

  // Save solution to file
  File file("poisson.pvd");
  file << U;

  return 0;
}
```

1.2.3 Compiling the program

The easiest way to compile the program is to create a `Makefile` that tells the standard Unix command `make` how to build the program. The following example shows how to write a `Makefile` for the above example:

```
CFLAGS  = `dolfin-config --cflags_dolfin`
LIBS    = `dolfin-config --libs_dolfin`
CXX     = `dolfin-config --compiler`

DEST    = dolfin-poisson
OBJECTS = main.o

all: $(DEST)
```

```
clean:
        -rm -f *.o core *.core $(OBJECTS) $(DEST)

$(DEST): $(OBJECTS)
         $(CXX) -o $@ $(OBJECTS) $(CFLAGS) $(LIBS)

.cpp.o:
        $(CXX) $(CFLAGS) -c $<
```

With the `Makefile` in place, we just need to type `make` to compile the program, generating the executable as the file `dolfin-poisson`. Note that this requires that the script `dolfin-config` is in your `PATH`. During `make install` of **DOLFIN**, this script will be automatically generated and placed in the subdirectory `bin` of the installation directory you have chosen for **DOLFIN**, which means you need to make sure this subdirectory is in your `PATH`.[2]

1.2.4 Running the program

To run the program, simply type the name of the executable:

```
# ./dolfin-poisson
Computing mesh connectivity:
Found 289 vertices
Found 512 cells
[...]
Solving linear system of size 289 x 289 (UMFPACK LU solver).
Saved function u (no description) to file poisson.pvd [...]
```

1.2.5 Visualizing the solution

DOLFIN relies on external programs for visualization. In this example, we chose to save the solution in VTK format, which can be imported into for

[2]You will be reminded of this during the initial configuration of **DOLFIN**.

example ParaView or MayaVi. A simple way to visualize the computed solution is to run the Python script `plot.py` which is present in the directory of the Poisson demo (and most other demos):

```
python plot.py
```

This script calls MayaVi to visualize the solution.

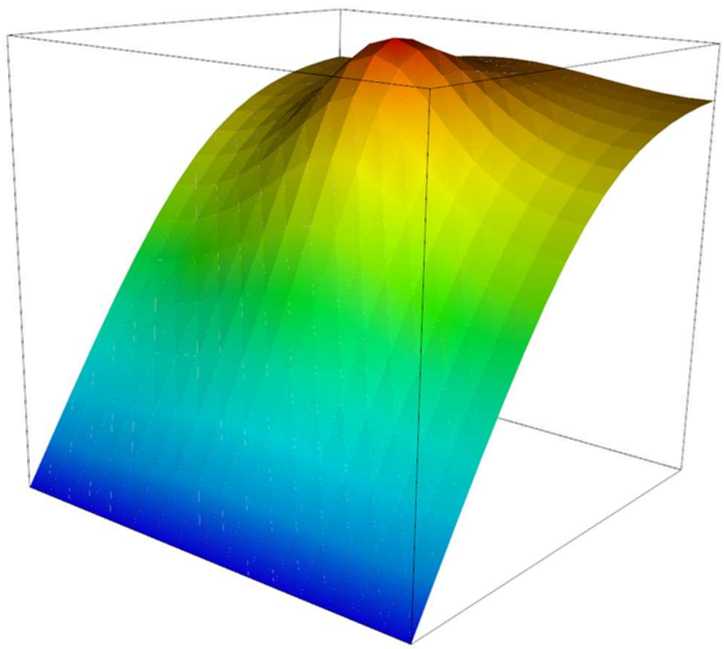

Figure 1.1: The solution of Poisson's equation (1.1) visualized in MayaVi.

Chapter 2

Linear algebra

DOLFIN provides a high-performance linear algebra library, including matrices and vectors, a set of linear solvers, preconditioners, and an eigenvalue solver. The core part of the functionality is provided through a wrappers that provide a common interface to the functionality of the linear algebra libraries uBlas [14] and PETSc [10].

2.1 Matrices and vectors

The two basic linear algebra data structures are the classes `Matrix` and `Vector`, representing a (sparse) $M \times N$ matrix and a vector of length N respectively.

The following code demonstrates how to create a matrix and a vector:

```
Matrix A(M, N);
Vector x(N);
```

Alternatively, the matrix and the vector may be created by

```
Matrix A;
```

```
Vector x;

A.init(M, N);
x.init(N);
```

The following code demonstrates how to access the size and the elements of a matrix and a vector:

```
A(5, 5) = 1.0;
real a = A(4, 3);

x(3) = 2.0;
real b = x(5);

unsigned int M = A.size(0);
unsigned int N = A.size(1);

N = x.size();
```

In addition, **DOLFIN** provides optimized functions for setting the values of a set of entries in a matrix or vector:

```
real block[] = {2, 4, 6};
int rows[] = {0, 1, 2};
int cols[] = {0, 1, 2};

A.set(block, rows, cols, 3);
```

Alternatively, the set of values given by the array `block` can be added to the entries given by the arrays `rows` and `cols`:

```
real block[] = {2, 4, 6};
int rows[] = {0, 1, 2};
int cols[] = {0, 1, 2};

A.add(block, rows, cols, 3);
```

These functions are particularly useful for efficient assembly of a (sparse) matrix from a bilinear form.

2.1.1 Sparse matrices

The default **DOLFIN** class `Matrix` is a sparse matrix, which efficiently represents the discretization of a partial differential equation where most entries in the matrix are zero. Alternatively, the class `SparseMatrix` may be used which is identical with the class `Matrix`[1].

If **DOLFIN** has been compiled with support for PETSc, then the sparse matrix is represented as a sparse PETSc matrix[2]. Alternatively, the class `PETScMatrix` may be used, together with the corresponding class `PETScVector`.

If **DOLFIN** has been compiled without support for PETSc, then the sparse matrix is represented as a uBlas sparse matrix. Alternatively, the class `uBlasSparseMatrix` may be used, together with the corresponding class `uBlasVector`.

2.1.2 Dense matrices

DOLFIN provides the class `DenseMatrix` for representation of dense matrices. A dense matrix representation is often preferable when computing with matrices of small to moderate size. In particular, accessing individual elements (and solving linear systems with a direct solver) is more efficient with a dense matrix representation.

A `DenseMatrix` is represented as uBlas dense matrix and alternatively the class `uBlasDenseMatrix` may be used, together with the corresponding class `uBlasVector`.

[1] The class `Matrix` is a `typedef` for the class `SparseMatrix`.

[2] By default, the sparse matrix is represented as a PETSc `MATSEQAIJ` matrix, but other PETSc representations are also available.

2.1.3 The common interface

Although **DOLFIN** differentiates between sparse and dense data structures, the two classes `GenericMatrix` and `GenericVector` define a common interface to all matrices and vectors. Refer to the *DOLFIN programmer's reference* for the exact specification of these interfaces.

2.2 Solving linear systems

DOLFIN provides a set of efficient solvers for linear systems of the form

$$Ax = b, \tag{2.1}$$

where A is an $N \times N$ matrix and where x and b are vectors of length N. Both iterative (Krylov subspace) solvers and direct (LU) solvers are provided.

2.2.1 Iterative methods

A linear system may be solved by the GMRES Krylov method as follows:

```
Matrix A;
Vector x, b:

GMRES::solve(A, x, b);
```

Alternatively, the linear system may be solved by first creating an object of the class `KrylovSolver`, which is more efficient for repeated solution of linear systems and also allows the specification of both the Krylov method and the preconditioner:

```
KrylovSolver solver(gmres, ilu);
solver.solve(A, x, b);
```

For uBlas matrices and vectors, the class `uBlasKrylovSolver` may be used and for PETSc matrices and vectors, the class `PETScKrylovSolver` may be used.

Krylov methods

DOLFIN provides the following set of Krylov methods:

`cg`	The conjugate gradient method
`gmres`	The GMRES method
`bicgstab`	The stabilized biconjugate gradient squared method
`default_method`	Default choice of Krylov method

Preconditioners

DOLFIN provides the following set of preconditioners:

`none`	No preconditioning
`jacobi`	Simple Jacobi preconditioning
`sor`	SOR, successive over-relaxation
`ilu`	Incomplete LU factorization
`icc`	Incomplete Cholesky factorization
`amg`	Algebraic multigrid (through Hypre when available)
`default_pc`	Default choice of preconditioner

Matrix-free solvers

The **DOLFIN** Krylov solvers may be used without direct access to a matrix representation. All that is needed is to provide the size of the linear system, the right-hand side, and a method implementing the multiplication of the matrix with any given vector.

Such a "virtual matrix" may be defined by implementing the interface defined by either the class `uBlasKrylovMatrix` of `PETScKrylovMatrix`. The matrix may then be used together with either the class `uBlasKrylovSolver` or `PETScKrylovSolver`.

2.2.2 Direct methods

A linear system may be solved by a direct LU factorization as follows:

```
Matrix A;
Vector x, b;

LU::solve(A, x, b);
```

Alternatively, the linear system may be solved by first creating an object of the class `LUSolver`, which may be more efficient for repeated solution of linear systems:

```
LUSolver solver;
solver.solve(A, x, b);
```

For uBlas matrices and vectors, the class `uBlasLUSolver` may be used and for PETSc matrices and vectors, the class `PETScLUSolver` may be used.

2.3 Solving eigenvalue problems

DOLFIN also provides a solver for eigenvalue problems. The solver is only available when **DOLFIN** has been compiled with support for PETSc and SLEPc [13].

For the basic eigenvalue problem

$$Ax = \lambda x, \tag{2.2}$$

the following code demonstrates how to compute the zeroth eigenpair:

```
PETScEigenvalueSolver esolver;
esolver.solve(A);

real lr, lc;
PETScVector xr, xc;
esolver.getEigenpair(lr, lc, xr, xc, 0);
```

The real and complex components of the eigenvalue are returned in `lr` and `lc`, respectively, and the real and complex parts of the eigenvector are returned in `xr` and `xc`, respectively.

For the generalized eigenvalue problem

$$Ax = \lambda Bx, \tag{2.3}$$

the following code demonstrates how to compute the second eigenpair:

```
PETScEigenvalueSolver esolver;
esolver.solve(A, B);

real lr, lc;
PETScVector xr, xc;
esolver.getEigenpair(lr, lc, xr, xc, 2);
```

2.4 Linear algebra backends

2.4.1 uBlas

uBlas is a C++ template library that provides BLAS level 1, 2 and 3 functionality (and more) for dense, packed and sparse matrices. The design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates.

DOLFIN wrappers for uBlas linear algebra is provided through the classes `uBlasSparseMatrix`, `uBlasDenseMatrix` and `uBlasVector`. These classes are implemented by subclassing the corresponding uBlas classes, which means that all standard `uBlas` operations are supported for these classes. For advanced usage not covered by the common **DOLFIN** interface specified by the classes `GenericMatrix` and `GenericVector`, refer directly to the documentation of `uBlas`.

2.4.2 PETSc

PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication.

DOLFIN wrappers for PETSc linear algebra is provided through the classes `PETScMatrix` and `PETScVector`. Direct access to the PETSc data structures is available through the member functions `mat()` and `vec()`, which return the PETSc `Mat` and `Vec` pointers respectively. For advanced usage not covered by the common **DOLFIN** interface specified by the classes `GenericMatrix` and `GenericVector`, refer directly to the documentation of PETSc.

Chapter 3

The mesh

► *Developer's note:* The **DOLFIN** mesh library has recently been reimplemented from scratch and replaces the old mesh library starting at **DOLFIN** version 0.6.3. The new mesh is simpler, faster and more general than the old mesh library, but adaptive mesh refinement is still missing from the old feature set. This will be added in the near future.

► *Developer's note:* This chapter is just a quick write-up of the most basic functionality of the mesh library and will be expanded.

3.1 Basic concepts

3.1.1 Mesh

A *mesh* consists of *mesh topology* and *mesh geometry*. These concepts are implemented by the classes `Mesh`, `MeshTopology` and `MeshGeometry`.

3.1.2 Mesh entities

A *mesh entity* is a pair (d, i), where d is the topological dimension of the mesh entity and i is a unique index of the mesh entity. Mesh entities are numbered within each topological dimension from 0 to $n_d - 1$, where n_d is the number of mesh entities of topological dimension d.

For convenience, mesh entities of topological dimension 0 are referred to as *vertices*, entities of dimension 1 *edges*, entities of dimension 2 *faces*, entities of *codimension* 1 *facets* and entities of codimension 0 *cells*. These concepts are summarized in Table 3.1.

Entity	Dimension	Codimension
Vertex	0	–
Edge	1	–
Face	2	–
Facet	–	1
Cell	–	0

Table 3.1: Named mesh entities.

These concepts are implemented by the classes `MeshEntity`, `Vertex`, `Edge`, `Face`, `Facet`, `Cell`.

3.2 Mesh iterators

Algorithms operating on a mesh can often be expressed in terms of *iterators*. The mesh library provides the general iterator `MeshEntityIterator` for iteration over mesh entities, as well as the specialized mesh iterators `VertexIterator`, `EdgeIterator`, `FaceIterator`, `FacetIterator` and `CellIterator`.

The following code illustrates how to iterate over all incident (connected) vertices of all vertices of all cells of a given mesh:

```
for (CellIterator c(mesh); !c.end(); ++c)
  for (VertexIterator v0(c); !v0.end(); ++v0)
    for (VertexIterator v1(v0); !v1.end(); ++v1)
      cout << *v1 << endl;
```

This may alternatively be implemented using the general iterator `MeshEntityIterator` as follows:

```
unsigned int dim = mesh.topology().dim();
for (MeshEntityIterator c(mesh, dim); !c.end(); ++c)
  for (MeshEntityIterator v0(c, 0); !v0.end(); ++v0)
    for (MeshEntityIterator v1(v0, 0); !v1.end(); ++v1)
      cout << *v1 << endl;
```

3.3 Mesh functions

A `MeshFunction` represents a discrete function that takes a value on each mesh entity of a given topological dimension. A `MeshFunction` may for example be used to store a global numbering scheme for the entities of a (parallel) mesh, marking sub domains or boolean markers for mesh refinement.

3.4 Mesh refinement

A mesh may be refined uniformly as follows:

```
mesh.refine();
```

DOLFIN does currently not support adaptive mesh refinement, but this will be supported in future versions.

3.5 Working with meshes

3.5.1 Reading a mesh from file

A mesh may be loaded from a file, either by specifying the file name to the constructor of the class `Mesh`:

```
Mesh mesh(''mesh.xml'');
```

or by creating a `File` object and streaming to a `Mesh`:

```
File file(''mesh.xml'');
Mesh mesh;
file >> mesh;
```

A mesh may be stored to file as follows:

```
File file(''mesh.xml'');
Mesh mesh;
file << mesh;
```

The **DOLFIN** mesh XML format has changed in **DOLFIN** version 0.6.3. Meshes in the old XML format may be converted to the new XML format using the script `dolfin-convert` included in the distribution of **DOLFIN**. For instructions, type `dolfin-convert --help`.

3.5.2 Extracting a boundary mesh

For any given mesh, a mesh of the boundary of the mesh (if any) may be created as follows:

```
BoundaryMesh boundary(mesh);
```

A `BoundaryMesh` is itself a `Mesh` of the same geometrical dimension and has the topological dimension of the mesh minus one.

The computation of a boundary mesh may also provide mappings from the vertices of the boundary mesh to the corresponding vertices in the original mesh, and from the cells of the boundary mesh to the corresponding facets of the original mesh:

```
MeshFunction<unsigned int> vertex_map,
MeshFunction<unsigned int> cell_map;
BoundaryMesh boundary(mesh, vertex_map, cell_map);
```

3.5.3 Built-in meshes

DOLFIN provides functionality for creating simple meshes, such as the mesh of the unit square and the unit cube. The following code demonstrates how to create a 16×16 triangular mesh of the unit square (consisting of $2\times16\times16 = 512$ triangles) and a $16\times16\times16$ tetrahedral mesh of the unit cube (consisting of $6 \times 16 \times 16 \times 16 = 24576$ tetrahedra):

```
UnitSquare mesh2D(16, 16);
UnitCube mesh3D(16, 16, 16);
```

► *Developer's note:* We could easily add other built-in meshes, like the unit interval, the unit disc, the unit sphere, rectangles, blocks etc. Any contributions are welcome.

3.5.4 Creating meshes

Simplicial meshes (meshes consisting of intervals, triangles or tetrahedra) may be constructed explicitly by specifying the cells and vertices of the mesh. A specialized interface for creating simplicial meshes is provided by the class `MeshEditor`. The following code demonstrates how to create a very simple mesh consisting of two triangles covering the unit square:

```
Mesh mesh;
MeshEditor editor(mesh, CellType::triangle, 2, 2);
editor.initVertices(4);
editor.initCells(2);
editor.addVertex(0, 0.0, 0.0);
editor.addVertex(1, 1.0, 0.0);
editor.addVertex(2, 1.0, 1.0);
editor.addVertex(3, 0.0, 1.0);
editor.addCell(0, 0, 1, 2);
editor.addCell(1, 0, 2, 3);
editor.close();
```

Note that the **DOLFIN** mesh library is not specialized to simplicial meshes, but supports general collections of mesh entities. However, tools like mesh refinement and mesh editors are currently only available for simplicial meshes.

Chapter 4

Functions

► *Developer's note:* Since this chapter was written, the `Function` class has seen a number of improvements which are not covered here. Chapter needs to be updated.

The central concept of a function on a domain $\Omega \subset \mathbb{R}^d$ is modeled by the class `Function`, which is used in **DOLFIN** to represent coefficients or solutions of partial differential equations.

4.1 Basic properties

The following basic properties hold for all `Function`s:

- A `Function` can be scalar or vector-valued;
- A `Function` can be evaluated at each `Vertex` of a `Mesh`;
- A `Function` can be restricted to each local `Cell` of a `Mesh`;
- The underlying representation of a `Function` may vary.

Depending on the actual underlying representation of a `Function`, it may also be possible to evaluate a `Function` at any given `Point`.

4.1.1 Representation

Currently supported representations of `Function`s include *discrete* `Function`s and *user-defined* `Function`s. These are discussed in detail below.

4.1.2 Evaluation

All `Functions` can be evaluated at the `Vertices` of a `Mesh`. The following example illustrates how to evaluate a scalar `Function` at each `Vertex` of a given `Mesh`:

```
Function u;
Mesh mesh;

for (VertexIterator vertex(mesh); !vertex.end(); ++vertex)
  cout << ``Value at vertex `` << *vertex << ``: ``
       << u(*vertex) << endl;
```

If the `Function` is vector-valued, an additional argument is needed to specify the component. The following example illustrates how to evaluate all components of a vector-valued `Function` at all each `Vertex` of a given `Mesh`:

```
Function u;
Mesh mesh;

for (VertexIterator vertex(mesh); !vertex.end(); ++vertex)
  for (unsigned int i = 0; i < u.vectordim(); i++)
    cout << ``Value of component `` << i << `` at vertex ``
         << *vertex << ``: `` << u(*vertex, i) << endl;
```

If allowed by the underlying representation, a `Function u` may also be evaluated directly at any given `Point`:

```
Point p(0.5, 0.5, 0.5);
cout << ``Value at p = `` << p << ``: `` << u(p) << endl;
```

As in the case of evaluation at a `Vertex`, the component index may be given as an additional argument for a vector-valued `Function`.

4.1.3 Assignment

One `Function` may be assigned to another `Function`:

```
Function v;
Function u = v;
```

Assignment creates a new `Function` sharing the same data. In particular, this means that modifying the data of one of the two `Function`s will also affect the other `Function`.

4.1.4 Components and sub functions

If a `Function` is vector-valued, a new `Function` may be created to represent any given component of the original `Function`, as illustrated by the following example:

```
Function u;         // Function with three components
Function u0 = u[0]; // first component
Function u1 = u[1]; // second component
Function u2 = u[2]; // third component
```

If a `Function` represents a *mixed* function (one defined in terms of a mixed `FiniteElement`, see below), then indexing has the effect of picking out sub functions. With `w` a `Function` representing the solution $w = (u, p)$ of a Stokes or Navier-Stokes system (with u the vector-valued velocity and p the scalar pressure), the following example illustrates how to pick sub functions and components of `w`:

```
Function w; // mixed Function (u, p)
u = w[0];   // first sub function (velocity)
p = w[1];   // second sub function (pressure)
u0 = u[0];  // first component of the velocity
u1 = u[1];  // second component of the velocity
u2 = u[2];  // third component of the velocity
```

Note that picking a component or sub function creates a new `Function` that shares data with the original `Function`.

4.1.5 Output

A `Function` can be written to a file in various file formats. To write a `Function u` to file in VTK format, suitable for viewing in ParaView or MayaVi, create a file with extension `.pvd`:

```
File file(‘‘solution.pvd’’);
file << u;
```

For further details on available file formats, see Chapter 8.

4.2 Discrete functions

A *discrete* `Function` is defined in terms of a `Vector` of nodal values (degrees of freedom), a `Mesh` and a `FiniteElement` specifying the distribution of the nodal values on the `Mesh`. In particular, a discrete `Function` is given by a linear combinations of basis functions:

$$v = \sum_{i=1}^{N} v_i \phi_i, \tag{4.1}$$

where $\{\phi_i\}_{i=1}^N$ is the global basis of the finite element space defined by the `Mesh` and the `FiniteElement`, and the nodal values $\{v_i\}_{i=1}^N$ are given by the values of a `Vector`.

Note that a *discrete* `Function` may not be evaluated at arbitrary points (only at each `Vertex` of a `Mesh`).

4.2.1 Creating a discrete function

A discrete `Function` can be initialized in several ways. In the simplest case, only a `Vector x` of nodal values needs to be specified:

```
Vector x;

Function u(x);
```

If possible, **DOLFIN** will then automatically try to determine the `Mesh` and the `FiniteElement`.

In some cases, it is necessary to also supply a `Mesh` when initializing a discrete `Function`:

```
Vector x;
Mesh mesh;

Function u(x, mesh);
```

If possible, **DOLFIN** will then automatically try to determine the `FiniteElement`.

In general however, a discrete `Function` must be initialized from a given `Vector`, a `Mesh` and a `FiniteElement`:

```
Vector x;
Mesh mesh;
FiniteElement element;

Function u(x, mesh, element);
```

4.2.2 Accessing discrete function data

It is possible to access the data of a discrete `Function`, including the associated `Vector`, `Mesh` and `FiniteElement`:

```
Vector& x              = u.vector();
Mesh& mesh             = u.mesh();
FiniteElement& element = u.element();
```

4.2.3 Attaching discrete function data

After a discrete `Function` has been initialized, it is possible to associate or reassociate data with the `Function`:

```
Vector x;
Mesh mesh;
FiniteElement element;

Function u(x);
u.attach(mesh);
u.attach(element);
```

Usually, the `FiniteElement` is given by the `BilinearForm` defining the problem. Considering the Poisson example in Chapter 1, a `Function u` representing the solution can be initialized as follows:

```
Vector x;
Mesh mesh;
```

```
Function u(x, mesh);

Poisson::BilinearForm a;

FiniteElement& element = a.trial();
u.attach(element);
```

In this example, the `Function u` represents a function in the trial space for the `BilinearForm a`.

4.3 User-defined functions

In the simplest case, a user-defined `Function` is just an expression in terms of the coordinates and is typically used for defining source terms and initial conditions. For example, a source term could be given by

$$f = f(x, y, z) = xy \sin(z/\pi). \tag{4.2}$$

There are two ways to create a user-defined `Function`; either by creating a sub class of `Function` or by creating a `Function` from a given function pointer.

4.3.1 Creating a sub class

A user-defined `Function` may be defined by creating a sub class of `Function` and overloading the `eval()` function. The following example illustrates how to create a `Function` representing the function in (4.2):

```
class Source : public Function
{
  real eval(const Point& p, unsigned int i)
  {
    return x*y*sin(z / DOLFIN_PI);
```

```
  }
};

Source f;
```

To create a vector-valued `Function`, the vector dimension must be supplied to the constructor of `Function`:

```
class Source : public Function
{
public:

  Source() : Function(3) {}

  real eval(const Point& p, unsigned int i)
  {
    if ( i == 0 )
      return 0.0;
    else if ( i == 1 )
      return x*y*sin(z / DOLFIN_PI);
    else
      return x + y;
  }
};

Source f;
```

4.3.2 Specifying a function-pointer

A user-defined `Function` may alternatively be defined by specifying a function pointer. The following example illustrates an alternative way of creating a `Function` representing the function in (4.2):

```
real source(const Point& p, unsigned int i)
{
```

```
    return x*y*sin(z / DOLFIN_PI);
  }

  Function f(source);
```

As before, for vector-valued `Function`s, the vector dimension must be supplied to the constructor of `Function`:

```
  real source(const Point& p, unsigned int i)
  {
    if ( i == 0 )
      return 0.0;
    else if ( i == 1 )
      return x*y*sin(z / DOLFIN_PI);
    else
      return x + y;
  }

  Function f(source, 3);
```

4.3.3 Cell-dependent functions

In some cases, it may be convenient to define a `Function` in terms of properties of the current `Cell`. One such example is a `Function` that at any given point takes the value of the mesh size at that point.

The following example illustrates how to create such as `Function` by overloading the `eval()` function:

```
  class MeshSize : public Function
  {
    real eval(const Point& p, unsigned int i)
    {
      return cell().diameter();
    }
```

```
}

MeshSize h;
```

Note that the current `Cell` is only available during assembly and has no meaning otherwise. It is thus not possible to write the `Function h` to file, since the current `Cell` is not available when evaluating a `Function` at any given `Vertex`. Furthermore, note that the current `Cell` is not available when creating a `Function` from a function pointer.

4.4 Time-dependent functions

► *Developer's note:* Write about time-dependent and pseudo time-dependent functions.

Chapter 5

Ordinary differential equations

► *Developer's note:* This chapter needs to be written. In the meantime, look at the demos in `src/demo/ode/` and the base class `ODE`.

Chapter 6

Partial differential equations

▶ *Developer's note:* This chapter is incomplete and inaccurate and needs to be updated. In the meantime, look at the demos in `src/demo/pde/`.

DOLFIN provides a general interface for defining a partial differential equation (PDE) in variational form. The variational form is compiled with **FFC**, which generates code for the assembly of a matrix and a vector, corresponding to the discretization of the PDE with a user-defined finite element method (FEM), where the basis functions of the FEM space are constructed using **FIAT**.

6.1 Boundary value problems

As a prototype of a boundary value problem in $\mathbb{R}^d$ we consider the scalar Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\Delta u(x) &= f(x) \quad x \in \Omega \subset \mathbb{R}^d \\ u(x) &= 0 \quad x \in \partial\Omega. \end{aligned} \tag{6.1}$$

6.2 Variational formulation

A variational formulation of (6.1) takes the form: find $u \in V$ such that

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \tag{6.2}$$

where $a(\cdot, \cdot) : \hat{V} \times V \to \mathbb{R}$ is a bilinear form acting on $\hat{V} \times V$, with $\hat{V}$ and V the *test space* and *trial space* respectively, defined by

$$a(v, u) = \int_\Omega \nabla v \cdot \nabla u \; dx = \int_\Omega \frac{\partial v}{\partial x_i} \frac{\partial u}{\partial x_i} \; dx, \tag{6.3}$$

where we employ tensor notation so that the double index i means summation from $i = 1, ..., d$, and $L(\cdot) : \hat{V} \to \mathbb{R}$ is a linear form acting on the test space $\hat{V}$, defined by

$$L(v) = \int_\Omega v f \; dx. \tag{6.4}$$

For this problem we typically use $V = \hat{V} = H_0^1(\Omega)$, with $H_0^1(\Omega)$ the standard Sobolev space of square integrable functions with also their first derivatives square integrable (in the Lebesgue sense), with the functions being zero on the boundary (in the sense of traces).

The FEM method for (6.2) is now: find $U \in V_h$ such that

$$a(v, U) = L(v) \quad \forall v \in \hat{V}_h, \tag{6.5}$$

where $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$ are finite dimensional subspaces of dimension M. The finite element spaces $V_h, \hat{V}_h$ are characterized by their sets of basis functions $\{\varphi_i\}_{i=1}^M, \{\hat{\varphi}_i\}_{i=1}^M$. The FEM method (6.5) is thus specified by the variational form and the basis functions of V_h and $\hat{V}_h$.

6.3 Finite elements and FIAT

Finite element basis functions in **DOLFIN** are defined using **FIAT**, which supports the generation of arbitrary order Lagrange finite elements on lines, triangles, and tetrahedra. Upcoming versions of **FIAT** will also support Hermite and nonconforming elements as well as H(div) and H(curl) elements such as Raviart-Thomas and Nedelec.

6.4 Compiling the variational form with FFC

In **DOLFIN** a PDE is defined in variational form using tensor notation in a `.form` file, which is compiled using **FFC**.

In the language of **FFC**, with $V_h = \hat{V}_h$ the space of piecewise linear Lagrange finite elements on a tetrahedral mesh, (6.5) is defined as:

```
element = FiniteElement("Lagrange", "tetrahedron", 1)

v = BasisFunction(element)
u = BasisFunction(element)
f = Function(element)

a = v.dx(i)*u.dx(i)*dx
L = v*f*dx
```

where `*dx` signifies integration over the domain Ω, and the finite element space is constructed using **FIAT**. **DOLFIN** is not communicating directly with **FIAT**, but only through **FFC** in the definition of the variational form in the `.form` file.

Compiling the `.form` file with

```
# ffc Poisson.form
```

generates a file `Poisson.h`, containing classes for the bilinear form $a(\cdot,\cdot)$ and the linear form $L(\cdot)$, and classes for the finite element spaces V_h and $\hat{V}_h$.

6.5 Element matrices and vectors

The element matrices and vectors for a given cell may be factored into two tensors, with one tensor depending on the geometry of the cell, and the other tensor only involving integration of products of basis functions and their derivatives over the reference element.

For efficiency in the computation of the element matrices and vectors, **FFC** precomputes the tensors that are independent of the geometry of a certain cell.

6.6 Assemble matrices and vectors

The class `FEM` automates the assembly algorithm, constructing a linear system of equations from a PDE, given in the form of a variational problem (6.2), with a bilinear form $a(\cdot, \cdot)$ and a linear form $L(\cdot)$.

The classes `BilinearForm` and `LinearForm` are automatically generated by **FFC**, and to assemble the corresponding matrix and vector for the Poisson problem (6.2) with source term f, we write:

```
Poisson::BilinearForm a;
Poisson::LinearForm L(f);

Mesh mesh;
Mat A;
Vec b;

FEM::assemble(a,L,A,b,mesh);
```

In the `assemble()` function the element matrices and vectors are computed by calling the function `eval()` in the classes `Bilinearform` and `Linearform`. The `eval()` functions at a certain element in the assembly algorithm take as argument an `AffineMap` object, describing the mapping from the reference element to the actual element, by computing the Jacobian J of the mapping (also J^{-1} and $det(J)$ are computed).

6.7 Specifying boundary conditions and data

The boundary conditions are specified by defining a new subclass of the class `BoundaryCondition`, which we here will name `MyBC`:

```
class MyBC : public BoundaryCondition
{
  const BoundaryValue operator() (const Point& p)
  {
    BoundaryValue value;
    if ( std::abs(p.x - 0.0) < DOLFIN_EPS ) value = 0.0;
    if ( std::abs(p.x - 1.0) < DOLFIN_EPS ) value = 0.0;
    if ( std::abs(p.y - 0.0) < DOLFIN_EPS ) value = 0.0;
    if ( std::abs(p.y - 1.0) < DOLFIN_EPS ) value = 0.0;
    if ( std::abs(p.z - 0.0) < DOLFIN_EPS ) value = 0.0;
    if ( std::abs(p.z - 1.0) < DOLFIN_EPS ) value = 0.0;
      return value;
  }
};
```

where we have assumed homogeneous Dirichlet boundary conditions for the unit cube. We only need to specify the boundary conditions explicitly on the Dirichlet boundary. On the remaining part of the boundary, homogeneous Neumann boundary conditions are automatically imposed weakly by the variational form.

The boundary conditions are then imposed as an argument to the `assemble()` function:

```
MyBC bc;
FEM::assemble(a,L,A,b,mesh,bc);
```

There is currently no easy way to impose non-homogeneous Neumann boundary conditions or other combinations of boundary conditions. This will be added to a future release of **DOLFIN**.

The right-hand side f of (6.1) is similarly specified by defining a new subclass of `Function`, which we here will name `MyFunction`, and overloading the evaluation operator:

```
class MyFunction : public Function
{
  real operator() (const Point& p) const
  {
    return p.x*p.z*sin(p.y);
  }
};
```

with the source $f(x, y, z) = xz\sin(y)$.

6.8 Initial value problems

A time dependent problem has to be discretized in time manually, and the resulting variational form is then discretized in space using **FFC** similarly to a stationary problem, with the solution at previous time steps provided as data to the `form` file.

For example, the form file `Heat.form` for the heat equation discretized in time with the implicit Euler method, takes the form:

```
v           = BasisFunction(scalar) # test function
u1          = BasisFunction(scalar) # value at next time step
u0          = Function(scalar)      # value at previous time step
f           = Function(scalar)      # source term
k           = Constant()            # time step

a = v*u1*dx + k*v.dx(i)*u1.dx(i)*dx
L = v*u0*dx + v*f*dx
```

which generates a file `Heat.h` when compiled with **FFC**. To initializations the corresponding forms we write:

```
real k;                  // time step
Vector x0;               // vector containing dofs for u0
Function u0(x0, mesh);   // solution at previous time step

Heat::BilinearForm a(k);
Heat::LinearForm L(u0,f,k);
```

Chapter 7

Nonlinear solver

▶ *Developer's note:* This chapter is currently being written...

DOLFIN provides tools for solving nonlinear equations of the form

$$F(u) = 0 \tag{7.1}$$

where $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$. To use the nonlinear solver, a nonlinear function must be defined. The nonlinear solver is then initialised with this function and a solution computed.

7.1 Nonlinear functions

To solve a nonlinear problem, the user must defined a class which . The class should be derived from the **DOLFIN**class `NonlinearFunction`. The class should contain the necessary functions to form the function $F(u)$ and the Jacobian matrix $J = \partial F/\partial u$. The precise form of the user defined class will depend on the PDE being solved and the numerical method. The structure of a user defined class `MyNonlinearFunction` is shown below.

```
class MyNonlinearFunction : public NonlinearFunction
{
```

```
  public:

    // Constructor
    MyNonlinearFunction() : NonlinearFunction(){}

    // Compute F(u)
    void F(Vector& b, const Vector& x)
    {
      // Insert F(u) into the vector b
    }

    // Compute J
    void J(Matrix& A, const Vector& x)
    {
      // Insert the Jacobian into the matrix A
    }

    dolfin::uint size()
    {
      // Return the dimension of the Jacobian matrix
    }

    dolfin::uint nzsize()
    {
      // Return the maximum number of zeroes per row of the Jacobian
    }

  private:
    // Pointers to objects with which F(u) is defined
};
```

7.2 Newton solver

7.2.1 Linear solver

7.2.2 Application of Dirichlet boundary conditions

The application of inhomogeneous Dirichlet boundary conditions in the context of a Newton solver requires particular attention.

7.2.3 Newton solver parameters

7.2.4 Application of Dirichlet boundary conditions

7.3 Incremental Newton solver

Chapter 8

Input/output

DOLFIN relies on external programs for pre- and post-processing, which means that computational meshes must be imported from file (pre-processing) and computed solutions must be exported to file and then imported into another program for visualization (post-processing). To simplify this process, **DOLFIN** provides support for easy interaction with files and includes output formats for a number of visualization programs.

8.1 Files and objects

A file in **DOLFIN** is represented by the class `File` and reading/writing data is done using the standard C++ operators `>>` (read) and `<<` (write).

Thus, if `file` is a `File` and `object` is an object of some class that can be written to file, then the object can be written to file as follows:

```
file << object;
```

Similarly, if `object` is an object of a class that can be read from file, then data can be read from file (overwriting any previous data held by the object) as follows:

```
file >> object;
```

The format (type) of a file is determined by its filename suffix, if not otherwise specified. Thus, the following code creates a `File` for reading/writing data in **DOLFIN** XML format:

```
File file(``data.xml'');
```

A complete list of file formats and corresponding file name suffixes is given in Table 8.1.

Alternatively, the format of a file may be explicitly defined. One may thus create a file named `data.xml` for reading/writing data in GNU Octave format:

```
File file(``data.xml'', File::octave);
```

Suffix	Format	Description
`.xml`/`.xml.gz`	`File::xml`	**DOLFIN** XML
`.pvd`	`File::vtk`	VTK
`.dx`	`File::opendx`	OpenDX
`.m`	`File::octave`	GNU Octave
(`.m`)	`File::matlab`	MATLAB
`.tec`	`File::tecplot`	Tecplot
`.msh`/`.res`	`File::gid`	GiD

Table 8.1: File formats and corresponding file name suffixes.

Although many of the classes in **DOLFIN** support file input/output, it is not supported by all classes and the support varies with the choice of file format. A summary of supported classes/formats is given in Table 8.2.

► *Developer's note:* Some of the file formats are partly broken after changing the linear algebra backend to PETSc. (Do `grep FIXME` in `src/kernel/io/`.)

Format	`Vector`	`Matrix`	`Mesh`	`Function`	`Sample`
`File::xml`	in/out	in/out	in/out	—	—
`File::vtk`	—	—	out	out	—
`File::opendx`	—	—	out	out	—
`File::octave`	out	out	out	out	out
`File::matlab`	out	out	out	out	out
`File::tecplot`	—	—	out	out	—
`File::gid`	—	—	out	out	—

Table 8.2: Matrix of supported combinations of classes and file formats for input/output in **DOLFIN**.

8.2 File formats

In this section, we give some pointers to each of the file formats supported by **DOLFIN**. For detailed information, we refer to the respective user manual of each format/program.

► *Developer's note:* This section needs to be improved and expanded. Any contributions are welcome.

8.2.1 DOLFIN XML

DOLFIN XML is the native format of **DOLFIN**. As the name says, data is stored in XML ASCII format. This has the advantage of being a robust and human-readable format, and if the files are compressed there is little overhead in terms of file size compared to a binary format.

DOLFIN automatically handles gzipped XML files, as illustrated by the following example which reads a `Mesh` from a compressed **DOLFIN** XML file and saves the mesh to an uncompressed **DOLFIN** XML file:

```
Mesh mesh;
```

```
File in(``mesh.xml.gz'');
in >> mesh;

File out(``mesh.xml'');
out << mesh;
```

The same thing can of course be accomplished by

```
# gunzip -c mesh.xml.gz > mesh.xml
```

on the command-line.

There is currently no visualization tool that can read **DOLFIN** XML files, so the main purpose of this format is to save and transfer data.

8.2.2 VTK

Data saved in VTK format [12] can be visualized using various packages. The powerful and freely available ParaView [9] is recommended. Alternatively, VTK data can be visualized in MayaVi [4], which is recommended for quality vector PostScript output. Time-dependent data is handled automatically in the VTK format.

The below code illustrates how to export a function in VTK format:

```
Function u;

File out(``data.pvd'');
out << u;
```

The sample code produces the file `data.pvd`, which can be read by ParaView. The file `data.pvd` contains a list of files which contain the results computed by **DOLFIN**. For the above example, these files would be named `dataXXX.vtu`, where `XXX` is a counter which is incremented each time the function is saved. If the function `u` was to be saved three times, the files

```
data000000.vtu
data000001.vtu
data000002.vtu
```

would be produced. Individual snapshots can be visualized by opening the desired file with the extension `.vtu` using ParaView.

ParaView can produce on-screen animations. High quality animations in various formats can be produced using a combination of ParaView and MEncoder [5].

▶ *Developer's note:* Add MEncoder example to create animation.

8.2.3 OpenDX

OpenDX [8] is a powerful free visualization tool based on IBM's *Visualization Data Explorer*. To visualize data with OpenDX, a user needs to build a *visual program* that instructs OpenDX how to extract and visualize relevant parts of your data. **DOLFIN** provides a ready-made visual program suitable for visualization of **DOLFIN** data in OpenDX. The visual program can be found in the subdirectory `src/utils/opendx/` of the **DOLFIN** source tree (file `dolfin.net` and accompanying configuration `dolfin.cfg`).

8.2.4 GNU Octave

GNU Octave [6] is a free clone of MATLAB that can be used to visualize solutions computed in **DOLFIN**, using the commands `pdemesh`, `pdesurf` and `pdeplot`. These commands are normally not part of GNU Octave but are provided by **DOLFIN** in the subdirectory `src/utils/octave/` of the **DOLFIN** source tree. These commands require the external program `ivview` included in the open source distribution of Open Inventor [7]. (Debian users install the package `inventor-clients`.)

To visualize a solution computed with **DOLFIN** and exported in GNU Octave format, first load the solution into GNU Octave by just typing the name of the file without the `.m` suffix. If the solution has been saved to the file `poisson.m`, then just type

```
octave:1> poisson
```

The solution can now be visualized using the command

```
octave:2> pdesurf(points, cells, u)
```

or to visualize just the mesh, type

```
octave:3> pdesurf(points, edges, cells)
```

8.2.5 MATLAB

Since MATLAB [3] is not free, users are encouraged to use GNU Octave whenever possible. That said, data is visualized in much the same way in MATLAB as in GNU Octave, using the MATLAB commands `pdemesh`, `pdesurf` and `pdeplot`.

8.2.6 Tecplot

Tecplot [11] is a proprietary visualization tool. The Tecplot format is not actively maintained and may be removed in future versions of **DOLFIN** (if there is not sufficient interest to maintain the format).

8.2.7 GiD

GiD [2] is a proprietary visualization tool. The GiD format is not actively maintained and may be removed in future versions of **DOLFIN** (if there is not sufficient interest to maintain the format).

8.3 Converting between file formats

DOLFIN supplies a script for easy conversion between different file formats. The script is named `dolfin-convert` and can be found in the directory `src/utils/convert/` of the **DOLFIN** source tree. The only supported file formats are currently the Medit `.mesh` format (which can be generated by TetGen) and the **DOLFIN** XML mesh format:

```
# dolfin-convert mesh.mesh mesh.xml
```

8.4 A note on new file formats

With some effort, **DOLFIN** can be expanded with new file formats. Any contributions are welcome. If you wish to contribute to **DOLFIN**, then adding a new file format (or improving upon an existing file format) is a good place to start. Take a look at one of the current formats in the subdirectory `src/kernel/io/` of the **DOLFIN** source tree to get a feeling for how to design the file format, or ask at `dolfin-dev@fenics.org` for directions.

Also consider contributing to the `dolfin-convert` script by adding a conversion routine for your favorite format. The script is written in Python and should be easy to extend with new formats.

Chapter 9

The log system

DOLFIN provides provides a simple interface for uniform handling of log messages, including warnings and errors. All messages are collected to a single stream, which allows the destination and formatting of the output from an entire program, including the **DOLFIN** library, to be controlled by the user.

9.1 Generating log messages

Log messages can be generated using the function `dolfin_info()` available in the `dolfin` namespace:

```
void dolfin_info(const char *message, ...);
```

which works similarly to the standard C library function `printf`. The following examples illustrate the usage of `dolfin_info()`:

```
dolfin_info(``Solving linear system.'');
dolfin_info(``Size of vector: %d.'', x.size());
dolfin_info(``R = %.3e (TOL = %.3e)'', R, TOL);
```

As an alternative to `dolfin_info()`, **DOLFIN** provides a C++ style interface to generating log messages. Thus, the above examples can also be implemented as follows:

```
cout << ‘‘Solving linear system.’’ << endl;
cout << ‘‘Size of vector: ‘‘ << x.size() << ‘‘.’’ << endl;
cout << ‘‘R = ‘‘ << R << ‘‘ (TOL = ‘‘ << TOL << ‘‘)’’ << endl;
```

Note the use of `dolfin::cout` and `dolfin::endl` from the `dolfin` namespace, corresponding to the standard standard `std::cout` and `std::endl` in namespace `std`. If log messages are directed to standard output (see below), then `dolfin::cout` and `std::cout` may be mixed freely.

Most classes provided by **DOLFIN** can be used together with `dolfin::cout` and `dolfin::endl` to display short informative messages about objects:

```
Matrix A(10, 10);
cout << A << endl;
```

To display detailed information for an object, use the member function `disp()`:

```
Matrix A(10, 10);
A.disp();
```

Use with caution for large objects. For a `Matrix`, calling `disp()` will displays all matrix entries.

9.2 Warnings and errors

Warnings and error messages can be generated using the macros

```
dolfin_warning(message);
dolfin_error(message);
```

In addition to displaying the given string message, the macro `dolfin_error()` also displays information about the location of the code that generated the error (file, function name and line number). Once an error is encountered, the program is stopped.

Note that in order to pass formatting strings and additional arguments to warnings or errors, the variations `dolfin_error1()`, `dolfin_error2()` and so on must be used, as illustrated by the following examples:

```
dolfin_error(``GMRES solver did not converge.'');
dolfin_error1(``Unable to find face opposite to node %d.'', n);
dolfin_error2(``Unable to find edge between nodes %d and %d.'', n0, n1);
```

9.3 Debug messages and assertions

The macro `dolfin_debug()` works similarly to `dolfin_info()`:

```
dolfin_debug(message);
```

but in addition to displaying the given message, information is printed about the location of the code that generated the debug message (file, function name and line number).

Note that in order to pass formatting strings and additional arguments with debug messages, the variations `dolfin_debug1()`, `dolfin_debug2()` and so on, depending on the number of arguments, must be used.

Assertions can often be a helpful programming tool. Use assertions whenever you assume something about about a variable in your code, such as assuming that given input to a function is valid. **DOLFIN** provides the macro `dolfin_assert()` for creating assertions:

```
dolfin_assert(check);
```

This macro accepts a boolean expression and if the expression evaluates to false, an error message is displayed, including the file, function name and line number of the assertion, and a segmentation fault is raised (to enable easy attachment to a debugger). The following examples illustrate the use of `dolfin_assert()`:

```
dolfin_assert(i >= 0);
dolfin_assert(i < n);
dolfin_assert(cell.type() == Cell::triangle);
dolfin_assert(cell.type() == Cell::tetrahedron);
```

Note that assertions are only active when compiling **DOLFIN** and your program with `DEBUG` defined (configure option `--enable-debug` or compiler flag `-DDEBUG`). Otherwise, the macro `dolfin_assert()` expands to nothing, meaning that liberal use of assertions does not affect performance, since assertions are only present during development and debugging.

9.4 Task notification

The two functions `dolfin_begin()` and `dolfin_end()` available in the `dolfin` name space can be used to notify the **DOLFIN** log system about the beginning and end of a task:

```
void dolfin_begin();
void dolfin_end();
```

Alternatively, a string message (or a formatting string with optional arguments) can be supplied:

```
void dolfin_begin(const char* message, ...);
void dolfin_end(const char* message, ...);
```

These functions enable the **DOLFIN** log system to display messages, warnings and errors hierarchically, by automatically indenting the output produced between calls to `dolfin_begin()` and `dolfin_end()`. A program may contain an arbitrary number of nested tasks.

9.5 Progress bars

The **DOLFIN** log system provides the class `Progress` for simple creation of progress sessions. A progress session automatically displays the progress of a computation using a progress bar.

If the number of steps of a computation is known, a progress session should be defined in terms of the number of steps and updated in each step of the computation as illustrated by the following example:

```
Progress p(‘‘Assembling’’, mesh.noCells());
for (CellIterator c(mesh); !c.end(); ++c)
{
  ...
  p++;
}
```

It is also possible to specify the step number explicitly by assigning an integer to the progress session:

```
Progress p(‘‘Iterating over vector’’, x.size())
for (uint i = 0; i < x.size(); i++)
{
  ...
  p = i;
}
```

Alternatively, if the number of steps is unknown, the progress session needs to be updated with the current percentage of the progress:

```
Progress p(‘‘Time-stepping’’);
while ( t < T )
{
  ...
  p = t / T;
}
```

The progress bar created by the progress session will only be updated if the progress has changed significantly since the last update (by default at least 10%). The amount of change needed for an update can be controlled using the parameter ``progress step'':

```
dolfin_set(``progress step'', 0.01);
```

Note that several progress sessions may be created simultaneously, or nested within tasks.

9.6 Controlling the destination of output

By default, the **DOLFIN** log system directs messages to standard output (the terminal). Other options include directing messages to a curses interface or turning of messages completely. To specify the output destination, use the function `dolfin_output()` available in the `dolfin` namespace:

```
void dolfin_output(const char* destination);
```

where `destination` is one of ``plain text'' (standard output), ``curses'' (curses interface) or `silent'' (no messages printed).

When messages are directed to the **DOLFIN** curses interface, a text-mode graphical and interactive user-interface is started in the current terminal window. To see a list of options, press 'h' for help. The curses-interface is updated periodically but the function `dolfin_update()` can be used to force a refresh of the display.

It is possible to switch the **DOLFIN** log system on or off using the function `dolfin_log()` available in the `dolfin` namespace. This function accepts as argument a `bool`, specifying whether or not messages should be directed to the current output destination. This function can be useful to suppress excessive logging, for example when calling a function that generates log messages multiple times:

```
GMRES gmres;
while ( ... )
{
  ...
  dolfin_log(false);
  gmres.solve(A, x, b);
  dolfin_log(true);
  ...
}
```

Chapter 10

Parameters

▶ *Developer's note:* Since this chapter was written, the **DOLFIN** parameter system has been completely redesigned and now supports localization of parameters to objects or hierarchies of objects. Chapter needs to be updated.

DOLFIN keeps a global database of parameters that control the behavior of the various components of **DOLFIN**. Parameters are controlled using a uniform type-independent interface that allows retrieving the values of existing parameters, modifying existing parameters and adding new parameters to the database.

10.1 Retrieving the value of a parameter

To retrieve the value of a parameter, use the function `get()` available in the `dolfin` namespace:

```
Parameter get(std::string key);
```

This function accepts as argument a string `key` and returns the value of the parameter matching the given key. An error message is printed through the log system if there is no parameter with the given key in the database.

The value of the parameter is automatically cast to the correct type when assigning the value of `get()` to a variable, as illustrated by the following examples:

```
real TOL = get(``tolerance'');
int num_samples = get(``number of samples'');
bool solve_dual = get(``solve dual problem'');
std::string filename = get(``file name'');
```

Note that there is a small cost associated with accessing the value of a parameter, so if the value of a parameter is to be used multiple times, then it should be retrieved once and stored in a local variable as illustrated by the following example:

```
int num_samples = get(``number of samples'');
for (int i = 0; i < num_samples; i++)
{
  ...
}
```

10.2 Modifying the value of a parameter

To modify the value of a parameter, use the function `set()` available in the `dolfin` namespace:

```
void set(std::string key, Parameter value);
```

This function accepts as arguments a string `key` together with the corresponding value. The value type should match the type of parameter that is being modified. An error message is printed through the log system if there is no parameter with the given key in the database.

The following examples illustrate the use of `set()`:

```
set(``tolerance'', 0.01);
set(``number of samples'', 10);
set(``solve dual problem'', true);
set(``file name'', ``solution.xml'');
```

Note that changing the values of parameters using `set()` does not change the values of already retrieved parameters; it only changes the values of parameters in the database. Thus, the value of a parameter must be changed before using a component that is controlled by the parameter in question.

10.3 Adding a new parameter

To add a parameter to the database, use the function `add()` available in the `dolfin` namespace:

```
void add(std::string key, Parameter value);
```

This function accepts two arguments: a unique key identifying the new parameter and the value of the new parameter.

The following examples illustrate the use of `add()`:

```
add(``tolerance'', 0.01);
add(``number of samples'', 10);
add(``solve dual problem'', true);
add(``file name'', ``solution.xml'');
```

10.4 Saving parameters to file

The following code illustrates how to save the current database of parameters to a file in **DOLFIN** XML format:

```
File file(‘‘parameters.xml’’);
file << ParameterSystem::parameters;
```

When running a simulation in **DOLFIN**, saving the parameter database to a file is an easy way to document the set of parameters used in the simulation.

10.5 Loading parameters from file

The following code illustrates how to load a set of parameters into the current database of parameters from a file in **DOLFIN** XML format:

```
File file(‘‘parameters.xml’’);
file >> ParameterSystem::parameters;
```

The following example illustrates how to specify a list of parameters in the **DOLFIN** XML format

```
<?xml version=’’1.0’’ encoding=’’UTF-8’’?>

<dolfin xmlns:dolfin=’’http://www.fenics.org/dolfin/’’>
  <parameters>
    <parameter name=’’tolerance’’ type=’’real’’ value=’’0.01’’/>
    <parameter name=’’number of samples’’ type=’’int’’ value=’’10’’/>
    <parameter name=’’solve dual problem’’ type=’’bool’’ value=’’false’’/>
    <parameter name=’’file name’’ type=’’string’’ value=’’solution.xml’’/>
  </parameters>
</dolfin>
```

Chapter 11

Solvers

▶ *Developer's note:* This chapter is currently being written...

DOLFIN provides a number of pre-defined PDE solvers (called "modules" in the source structure) by default. The solver interface is intentionally very simple to facilitate users writing their own solvers. These are the pre-defined solvers:

1. Poisson
2. Convection-Diffusion
3. Navier-Stokes
4. Elasticity

A solver for a PDE should provide the following interface:

1. a constructor which takes a mesh, equation coefficients and possibly additional data.
2. a `solve()` method which solves the equation given the specified data.
3. a static `solve()` function which constructs and solves the equation.

FIXME: List solvers, then present in detail, include lots of nice images with solver output

11.1 Poisson's equation

Poisson's equation with Dirichlet and homogeneous Neumann boundary conditions:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= g_D \quad \text{on } \Gamma_1, \\ -\partial_n u &= 0 \quad \text{on } \Gamma_2 \end{aligned} \tag{11.1}$$

The variational formulation is given by

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx \quad \forall v. \tag{11.2}$$

The boundary conditions are enforced strongly and thus don't appear in the variational formulation.

11.1.1 Usage

The API for the Poisson solver:

```
// Create Poisson solver
PoissonSolver(Mesh& mesh, Function& f, BoundaryCondition& bc);

// Solve Poisson's equation
void solve();

// Solve Poisson's equation (static version)
static void solve(Mesh& mesh, Function& f, BoundaryCondition& bc);
```

A simple example of using the solver:

```
int main()
{
  Mesh mesh("mesh.xml.gz");
  MyFunction f;
  MyBC bc;

  PoissonSolver::solve(mesh, f, bc);

  return 0;
}
```

Where “f” is a Function specifying the right-hand side of the equation and “bc” is a BoundaryCondition.

11.1.2 Performance

The solver is an illustrative example and performance has not been a goal. It uses a GMRES linear solver, where a multi-grid linear solver would give optimal performance.

11.1.3 Limitations

The solver is meant to be the simplest example solver, and therefore some simplifications have been made. Typically the general form of Poisson’s equation includes a diffusion coefficient which has been omitted here.

11.2 Convection–diffusion

The convection-diffusion equation with Dirichlet and homogeneous Neumann boundary conditions is given by:

$$
\begin{aligned}
\dot{u} + b \cdot \nabla u - \nabla \cdot (a \nabla u) &= f \quad \text{in } \Omega \times (0, T], \\
u &= g_D \quad \text{on } \Gamma_1 \times (0, T], \\
-\partial_n u &= 0 \quad \text{on } \Gamma_2 \times (0, T], \\
u(\cdot, 0) &= u_0 \quad \text{in } \Omega,
\end{aligned}
\tag{11.3}
$$

where the convection is given by the vector $b = b(x, t)$ and the diffusion is given by $a = a(x, t)$.

The variational formulation is:

FIXME: Stabilized convection-diffusion

This is a stabilized FEM-formulation, so the solver can handle convection-dominated problems.

The time integration is done using cG(1) (Crank-Nicolson).

11.2.1 Usage

The API for the convection-diffusion solver:

```
// Create convection-diffusion solver
ConvectionDiffusionSolver(Mesh& mesh, Function& w, Function& f,
  BoundaryCondition& bc);

// Solve convection-diffusion
void solve();

// Solve convection-diffusion (static version)
static void solve(Mesh& mesh, Function& w, Function& f,
  BoundaryCondition& bc);
```

A simple example of using the solver:

```
int main()
{
  dolfin_output("curses");

  Mesh mesh("dolfin.xml.gz");
  Convection w;
  Source f;
  MyBC bc;

  ConvectionDiffusionSolver::solve(mesh, w, f, bc);

  return 0;
}
```

11.2.2 Performance

There are no particular performance issues with the solver. GMRES is used for solving the linear system.

11.2.3 Limitations

Currently many coefficients (such as diffusivity) are not user-definable, they need to be exposed by the interface.

11.3 Incompressible Navier–Stokes

Write introduction here, equations etc.

11.3.1 Usage

Present API of solver and give an example.

11.3.2 Performance

Write something about the performance of the solver.

11.3.3 Limitations

Write something about the limitations of the solver.

11.4 Elasticity

Navier's equations of elasticity with Dirichlet and homogeneous Neumann boundary conditions:

$$
\begin{aligned}
u &= x - X, \\
\dot{u} - v &= 0 \quad \text{in } \Omega^0, \\
\dot{v} - \nabla \cdot \sigma &= f \quad \text{in } \Omega^0, \\
\sigma &= E\epsilon(u) = E(\nabla u^\top + \nabla u) \\
E\epsilon &= \lambda tr(\epsilon) I + 2\mu\epsilon, \\
v(0, \cdot) &= v^0, \quad u(0, \cdot) = u^0 \quad \text{in } \Omega^0, \\
u &= g_D \quad \text{on } \Gamma_1 \times (0, T], \\
-\partial_n u &= 0 \quad \text{on } \Gamma_2 \times (0, T]
\end{aligned}
$$

The variational form:

$$
\int_\Omega \dot{v} w \, dx = \int_\Omega -\sigma(u)\epsilon(v) + f w \, dx, \quad \forall w. \tag{11.4}
$$

The time integration is done using dG(0) (backward Euler).

The mass matrix appearing from $\int_\Omega \dot{v} w dx$ is lumped (equivalent to computing it using nodal quadrature).

11.4.1 Usage

Present API of solver and give an example.

11.4.2 Performance

Write something about the performance of the solver.

11.4.3 Limitations

Write something about the limitations of the solver.

Bibliography

[1] *Cygwin*, 2005. http://cygwin.com/.

[2] *GiD*, 2005. http://gid.cimne.upc.es/.

[3] *MATLAB*, 2005. http://www.mathworks.com/.

[4] *MayaVi*, 2005. http://mayavi.sourceforge.net/.

[5] *MEncoder*, 2005. http://www.mplayerhq.hu/.

[6] *Octave*, 2005. http://www.octave.org/.

[7] *Open Inventor*, 2005. http://http://oss.sgi.com/projects/inventor/.

[8] *OpenDX*, 2005. http://www.opendx.org/.

[9] *ParaView*, 2005. http://www.paraview.org/.

[10] *Portable, extensible toolkit for scientific computation petsc*, 2005. http://www-unix.mcs.anl.gov/petsc/petsc-2/.

[11] *Tecplot*, 2005. http://www.tecplot.com/.

[12] *The Visualization Toolkit (VTK)*, 2005. http://www.vtk.org/.

[13] *SLEPc*, 2006. http://www.grycap.upv.es/slepc/.

[14] *uBLAS*, 2006. http://www.boost.org/libs/numeric/ublas/.

Appendix A

Reference elements

A.1 The reference triangle

The reference triangle (Figure A.1) is defined by the following three vertices:

$$\begin{aligned} v^0 &= (0,0), \\ v^1 &= (1,0), \\ v^2 &= (0,1). \end{aligned} \tag{A.1}$$

Note that this corresponds to a counter-clockwise orientation of the vertices in the plane.

The edges of the reference triangle are ordered following the convention that edge e^i should be opposite to vertex v^i for $i = 0, 1, 2$, with the vertices of each edge ordered to give a counter-clockwise orientation of the triangle in the plane:

$$\begin{aligned} e^0 &: (v^1, v^2), \\ e^1 &: (v^2, v^0), \\ e^2 &: (v^0, v^1). \end{aligned} \tag{A.2}$$

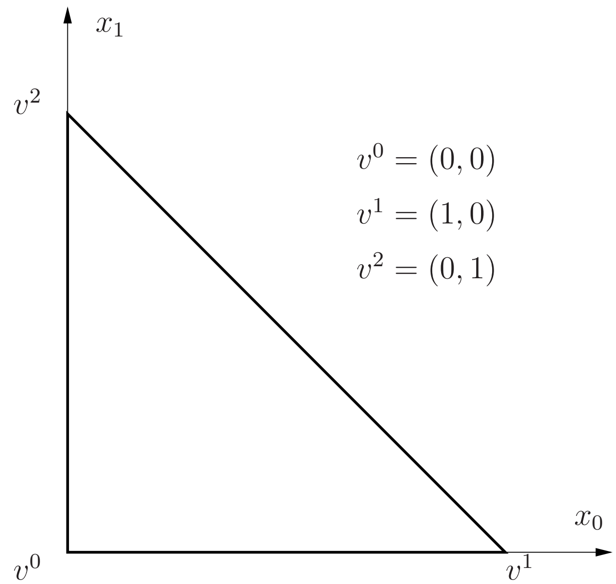

Figure A.1: Physical coordinates of the reference triangle.

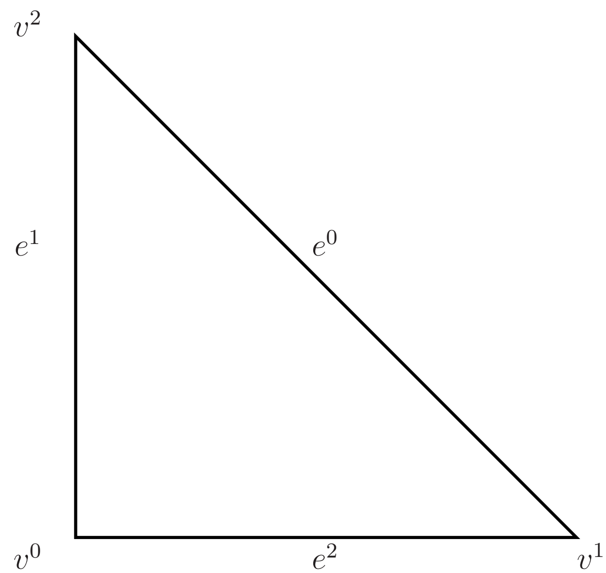

Figure A.2: Ordering of mesh entities (vertices and edges) for the reference triangle.

A.2 The reference tetrahedron

The reference tetrahedron (Figure A.3) is defined by the following four vertices:

$$\begin{aligned} v^0 &= (0,0,0), \\ v^1 &= (1,0,0), \\ v^2 &= (0,1,0), \\ v^4 &= (0,0,1). \end{aligned} \tag{A.3}$$

The faces of the reference tetrahedron are ordered following the convention that face f^i should be opposite to vertex v^i for $i = 0,1,2,3$, with the vertices of each face ordered to give a counter-clockwise orientation of each face as seen from the outside of the tetrahedron and the first vertex of face f^i given by vertex $v^{i+1 \mod 4}$:

$$\begin{aligned} f^0 &: (v^1, v^3, v^2), \\ f^1 &: (v^2, v^3, v^0), \\ f^2 &: (v^3, v^1, v^0), \\ f^3 &: (v^0, v^1, v^2). \end{aligned} \tag{A.4}$$

The edges of the reference tetrahedron are ordered following the convention that edges e^0, e^1, e^2 should correspond to the edges of the reference triangle. Edges e^3, e^4, e^5 all ending up at vertex v^3 are ordered based on their first vertex:

$$\begin{aligned} e^0 &: (v^1, v^2), \\ e^1 &: (v^2, v^0), \\ e^2 &: (v^0, v^1), \\ e^3 &: (v^0, v^3), \\ e^4 &: (v^1, v^3), \\ e^5 &: (v^2, v^3). \end{aligned} \tag{A.5}$$

The ordering of vertices on faces implicitly defines an ordering of edges on faces by identifying an edge on a face with the opposite vertex on the face:

$$\begin{aligned} f^0 &: (e^5, e^0, e^4), \\ f^1 &: (e^3, e^1, e^5), \\ f^2 &: (e^2, e^3, e^4), \\ f^3 &: (e^0, e^1, e^2). \end{aligned} \tag{A.6}$$

Note that the ordering of edges on f^3 is the same as the ordering of edges on the reference triangle. Also note that the internal ordering of vertices on edges does not always follow the orientation of the face (which is not possible).

A.3 Ordering of degrees of freedom

The local and global orderings of degrees of freedom or *nodes* are obtained by associating each node with a mesh entity, locally and globally.

A.3.1 Mesh entities

We distinguish between mesh entities of different topological dimensions:

vertices	topological dimension 0
edges	topological dimension 1
faces	topological dimension 2
cells	topological dimension 2 or 3

A cell can be either a triangle or a tetrahedron depending on the type of mesh. For a mesh consisting of triangles, the mesh entities involved are vertices, edges and cells, and for a mesh consisting of tetrahedrons, the mesh entities involved are vertices, edges, faces and cells.

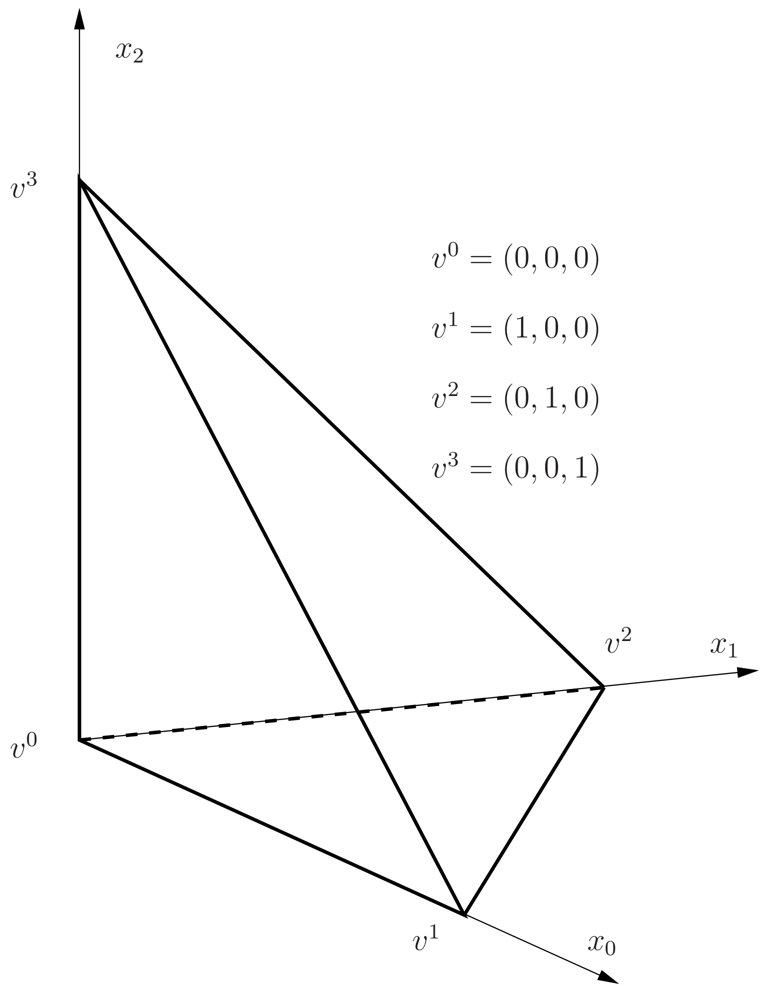

Figure A.3: Physical coordinates of the reference tetrahedron.

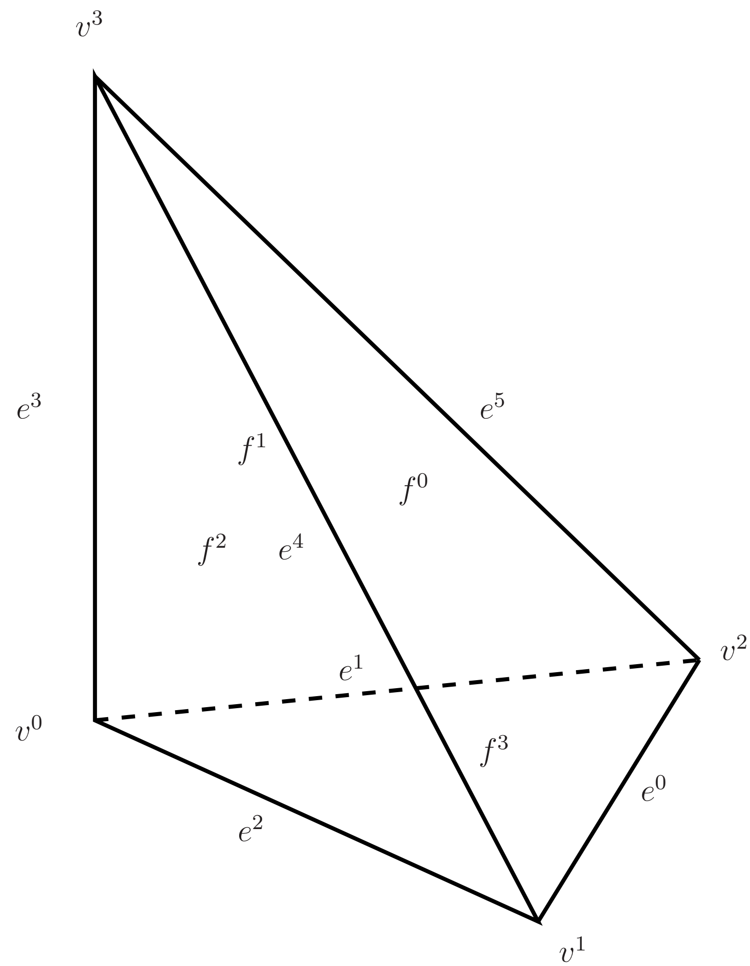

Figure A.4: Ordering of mesh entities (vertices, edges, faces) for the reference tetrahedron.

A.3.2 Ordering among mesh entities

With each mesh entity, there can be associated zero or more nodes and the nodes are ordered locally and globally based on the topological dimension of the mesh entity with which they are associated. Thus, any nodes associated with vertices are ordered first and nodes associated with cells last.

If more than one node is associated with a single mesh entity, the internal ordering of the nodes associated with the mesh entity becomes important, in particular for edges and faces, where the nodes of two adjacent cells sharing a common edge or face must line up.

A.3.3 Internal ordering on edges

For edges containing more than one node, the nodes are ordered in the direction from the first vertex (v_e^0) of the edge to the second vertex (v_e^1) of the edge as in Figure A.5.

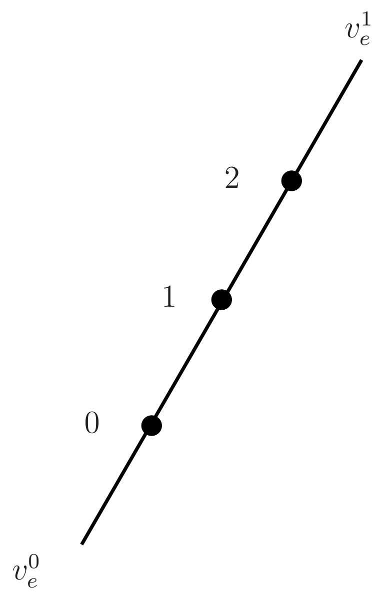

Figure A.5: Internal ordering of nodes on edges.

A.3.4 Alignment of edges

Depending on the orientation of any given cell, an edge on the cell may be aligned or not aligned with the corresponding edge on the reference cell if the vertices of the cell are mapped to the reference cell. We define the *alignment* of an edge with respect to a cell to be 0 if the edge is aligned with the orientation of the reference cell and 1 otherwise.

Example 1: The alignment of the first edge (e^0) on a triangle is 0 if the first vertex of the edge is the second vertex (v^1) of the triangle.

Example 2: The alignment of the second edge (e^1) on a tetrahedron is 0 if the first vertex of the edge is the third vertex (v^2) of the tetrahedron.

If two cells share a common edge and the edge is aligned with one of the cells and not the other, we must reverse the order in which the local nodes are mapped to global nodes on one of the two cells. As a convention, the order is kept if the alignment is 0 and reversed if the alignment is 1.

A.3.5 Internal ordering on faces

For faces containing more than one node, the ordering of nodes is nested going from the first to the third vertex and in each step going from the first to the second vertex as in Figure A.6.

A.3.6 Alignment of faces

There are six different ways for a face to be aligned on a tetrahedron; there are three ways to pick the first edge of the face, and once the first edge is picked, there are two ways to pick the second edge. To define an alignment of faces as an integer between 0 and 5, we compare the ordering of edges on a face with the ordering of edges on the corresponding face on the reference tetrahedron. If the first edge of the face matches the first edge on the corresponding face on the reference tetrahedron and also the second edge matches the second edge on the reference tetrahedron, then the alignment is 0. If only the first

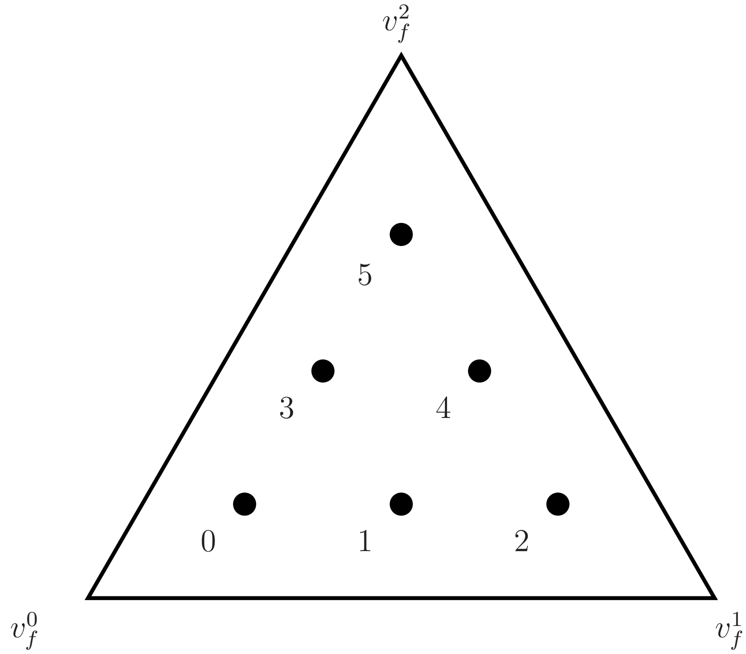

Figure A.6: Internal ordering of nodes on faces.

edge matches, then the alignment is 1. We similarly define alignments $2, 3$ by matching the first and second edges with the second and third edges on the corresponding face on the reference tetrahedron, and alignments $4, 5$ by matching the first and second edges with the third and first edges on the corresponding face on the reference tetrahedron.

Example 1: The alignment of the first face of a tetrahedron is 0 if the first edge of the face is edge number 5 and the second edge is edge number 0.

Example 2: The alignment of the first face of a tetrahedron is 1 if the first edge of the face is edge number 5 and the second edge is not edge number 0. (It must then be edge number 4.)

Example 3: The alignment of the first face of a tetrahedron is 4 if the first edge of the face is edge number 4 and the second edge is edge number 5.

Example 4: The alignment of the first face of a tetrahedron is 5 if the first edge of the face is edge number 4 and the second edge is not edge number 5. (It must then be edge number 0.)

Appendix B

Design

This chapter discusses details of the design of **DOLFIN** and is intended mainly for developers of **DOLFIN**.

B.1 Linear algebra

The linear algebra library provides a uniform interface to uBlas and PETSc linear algebra through a set of wrappers for basic data structures (matrices and vectors) and solvers, such as Krylov subspace solvers with preconditioners.

For both sets of wrappers, a common interface is defined by the classes `GenericMatrix` and `GenericVector`. **DOLFIN** provides a number of algorithms, most notably the assembly algorithms, that work only through the common interface, which means that these algorithms work for any given representation that implements the interface specified by `GenericMatrix` or `GenericVector`. A class diagram for the **DOLFIN** linear algebra implementation is given in Figure B.1.

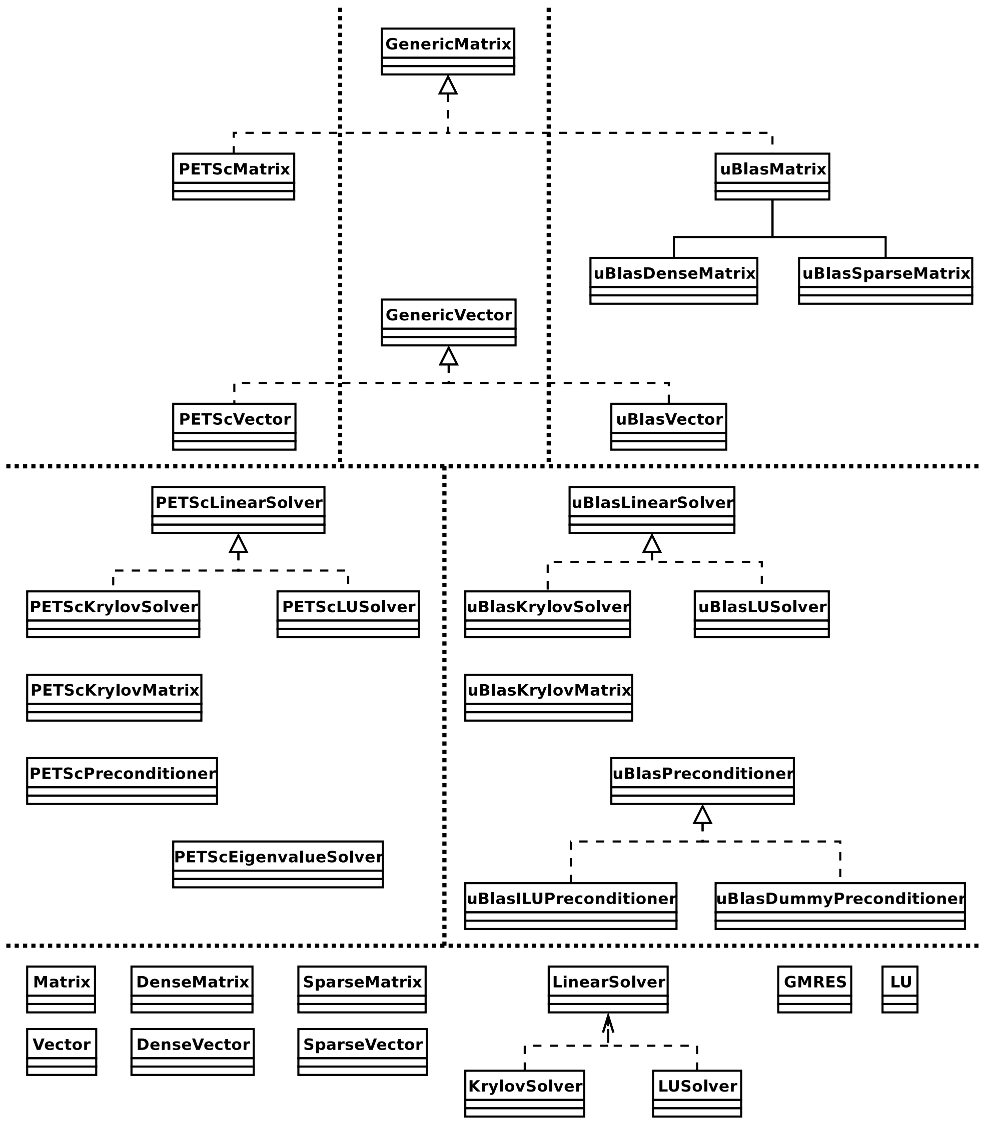

Figure B.1: Class diagram of the linear algebra classes in **DOLFIN**.

Appendix C

Installation

The source code of **DOLFIN** is portable and should compile on any Unix system, although it is developed mainly under GNU/Linux (in particular Debian GNU/Linux). **DOLFIN** can be compiled under Windows through Cygwin [1]. Questions, bug reports and patches concerning the installation should be directed to the **DOLFIN** mailing list at the address

```
dolfin-dev@fenics.org
```

DOLFIN must currently be compiled directly from source, but an effort is underway to provide precompiled Debian packages of **DOLFIN** and other **FEniCS** components.

C.1 Installing from source

C.1.1 Dependencies and requirements

DOLFIN depends on a number of libraries that need to be installed on your system. These libraries include Boost, Libxml2 and (optionally) PETSc. In addition to these libraries, you need to install **FIAT** and **FFC** if you want to define your own variational forms.

Installing Boost

Boost is a collection of C++ source libraries. Boost can be obtained from

`http://www.boost.org/`

Packages are available for most Linux distributions. For Ubuntu/Debian users, the package to install is `boost-dev`.

Installing Libxml2

Libxml2 is a library used by **DOLFIN** to parse XML data files. Libxml2 can be obtained from

`http://xmlsoft.org/`

Packages are available for most Linux distributions. For Ubuntu/Debian users, the package to install is `libxml2-dev`.

Installing PETSc

Optionally, **DOLFIN** may be compiled with support for PETSc. To compile **DOLFIN** with PETSc, add the flag `--enable-petsc` during the initial configuration of **DOLFIN**.

PETSc is a library for the solution of linear and nonlinear systems, functioning as the backend for the **DOLFIN** linear algebra classes. **DOLFIN** depends on PETSc version 2.3.1, which can be obtained from

```
http://www-unix.mcs.anl.gov/petsc/petsc-2/
```

Follow the installation instructions on the PETSc web page. Normally, you should only have to perform the following simple steps in the PETSc source directory:

```
# export PETSC_DIR=‘pwd‘
# ./config/configure.py --with-clanguage=cxx --with-shared=1
# make all
```

Add `--download-hypre=yes` to configure.py if you want to install Hypre which provides a collection of preconditioners, including algebraic multigrid (AMG), and `--download-umfpack=yes` to configure.py if you want to install UMFPACK which provided as fast direct linear solver. Both packages are highly recommended.

DOLFIN assumes that `PETSC_DIR` is `/usr/local/lib/petsc/` but this can be controlled using the flag `--with-petsc-dir=<path>` when configuring DOLFIN (see below).

Installing FFC

DOLFIN uses the FEniCS Form Compiler **FFC** to process variational forms. **FFC** can be obtained from

```
http://www.fenics.org/
```

Follow the installation instructions given in the **FFC** manual. **FFC** follows the standard for Python packages, which means that normally you should only have to perform the following simple step in the **FFC** source directory:

```
# python setup.py install
```

Note that **FFC** depends on **FIAT** , which in turn depends on the Python packages Numeric (Debian package `python-numeric`) and LinearAlgebra (Debian package `python-numeric-ext`). Refer to the **FFC** manual for further details.

C.1.2 Downloading the source code

The latest release of **DOLFIN** can be obtained as a `tar.gz` archive in the download section at

```
http://www.fenics.org/
```

Download the latest release of **DOLFIN**, for example `dolfin-0.1.0.tar.gz`, and unpack using the command

```
# tar zxfv dolfin-0.1.0.tar.gz
```

This creates a directory `dolfin-0.1.0` containing the **DOLFIN** source code.

If you want the very latest version of **DOLFIN**, there is also a version named `dolfin-cvs-current.tar.gz` which provides a snapshot of the current CVS version of **DOLFIN**, updated automatically from the CVS repository each hour. This version may contain features not yet present in the latest release, but may also be less stable and even not work at all.

C.1.3 Compiling the source code

DOLFIN is built using the standard GNU Autotools (Automake, Autoconf) and libtool, which means that the installation procedure is simple:

```
# ./configure
# make
```

followed by an optional

```
# make install
```

to install **DOLFIN** on your system.

The configure script will check for a number of libraries and try to figure out how compile **DOLFIN** against these libraries. The configure script accepts a collection of optional arguments that can be used to control the compilation process. A few of these are listed below. Use the command

```
# ./configure --help
```

for a complete list of arguments.

- Use the option `--prefix=<path>` to specify an alternative directory for installation of **DOLFIN**. The default directory is `/usr/local/`, which means that header files will be installed under `/usr/local/include/` and libraries will be installed under `/usr/local/lib/`. This option can be useful if you don't have root access but want to install **DOLFIN** locally on a user account as follows:

  ```
  # mkdir ~/local
  # ./configure --prefix=~/local
  # make
  # make install
  ```

- Use the option `--enable-debug` to compile **DOLFIN** with debugging symbols and assertions.

- Use the option `--enable-optimization` to compile an optimized version of **DOLFIN** without debugging symbols and assertions.

- Use the option `--disable-curses` to compile **DOLFIN** without the curses interface (a text-mode graphical user interface).

- Use the option `--enable-petsc` to compile **DOLFIN** with support for PETSc.

- Use the option `--disable-pydolfin` to compile without support for PyDOLFIN.

- Use the option `--disable-mpi` to compile **DOLFIN** without support for MPI (Message Passing Interface), assuming PETSc has been compiled without support for MPI.
- Use the option `--with-petsc-dir=<path>` to specify the location of the PETSc directory. By default, **DOLFIN** assumes that PETSc has been installed in `/usr/local/lib/petsc/`.

C.1.4 Compiling the demo programs

After compiling the **DOLFIN** library according to the instructions above, you may want to try one of the demo programs in the subdirectory `src/demo/` of the **DOLFIN** source tree. Just enter the directory containing the demo program you want to compile and type `make`. You may also compile all demo programs at once using the command

```
# make demo
```

C.1.5 Compiling a program against DOLFIN

Whether you are writing your own Makefiles or using an automated build system such as GNU Autotools or BuildSystem, it is straightforward to compile a program against **DOLFIN**. The necessary include and library paths can be obtained through the script `dolfin-config` which is automatically generated during the compilation of **DOLFIN** and installed in the `bin` subdirectory of the `<path>` specified with `--prefix`. Assuming this directory is in your executable path (environment variable `PATH`), the include path for building **DOLFIN** can be obtained from the command

```
dolfin-config --cflags
```

and the path to **DOLFIN** libraries can be obtained from the command

```
dolfin-config --libs
```

If `dolfin-config` is not in your executable path, you need to provide the full path to `dolfin-config`.

Examples of how to write a proper `Makefile` are provided with each of the example programs in the subdirectory `src/demo/` in the **DOLFIN** source tree.

C.2 Debian package

In preparation.

C.3 Installing from source under Windows

DOLFIN can be used under Windows using Cygwin, which provides a Linux-like environment. The installation process is the same as under GNU/Linux. To use **DOLFIN** under Cygwin, the Cygwin development tools must be installed. Instructions for installing PETSc under Cygwin can be found on the PETSc web page. Installation of **FFC** and **FIAT** is the same as under GNU/Linux. The Python package Numeric is not available as a Cygwin package and must be installed manually. To compile **DOLFIN**, the Cygwin package `libxml2-devel` must be installed. The compilation procedure is then the same as under GNU/Linux. If MPI has not been installed:

```
# ./configure --disable-mpi
# make
```

followed by an optional

```
# make install
```

will compile **DOLFIN** on your system.

Appendix D

Contributing code

If you have created a new module, fixed a bug somewhere, or have made a small change which you want to contribute to **DOLFIN**, then the best way to do so is to send us your contribution in the form of a patch. A patch is a file which describes how to transform a file or directory structure into another. The patch is built by comparing a version which both parties have against the modified version which only you have.

D.1 Creating a patch

The tool used to create a patch is called `diff` and the tool used to apply the patch is called `patch`. These tools are free software and are standard on most Unix systems.

Here's an example of how it works. Start from the latest release of **DOLFIN**, which we here assume is release 0.1.0. You then have a directory structure under `dolfin-0.1.0` where you have made modifications to some files which you think could be useful to other users.

1. Clean up your modified directory structure to remove temporary and binary files which will be rebuilt anyway:

```
# make clean
```

2. From the parent directory, rename the **DOLFIN** directory to something else:

```
# mv dolfin-0.1.0 dolfin-0.1.0-mod
```

3. Unpack the version of **DOLFIN** that you started from:

```
# tar zxfv dolfin-0.1.0.tar.gz
```

4. You should now have two **DOLFIN** directory structures in your current directory:

```
# ls
dolfin-0.1.0
dolfin-0.1.0-mod
```

5. Now use the `diff` tool to create the patch:

```
# diff -u --new-file --recursive dolfin-0.1.0
  dolfin-0.1.0-mod > dolfin-<identifier>-<date>.patch
```

   written as one line, where `<identifier>` is a keyword that can be used to identify the patch as coming from you (your username, last name, first name, a nickname etc) and `<date>` is today's date in the format `yyyy-mm-dd`.

6. The patch now exists as `dolfin-<identifier>-<date>.patch` and can be distributed to other people who already have `dolfin-0.1.0` to easily create your modified version. If the patch is large, compressing it with for example `gzip` is advisable:

```
# gzip dolfin-<identifier>-<date>.patch
```

D.2 Sending patches

Patch files should be sent to the **DOLFIN** mailing list at the address

```
dolfin-dev@fenics.org
```

Include a short description of what your patch accomplishes. Small patches have a better chance of being accepted, so if you are making a major contribution, please consider breaking your changes up into several small self-contained patches if possible.

D.3 Applying a patch (maintainers)

Let's say that a patch has been built relative to **DOLFIN** release 0.1.0. The following description then shows how to apply the patch to a clean version of release 0.1.0.

1. Unpack the version of **DOLFIN** which the patch is built relative to:

   ```
   # tar zxfv dolfin-0.1.0.tar.gz
   ```

2. Check that you have the patch `dolfin-<identifier>-<date>.patch` and the **DOLFIN** directory structure in the current directory:

   ```
   # ls
   dolfin-0.1.0
   dolfin-<identifier>-<date>.patch
   ```

   Unpack the patch file using `gunzip` if necessary.

3. Enter the **DOLFIN** directory structure:

   ```
   # cd dolfin-0.1.0
   ```

4. Apply the patch:

   ```
   # patch -p1 < ../dolfin-<identifier>-<date>.patch
   ```

   The option `-p1` strips the leading directory from the filename references in the patch, to match the fact that we are applying the patch from inside the directory. Another useful option to `patch` is `--dry-run` which can be used to test the patch without actually applying it.

5. The modified version now exists as `dolfin-0.1.0`.

D.4 License agreement

By contributing a patch to **DOLFIN**, you agree to license your contributed code under the GNU General Public License (a condition also built into the GPL license of the code you have modified). Before creating the patch, please update the author and date information of the file(s) you have modified according to the following example:

```
// Copyright (C) 2004-2005 Johan Hoffman and Anders Logg.
// Licensed under the GNU GPL Version 2.
//
// Modified by Johan Jansson 2005.
// Modified by Garth N. Wells 2005.
//
// First added:  2004-06-22
// Last changed: 2005-09-01
```

As a rule of thumb, the original author of a file holds the copyright.

Appendix E

License

DOLFIN is licensed under the GNU General Public License (GPL) version 2, included verbatim below.

```
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

  The precise terms and conditions for copying, distribution and
modification follow.

    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
```

```
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)
```

```
These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
```

```
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

  5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
```

```
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded.  In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number.  If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation.  If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.
```

Index