DOLFIN User Manual

September 10, 2010

Logg, Wells, et al.

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 **11**

1 Quickstart **13**

1.1 Downloading and installing **DOLFIN** 13

1.2 Solving Poisson's equation with **DOLFIN** 14

1.2.1 Setting up the variational formulation 15

1.2.2 Writing the solver 15

1.2.3 Compiling the program 21

1.2.4 Running the program 21

1.2.5 Visualizing the solution 22

2 Linear algebra **23**

2.1 Matrices and vectors 23

2.1.1 Sparse matrices 25

2.1.2 Dense matrices 26

2.1.3 The common interface . . . 26
2.2 Solving linear systems . . . 26
2.2.1 Iterative methods . . . 27
2.2.2 Direct methods . . . 28
2.3 Eigenvalue problems . . . 29
2.4 Linear algebra backends . . . 30
2.4.1 uBLAS . . . 30
2.4.2 PETSc . . . 30
2.4.3 Trilinos . . . 31
2.4.4 Matrix Template Library (MTL4) . . . 31
2.4.5 User provided linear algebra backends . . . 31

3 The mesh **33**

3.1 Basic concepts . . . 33
3.1.1 Mesh . . . 33
3.1.2 Mesh entities . . . 33
3.2 Mesh iterators . . . 34
3.3 Mesh functions . . . 35
3.4 Mesh refinement . . . 35
3.5 Working with meshes . . . 35
3.5.1 Reading a mesh from file . . . 35

3.5.2 Extracting a boundary mesh 36
3.5.3 Built-in meshes 37
3.5.4 Creating meshes 37

4 Functions **39**

4.1 Basic properties 39
4.1.1 Representation 40
4.1.2 Assignment 40
4.1.3 Components and subfunctions 40
4.1.4 Output 41
4.2 Discrete functions 41
4.3 User-defined functions 42
4.3.1 Cell-dependent functions 42
4.3.2 Simple definition of functions in Python 43

5 Ordinary differential equations **47**

6 Partial differential equations **49**

7 Nonlinear solver **51**

7.1 Nonlinear functions 51
7.2 Newton solver 52
7.2.1 Linear solver 53

7.2.2 Application of Dirichlet boundary conditions 54
7.3 Incremental Newton solver 55

8 Input/output **57**

8.1 Files and objects 57
8.2 File formats 59
8.2.1 **DOLFIN** XML 59
8.2.2 VTK 60
8.2.3 GNU Octave 61
8.2.4 MATLAB 62
8.3 Converting between file formats 62
8.4 A note on new file formats 63

9 The log system **65**

9.1 Generating log messages 65
9.2 Warnings and errors 66
9.3 Debug messages and assertions 67
9.4 Task notification 68
9.5 Progress bars 69
9.6 Controlling the destination of output 70

10 Parameters **71**

10.1 Retrieving the value of a parameter 71
10.2 Modifying the value of a parameter 72
10.3 Adding a new parameter 73
10.4 Saving parameters to file 74
10.5 Loading parameters from file 74

11 PyDOLFIN **75**

A Reference cells **79**

A.1 The reference interval 80
A.2 The reference triangle 80
A.3 The reference quadrilateral 81
A.4 The reference tetrahedron 82
A.5 The reference hexahedron 83

B Numbering of mesh entities **85**

B.1 Basic concepts 85
B.2 Numbering of vertices 86
B.3 Numbering of other mesh entities 88
B.3.1 Relative ordering 89
B.3.2 Limitations 90
B.4 Numbering schemes for reference cells 91

B.4.1 Numbering of mesh entities on intervals 91
B.4.2 Numbering of mesh entities on triangular cells 91
B.4.3 Numbering of mesh entities on quadrilateral cells . . . 92
B.4.4 Numbering of mesh entities on tetrahedral cells 92
B.4.5 Numbering of mesh entities on hexahedral cells 93

C Design **95**

C.1 Linear algebra 95

D Installation **97**

D.1 Installing from source 97
D.1.1 Dependencies and requirements 97
D.1.2 Downloading the source code 101
D.1.3 Compiling the source code 101
D.1.4 Compiling the demo programs 103
D.1.5 Compiling a program against **DOLFIN** 103
D.2 Debian package 104
D.3 Installing from source under Windows 104

E Contributing code **107**

E.1 Creating bundles/patches 107
E.1.1 Creating a Mercurial (hg) bundle 107

E.1.2 Creating a standard (diff) patch file . . . 109
E.2 Sending bundles/patches . . . 110
E.3 Applying changes . . . 111
E.3.1 Applying a Mercurial bundle . . . 111
E.3.2 Applying a standard patch file . . . 111
E.4 License agreement . . . 112

F Contributors **115**

G Coding style **117**

G.1 Naming conventions . . . 117
G.1.1 Class names . . . 117
G.1.2 Function names . . . 118
G.1.3 Variable names . . . 119
G.1.4 Enum variables and constants . . . 119
G.1.5 File names . . . 119
G.2 Miscellaneous . . . 120
G.2.1 Comments . . . 120
G.2.2 Integers and reals . . . 120
G.2.3 Placement of brackets . . . 121
G.2.4 Indentation . . . 121
G.2.5 Header file layout . . . 121

G.2.6 Implementation file layout 122
G.2.7 Including header files 123
G.2.8 Forward declarations 123
G.2.9 Explicit constructors 124
G.2.10 Virtual functions 124
G.3 Use of libraries 125
G.3.1 Prefer C++ strings and streams to old C-style `char*` . 125

H License **127**

About this manual

This manual is currently being written. As a consequence, some sections may be incomplete or inaccurate. In particular, only the C++ interface (not the Python interface) of **DOLFIN** is documented, and only to a certain extent. Care has been taken that the quickstart chapter is accurate, but other than that, inconsistencies and inaccuracies can be expected.

We apologize for any inconvenience, but take comfort in the fact that (i) with the release of **DOLFIN** 0.7.0, the interface is starting to mature and will undergo less dramatic changes in the future (which will actually make it possible to write documentation) and (ii) most of the code is pretty well documented through the demos. If you have some writing skills and are willing to contribute, please consider writing a section or two and submit to the mailing list!

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 D 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 x.y.z of **DOLFIN**:

```
# tar zxfv dolfin-x.y.z.tar.gz
# cd dolfin-x.y.z
```

```
# sudo scons install
```

Since **DOLFIN** depends on a number of other packages, you may also need to download and install those packages before you can compile **DOLFIN**. (See Appendix D 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 $\Gamma = \{(x,y) \in \partial\Omega : x = 0 \text{ or } x = 1\}$, homogeneous Neumann boundary conditions $\partial_n u = 0$ on the remaining part of the boundary, and right-hand side given by $f(x,y) = 500\exp(-((x-0.5)^2 + (y-0.5)^2)/0.02)$:

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

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

$$a(v, u_h) = L(v) \quad \forall v \in \hat{V}_h, \tag{1.4}$$

with $(\hat{V}_h, V_h)$ a pair of suitable discrete function spaces (the test and trial spaces). The bilinear form $a : \hat{V}_h \times V_h \to \mathbb{R}$ is given by

$$a(v, u_h) = \int_\Omega \nabla v \cdot \nabla u_h \, \mathrm{d}x \tag{1.5}$$

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

$$L(v) = \int_\Omega v f \, \mathrm{d}x. \tag{1.6}$$

1.2.1 Setting up the variational formulation

The variational formulation (1.4) must be given to **DOLFIN** as a pair of bilinear and linear forms (a, L) using a form compiler like **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)

a = dot(grad(v), grad(u))*dx
L = v*f*dx
```

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 -l dolfin 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 (1.4) 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 `demo/pde/poisson/cpp/` of the **DOLFIN** source tree.[1]

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:

```
// Include headers and define classes here

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` (which we here name `Source`) and by 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
{
```

[1] A corresponding Python demo is available in `demo/pde/poisson/python/`

```
  void eval(double* values, const double* x) const
  {
    double dx = x[0] - 0.5;
    double dy = x[1] - 0.5;
    values[0] = 500.0*exp(-(dx*dx + dy*dy) / 0.02);
  }
};
```

To define the Dirichlet boundary condition, we define a `SubDomain` for the Dirichlet boundary:

```
class DirichletBoundary : public SubDomain
{
  bool inside(const double* x, bool on_boundary) const
  {
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS;
  }
};
```

Next, we need to create a mesh. **DOLFIN** relies on external programs for mesh generation, and imports meshes in **DOLFIN** XML format. Meshes in other formats can be converted to the **DOLFIN** XML format using the script `dolfin-convert`. 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 $1/32$ (with a total of $2 \cdot 32^2 = 2048$ triangles), we can just type

```
UnitSquare mesh(32, 32);
```

Next, we instantiate a finite element function space on the mesh as follows:

```
PoissonFunctionSpace V(mesh);
```

Here, we make use of the code generated by the form compiler (the class `PoissonFunctionSpace`).

Next, we define the Dirichlet boundary condition by specifying the value that the solution should take and the subset of the boundary where it should be applied:

```
Constant u0(0.0);
DirichletBoundary boundary;
DirichletBC bc(V, u0, boundary);
```

Note the difference in how Dirichlet and Neumann boundary conditions are applied. Dirichlet conditions are here applied strongly (but may also be applied weakly) using the `DirichletBC` class, while Neumann boundary conditions are defined as part of the variational problem. (It is a *natural* boundary condition for this variational formulation of Poisson's equation.)

Next, we initialize the pair of bilinear and linear forms that we have previously compiled with **FFC**. We also instantiate the right-hand side function and attach it to the linear form:

```
PoissonBilinearForm a(V, V);
PoissonLinearForm L(V);
Source f;
L.f = f;
```

To solve the variational problem, we use the class `VariationalProblem`, which automatically assembles and solves the corresponding linear system:

```
VariationalProblem problem(a, L, bc);
Function u;
problem.solve(u);
```

To plot the solution, one may simply type

```
plot(u);
```

This requires the installation of PyDOLFIN and Viper. (A warning will be issued if plotting is not available.)

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;

// Source term
class Source : public Function
{
  void eval(double* values, const double* x) const
  {
    double dx = x[0] - 0.5;
    double dy = x[1] - 0.5;
    values[0] = 500.0*exp(-(dx*dx + dy*dy) / 0.02);
  }
};

// Sub domain for Dirichlet boundary condition
class DirichletBoundary : public SubDomain
{
  bool inside(const double* x, bool on_boundary) const
  {
```

```
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS;
  }
};

int main()
{
  // Create mesh and function space
  UnitSquare mesh(32, 32);
  PoissonFunctionSpace V(mesh);

  // Define boundary condition
  Constant u0(0.0);
  DirichletBoundary boundary;
  DirichletBC bc(V, u0, boundary);

  // Define variational problem
  PoissonBilinearForm a(V, V);
  PoissonLinearForm L(V);
  Source f;
  L.f = f;

  // Compute solution
  VariationalProblem problem(a, L, bc);
  Function u;
  problem.solve(u);

  // Plot solution
  plot(u);

  // Save solution in VTK format
  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 `SConstruct` file that tells `scons` how to build the program. The following example shows how to write a `SConstruct` file for the above example:

```
import os, commands

# Get compiler from pkg-config
compiler = commands.getoutput('pkg-config --variable=compiler dolfin')

# Create a SCons Environment based on the main os environment
env = Environment(ENV=os.environ, CXX=compiler)

# Get compiler flags from pkg-config
env.ParseConfig('pkg-config --cflags --libs dolfin')

# Program name
env.Program('demo', 'main.cpp')
```

With this file in place, we just need to type `scons` to compile the program, generating the executable as the file `demo`. Note that this requires `pkg-config` to be able to find the file `dolfin.pc`. That file is generated by the `configure` script during the configuration of **DOLFIN**. If `pkg-config` fails to find it, you need to add the directory containing it to the environment variable `PKG_CONFIG_PATH`.

1.2.4 Running the program

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

```
# ./demo
```

1.2.5 Visualizing the solution

The solution may be visualized either by the built-in `plot()` command or by calling an external application. The built-in plotting requires the installation of both PyDOLFIN and Viper. In this example, we chose to save the solution in VTK format, which can be imported into for example ParaView or MayaVi.

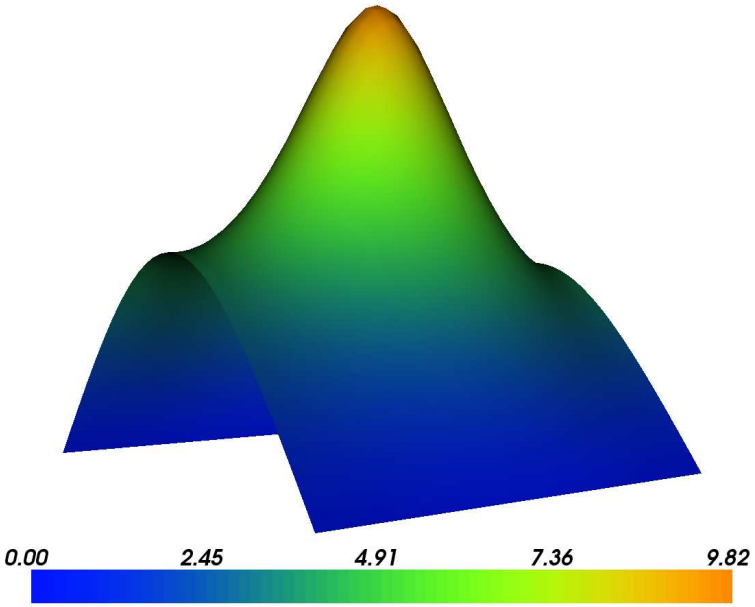

Figure 1.1: The solution of Poisson's equation (1.1) visualized using the `plot()` command (Viper).

Chapter 2

Linear algebra

▶ *Developer's note:* Since this chapter was written, the **DOLFIN** linear algebra interface has undergone some changes that are not documented here. As a result, some of the information in this chapter may be outdated, incomplete or simply wrong.

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 wrappers that provide a common interface to the functionality of the linear algebra libraries uBLAS [13], PETSc [8] and Trilinos [**?**].

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;
double a = A(4, 3);

x(3) = 2.0;
double 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:

```
double 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`:

```
double 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`.

[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.

2.1.2 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 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`.

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.

The following points should be noted:

1. If you want a specific backend like PETSc, then use `PETScVector`/`PETScMatrix`.
2. If you don't care about the backend, then use `Vector`/`Matrix`.
3. If you write a *function* that should be able to accept vectors and matrices from any backend as input, then use `GenericVector`/`GenericMatrix`.

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 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 [12].

For the basic eigenvalue problem

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

the following code demonstrates how to compute the first eigenpair:

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

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

The double and complex components of the eigenvalue are returned in `lr` and `lc`, respectively, and the double 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 third eigenpair:

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

double 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.

2.4.3 Trilinos

Trilinos . . . .

2.4.4 Matrix Template Library (MTL4)

MTL4 . . . .

2.4.5 User provided linear algebra backends

▶ *Developer's note:* This section will explain how a user-provided linear alegra backend can be used with **DOLFIN**.

Chapter 3

The mesh

▶ *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();
```

A mesh may also be refined locally by supplying a `MeshFunction` with boolean markers for the cells that should be refined.

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 $2 \times 16 \times 16 = 512$ triangles) and a $16 \times 16 \times 16$ tetrahedral mesh of the unit cube (consisting of $6 \times 16 \times 16 \times 16 = 24576$ tetrahedra):

```
UnitInterval mesh1D(16);
UnitSquare mesh2D(16, 16);
UnitCube mesh3D(16, 16, 16);
```

► *Developer's note:* We could easily add other built-in meshes, like 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 `Functions`:

- A `Function` can be scalar or vector-valued;
- A `Function` can be restricted (interpolated) 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 `Functions` include *discrete* `Functions`, *user-defined* `Functions` and *constant* `Functions`. These are discussed in detail below.

4.1.2 Assignment

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

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

4.1.3 Components and subfunctions

If a `Function` is vector-valued, or in general *nested* (mixed), a new `Function` may be created to represent any given *subsystem* (component) of the original `Function`, as illustrated by the following example:

```
Function u;         // Function with two components
Function u0 = u[0]; // First subsystem (component) of u
Function u1 = u[1]; // First subsystem (component) of u
```

If a `Function` represents a nested function (one defined in terms of a mixed finite element, see below), then indexing has the effect of picking out subfunctions. 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 subfunctions and components of `w`:

```
Function w; // Mixed Function (u, p)
u = w[0];   // First subfunction (velocity)
```

```
p = w[1];   // Second subfunction (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
```

4.1.4 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 degrees of freedom, a `Mesh`, a local-to-global mapping (`DofMap`) and a finite element. 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 finite element, and the nodal values $\{v_i\}_{i=1}^N$ are given by the values of a `Vector`.

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

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

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
{
public:

  Source(Mesh& mesh) : Function(mesh) {}

  double eval(const double* x) const
  {
    return x[0]*x[1]*sin(x[2] / DOLFIN_PI);
  }

};

Source f;
```

▶ *Developer's note:* Write about how to define vector-valued functions.

4.3.1 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
{
public:

  MeshSize(Mesh& mesh) : Function(mesh) {}

  double eval(const double* x) const
  {
    return cell().diameter();
  }

};

MeshSize h;
```

Note that the current `Cell` is only available during assembly and has no meaning otherwise. For example, it is not possible to write the `Function h` to file.

▶ *Developer's note:* Write about predefined special functions like `MeshSize` and `FacetNormal`.

4.3.2 Simple definition of functions in Python

▶ *Developer's note:* This is a verbatim copy from the mailing list. It should be edited for the manual.

```
On Wed, Jun 04, 2008 at 12:44:17PM +0200, Martin Sandve Alnaes wrote:
2008/6/3 Anders Logg <logg@simula.no>:
On Tue, Jun 03, 2008 at 11:23:41PM +0200, Martin Sandve Alnaes wrote:
>> It's easy with some helper functions I have, I can show you tomorrow.
>> It only depends on dolfin::Function (i.e. dolfin.cpp_Function) and Instant.
```

```
>
> ok, nice. Maybe we can add the helper functions to assemble.py.

Now you can do things like this:

A = assemble(..., coefficients=[myfunction, 1.23, "sin(x[0])"])

1.23 is wrapped in a cpp_Function (i.e. a constant Function), and
the string is compiled as a C++ dolfin::Function like "v[0] = ...;".

If you want more control over the implementation
(i.e. temporary variables in eval for efficiency, if-checks, etc.)
you can also do this:

code = """
class F: dolfin::Function {
 ...
};
class G: dolfin::Function {
 ...
};
"""

f, g = compile_functions(code, mesh)


Or, if you just want to precompile but don't need anything too fancy:

expressions = ["sin(x[0])",
                         ("a*x[1]", "b*x[0]")]
f, g = compile_functions(expressions, mesh)
assert f.rank() == 0
assert g.rank() == 1
g.a = 1.23
g.b = 3.21

Note that:
- a tuple is interpreted as a vector expression

- a tuple of tuples is interpreted as a matrix expression

- variables like a and b in the strings above are detected
  automatically, but this requires all builtin names to be
  registered in the variable dolfin.compile_functions._builtins,
  which currently contains some common mathematical
```

```
  functions like sqrt, pow, exp, sin, pi etc.

- variables are initialized to 0.0, so use compile_functions if
  you need control over variable values.

- assemble also accepts tuples like in compile_functions

- assemble only calls on instant once independently of
  the number of string expressions, to reduce compilation time

- Error checks are not extensive!
```

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 needs to be written. In the meantime, look at the demos in `src/demo/pde/`.

Chapter 7

Nonlinear solver

DOLFIN provides tools for solving nonlinear equations of the form

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

where $F : \mathbb{R}^n \to \mathbb{R}^n$. To use the nonlinear solver, a nonlinear function must be defined. The nonlinear solver is then initialised with this function and a solution computed. The source code for examples of solving nonlinear problems can be found in `src/demo/nls`.

7.1 Nonlinear functions

To solve a nonlinear problem, the user must defined a class which represents the nonlinear function $F(u)$. The class should be derived from the **DOLFIN** class `NonlinearFunction`. It contains 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 problem being solved. The structure of a user defined class `MyNonlinearFunction` is shown below.

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

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

 // Compute F(u) and J
 void form(GenericMatrix& A, GenericVector& b,
           const GenericVector& u)
 {
   // Insert F(u) into the vector b and J into the matrix A
 }

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

The above class computes the function $F(u)$ and its Jacobian J concurrently. In the future, it will be possible to compute $F(u)$ and J either concurrently or separately.

7.2 Newton solver

DOLFIN provides tools to solve nonlinear systems using Newton's method and variants of it. The following code solves a nonlinear problem, defined by `MyNonlinearFunction` using Newton's method.

```
Vector u;
MyNonlinearFunction F;
NewtonSolver newton_solver;

nonlinear_solver.solve(F, x);
```

The maximum number of iterations before the Newton procedure is exited can be set through the **DOLFIN** parameter system, along with the relative and absolute tolerances the residual. This is illustrated below.

```
NewtonSolver nonlinear_solver;
nonlinear_solver.set("Newton maximum iterations", 50);
nonlinear_solver.set("Newton relative tolerance", 1e-10);
nonlinear_solver.set("Newton absolute tolerance", 1e-10);
```

The Newton procedure is considered to have converged when the residual r_n at iteration n is less than the absolute tolerance or the relative residual r_n/r_0 is less than the relative tolerance. By default, the residual at iteration n is given by

$$r_n = \|F(u_n)\|. \tag{7.2}$$

Computation of the residual in this way can be set by

```
NewtonSolver newton_solver;
newton_solver.set("Newton convergence criterion", "residual");
```

For some problems, it is more appropriate to consider changes in the solution u in testing for convergence. At iteration n, the solution is updated via

$$u_n = u_{n-1} + du_n \tag{7.3}$$

where du_n is the increment. When using an incremental criterion for convergence, the 'residual' is defined as

$$r_n = \|du_n\|. \tag{7.4}$$

Computation of the incremental residual can be set by

```
NewtonSolver newton_solver;
newton_solver.set("Newton convergence criterion", "incremental");
```

7.2.1 Linear solver

The solution to the nonlinear problems is returned in the vector `x`. By default, the `NewtonSolver` uses a direct solver to solve systems of linear

equations. It is possible to set the type linear solver to be used when creating a `NewtonSolver`. For example,

```
NewtonSolver newton_solver(gmres);
```

creates a solver which will use GMRES to solve linear systems. For iterative solvers, the preconditioner can also be selected,

```
NewtonSolver newton_solver(gmres, ilu);
```

The above Newton solver will use GMRES in combination with incomplete LU factorisation.

7.2.2 Application of Dirichlet boundary conditions

The application of Dirichlet boundary conditions to finite element problems in the context of a Newton solver requires particular attention. The 'residual' $F(u)$ at nodes where Dirichlet boundary conditions are applied is the equal to difference between the imposed boundary condition value and the current solution u. The function

```
DirichletBC bc;
bc.apply(Matrix& A, Matrix& b, const Matrix& x, const Form& form);
```

applies Dirichlet boundary conditions correctly. For a nonlinear finite element problem, the below code assembles the function $F(u)$ and its Jacobian, and applied Dirichlet boundary conditions in the appropriate manner.

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

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

  // Compute F(u) and J
  void form(GenericMatrix& A, GenericVector& b,
            const GenericVector& u)
  {
    // Insert F(u) into the vector b and J into the matrix A
    Assembler assembler;
    assembler.assemble(A, a, mesh);
    assembler.assemble(b, L, mesh);
    bc.apply(A, b, x, a);
  }

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

7.3 Incremental Newton solver

Newton solvers are commonly used to solve nonlinear equations in a series of steps. This can be done by building a simple loop around a Newton solver, and is shown in the following code.

```
Function u;
Vector x;
MyNonlinearProblem F(u,x); // Initialise u with the vector x
NewtonSolver nonlinear_solver;

// Solve nonlinear problem in a series of steps
double dt = 1.0; double t  = 0.0; double T  = 3.0;
while( t < T)
{
  t += dt;
  nonlinear_solver.solve(F, x);
```

```
}
```

Typically, the boundary conditions and/or source terms will be dependent on `t`.

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
`.m`	`File::octave`	GNU Octave
(`.m`)	`File::matlab`	MATLAB

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::octave`	out	out	out	out	out
`File::matlab`	out	out	out	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.

8.2.2 VTK

Data saved in VTK format [9] can be visualized using various packages. The powerful and freely available ParaView [7] is recommended. Alternatively, VTK data can be visualized in MayaVi [3], 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 [4].

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

8.2.3 GNU Octave

GNU Octave [5] 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 [6]. (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)
```

8.2.4 MATLAB

Since MATLAB [2] 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`.

8.3 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, the Gmsh `.msh` version 2.0 format and the **DOLFIN** XML (`.xml`) mesh format.

To convert a mesh in Medit `.mesh` format generated by TetGen with the -g option, type

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

To convert a mesh in Gmsh `.msh` format type

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

In generating a Gmsh mesh, make sure to define a physical surface/volume. It is also possible to convert a mesh from the old **DOLFIN** XML (`.xml`) mesh format to the current one by typing

```
# dolfin-convert -i xml-old old_mesh.xml new_mesh.xml
```

Example meshes can be found in the directory `src/utils/convert/` of the **DOLFIN** source tree.

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.g

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 `info()` available in the `dolfin` namespace:

```
void info(std::string format, ...);
```

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

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

As an alternative to `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()` displays all matrix entries.

9.2 Warnings and errors

Warnings and error messages can be generated using the functions

```
warning(std::string format, ...);
error(std::string format, ...);
```

Once an error is encountered, the program throws an exception.

9.3 Debug messages and assertions

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

```
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()`:

```
assert(i >= 0);
assert(i < n);
assert(cell.type() == Cell::triangle);
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 begin();
void end();
```

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

```
void begin(std::string format, ...);
void end();
```

These functions enable the **DOLFIN** log system to display messages, warnings and errors hierarchically, by automatically indenting the output produced between calls to `begin()` and `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). Messages may also be turned off completely. To specify the destination, set the value of the parameter "output destination":

```
dolfin_set("output destination", "terminal");
dolfin_set("output destination", "silent");
```

One may also set the debug level for the **DOLFIN** log system so that only messages with a debug level higher than or equal to the current debug level are printed:

```
dolfin_set("debug level", "1");
```

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 `dolfin_get()` available in the `dolfin` namespace:

```
Parameter dolfin_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 `dolfin_get()` to a variable, as illustrated by the following examples:

```
double TOL = dolfin_get("tolerance");
int num_samples = dolfin_get("number of samples");
bool solve_dual = dolfin_get("solve dual problem");
std::string filename = dolfin_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 = dolfin_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 `dolfin_set()` available in the `dolfin` namespace:

```
void dolfin_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 `dolfin_set()`:

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

Note that changing the values of parameters using `dolfin_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

PyDOLFIN

▶ *Developer's note:* Describe Python interface, especially in areas where it differs from the C++ interface,

Bibliography

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

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

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

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

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

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

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

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

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

[10] *Mercurial*, 2006. http://www.selenic.com/mercurial/.

[11] *NumPy*, 2006. http://numpy.scipy.org/.

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

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

[14] M. S. Alnæs, H. P. Langtangen, A. Logg, K.-A. M. dal, and O. Skavhaug, *UFC*, 2007. URL: http://www.fenics.org/ufc/.

Appendix A

Reference cells

The definition of reference cells used in **DOLFIN** follows the UFC specification. [14]

The following five reference cells are covered by the UFC specification: the reference *interval*, the reference *triangle*, the reference *quadrilateral*, the reference *tetrahedron* and the reference *hexahedron* (see Table A.1).

The UFC specification assumes that each cell in a finite element mesh is always isomorphic to one of the reference cells.

Reference cell	Dimension	#Vertices	#Facets
The reference interval	1	2	2
The reference triangle	2	3	3
The reference quadrilateral	2	4	4
The reference tetrahedron	3	4	4
The reference hexahedron	3	8	6

Table A.1: Reference cells covered by the UFC specification.

Figure A.1: The reference interval.

Vertex	Coordinate
v_0	$x = 0$
v_1	$x = 1$

Table A.2: Vertex coordinates of the reference interval.

A.1 The reference interval

The reference interval is shown in Figure A.1 and is defined by its two vertices with coordinates as specified in Table A.2.

A.2 The reference triangle

The reference triangle is shown in Figure A.2 and is defined by its three vertices with coordinates as specified in Table A.3.

Vertex	Coordinate
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (0, 1)$

Table A.3: Vertex coordinates of the reference triangle.

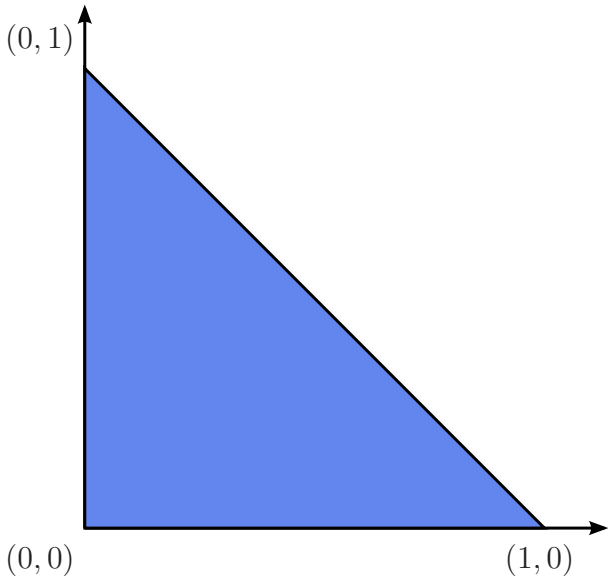

Figure A.2: The reference triangle.

A.3 The reference quadrilateral

The reference quadrilateral is shown in Figure A.3 and is defined by its four vertices with coordinates as specified in Table A.4.

Vertex	Coordinate
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (1, 1)$
v_3	$x = (0, 1)$

Table A.4: Vertex coordinates of the reference quadrilateral.

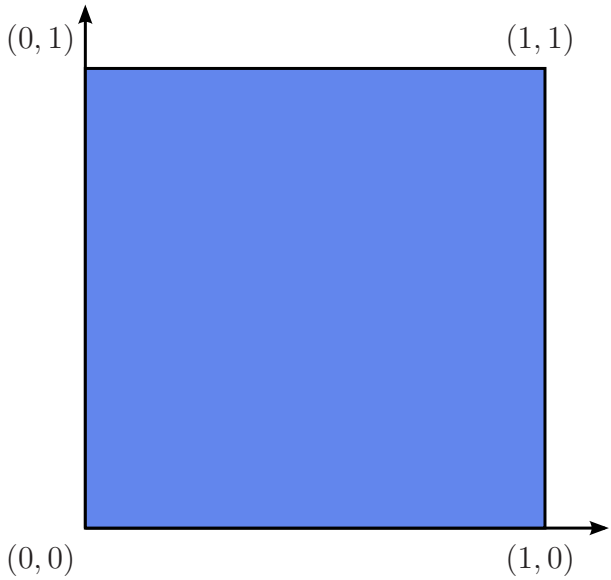

Figure A.3: The reference quadrilateral.

A.4 The reference tetrahedron

The reference tetrahedron is shown in Figure A.4 and is defined by its four vertices with coordinates as specified in Table A.5.

Vertex	Coordinate
v_0	$x = (0, 0, 0)$
v_1	$x = (1, 0, 0)$
v_2	$x = (0, 1, 0)$
v_3	$x = (0, 0, 1)$

Table A.5: Vertex coordinates of the reference tetrahedron.

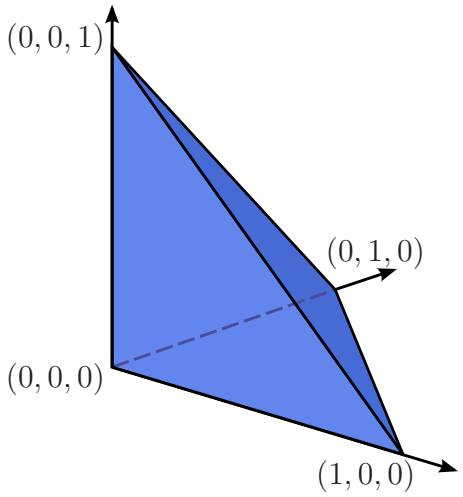

Figure A.4: The reference tetrahedron.

Vertex	Coordinate	Vertex	Coordinate
v_0	$x = (0, 0, 0)$	v_4	$x = (0, 0, 1)$
v_1	$x = (1, 0, 0)$	v_5	$x = (1, 0, 1)$
v_2	$x = (1, 1, 0)$	v_6	$x = (1, 1, 1)$
v_3	$x = (0, 1, 0)$	v_7	$x = (0, 1, 1)$

Table A.6: Vertex coordinates of the reference hexahedron.

A.5 The reference hexahedron

The reference hexahedron is shown in Figure A.5 and is defined by its eight vertices with coordinates as specified in Table A.6.

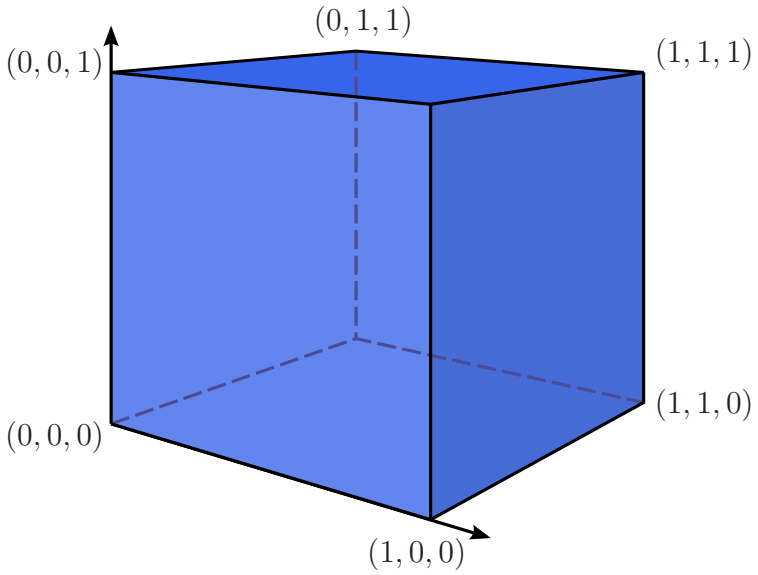

Figure A.5: The reference hexahedron.

Appendix B

Numbering of mesh entities

The numbering of mesh entities used in DOLFIN follows the UFC specification [14] for each mesh that has been ordered.[1]

The UFC specification dictates a certain numbering of the vertices, edges etc. of the cells of a finite element mesh. First, an *ad hoc* numbering is picked for the vertices of each cell. Then, the remaining entities are ordered based on a simple rule, as described in detail below.

B.1 Basic concepts

The topological entities of a cell (or mesh) are referred to as *mesh entities*. A mesh entity can be identified by 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 as *edges*, entities of dimension 2 as *faces*, entities of *codimension* 1 as *facets* and entities of codimension 0 as *cells*.

[1] To order a mesh, call the `order()` function: `mesh.order()`.

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

Table B.1: Named mesh entities.

These concepts are summarized in Table B.1.

Thus, the vertices of a tetrahedron are identified as $v_0 = (0, 0)$, $v_1 = (0, 1)$ and $v_2 = (0, 2)$, the edges are $e_0 = (1, 0)$, $e_1 = (1, 1)$, $e_2 = (1, 2)$, $e_3 = (1, 3)$, $e_4 = (1, 4)$ and $e_5 = (1, 5)$, the faces (facets) are $f_0 = (2, 0)$, $f_1 = (2, 1)$, $f_2 = (2, 2)$ and $f_3 = (2, 3)$, and the cell itself is $c_0 = (3, 0)$.

B.2 Numbering of vertices

For simplicial cells (intervals, triangles and tetrahedra) of a finite element mesh, the vertices are numbered locally based on the corresponding global vertex numbers. In particular, a tuple of increasing local vertex numbers corresponds to a tuple of increasing global vertex numbers. This is illustrated in Figure B.1 for a mesh consisting of two triangles.

For non-simplicial cells (quadrilaterals and hexahedra), the numbering is arbitrary, as long as each cell is isomorphic to the corresponding reference cell by matching each vertex with the corresponding vertex in the reference cell. This is illustrated in Figure B.2 for a mesh consisting of two quadrilaterals.

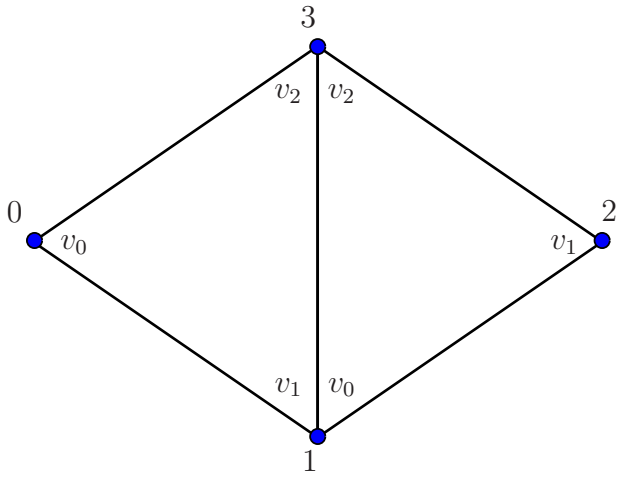

Figure B.1: The vertices of a simplicial mesh are numbered locally based on the corresponding global vertex numbers.

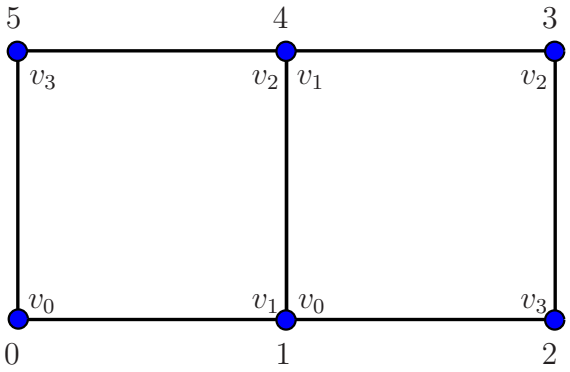

Figure B.2: The local numbering of vertices of a non-simplicial mesh is arbitrary, as long as each cell is isomorphic to the reference cell by matching each vertex to the corresponding vertex of the reference cell.

B.3 Numbering of other mesh entities

When the vertices have been numbered, the remaining mesh entities are numbered within each topological dimension based on a *lexicographical ordering* of the corresponding ordered tuples of *non-incident vertices*.

As an illustration, consider the numbering of edges (the mesh entities of topological dimension one) on the reference triangle in Figure B.3. To number the edges of the reference triangle, we identify for each edge the corresponding non-incident vertices. For each edge, there is only one such vertex (the vertex opposite to the edge). We thus identify the three edges in the reference triangle with the tuples (v_0), (v_1) and (v_2). The first of these is edge e_0 between vertices v_1 and v_2 opposite to vertex v_0, the second is edge e_1 between vertices v_0 and v_2 opposite to vertex v_1, and the third is edge e_2 between vertices v_0 and v_1 opposite to vertex v_2.

Similarly, we identify the six edges of the reference tetrahedron with the corresponding non-incident tuples (v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_1, v_3) and (v_2, v_3). The first of these is edge e_0 between vertices v_2 and v_3 opposite to vertices v_0 and v_1 as shown in Figure B.4.

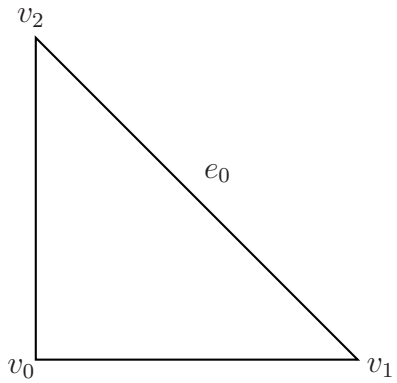

Figure B.3: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertex v_0.

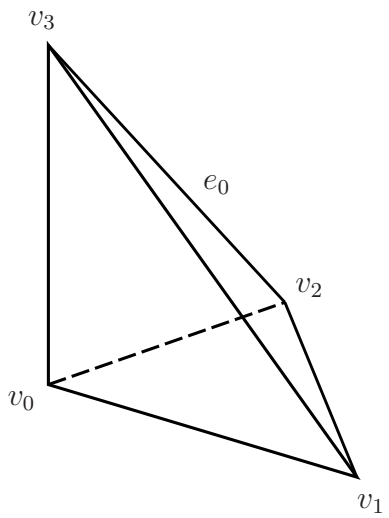

Figure B.4: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertices v_0 and v_1.

B.3.1 Relative ordering

The relative ordering of mesh entities with respect to other incident mesh entities follows by sorting the entities by their (global) indices. Thus, the pair of vertices incident to the first edge e_0 of a triangular cell is (v_1, v_2), not (v_2, v_1). Similarly, the first face f_0 of a tetrahedral cell is incident to vertices (v_1, v_2, v_3).

For simplicial cells, the relative ordering in combination with the convention of numbering the vertices locally based on global vertex indices means that two incident cells will always agree on the orientation of incident subsimplices. Thus, two incident triangles will agree on the orientation of the common edge and two incident tetrahedra will agree on the orientation of the common edge(s) and the orientation of the common face (if any). This is illustrated in Figure B.5 for two incident triangles sharing a common edge.

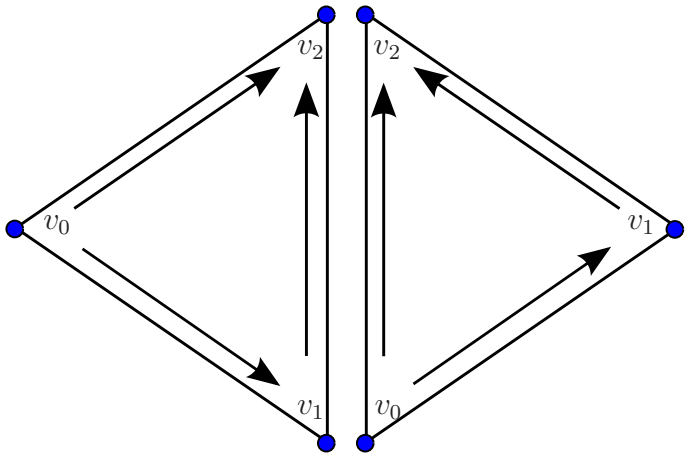

Figure B.5: Two incident triangles will always agree on the orientation of the common edge.

B.3.2 Limitations

The UFC specification is only concerned with the ordering of mesh entities with respect to entities of larger topological dimension. In other words, the UFC specification is only concerned with the ordering of incidence relations of the class $d - d'$ where $d > d'$. For example, the UFC specification is not concerned with the ordering of incidence relations of the class $0 - 1$, that is, the ordering of edges incident to vertices.

B.4 Numbering schemes for reference cells

The numbering scheme is demonstrated below for cells isomorphic to each of the five reference cells.

B.4.1 Numbering of mesh entities on intervals

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1)
$v_1 = (0, 1)$	(v_1)	(v_0)
$c_0 = (1, 0)$	(v_0, v_1)	$\emptyset$

B.4.2 Numbering of mesh entities on triangular cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1)
$e_0 = (1, 0)$	(v_1, v_2)	(v_0)
$e_1 = (1, 1)$	(v_0, v_2)	(v_1)
$e_2 = (1, 2)$	(v_0, v_1)	(v_2)
$c_0 = (2, 0)$	(v_0, v_1, v_2)	$\emptyset$

B.4.3 Numbering of mesh entities on quadrilateral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0, 3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1, 0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1, 1)$	(v_1, v_2)	(v_0, v_3)
$e_2 = (1, 2)$	(v_0, v_3)	(v_1, v_2)
$e_3 = (1, 3)$	(v_0, v_1)	(v_2, v_3)
$c_0 = (2, 0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

B.4.4 Numbering of mesh entities on tetrahedral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0, 3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1, 0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1, 1)$	(v_1, v_3)	(v_0, v_2)
$e_2 = (1, 2)$	(v_1, v_2)	(v_0, v_3)
$e_3 = (1, 3)$	(v_0, v_3)	(v_1, v_2)
$e_4 = (1, 4)$	(v_0, v_2)	(v_1, v_3)
$e_5 = (1, 5)$	(v_0, v_1)	(v_2, v_3)
$f_0 = (2, 0)$	(v_1, v_2, v_3)	(v_0)
$f_1 = (2, 1)$	(v_0, v_2, v_3)	(v_1)
$f_2 = (2, 2)$	(v_0, v_1, v_3)	(v_2)
$f_3 = (2, 3)$	(v_0, v_1, v_2)	(v_3)
$c_0 = (3, 0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

B.4.5 Numbering of mesh entities on hexahedral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	$(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_1 = (0, 1)$	(v_1)	$(v_0, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_2 = (0, 2)$	(v_2)	$(v_0, v_1, v_3, v_4, v_5, v_6, v_7)$
$v_3 = (0, 3)$	(v_3)	$(v_0, v_1, v_2, v_4, v_5, v_6, v_7)$
$v_4 = (0, 4)$	(v_4)	$(v_0, v_1, v_2, v_3, v_5, v_6, v_7)$
$v_5 = (0, 5)$	(v_5)	$(v_0, v_1, v_2, v_3, v_4, v_6, v_7)$
$v_6 = (0, 6)$	(v_6)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_7)$
$v_7 = (0, 7)$	(v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
$e_0 = (1, 0)$	(v_6, v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5)$
$e_1 = (1, 1)$	(v_5, v_6)	$(v_0, v_1, v_2, v_3, v_4, v_7)$
$e_2 = (1, 2)$	(v_4, v_7)	$(v_0, v_1, v_2, v_3, v_5, v_6)$
$e_3 = (1, 3)$	(v_4, v_5)	$(v_0, v_1, v_2, v_3, v_6, v_7)$
$e_4 = (1, 4)$	(v_3, v_7)	$(v_0, v_1, v_2, v_4, v_5, v_6)$
$e_5 = (1, 5)$	(v_2, v_6)	$(v_0, v_1, v_3, v_4, v_5, v_7)$
$e_6 = (1, 6)$	(v_2, v_3)	$(v_0, v_1, v_4, v_5, v_6, v_7)$
$e_7 = (1, 7)$	(v_1, v_5)	$(v_0, v_2, v_3, v_4, v_6, v_7)$
$e_8 = (1, 8)$	(v_1, v_2)	$(v_0, v_3, v_4, v_5, v_6, v_7)$
$e_9 = (1, 9)$	(v_0, v_4)	$(v_1, v_2, v_3, v_5, v_6, v_7)$
$e_{10} = (1, 10)$	(v_0, v_3)	$(v_1, v_2, v_4, v_5, v_6, v_7)$
$e_{11} = (1, 11)$	(v_0, v_1)	$(v_2, v_3, v_4, v_5, v_6, v_7)$
$f_0 = (2, 0)$	(v_4, v_5, v_6, v_7)	(v_0, v_1, v_2, v_3)
$f_1 = (2, 1)$	(v_2, v_3, v_6, v_7)	(v_0, v_1, v_4, v_5)
$f_2 = (2, 2)$	(v_1, v_2, v_5, v_6)	(v_0, v_3, v_4, v_7)
$f_3 = (2, 3)$	(v_0, v_3, v_4, v_7)	(v_1, v_2, v_5, v_6)
$f_4 = (2, 4)$	(v_0, v_1, v_4, v_5)	(v_2, v_3, v_6, v_7)
$f_5 = (2, 5)$	(v_0, v_1, v_2, v_3)	(v_4, v_5, v_6, v_7)
$c_0 = (3, 0)$	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$	$\emptyset$

Appendix C

Design

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

C.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 C.1.

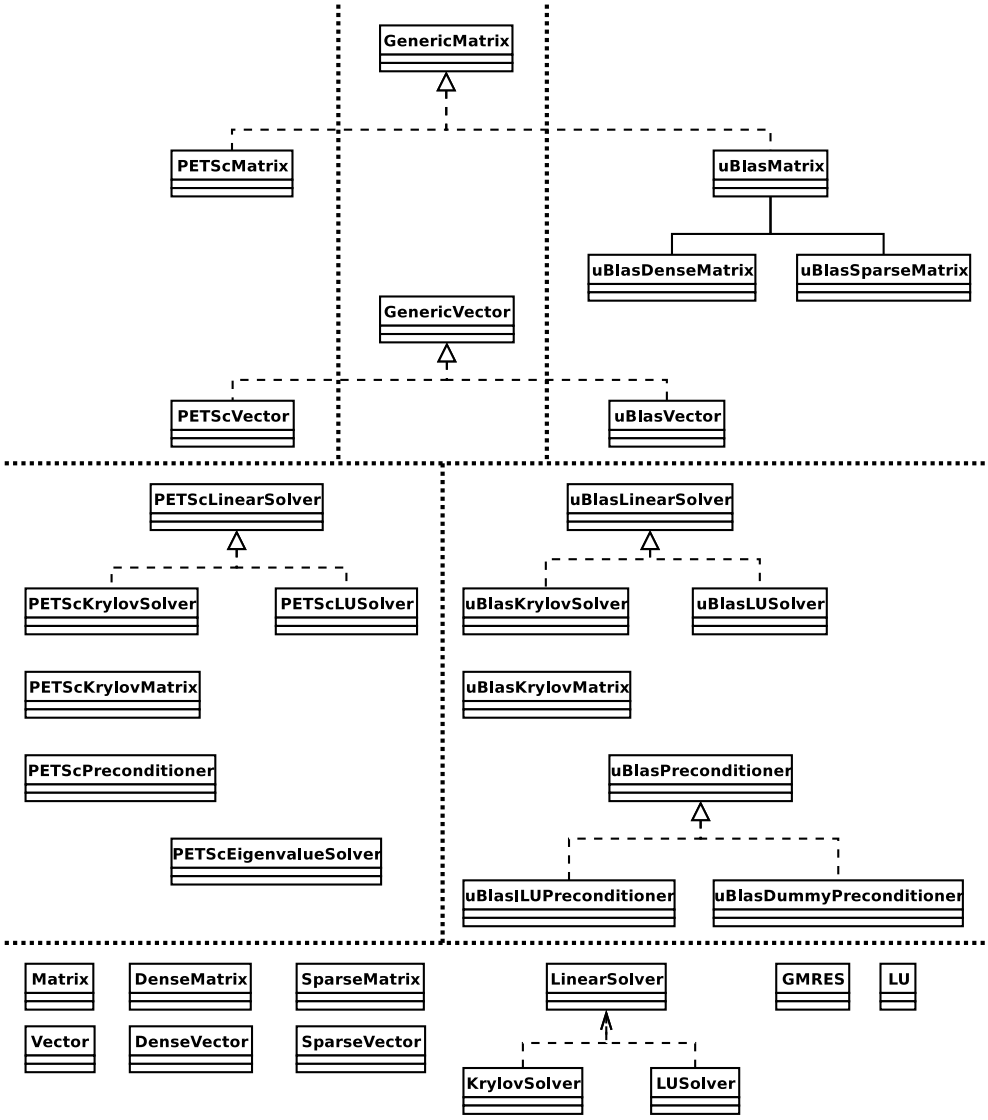

Figure C.1: Class diagram of the linear algebra classes in **DOLFIN**.

Appendix D

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.

D.1 Installing from source

D.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 and

UMFPACK. If you wish to use the Python interface **PyDOLFIN**, the libraries SWIG and NumPy are required. 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 NumPy

NumPy is required for generating the Python interface **PyDOLFIN**. It can be obtained from

```
http://numpy.scipy.org/
```

Packages are available for most Linux distributions. For Ubuntu/Debian users, the packages `python-numpy` `python-numpy-ext` should be installed.

Installing SWIG

SWIG is also required for generating the Python interface **PyDOLFIN**. It can be obtained from

```
http://www.swig.org/
```

Packages are available for most Linux distributions. For Ubuntu/Debian users, the package `swig` should be installed.

Installing UMFPACK

UMFAPCK is part of the SuiteSparse collection and is a library for the direct solution of linear systems. It is highly recommended if PESTSc is not installed. If PETSc is not installed, UMFPACK is the default direct linear solver. The SuiteSparse collection can be downloaded from

```
http://www.cise.ufl.edu/research/sparse/SuiteSparse/
```

UMFPACK is available as a package for many Linux distributions. For Ubuntu/Debian users, the packages `libumfpack4` and `libumfpack4-dev` should be installed. For other distributions, the package may be called `libufsparse`.

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 NumPy [11] (Debian package `python-numpy`) and LinearAlgebra (Debian package `python-numpy-ext`). Refer to the **FFC** manual for further details.

D.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-x.y.z.tar.gz`, and unpack using the command

```
# tar zxfv dolfin-x.y.z.tar.gz
```

This creates a directory `dolfin-x.y.z` containing the **DOLFIN** source code.

If you want the very latest version of **DOLFIN**, it can be accessed directly from the development repository through `hg` (Mercurial):

```
# hg clone http://www.fenics.org/hg/dolfin
```

This version may contain features not yet present in the latest release, but may also be less stable and even not work at all.

D.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/`.

D.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
```

D.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.

D.2 Debian package

In preparation.

D.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 NumPy is not available as a Cygwin

package and must be installed manually. To compile **DOLFIN**, the Cygwin package `libxml2-devel` must be installed. For **PyDOLFIN** the package `swig` must be installed. `NumPy` is not available as a package for Cygwin, therefore it must be installed manually if you wish to use **PyDOLFIN**. 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 E

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. Patches can be created with Mercurial or `diff`.

E.1 Creating bundles/patches

E.1.1 Creating a Mercurial (hg) bundle

Creating bundles is the preferred way of submitting patches. It has several advantages over plain diffs. If you are a frequent contributor, consider publishing your source tree so that the **FFC** maintainers (and other users) may pull your changes directly from your tree.

A bundle contains your contribution to **DOLFIN** in the form of a binary patch file generated by Mercurial [10], the revision control system used by **DOLFIN**. Follow the procedure described below to create your bundle.

1. Clone the **DOLFIN** repository:

```
# hg clone http://www.fenics.org/hg/dolfin
```

2. If your contribution consists of new files, add them to the correct location in the **DOLFIN** directory tree. Enter the **DOLFIN** directory and add these files to the local repository by typing:

```
# hg add <files>
```

   where `<files>` is the list of new files. You do not have to take any action for previously existing files which have been modified. Do not add temporary or binary files.

3. Enter the **DOLFIN** directory and commit your contribution:

```
# hg commit -m "<description>"
```

   where `<description>` is a short description of what your patch accomplishes.

4. Create the bundle:

```
# hg bundle dolfin-<identifier>-<date>.hg
  http://www.fenics.org/hg/dolfin
```

   written as one line, where `<identifier>` is a keyword that can be used to identify the bundle 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`.
   The bundle now exists as `dolfin-<identifier>-<date>.hg`.

When you add your contribution at point 2, make sure that only the files that you want to share are present by typing:

```
# hg status
```

This will produce a list of files. Those marked with a question mark are not tracked by Mercurial. You can track them by using the `add` command as shown above. Once you have added these files, their status changes form `?` to `A`.

E.1.2 Creating a standard (diff) patch file

The tool used to create a patch is called `diff` and the tool used to apply the patch is called `patch`.

Here's an example of how it works. Start from the latest release of **DOLFIN**, which we here assume is release x.y.z. You then have a directory structure under `dolfin-x.y.z` 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-x.y.z dolfin-x.y.z-mod
```

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

```
# tar zxfv dolfin-x.y.z.tar.gz
```

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

```
# ls
dolfin-x.y.z
dolfin-x.y.z-mod
```

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

```
# diff -u --new-file --recursive dolfin-x.y.z
  dolfin-x.y.z-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-x.y.z` to easily create your modified version. If the patch is large, compressing it with for example `gzip` is advisable:

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

E.2 Sending bundles/patches

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

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

Include a short description of what your patch/bundle accomplishes. Small patches/bundles 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/bundles if possible.

E.3 Applying changes

E.3.1 Applying a Mercurial bundle

You have received a patch in the form of a Mercurial bundle. The following procedure shows how to apply the patch to your version of **DOLFIN**.

1. Before applying the patch, you can check its content by entering the **DOLFIN** directory and typing:

   ```
   # hg incoming -p
     bundle://<path>/dolfin-<identifier>-<date>.hg
   ```

   written as one line, where `<path>` is the path to the bundle. `<path>` can be omitted if the bundle is in the **DOLFIN** directory. The option `-p` can be omitted if you are only interested in a short summary of the changesets found in the bundle.

2. To apply the patch to your version of **DOLFIN** type:

   ```
   # hg unbundle <path>/dolfin-<identifier>-<date>.hg
   ```

   followed by:

   ```
     # hg update
   ```

E.3.2 Applying a standard patch file

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

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

```
# tar zxfv dolfin-x.y.z.tar.gz
```

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

```
# ls
dolfin-x.y.z
dolfin-<identifier>-<date>.patch
```

   Unpack the patch file using `gunzip` if necessary.

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

```
# cd dolfin-x.y.z
```

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-x.y.z`.

E.4 License agreement

By contributing a patch to **DOLFIN**, you agree to license your contributed code under the GNU Lesser General Public License (a condition also built into the LGPL 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 LGPL Version 2.1.
//
// 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 F

Contributors

▶ *Developer's note:* List all contributors here.

Appendix G

Coding style

To streamline the **DOLFIN** source code and ease the job for maintainers that need to read and edit large amounts of code, developers should try to follow the below coding style when submitting patches to **DOLFIN**.

The guideline below is for C++ but may in some cases be extrapolated to Python.

G.1 Naming conventions

G.1.1 Class names

Use camel caps for class names:

```
class FooBar
{
  ...
};
```

G.1.2 Function names

Use lower-case for function names and underscore to separate words:

```
void foo();
void bar();
void foo_bar(...);
```

Functions returning a value should be given the name of that value, for example:

```
class Array:
{
public:

  /// Return size of array (number of entries)
  uint size() const;

};
```

In the above example, the function should be named `size` rather than `get_size`. On the other hand, a function not returning a value but rather taking a variable (by reference) and assigning a value to it, should use the `get_foo` naming scheme, for example:

```
class Parameters:
{
public:

  /// Retrieve all parameter keys
  void get_parameter_keys(std::vector<std::string>& parameter_keys) const;

};
```

G.1.3 Variable names

Use lower-case for variable names and underscore to separate words:

```
Foo foo;
Bar bar;
FooBar foo_bar;
```

G.1.4 Enum variables and constants

Enum variables should be lower-case with underscore to separate words:

```
enum Type {foo, bar, foo_bar};
```

We try to avoid using `#define` to define constants, but when necessary constants should be capitalized:

```
#define FOO 3.14159265358979
```

G.1.5 File names

Use camel caps for file names if they contain the declaration/definition of a class. Header files should have the suffix `.h` and implementation files should have the suffix `.cpp`:

```
FooBar.h
FooBar.cpp
```

Use lower-case for file names that contain utilities/functions (not classes).

G.2 Miscellaneous

G.2.1 Comments

Comment your code, and do it often. Capitalize the first letter and don't use punctuation (unless the comment runs over several sentences). Here's a good example from `TopologyComputation.cpp`:

```
// Check if connectivity has already been computed
if (connectivity.size() > 0)
  return;

// Invalidate ordering
mesh._ordered = false;

// Compute entities if they don't exist
if (topology.size(d0) == 0)
  computeEntities(mesh, d0);
if (topology.size(d1) == 0)
  computeEntities(mesh, d1);

// Check if connectivity still needs to be computed
if (connectivity.size() > 0)
  return;

...
```

G.2.2 Integers and reals

Use `dolfin::uint` instead of `int` (unless you really want to use negative integers which is rare) and `dolfin::real` instead of `double`:

```
uint i = 0;
double x = 0.0;
```

These are typedefs for the standard C++ types `unsigned int` and `double` (defined in `dolfin/common/types.h`).

G.2.3 Placement of brackets

Curly brackets following a control statement should appear in the next line and not be indented:

```
for (uint i = 0; i < 10; i++)
{
  ...
}
```

G.2.4 Indentation

Indentation should be two spaces and it should be spaces, *not* tab(s).

G.2.5 Header file layout

Header files should follow the below template:

```
// Copyright (C) 2008 Foo Bar.
// Licensed under the GNU LGPL Version 2.1.
//
// Modified by Bar Foo, 2008.
//
// First added:  2008-01-01
// Last changed: 2008-02-01
```

```
#ifndef __FOO_H
#define __FOO_H

namespace dolfin
{

  class Bar; // Forward declarations here

  /// Documentation of class

  class Foo
  {
  public:

    ...

  private:

    ...

  };

}

#endif
```

G.2.6 Implementation file layout

Implementation files should follow the below template:

```
// Copyright (C) 2008 Foo Bar.
// Licensed under the GNU LGPL Version 2.1.
//
// Modified by Bar Foo, 2008.
```

```
//
// First added:  2008-01-01
// Last changed: 2008-02-01

#include <dolfin/Foo.h>

using namespace dolfin;

//-----------------------------------------------------------
Foo::Foo() : // variable initialization here
{
  ...
}
//-----------------------------------------------------------
Foo::~Foo()
{
  // Do nothing
}
//-----------------------------------------------------------
```

The horizontal lines above should be exactly 79 characters wide but have been shortened here to fit the page.

G.2.7 Including header files

Don't use `#include <dolfin.h>` or `#include <dolfin/dolfin_foo.h>` inside the DOLFIN kernel. Only include the portions of DOLFIN you are actually using.

G.2.8 Forward declarations

Actually, try to include as little as possible and use forward declarations whenever possible (in header files). Put the `#include` in the implementation file.

G.2.9 Explicit constructors

Make all constructors (except copy constructors) explicit if there is no particular reason not to do so:

```
class Foo
{
  explicit Foo(uint i);
};
```

G.2.10 Virtual functions

Always declare inherited virtual functions as virtual in the subclasses. This makes it easier to spot which functions are virtual.

```
class Foo
{
  virtual void foo();
  virtual void bar() = 0;
};

class Bar
{
  virtual void foo();
  virtual void bar();
};
```

G.3 Use of libraries

G.3.1 Prefer C++ strings and streams to old C-style `char*`

Use std::string instead of `const char*` and use std::istream and std::ostream instead of `FILE`. Avoid `printf`, `sprintf` and the like.

There are exceptions to this rule where we need to use old C-style function calls. One such exception is handling of command-line arguments (`char* argv[]`).

Appendix H

License

DOLFIN is licensed under the GNU Lesser General Public License (LGPL) version 2.1, included verbatim below.

```
  GNU LESSER GENERAL PUBLIC LICENSE
       Version 2.1, February 1999

 Copyright (C) 1991, 1999 Free Software Foundation, Inc.
 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]

    Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

  This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it.  You
can use it too, but we suggest you first think carefully about whether
```

this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

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

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and

is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

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

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, 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 library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete 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 distribute a copy of this License along with the Library.

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 Library or any portion
of it, thus forming a work based on the Library, 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) The modified work must itself be a software library.

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

    c) You must cause the whole of the work to be licensed at no
    charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a
    table of data to be supplied by an application program that uses
    the facility, other than as an argument passed when the facility
    is invoked, then you must make a good faith effort to ensure that,
    in the event an application does not supply such function or
    table, the facility still operates, and performs whatever part of
    its purpose remains meaningful.

    (For example, a function in a library to compute square roots has
    a purpose that is entirely well-defined independent of the
    application.  Therefore, Subsection 2d requires that any
    application-supplied function or table used by this function must
    be optional: if the application does not supply it, the square
    root function must still compute square roots.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
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 Library, 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 Library.

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

  3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.

  Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of
the Library into a program that is not a library.

  4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you 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.

  If distribution of 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 satisfies the requirement to
distribute the source code, even though third parties are not
compelled to copy the source along with the object code.

  5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a "work that uses the Library".  Such a
work, in isolation, is not a derivative work of the Library, and
therefore falls outside the scope of this License.

  However, linking a "work that uses the Library" with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a "work that uses the
```

```
library".  The executable is therefore covered by this License.
Section 6 states terms for distribution of such executables.

  When a "work that uses the Library" uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is not.
Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library.  The
threshold for this to be true is not precisely defined by law.

  If such an object file uses only numerical parameters, data
structure layouts and accessors, and small macros and small inline
functions (ten lines or less in length), then the use of the object
file is unrestricted, regardless of whether it is legally a derivative
work.  (Executables containing this object code plus portions of the
Library will still fall under Section 6.)

  Otherwise, if the work is a derivative of the Library, you may
distribute the object code for the work under the terms of Section 6.
Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.

  6. As an exception to the Sections above, you may also combine or
link a "work that uses the Library" with the Library to produce a
work containing portions of the Library, and distribute that work
under terms of your choice, provided that the terms permit
modification of the work for the customer's own use and reverse
engineering for debugging such modifications.

  You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License.  You must supply a copy of this License.  If the work
during execution displays copyright notices, you must include the
copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License.  Also, you must do one
of these things:

    a) Accompany the work with the complete corresponding
    machine-readable source code for the Library including whatever
    changes were used in the work (which must be distributed under
    Sections 1 and 2 above); and, if the work is an executable linked
    with the Library, with the complete machine-readable "work that
    uses the Library", as object code and/or source code, so that the
    user can modify the Library and then relink to produce a modified
    executable containing the modified Library.  (It is understood
```

```
    that the user who changes the contents of definitions files in the
    Library will not necessarily be able to recompile the application
    to use the modified definitions.)

    b) Use a suitable shared library mechanism for linking with the
    Library.  A suitable mechanism is one that (1) uses at run time a
    copy of the library already present on the user's computer system,
    rather than copying library functions into the executable, and (2)
    will operate properly with a modified version of the library, if
    the user installs one, as long as the modified version is
    interface-compatible with the version that the work was made with.

    c) Accompany the work with a written offer, valid for at
    least three years, to give the same user the materials
    specified in Subsection 6a, above, for a charge no more
    than the cost of performing this distribution.

    d) If distribution of the work is made by offering access to copy
    from a designated place, offer equivalent access to copy the above
    specified materials from the same place.

    e) Verify that the user has already received a copy of these
    materials or that you have already sent this user a copy.

  For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it.  However, as a special exception,
the materials to be 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.

  It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system.  Such a contradiction means you cannot
use both them and the Library together in an executable that you
distribute.

  7. You may place library facilities that are a work based on the
Library side-by-side in a single library together with other library
facilities not covered by this License, and distribute such a combined
library, provided that the separate distribution of the work based on
the Library and of the other library facilities is otherwise
permitted, and provided that you do these two things:
```

```
    a) Accompany the combined library with a copy of the same work
    based on the Library, uncombined with any other library
    facilities.  This must be distributed under the terms of the
    Sections above.

    b) Give prominent notice with the combined library of the fact
    that part of it is a work based on the Library, and explaining
    where to find the accompanying uncombined form of the same work.

  8. You may not copy, modify, sublicense, link with, or distribute
the Library except as expressly provided under this License.  Any
attempt otherwise to copy, modify, sublicense, link with, or
distribute the Library 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.

  9. 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 Library or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Library or works based on it.

  10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
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 with
this License.

  11. 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 Library at all.  For example, if a patent
license would not permit royalty-free redistribution of the Library 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 Library.

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.

  12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library 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.

  13. The Free Software Foundation may publish revised and/or new
versions of the Lesser 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 Library
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 Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.

  14. If you wish to incorporate parts of the Library into other free
programs whose distribution conditions are incompatible with these,
write to the author to ask for permission.  For software which is
```

```
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this.  Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.

    NO WARRANTY

  15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU.  SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

     END OF TERMS AND CONDITIONS

           How to Apply These Terms to Your New Libraries

  If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change.  You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of the
ordinary General Public License).

  To apply these terms, attach the following notices to the library.  It is
safest to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.

    <one line to give the library's name and a brief idea of what it does.>
```

```
    Copyright (C) <year>  <name of author>

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the library, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  library ‘Frob’ (a library for tweaking knobs) written by James Random Hacker.

  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice

That’s all there is to it!
```

Index

`File`, 57
`GMRES`, 27
`GenericMatrix`, 26
`GenericVector`, 26
`KrylovSolver`, 27
`LUSolver`, 28
`LU`, 28
`Matrix`, 23
`Progress`, 69
`Vector`, 23
`add()`, 73
`cout`, 65
`dolfin_assert()`, 67
`dolfin_begin()`, 68
`dolfin_debug()`, 67
`dolfin_end()`, 68
`dolfin_get()`, 71
`dolfin_set()`, 72
`endl`, 65
`error()`, 66
`info()`, 65
`warning()`, 66

algebraic multigrid, 28
AMG, 28
assertions, 67

BiCGStab, 27
Boost, 98
bundle, 107, 110, 111

coding style, 117
compiling, 101, 103
conjugate gradient method, 27
contact, 12
contributing, 107
Cygwin, 104

Debian package, 104
debugging, 67
demo programs, 103
dense matrix, 26
dependencies, 97
diff, 109
direct methods, 28
downloading, 13, 101

eigenvalue problems, 29
eigenvalue solver, 29
enumeration, 12
errors, 66

FFC, 100
ffc, 15
FIAT, 101
file formats, 59
Function, 39
functions, 39

GMRES method, 27
GNU Lesser General Public License, 127

hexahedron, 83

hg, 107, 111

I/O, 57
ILU, 28
incomplete LU factorization, 28
indices, 12
input/output, 57
installation, 13, 97
interval, 80
iterative methods, 27

Jacobi, 28

Krylov subspace methods, 27

LGPL, 127
Libxml2, 98
license, 112, 127
linear algebra backends, 30
linear systems, 26
log system, 65
LU factorization, 28

Matrix Template Library, 31
matrix-free solvers, 28
MayaVi, 41
Mercurial, 107, 111
mesh entity, 85
MTL4, 31

Newton's method, 52
NewtonSolver, 52
nonlinear solver, 51
NonlinearFunction, 51
numbering, 85
NumPy, 98

object, 57
output destination, 70

parameters, 71
ParaView, 41
partial differential equations, 49
patch, 109–111
PETSc, 30, 99
Poisson's equation, 14
post-processing, 57
pre-processing, 57
preconditioners, 28
progress bar, 69
PyDOLFIN, 75
Python, 75

quadrilateral, 81
quickstart, 13

reference cells, 79

SLEPc, 29
SOR, 28
source code, 101
sparse matrix, 25
successive over-relaxation, 28
SWIG, 99

tasks, 68
tetrahedron, 82
topological dimension, 85
triangle, 80
Trilinos, 31
typographic conventions, 12

uBLAS, 30
UMFPACK, 99
user-defined functions, 42

vertex numbering, 86
virtual matrix, 28

warnings, 66

XML, 59, 74