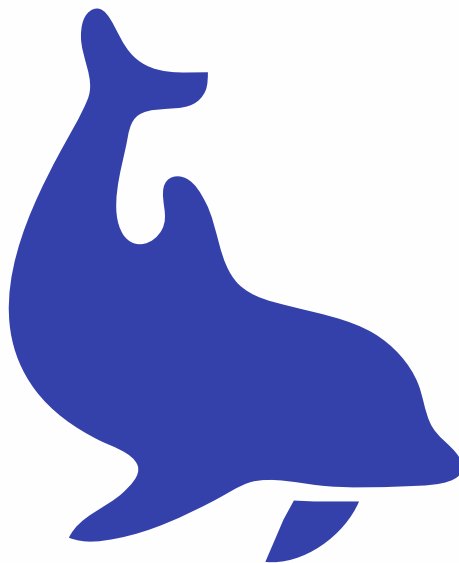


DOLFIN User Manual

September 23, 2005



Hoffman, Jansson, Logg

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	8
1 Introduction	11
1.1 The FEniCS project	11
1.2 The finite element method	11
1.3 Overview	12
2 Quickstart	13
2.1 Downloading and installing DOLFIN	13
2.2 Solving Poisson's equation with DOLFIN	14
2.2.1 Setting up the variational formulation	15
2.2.2 Writing the solver	15
2.2.3 Compiling the program	20
2.2.4 Running the program	20
2.2.5 Visualizing the solution	21

3	Linear algebra	23
3.1	The Matrix class	23
3.2	The VirtualMatrix class	23
3.3	The Vector class	24
3.4	The LinearSolver class	24
3.5	The GMRES class	24
3.6	The LU class	24
3.7	The EigenvalueSolver class	25
3.8	The Preconditioner class	25
3.9	The PETScManager class	25
3.10	The PETSc system	25
3.11	The Hypre system	25
4	Functions	27
5	The mesh	29
6	Ordinary differential equations	31
7	Partial differential equations	33
7.1	Boundary value problems	33
7.2	Variational formulation	33
7.3	Compiling the variational form with FFC	34

7.4	The FEM class	35
7.5	The BilinearForm class	36
7.6	The LinearForm class	36
7.7	The AffineMap class	37
7.8	The FiniteElement class	37
7.9	The PDE class	38
7.10	Computation of Element matrices and vectors	38
7.11	Boundary conditions	39
7.12	Finite elements	39
7.13	Initial value problems	39
8	Input/output	41
8.1	Pre- and post-processing	41
8.2	Files and objects	41
8.3	File formats	41
8.3.1	DOLFIN XML	42
8.3.2	Another format	42
8.3.3	Another format	42
8.3.4	Another format	42
8.4	Adding a new file format	42
9	The log system	43

9.1	Generating log messages	43
9.2	Warnings and errors	44
9.3	Debug messages and assertions	45
9.4	Task notification	46
9.5	Progress bars	47
9.6	Controlling the destination of output	48
10	Parameters	51
10.1	Retrieving the value of a parameter	51
10.2	Modifying the value of a parameter	52
10.3	Adding a new parameter	53
10.4	Saving parameters to file	54
10.5	Loading parameters from file	54
11	Solvers	55
11.1	Poisson’s equation	56
11.1.1	Usage	56
11.1.2	Performance	57
11.1.3	Limitations	57
11.2	Convection–diffusion	57
11.2.1	Usage	58
11.2.2	Performance	59

11.2.3	Limitations	59
11.3	Incompressible Navier–Stokes	59
11.3.1	Usage	59
11.3.2	Performance	60
11.3.3	Limitations	60
11.4	Elasticity	60
11.4.1	Usage	61
11.4.2	Performance	61
11.4.3	Limitations	61
A	Reference elements	63
A.1	The reference triangle	63
A.2	The reference tetrahedron	65
A.3	Ordering of degrees of freedom	66
A.3.1	Mesh entities	66
A.3.2	Ordering among mesh entities	69
A.3.3	Internal ordering on edges	69
A.3.4	Alignment of edges	70
A.3.5	Internal ordering on faces	70
A.3.6	Alignment of faces	70
B	Installation	73

B.1	Installing from source	73
B.1.1	Dependencies and requirements	73
B.1.2	Downloading the source code	75
B.1.3	Compiling the source code	76
B.1.4	Compiling the demo programs	77
B.1.5	Compiling a program against DOLFIN	77
B.2	Debian package	78
C	Contributing code	79
C.1	Creating a patch	79
C.2	Sending patches	80
C.3	Applying a patch (maintainers)	81
C.4	License agreement	82
D	License	83

About this manual

This manual is currently being written. A first version of this manual should be ready sometime in the fall of 2005.

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: $\frac{\partial}{\partial x_0}, \frac{\partial}{\partial x_1}, \dots, \frac{\partial}{\partial x_{n-1}}$.

Contact

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

`dolfin-dev@fenics.org`

Chapter 1

Introduction

FIXME: Automation of CMM, FEniCS, purpose of DOLFIN: PSE for differential equations, C++ interface of FEniCS, etc

1.1 The FEniCS project

FIXME: Automation of CMM, other components of **FEniCS**

1.2 The finite element method

FIXME: Automation of discretization

1.3 Overview

FIXME: Component diagram, user, module, kernel

FIXME: Write about `real`, `uint`, namespace `dolfin`

Chapter 2

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 B for detailed installation instructions and Chapter 7 for a detailed discussion of how to solve partial differential equations with **DOLFIN**.

2.1 Downloading and installing DOLFIN

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

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

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

```
# tar zxfv dolfin-0.1.0.tar.gz
# cd dolfin-0.1.0
# make
# make install
```

Note that you may need to be root on your system to do the last step. **DOLFIN** depends on a number of other packages, including the linear algebra package PETSc and the form compiler **FFC**. (See Appendix B for detailed instructions.)

2.2 Solving Poisson's equation with DOLFIN

Let's say that we want to solve Poisson's equation on the unit square $\Omega = (0, 1) \times (0, 1)$ with homogeneous Dirichlet boundary conditions on the boundary $\Gamma_0 = \{(x, y) \in \partial\Omega : x = 1\}$, homogeneous Neumann boundary conditions on the remaining part of the boundary and right-hand side given by $f = x \sin(y)$,

$$-\Delta u(x, y) = x \sin(y), \quad x \in \Omega = (0, 1) \times (0, 1), \quad (2.1)$$

$$u(x) = 0, \quad x \in \Gamma_0 = \{(x, y) \in \partial\Omega : x = 1\}, \quad (2.2)$$

$$\partial_n u(x) = 0, \quad x \in \partial\Omega \setminus \Gamma_0. \quad (2.3)$$

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

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \quad (2.4)$$

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

$$a(v, u) = \int_{\Omega} \nabla u \cdot \nabla v \, dx \quad (2.5)$$

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

$$L(v) = \int_{\Omega} f v \, dx. \quad (2.6)$$

2.2.1 Setting up the variational formulation

The variational formulation (2.4) must be given to **DOLFIN** as a pair of bilinear and linear forms (a, L) using the form compiler **FFC**. This is done by entering the definition of the forms in a text file with extension `.form`, e.g. `Poisson.form`, as follows:

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

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

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

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

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

```
# ffc Poisson.form
```

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

2.2.2 Writing the solver

Having now compiled the variational formulation (2.4) 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 `src/demo/poisson/` of the **DOLFIN** source tree.

At the beginning of our C++ program, which we write in a text file named `main.cpp`, we must first include the header file `dolfin.h`, which gives our program access to the **DOLFIN** class library. In addition, we include the header file `Poisson.h` generated by the form compiler. Since all classes of 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;
```

Next, we specify the right-hand side f of (2.1). This is done by defining a new subclass of `Function`, which we here will name `MyFunction`, and overloading the evaluation operator to return the value $f(x, y) = x \sin(y)$:

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

The boundary condition is specified similarly, by overloading the evaluation operator for a subclass of `BoundaryCondition`:

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


We only need to specify the boundary condition explicitly on the Dirichlet boundary. On the remaining part of the boundary, **DOLFIN** assumes homogeneous Neumann boundary conditions by default.

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

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

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

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

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

Next, we initialize the right-hand side, the boundary condition and the pair of forms that we have previously defined:

```
MyFunction f;
MyBC bc;
Poisson::BilinearForm a;
Poisson::LinearForm L(f);
```

All that remains is now to assemble the linear system $Ax = b$ representing the variational problem (2.4) and solve for the vector x . To assemble the system, we define a `Matrix A`, a `Vector b` and call the function `FEM::assemble()`:

```
Matrix A;
Vector x, b;
FEM::assemble(a, L, A, b, mesh, bc);
```

We may then solve the linear system $Ax = b$ for the degrees of freedom of the solution u using the GMRES method:

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

Finally, we export the solution to a file for visualization. To do this, we define a `Function` which represents a field with given degrees of freedom in a function space defined by a mesh and a finite element, which we may obtain from the bilinear form by calling the member function `trial()`. Here, we choose to save the solution in Octave/MATLAB format, which we do by specifying a file name with extension `.m`:

```
Function u(x, mesh, a.trial());
File file('poisson.m');
file << u;
```

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

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

using namespace dolfin;

// Right-hand side
class MyFunction : public Function
{
  real operator() (const Point& p) const
  {
```

```
        return p.x*sin(p.y);
    }
};

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

int main()
{
    // Set up problem
    UnitSquare mesh(16, 16);
    MyFunction f;
    MyBC bc;
    Poisson::BilinearForm a;
    Poisson::LinearForm L(f);

    // Assemble linear system
    Matrix A;
    Vector x, b;
    FEM::assemble(a, L, A, b, mesh, bc);

    // Solve the linear system
    GMRES solver;
    solver.solve(A, x, b);

    // Save function to file
    Function u(x, mesh, a.trial());
    File file("poisson.m");
    file << u;

    return 0;
}
```

2.2.3 Compiling the program

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

```
CFLAGS = 'dolphin-config --cflags_dolphin'
LIBS    = 'dolphin-config --libs_dolphin'
CXX     = 'dolphin-config --compiler'

DEST    = dolphin-poisson
OBJECTS = main.o

all: $(DEST)

install:

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

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

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

With the **Makefile** in place, we just need to type **make** to compile the program, generating the executable as the file **dolphin-poisson**.

2.2.4 Running the program

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

```
# ./dolphin-poisson
Computing mesh connectivity:
Found 289 nodes
Found 512 cells
[...]
GMRES converged in 21 iterations.
```

```
Saved mesh mesh [...]
Saved function u [...]
```

2.2.5 Visualizing the solution

DOLFIN relies on external programs for visualization. In this example we chose to save the solution in Octave/MATLAB format, so we start Octave (or MATLAB) using the command `octave`. Then, in Octave, we import the solution and plot it using the `pdesurf` command:

```
octave:1> poisson
octave:2> pdesurf(points, cells, u)
```

Note that the commands `pdemesh`, `pdesurf` and `pdeplot` are not included with a standard installation of Octave, but are available in the subdirectory `src/utils/octave/` of the **DOLFIN** source tree.

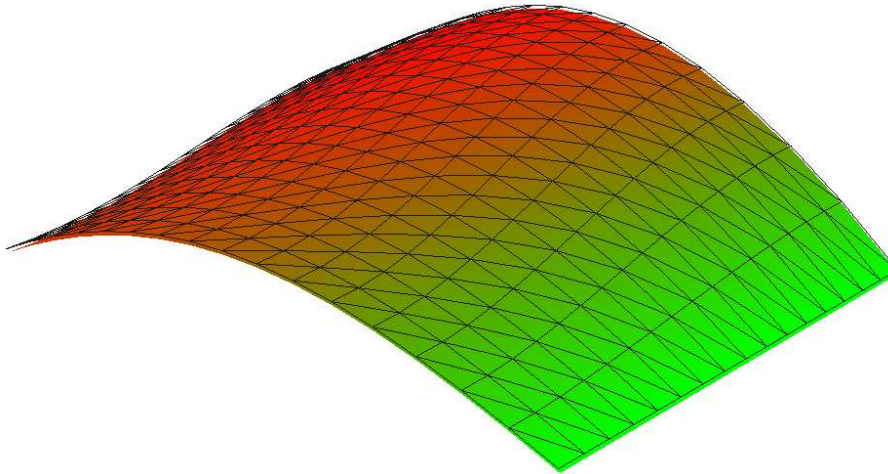


Figure 2.1: The solution of Poisson's equation (2.1) visualized in Octave.

Chapter 3

Linear algebra

DOLFIN uses PETSc for the linear algebra. For convenience **DOLFIN** provide wrappers for some of the most common linear algebra functionality.

3.1 The Matrix class

The matrix class represents a matrix of dimension $m \times n$. It is a simple wrapper for a PETSc matrix `Mat`. The interface is intentionally simple. For advanced usage, access the PETSc Mat pointer using the function `mat()` and use the standard PETSc interface.

3.2 The VirtualMatrix class

This class represents a matrix-free matrix of dimension $m \times m$. It is a simple wrapper for a PETSc shell matrix. The interface is intentionally simple. For advanced usage, access the PETSc Mat pointer using the function `mat()` and use the standard PETSc interface.

The class `VirtualMatrix` enables the use of Krylov subspace methods for lin-

ear systems $Ax = b$, without having to explicitly store the matrix A . All that is needed is that the user-defined `VirtualMatrix` implements multiplication with vectors. Note that the multiplication operator needs to be defined in terms of PETSc data structures (`Vec`), since it will be called from PETSc.

3.3 The Vector class

The vector class represents a vector of dimension n . It is a simple wrapper for a PETSc vector `Vec`. The interface is intentionally simple. For advanced usage, access the PETSc `Vec` pointer using the function `vec()` and use the standard PETSc interface.

3.4 The LinearSolver class

This class defines the interface of all linear solvers for systems of the form $Ax = b$.

3.5 The GMRES class

This class implements the GMRES method for linear systems of the form $Ax = b$. It is a wrapper for the GMRES solver of PETSc.

3.6 The LU class

This class implements the direct solution (LU factorization) for linear systems of the form $Ax = b$. It is a wrapper for the LU solver of PETSc.

3.7 The EigenvalueSolver class

This class computes eigenvalues of a matrix. It is a wrapper for the eigenvalue solver of PETSc.

3.8 The Preconditioner class

This class specifies the interface for user-defined Krylov method preconditioners. A user wishing to implement her own preconditioner needs only supply a function that approximately solves the linear system given a right-hand side.

3.9 The PETScManager class

This class is responsible for initializing and (automatically) finalizing PETSc. To initialize PETSc, call `PETScManager::init()` once (additional calls will be ignored). Finalization will be handled automatically.

3.10 The PETSc system

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.

3.11 The Hypre system

As a complement to PETSc, **DOLFIN** also uses Hypre, which is a library for solving large, sparse linear systems of equations on massively parallel computers.

To use a preconditioner from Hypre with your PETSc solver in **DOLFIN**, write

```
PCSetType(PC pc, PCHYPRE );  
PCHYPRESetType(PC pc, "boomeramg");
```

In particular, the above preconditioner **boomeramg** is an algebraic multigrid preconditioner, which may be very useful for some problems.

FIXME: Write about the wrappers, PETSc, using `mat()` and `vec()` to do more advanced operations with PETSc etc.

Chapter 4

Functions

FIXME: Discuss the Function class and the different representations.

Chapter 5

The mesh

FIXME: Triangular, tetrahedral, include some images, mesh refinement, connectivity, iterators, file formats, local ordering
--

Chapter 6

Ordinary differential equations

FIXME: Mono-adaptive, multi-adaptive, ODE base class, simple example, error control, adaptivity, complex ODE, implicit, homotopies

Chapter 7

Partial differential equations

7.1 Boundary value problems

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

$$\begin{aligned} -\Delta u(x) &= f(x) & x \in \Omega \subset \mathbb{R}^d \\ u(x) &= 0 & x \in \partial\Omega. \end{aligned} \tag{7.1}$$

7.2 Variational formulation

A variational formulation of (7.1) take the form: find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall v \in V, \tag{7.2}$$

where $a(\cdot, \cdot) : V \times V \rightarrow \mathbb{R}$ is a bilinear form on V defined by

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i} \, dx, \tag{7.3}$$

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

$$L(v) = \int_{\Omega} f v \, dx. \quad (7.4)$$

$V = H_0^1(\Omega)$ is the standard Sobolev space of square integrable functions with also their first derivatives square integrable (in the Lebesgue sense), with the functions being zero on the boundary (in the sense of traces).

The Finite Element Method FEM for (7.2) is now: find $U \in V_h$ such that

$$a(U, v) = L(v) \quad \forall v \in V_h, \quad (7.5)$$

where $V_h \subset V$ is a finite dimensional subspace of dimension N . The finite element space V_h is characterized by the set of basis functions $\{\varphi_i\}_{i=1}^N$, and thus the FEM method (7.5) is specified by the variational form and the basis functions of V_h .

7.3 Compiling the variational form with FFC

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

At `/dolfin/src/demo/solvers/poisson/dolfin/` the following `poisson.form` file for (7.5) can be found

```
# Copyright (C) 2005 Johan Hoffman and Anders Logg.
# Licensed under the GNU GPL Version 2.
#
# First added: 2005-04-04
# Last changed: 2005
#
# The bilinear form a(u,v) and linear form L(v) for
# Poisson's equation.
#
# Compile this form with FFC: ffc poisson.form.
```

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

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

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

Compiling the file with

```
# ffc Poisson.form
```

generate a file `Poisson.h` containing the classes `BilinearForm` and `LinearForm`, and classes for the finite element used in the forms.

7.4 The FEM class

The FEM class automates the assembly algorithm, constructing a linear system of equations from a given partial differential equation, specified as a variational problem (7.2), by a bilinear form $a(\cdot, \cdot)$ and a linear form $L(\cdot)$, according to

```
/// Assemble bilinear and linear forms
static void assemble(BilinearForm& a, LinearForm& L,
                    Matrix& A, Vector& b, Mesh& mesh);
```

The assemble function is called in the following way

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

where A and b is a matrix and vector respectively.

7.5 The BilinearForm class

BilinearForm represents a bilinear form $a(u, v)$ with arguments v and u basis functions of the finite element space defined by a pair of finite elements (test and trial).

This class is automatically generated by FFC when compiling the variational form.

The function used in the assembly algorithm is an eval function

```
/// Compute element matrix (interior contribution)
virtual void eval(real block[], const AffineMap& map) const;
```

where the element matrix is returned in `block[]`, and `map` describes the affine map used for mapping the reference element to the actual element.

7.6 The LinearForm class

LinearForm represents a linear form $L(v)$ with argument v (the test function) a basis function of the finite element space defined by a finite element.

This class is automatically generated by FFC when compiling the variational form.

The function used in the assembly algorithm is an eval function

```
/// Compute element vector (interior contribution)
virtual void eval(real block[], const AffineMap& map) const;
```

where the element vector is returned in `block[]`, and `map` describes the affine map used for mapping the reference element to the actual element.

7.7 The AffineMap class

This class represents the affine map from the reference element to the current element.

The 2D reference element is given by $(0,0)-(1,0)-(0,1)$. The 3D reference element is given by $(0,0,0)-(1,0,0)-(0,1,0)-(0,0,1)$.

The dimension d of the map is automatically determined from the arguments used when calling the map.

The map is initialized by the function

```
/// Update map for current element
void update(const Cell& cell);
```

When `update` is called the following local variables in the class are computed for the current cell

```
// Determinant of Jacobian of map
real det;

// Jacobian of map
real f00, f01, f02, f10, f11, f12, f20, f21, f22;

// Inverse of Jacobian of map
real g00, g01, g02, g10, g11, g12, g20, g21, g22;
```

7.8 The FiniteElement class

This is the base class for finite elements automatically generated by FFC.

```
/// Return dimension of the finite element space
virtual unsigned int spacedim() const = 0;
```

```
/// Return dimension of the underlying shape
virtual unsigned int shapedim() const = 0;

/// Return vector dimension of the finite element space
virtual unsigned int tensordim(unsigned int i) const = 0;

/// Return vector dimension of the finite element space
virtual unsigned int rank() const = 0;

/// Compute map from local to global degrees of freedom
virtual void dofmap(int dofs[], const Cell& cell,
                   const Mesh& mesh) const = 0;

/// Compute map from local to global coordinates
virtual void pointmap(Point points[], uint components[],
                     const AffineMap& map) const = 0;
```

7.9 The PDE class

A PDE represents a (linearized) partial differential equation, given by a bilinear form a and a linear form L .

```
/// Constructor
PDE(BilinearForm& a, LinearForm& L);
```

7.10 Computation of Element matrices and vectors

divide element matrix into geometry tensor and integration over reference element FErari

precomputation of integrals, quadrature, tensorrepresentation factored out, FFC

7.11 Boundary conditions

7.12 Finite elements

Finite Element by Ciarlet, FIAT

7.13 Initial value problems

semidiscretization, space-time FEM,

Chapter 8

Input/output

8.1 Pre- and post-processing

FIXME: **DOLFIN** relies on external programs for pre- and post-processing

8.2 Files and objects

FIXME: Discuss operators `>>` and `<<`

8.3 File formats

FIXME: Insert table here of filename suffixes and corresponding formats.

8.3.1 DOLFIN XML

FIXME: The native format

8.3.2 Another format

8.3.3 Another format

8.3.4 Another format

8.4 Adding a new file format

FIXME: Discuss classes File, GenericFile etc

Chapter 9

The log system

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

9.1 Generating log messages

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

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

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

```
dolfin_info('Solving linear system.');
```

```
dolfin_info('Size of vector: %d.', x.size());
```

```
dolfin_info('R = %.3e (TOL = %.3e)', R, TOL);
```

As an alternative to `dolphin_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 `dolphin::cout` and `dolphin::endl` from the `dolphin` namespace, corresponding to the standard `std::cout` and `std::endl` in namespace `std`. If log messages are directed to standard output (see below), then `dolphin::cout` and `std::cout` may be mixed freely.

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

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

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

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

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

9.2 Warnings and errors

Warnings and error messages can be generated using the macros

```
dolphin_warning(message);
dolphin_error(message);
```

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

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

```
dolfin_error('GMRES solver did not converge.');
```

```
dolfin_error1('Unable to find face opposite to node %d.', n);
```

```
dolfin_error2('Unable to find edge between nodes %d and %d.', n0, n1);
```

9.3 Debug messages and assertions

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

```
dolfin_debug(message);
```

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

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

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

```
dolfin_assert(check);
```

This macro accepts a boolean expression and if the expression evaluates to false, an error message is displayed, including the file, function name and

line number of the assertion, and a segmentation fault is raised (to enable easy attachment to a debugger). The following examples illustrate the use of `dolfin_assert()`:

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

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

9.4 Task notification

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

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

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

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

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

9.5 Progress bars

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

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

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

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

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

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

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

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

```
dolphin_set(‘‘progress step’’, 0.01);
```

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

9.6 Controlling the destination of output

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

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

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

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

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


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


Chapter 10

Parameters

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(const char* 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:

```
real 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 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(const char* key, ...);
```

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

```
void dolfin_parameter(Parameter::Type type,
                      const char* key, ...);
```

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

Possible values for `type` are

- `Parameter::REAL`, corresponding to `real`;
- `Parameter::INT`, corresponding to `int`;
- `Parameter::BOOL`, corresponding to `bool`;
- `Parameter::STRING`, corresponding to `std::string`.

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

```
dolfin_parameter(Parameter::REAL, 'tolerance', 0.01);
dolfin_parameter(Parameter::INT, 'number of samples', 10);
dolfin_parameter(Parameter::BOOL, 'solve dual problem', true);
dolfin_parameter(Parameter::STRING, 'file name', 'solution.xml');
```

10.4 Saving parameters to file

To save the current database of parameters to a file in **DOLFIN** XML format, use the function `dolfin_save()` available in the `dolfin` namespace:

```
void dolfin_save(const char* filename);
```

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

To load a set of parameters from a file into the parameter database, use the function `dolfin_load()` available in the `dolfin` namespace:

```
void dolfin_load(const char* filename);
```

This function accepts as argument the name of a file containing a list of a parameters in **DOLFIN** XML format, as illustrated below:

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

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

Chapter 11

Solvers

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

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

A solver for a PDE should provide the following interface:

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

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

11.1 Poisson's equation

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

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

The variational formulation is given by

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

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

11.1.1 Usage

The API for the Poisson solver:

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

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

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


A simple example of using the solver:

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

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

    return 0;
}
```

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

11.1.2 Performance

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

11.1.3 Limitations

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

11.2 Convection–diffusion

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

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

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

The variational formulation is:

FIXME: Stabilized convection-diffusion

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

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

11.2.1 Usage

The API for the convection-diffusion solver:

```

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

// Solve convection-diffusion
void solve();

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

```

A simple example of using the solver:

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

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

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

    return 0;
}
```

11.2.2 Performance

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

11.2.3 Limitations

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

11.3 Incompressible Navier–Stokes

Write introduction here, equations etc.

11.3.1 Usage

Present API of solver and give an example.

11.3.2 Performance

Write something about the performance of the solver.

11.3.3 Limitations

Write something about the limitations of the solver.

11.4 Elasticity

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

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

The variational form:

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

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

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

11.4.1 Usage

Present API of solver and give an example.

11.4.2 Performance

Write something about the performance of the solver.

11.4.3 Limitations

Write something about the limitations of the solver.

Appendix A

Reference elements

A.1 The reference triangle

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

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

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

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

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

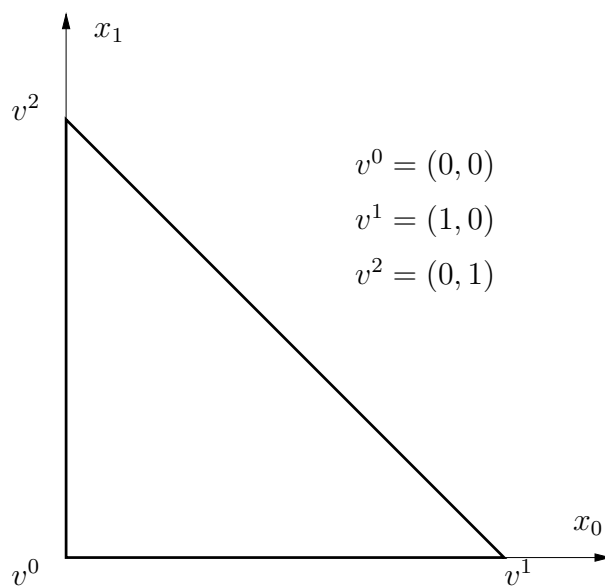


Figure A.1: Physical coordinates of the reference triangle.

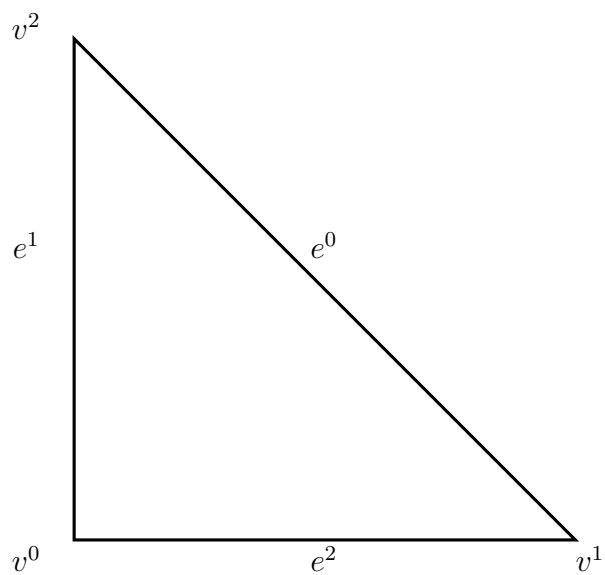


Figure A.2: Ordering of mesh entities (vertices and edges) for the reference triangle.

A.2 The reference tetrahedron

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

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

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

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

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

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

The ordering of vertices on faces implicitly defines an ordering of edges on

faces by identifying an edge on a face with the opposite vertex on the face:

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

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

A.3 Ordering of degrees of freedom

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

A.3.1 Mesh entities

We distinguish between mesh entities of different topological dimensions:

<i>vertices</i>	topological dimension 0
<i>edges</i>	topological dimension 1
<i>faces</i>	topological dimension 2
<i>cells</i>	topological dimension 2 or 3

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

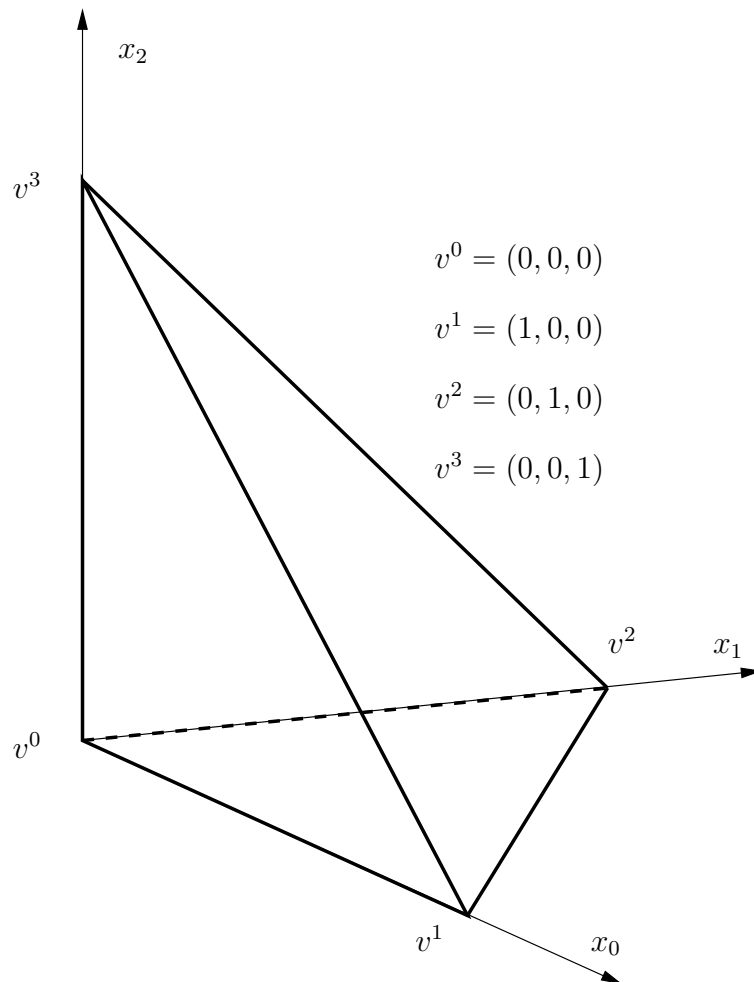


Figure A.3: Physical coordinates of the reference tetrahedron.

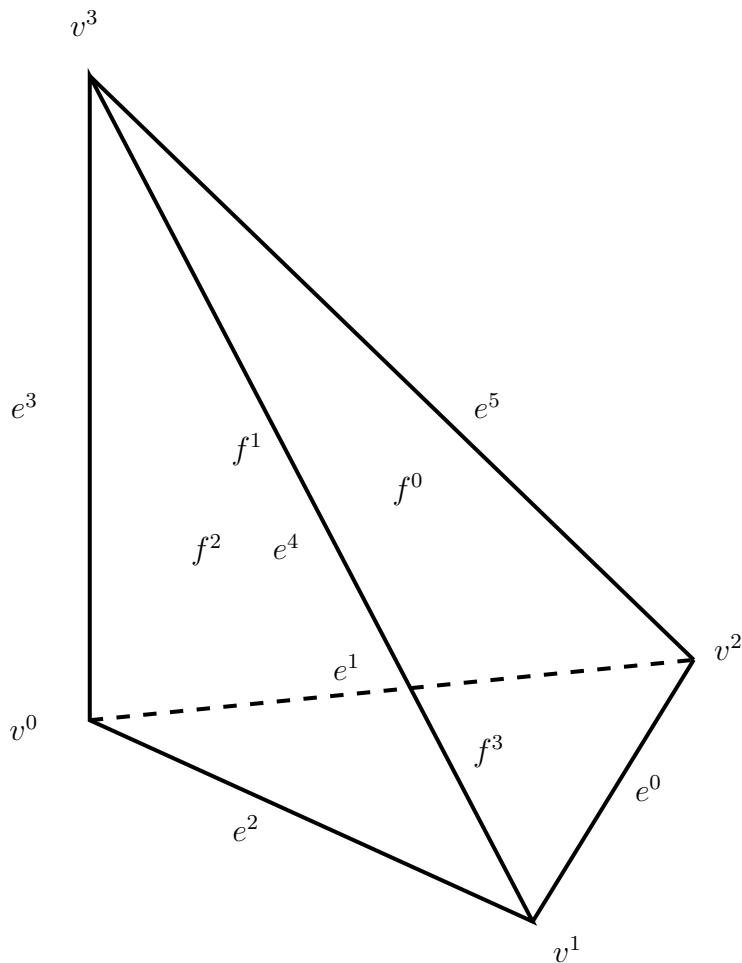


Figure A.4: Ordering of mesh entities (vertices, edges, faces) for the reference tetrahedron.

A.3.2 Ordering among mesh entities

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

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

A.3.3 Internal ordering on edges

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

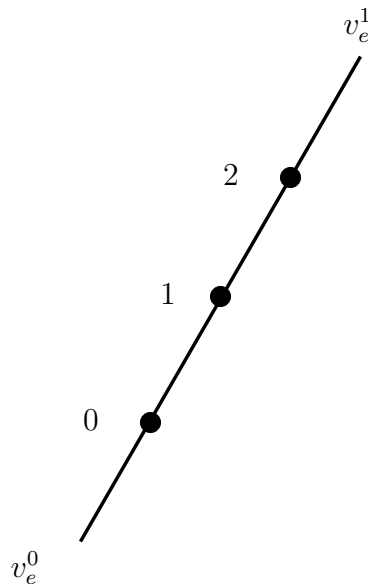


Figure A.5: Internal ordering of nodes on edges.

A.3.4 Alignment of edges

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

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

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

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

A.3.5 Internal ordering on faces

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

A.3.6 Alignment of faces

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

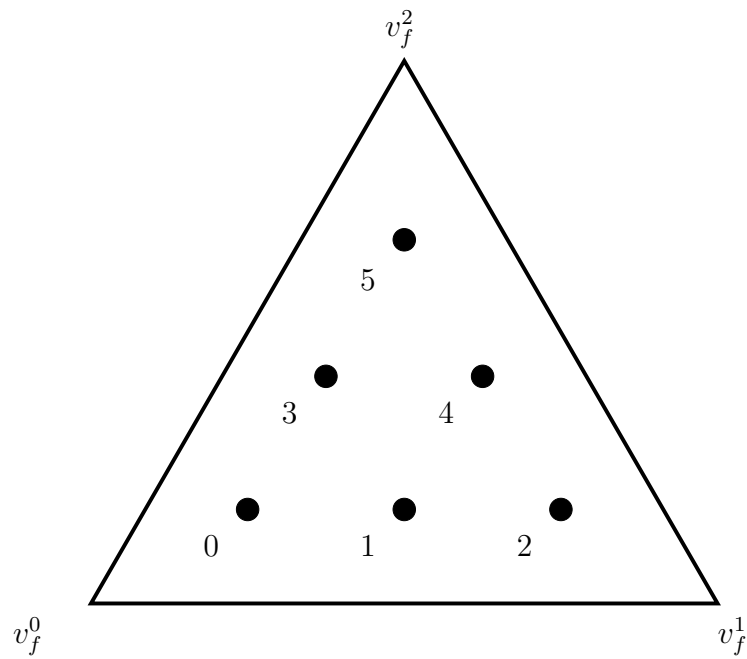


Figure A.6: Internal ordering of nodes on faces.

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

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

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

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

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

Appendix B

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). 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 effort is underway to provide precompiled Debian packages of **DOLFIN** and other **FENICS** components.

B.1 Installing from source

B.1.1 Dependencies and requirements

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

Installing Libxml2

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

<http://xmlsoft.org/>

For Debian users, the package to install is `libxml2-dev`.

Installing PETSc

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 2.3.0, 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).

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

Installing FFC

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

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

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

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

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

B.1.2 Downloading the source code

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

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

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

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

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

If you want the very latest version of **DOLFIN**, there is also a version named `dolfin-cvs-current.tar.gz` which provides a snapshot of the current CVS

version of **DOLFIN**, updated automatically from the CVS repository each hour. This version may contain features not yet present in the latest release, but may also be less stable and even not work at all.

B.1.3 Compiling the source code

DOLFIN is built using the standard GNU Autotools (Automake, Autoconf), 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 `--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/`.

B.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
```

B.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 `dolphin-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

```
dolphin-config --cflags
```

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

```
dolphin-config --libs
```

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

B.2 Debian package

In preparation.

Appendix C

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.

C.1 Creating a patch

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

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

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

```
# make clean
```

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

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

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

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

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

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

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

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

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

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

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

C.2 Sending patches

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

dolfin-dev@fenics.org

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

C.3 Applying a patch (maintainers)

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

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

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

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

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

Unpack the patch file using `gunzip` if necessary.

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

```
# cd dolfin-0.1.0
```

4. Apply the patch:

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

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

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

C.4 License agreement

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

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

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

Appendix D

License

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

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, 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 or use pieces of it in new free programs; and that you know you can do these things.

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

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

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

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

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

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

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

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

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

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

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

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

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

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

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

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

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

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

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

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

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, 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

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "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 PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. 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 PROGRAM 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 PROGRAM (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 PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index

Progress, 47
cout, 43
dolfin_assert(), 45
dolfin_begin(), 46
dolfin_debug(), 45
dolfin_end(), 46
dolfin_error(), 44
dolfin_get(), 51
dolfin_info(), 43
dolfin_load(), 54
dolfin_log(), 48
dolfin_output(), 48
dolfin_parameter(), 53
dolfin_save(), 54
dolfin_set(), 52
dolfin_warning(), 44
endl, 43

assertions, 45
automation, 11

compiling, 76, 77
contact, 10
contributing, 79
convection–diffusion, 57
curses interface, 48

Debian package, 78
debugging, 45
demo programs, 77
dependencies, 73

diff, 79
downloading, 13, 75

enumeration, 10
errors, 44

FEniCS, 11
FFC, 75
ffc, 15
FIAT, 75
file formats, 41
finite element method, 11
Function, 27
functions, 27

GNU General Public License, 83
GPL, 83

I/O, 41
incompressible Navier–Stokes, 59
indices, 10
input/output, 41
installation, 13, 73

Libxml2, 74
license, 82, 83
log system, 43

Navier–Stokes, 59

output destination, 48

parameters, 51

partial differential equations, 33
patch, 79–81
PETSc, 74
Poisson’s equation, 14, 56
post-processing, 41
pre-processing, 41
progress bar, 47

quickstart, 13

reference tetrahedron, 65
reference triangle, 63

source code, 75

tasks, 46
typographic conventions, 9

warnings, 44

XML, 42, 54