# 10 DOLFIN: a C++/Python finite element library

By Anders Logg, Garth N. Wells and Johan Hake

DOLFIN is a C++/Python library that functions as the main user interface of FEniCS. In this chapter, we review the functionality of DOLFIN. We also discuss the implementation of some key features of DOLFIN in detail. For a general discussion on the design and implementation of DOLFIN, we refer to Logg and Wells (2010).

## 10.1 Overview

A large part of the functionality of FEniCS is implemented as part of DOLFIN. It provides a problem solving environment for models based on partial differential equations and implements core parts of the functionality of FEniCS, including data structures and algorithms for computational meshes and finite element assembly. To provide a simple and consistent user interface, DOLFIN wraps the functionality of other FEniCS components and external software, and handles the communication between these components.

Figure 10.1 presents an overview of the relationships between the components of FEniCS and external software. The software map presented in the figure shows a user application implemented on top of the DOLFIN user interface, either in C++ or in Python. User applications may also be developed using FEniCS Apps, a collection of solvers implemented on top of FEniCS/DOLFIN. DOLFIN itself functions as both a user interface and a core component of FEniCS. All communication between a user program, other core components of FEniCS and external software is routed through wrapper layers that are implemented as part of the DOLFIN user interface. In particular, variational forms expressed in the UFL form language (Chapter 17) are passed to the form compiler FFC (Chapter 11) or SFC (Chapter 15) to generate UFC code (Chapter 16), which can then be used by DOLFIN to assemble linear systems. In the case of FFC, this code generation depends on the finite element backend FIAT (Chapter 13), the just-in-time compilation utility Instant (Chapter 14) and the optional optimizing backend FErari (Chapter 12). Finally, the plotting capabilities provided by DOLFIN are implemented by Viper. Some of this communication is exposed to users of the DOLFIN C++ interface, which requires a user to explicitly generate UFC code from a UFL form file by calling a form compiler on the command-line.

DOLFIN also relies on external software for important functionality such as the linear algebra libraries PETSc, Trilinos, uBLAS and MTL4, and the mesh partitioning libraries ParMETIS and SCOTCH (Pellegrini).

A. Logg et al., *Automated Solution of Differential Equations by the Finite Element Method*, Lecture Notes in Computational Science and Engineering 84, DOI 10.1007/978-3-642-23099-8_10,

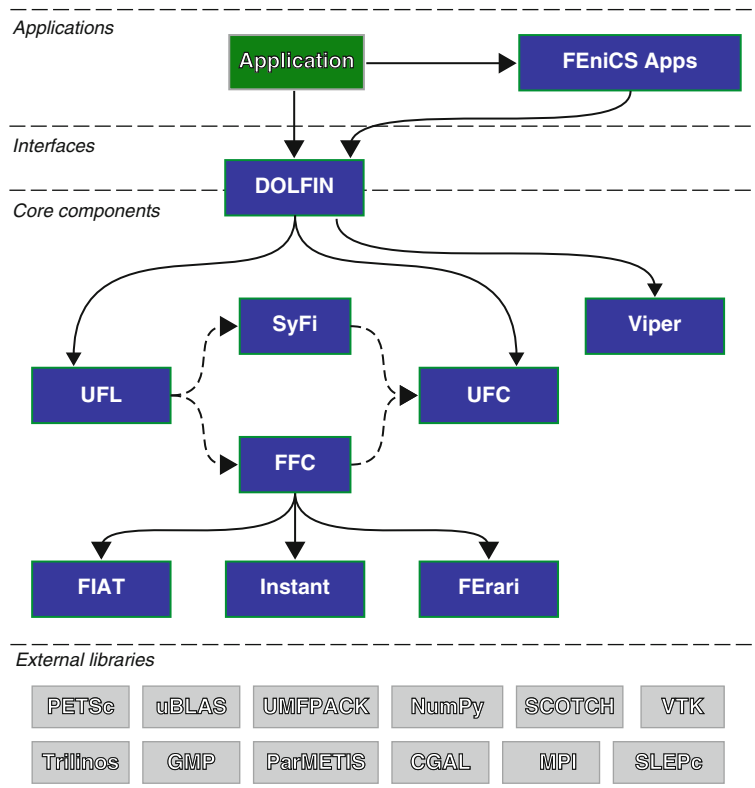


Figure 10.1: DOLFIN functions as the main user interface of FEniCS and handles the communication between the various components of FEniCS and external software. Solid lines indicate dependencies and dashed lines indicate data flow.

## 10.2 User interfaces

DOLFIN provides two user interfaces. One interface is implemented as a traditional C++ library, and another interface is implemented as a standard Python module. The two interfaces are near-identical, but in some cases particular language features of either C++ or Python require variations in the interfaces. In particular, the Python interface adds an additional level of automation by employing run-time (just-in-time) code generation. Below, we comment on the design and implementation of the two user interfaces of DOLFIN.

### 10.2.1 C++ interface

The DOLFIN C++ interface is designed as a standard object-oriented C++ library. It provides classes such as `Matrix`, `Vector`, `Mesh`, `FiniteElement`, `FunctionSpace` and `Function`, which model important concepts for finite element computing (see Figure 10.2). It also provides a small number of free functions (a function that is not a member function of a class), most notably `assemble` and `solve`, which can be used in conjunction with DOLFIN class objects to implement finite element solvers. The interface is designed to be as simple as possible, and without compromising on generality. When external software is wrapped, a simple and consistent user interface is provided to allow the rapid development of solvers without needing to deal with differences in the interfaces of external libraries. However, DOLFIN has been designed to interact flexibly with external software. In particular, in cases where DOLFIN provides wrappers for external libraries, such as the `Matrix` and `Vector` classes which wrap data structures from linear algebra libraries like PETSc and Trilinos, advanced users may, if necessary, access the underlying data structures in order to use native functionality from the wrapped external libraries.

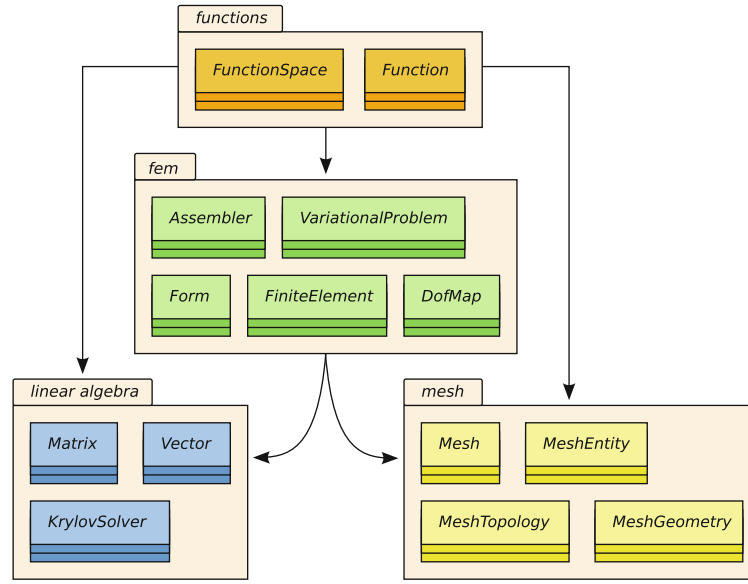


Figure 10.2: Schematic overview of some of the most important components and classes of DOLFIN. Arrows indicate dependencies.

To solve partial differential equations using the DOLFIN C++ interface, users must express finite element variational problems in the UFL form language. This is accomplished by entering the forms into separate `.ufl` files and compiling those files using a form compiler to generate UFC-compliant C++ code. The generated code may then be included in a DOLFIN C++ program. We return to this issue in Section 10.3.

To use DOLFIN from C++, users need to include one or more header files from the DOLFIN C++ library. In the simplest case, one includes the header file `dolfin.h`, which in turn includes all other DOLFIN header files:

*C++ code*

```
#include <dolfin.h>

using namespace dolfin;

int main()
{

  return 0;
}
```

### 10.2.2 Python interface

Over the last decade, Python has emerged as an attractive choice for the rapid development of simulation codes for scientific computing. Python brings the benefits of a high-level scripting language, the strength of an object-oriented language and a wealth of libraries for numerical computation.

The bulk of the DOLFIN Python interface is automatically generated from the C++ interface using SWIG (Beazley, 1996; SWIG). Since the functionality of both the C++ and Python interfaces are implemented as part of the DOLFIN C++ library, DOLFIN is equally efficient via the C++ and Python interfaces for most operations.

The DOLFIN Python interface offers some functionality that is not available from the C++ interface. In particular, the UFL form language is seamlessly integrated into the Python interface and code generation is automatically handled at run-time. To use DOLFIN from Python, users need to import

functionality from the DOLFIN Python module. In the simplest case, one includes all functionality from the Python module named `dolfin`:

*Python code*

```
from dolfin import *
```

## 10.3 Functionality

DOLFIN is organized as a collection of libraries (modules), with each covering a certain area of functionality. We review here these areas and explain the purpose and usage of the most commonly used classes and functions. The review is bottom-up; that is, we start by describing the core low-level functionality of DOLFIN (linear algebra and meshes) and then move upwards to describe higher level functionality. For further details, we refer to the DOLFIN Programmer's Reference on the FEniCS Project web page and to Logg and Wells (2010).

### 10.3.1 Linear algebra

DOLFIN provides a range of linear algebra objects and functionality, including vectors, dense and sparse matrices, direct and iterative linear solvers and eigenvalues solvers, and does so via a simple and consistent interface. For the bulk of underlying functionality, DOLFIN relies on third-party libraries such as PETSc and Trilinos. DOLFIN defines the abstract base classes `GenericTensor`, `GenericMatrix` and `GenericVector`, and these are used extensively throughout the library. Implementations of these generic interfaces for a number of backends are provided in DOLFIN, thereby achieving a common interface for different backends. Users can also wrap other linear algebra backends by implementing the generic interfaces.

*Matrices and vectors.* The simplest way to create matrices and vectors is via the classes `Matrix` and `Vector`. In general, `Matrix` and `Vector` represent distributed linear algebra objects that may be stored across (MPI) processes when running in parallel. Consistent with the most common usage in a finite element library, a `Vector` uses dense storage and a `Matrix` uses sparse storage. A `Vector` can be created as follows:

*C++ code*

```
Vector x;
```

*Python code*

```
x = Vector()
```

and a matrix can be created by:

*C++ code*

```
Matrix A;
```

*Python code*

```
A = Matrix()
```

In most applications, a user may need to create a matrix or a vector, but most operations on the linear algebra objects, including resizing, will take place inside the library and a user will not have to operate on the objects directly.

The following code illustrates how to create a vector of size 100:

*C++ code*

```
Vector x(100);
```

*Python code*

```
x = Vector(100)
```

A number of backends support distributed linear algebra for parallel computation, in which case the vector `x` will have global size 100, and DOLFIN will partition the vector across processes in (near) equal-sized portions.

Creating a `Matrix` of a given size is more involved as the matrix is sparse and in general needs to be initialized (data structures allocated) based on the structure of the sparse matrix (its sparsity pattern). Initialization of sparse matrices is handled by DOLFIN when required.

While DOLFIN supports distributed linear algebra objects for parallel computation, it is rare that a user is exposed to details at the level of parallel data layouts. The distribution of objects across processes is handled automatically by the library.

*Solving linear systems.* The simplest approach to solving the linear system $Ax = b$ is to use

*C++ code*

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

*Python code*

```
solve(A, x, b)
```

DOLFIN will use a default method to solve the system of equations. Optional arguments may be given to specify which algorithm to use when solving the linear system and, in the case of an iterative method, which preconditioner to use:

*C++ code*

```
solve(A, x, b, "lu");
solve(A, x, b, "gmres", "ilu");
```

*Python code*

```
solve(A, x, b, "lu");
solve(A, x, b, "gmres", "ilu")
```

Which methods and preconditioners that are available depends on which linear algebra backend DOLFIN has been configured with. To list the available solver methods and preconditioners, the following commands may be used:

*C++ code*

```
list_lu_solver_methods();
list_krylov_solver_methods();
list_krylov_solver_preconditioners();
```

*Python code*

```
list_lu_solver_methods()
list_krylov_solver_methods()
list_krylov_solver_preconditioners()
```

Using the function `solve` is straightforward, but it offers little control over details of the solution process. For many applications, it is desirable to exercise a degree of control over the solution process and reuse solver objects throughout a simulation.

The linear system $Ax = b$ can be solved using LU decomposition (a direct method) as follows:

*C++ code*

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

*Python code*

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

Alternatively, the operator $A$ associated with the linear solver can be set post-construction:

*C++ code*

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

*Python code*

```
solver = LUSolver()
solver.set_operator(A)
solver.solve(x, b)
```

This can be useful when passing a linear solver via a function interface and setting the operator inside a function.

In some cases, the system $Ax = b$ may be solved a number of times for a given matrix $A$ and different vectors $b$, or for different $A$ but with the same nonzero structure. If the nonzero structure of $A$ does not change, then some efficiency gains for repeated solves can be achieved by informing the LU solver of this fact:

*C++ code*

```
solver.parameters["same_nonzero_pattern"] = true;
```

*Python code*

```
solver.parameters["same_nonzero_pattern"] = True
```

In the case that $A$ does not change, the solution time for subsequent solves can be reduced dramatically by re-using the LU factorization of $A$. Re-use of the factorization is controlled by the parameter `"reuse_factorization"`.

It is possible for some backends to prescribe the specific LU solver to be used. This depends on the backend, which solvers that have been configured by DOLFIN and how third-party linear algebra backends have been configured.

The system of equations $Ax = b$ can be solved using a preconditioned Krylov solver by:

*C++ code*

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

*Python code*

```
solver = KrylovSolver(A)
solver.solve(x, b)
```

The above will use a default preconditioner and solver, and default parameters. If a `KrylovSolver` is constructed without a matrix operator $A$, the operator can be set post-construction:

*C++ code*

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

*Python code*

```
solver = KrylovSolver()
solver.set_operator(A)
solver.solve(x, b)
```

In some cases, it may be useful to use a preconditioner matrix $P$ that differs from $A$:

*C++ code*

```
KrylovSolver solver;
solver.set_operators(A, P);
solver.solve(x, b);
```

*Python code*

```
solver = KrylovSolver()
solver.set_operators(A, P)
solver.solve(x, b)
```

Various parameters for Krylov solvers can be set. Some common parameters are:

*Python code*

```
solver = KrylovSolver()
solver.parameters["relative_tolerance"]      = 1.0e-6
solver.parameters["absolute_tolerance"]      = 1.0e-15
solver.parameters["divergence_limit"]        = 1.0e4
solver.parameters["maximum_iterations"]      = 10000
solver.parameters["error_on_nonconvergence"] = True
solver.parameters["nonzero_initial_guess"]   = False
```

The parameters may be set similarly from C++. Printing a summary of the convergence of a `KrylovSolver` and printing details of the convergence history can be controlled via parameters:

*C++ code*

```
KrylovSolver solver;
solver.parameters["report"] = true;
solver.parameters["monitor_convergence"] = true;
```

*Python code*

```
solver = KrylovSolver()
solver.parameters["report"] = True
solver.parameters["monitor_convergence"] = True
```

The specific Krylov solver and preconditioner to be used can be set at construction of a solver object. The simplest approach is to set the Krylov method and the preconditioner via string descriptions. For example:

*C++ code*

```
KrylovSolver solver("gmres", "ilu");
```

*Python code*

```
solver = KrylovSolver("gmres", "ilu")
```

The above specifies the Generalized Minimum Residual (GMRES) method as a solver, and incomplete LU (ILU) preconditioning.

When backends such as PETSc and Trilinos are configured, a wide range of Krylov methods and preconditioners can be applied, and a large number of solver and preconditioner parameters can be set. In addition to what is described here, DOLFIN provides more advanced interfaces which permit finer control of the solution process. It is also possible for users to provide their own preconditioners.

*Solving eigenvalue problems.* DOLFIN uses the library SLEPc, which builds on PETSc, to solve eigenvalue problems. The SLEPc interface works only with PETSc-based linear algebra objects. Therefore, it is necessary to use PETSc-based objects, or to set the default linear algebra backend to PETSc and downcast objects (as explained in the next section). The following code illustrates the solution of the eigenvalue problem $Ax = \lambda x$:

*C++ code*

```
// Create matrix
PETScMatrix A;

// Code omitted for setting the entries of A

// Create eigensolver
SLEPcEigenSolver eigensolver(A);

// Compute all eigenvalues of A
eigensolver.solve();

// Get first eigenpair
double lambda_real, lambda_complex;
PETScVector x_real, x_complex;
eigensolver.get_eigenpair(lambda_real, lambda_complex, x_real, x_complex, 0);
```

*Python code*

```
# Create matrix
A = PETScMatrix()

# Code omitted for setting the entries of A

# Create eigensolver
eigensolver = SLEPcEigenSolver(A)

# Compute all eigenvalues of A
eigensolver.solve()

# Get first eigenpair
lambda_r, lambda_c, x_real, x_complex = eigensolver.get_eigenpair(0)
```

The real and complex components of the eigenvalue are returned in `lambda_real` and `lambda_complex`, respectively, and the real and complex components of the eigenvector are returned in `x_real` and `x_complex`, respectively.

To create a solver for the generalized eigenvalue problem $Ax = \lambda Mx$, the eigensolver can be constructed using $A$ and $M$:

*C++ code*

```
PETScMatrix A;
PETScMatrix M;

// Code omitted for setting the entries of A and M

SLEPcEigenSolver eigensolver(A, M);
```

*Python code*

```
A = PETScMatrix()
M = PETScMatrix()

# Code omitted for setting the entries of A and M

eigensolver = SLEPcEigenSolver(A, M)
```

There are many options that a user can set via the parameter system to control the eigenvalue problem solution process. To print a list of available parameters, call `info(eigensolver.parameters, true)` and `info(eigensolver.parameters, True)` from C++ and Python, respectively.

*Selecting a linear algebra backend.* The `Matrix`, `Vector`, `LUSolver` and `KrylovSolver` objects are all based on a specific linear algebra backend. The default backend depends on which backends are enabled when DOLFIN is configured. The backend can be set via the global parameter `"linear_algebra_backend"`. To use PETSc as the linear algebra backend:

*C++ code*

```
parameters["linear_algebra_backend"] = "PETSc";
```

*Python code*

```
parameters["linear_algebra_backend"] = "PETSc"
```

This parameter should be set before creating linear algebra objects. To use Epetra from the Trilinos collection, the parameter `"linear_algebra_backend"` should be set to `"Epetra"`. For uBLAS, the parameter should be set to `"uBLAS"` and for MTL4, the parameter should be set to `"MTL4"`.

Users can explicitly create linear algebra objects that use a particular backend. Generally, such objects are prefixed with the name of the backend. For example, a PETSc-based vector and LU solver are created by:

*C++ code*

```
PETScVector x;
PETScLUSolver solver;
```

*Python code*

```
x = PETScVector()
solver = PETScLUSolver()
```

*Solving nonlinear systems.* DOLFIN provides a Newton solver in the form of the class `NewtonSolver` for solving nonlinear systems of equations of the form

$$F(x) = 0, \tag{10.1}$$

where $x \in \mathbb{R}^n$ and $F : \mathbb{R}^n \to \mathbb{R}^n$. To solve such a problem using the DOLFIN Newton solver, a user needs to provide a subclass of `NonlinearProblem`. The purpose of a `NonlinearProblem` object is to evaluate $F$ and the Jacobian of $F$, which will be denoted by $J : \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n$. An outline of a user-provided `MyNonlinearProblem` class for solving a nonlinear differential equation is shown below.

*C++ code*

```
class MyNonlinearProblem : public NonlinearProblem
{
public:

  // Constructor
  MyNonlinearProblem(const Form& L, const Form& a,
                     const BoundaryCondition& bc) : L(L), a(a), bc(bc) {}

  // User-defined residual vector F
  void F(GenericVector& b, const GenericVector& x)
  {
    assemble(b, L);
    bc.apply(b, x);
  }

  // User-defined Jacobian matrix J
  void J(GenericMatrix& A, const GenericVector& x)
  {
    assemble(A, a);
    bc.apply(A);
  }

private:

  const Form& L;
  const Form& a;
  const BoundaryCondition& bc;

};
```

A `MyNonlinearProblem` object is constructed using a linear form `L`, that when assembled corresponds to $F$, and a bilinear form `a`, that when assembled corresponds to $J$. The classes `Form` and `BoundaryCondition` used in the example are discussed in more detail later. The same `MyNonlinearProblem` class can be defined in Python:

*Python code*

```
class MyNonlinearProblem(NonlinearProblem):
    def __init__(self, L, a, bc):
        NonlinearProblem.__init__(self)
        self.L = L
        self.a = a
        self.bc = bc
    def F(self, b, x):
        assemble(self.L, tensor=b)
        self.bc.apply(b, x)
    def J(self, A, x):
        assemble(self.a, tensor=A)
        self.bc.apply(A)
```

Once a nonlinear problem class has been defined, a `NewtonSolver` object can be created and the Newton solver can be used to compute the solution vector $x$ to the nonlinear problem:

*C++ code*

```
MyNonlinearProblem problem(L, a, bc);
NewtonSolver newton_solver;

newton_solver.solve(problem, u.vector());
```

*Python code*

```
problem = MyNonlinearProblem(L, a, bc)
newton_solver = NewtonSolver()

newton_solver.solve(problem, u.vector())
```

A number of parameters can be set for a `NewtonSolver`. Some parameters that determine the behavior of the Newton solver are:

*Python code*

```
newton_solver = NewtonSolver()
newton_solver.parameters["maximum_iterations"] = 20
newton_solver.parameters["relative_tolerance"] = 1.0e-6
newton_solver.parameters["absolute_tolerance"] = 1.0e-10
newton_solver.parameters["error_on_nonconvergence"] = False
```

The parameters may be set similarly from C++. When testing for convergence, usually a norm of the residual $F$ is checked. Sometimes it is useful instead to check a norm of the iterative correction $dx$. This is controlled by the parameter `"convergence_criterion"`, which can be set to `"residual"`, for checking the size of the residual $F$, or `"incremental"`, for checking the size of the increment $dx$.

For more advanced usage, a `NewtonSolver` can be constructed with arguments that specify the linear solver and preconditioner to be used in the solution process.

### *10.3.2 Meshes*

A central part of DOLFIN is its mesh library and the `Mesh` class. The mesh library provides data structures and algorithms for computational meshes, including the computation of mesh connectivity (incidence relations), mesh refinement, mesh partitioning and mesh intersection.

The mesh library is implemented in C++ and has been optimized to minimize storage requirements and to enable efficient access to mesh data. In particular, a DOLFIN mesh is stored in a small number of contiguous arrays, on top of which a light-weight object-oriented layer provides a *view* to the underlying data. For a detailed discussion on the design and implementation of the mesh library, we refer to Logg (2009).

*Creating a mesh.* DOLFIN provides functionality for creating simple meshes, such as meshes of unit squares and unit cubes, spheres, rectangles and boxes. The following code demonstrates how to create a $16 \times 16$ 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 $6 \times 16 \times 16 \times 16 = 24,576$ tetrahedra).

*C++ code*

```
UnitSquare unit_square(16, 16);
UnitCube unit_cube(16, 16, 16);
```

*Python code*

```
unit_square = UnitSquare(16, 16)
unit_cube = UnitCube(16, 16, 16)
```

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

*C++ code*

```
Mesh mesh;
MeshEditor editor;
editor.open(mesh, 2, 2);
editor.init_vertices(4);
editor.init_cells(2);
editor.add_vertex(0, 0.0, 0.0);
editor.add_vertex(1, 1.0, 0.0);
editor.add_vertex(2, 1.0, 1.0);
editor.add_vertex(3, 0.0, 1.0);
editor.add_cell(0, 0, 1, 2);
editor.add_cell(1, 0, 2, 3);
editor.close();
```

*Python code*

```
mesh = Mesh();
editor = MeshEditor();
editor.open(mesh, 2, 2)
editor.init_vertices(4)
editor.init_cells(2)
editor.add_vertex(0, 0.0, 0.0)
editor.add_vertex(1, 1.0, 0.0)
editor.add_vertex(2, 1.0, 1.0)
editor.add_vertex(3, 0.0, 1.0)
editor.add_cell(0, 0, 1, 2)
editor.add_cell(1, 0, 2, 3)
editor.close()
```

*Reading a mesh from file.* Although the built-in classes `UnitSquare` and `UnitCube` are useful for testing, a typical application will need to read from file a mesh that has been generated by an external mesh generator. To read a mesh from file, simply supply the filename to the constructor of the `Mesh` class:

*C++ code*

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

*Python code*

```
mesh = Mesh("mesh.xml")
```

Meshes must be stored in the DOLFIN XML format. The following example illustrates the XML format for a $2 \times 2$ mesh of the unit square:

*XML code*

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

<dolfin xmlns:dolfin="http://fenicsproject.org">
  <mesh celltype="triangle" dim="2">
    <vertices size="9">
      <vertex index="0" x="0" y="0"/>
      <vertex index="1" x="0.5" y="0"/>
      <vertex index="2" x="1" y="0"/>
      <vertex index="3" x="0" y="0.5"/>
      <vertex index="4" x="0.5" y="0.5"/>
      <vertex index="5" x="1" y="0.5"/>
      <vertex index="6" x="0" y="1"/>
      <vertex index="7" x="0.5" y="1"/>
      <vertex index="8" x="1" y="1"/>
    </vertices>
    <cells size="8">
      <triangle index="0" v0="0" v1="1" v2="4"/>
      <triangle index="1" v0="0" v1="3" v2="4"/>
      <triangle index="2" v0="1" v1="2" v2="5"/>
      <triangle index="3" v0="1" v1="4" v2="5"/>
      <triangle index="4" v0="3" v1="4" v2="7"/>
      <triangle index="5" v0="3" v1="6" v2="7"/>
      <triangle index="6" v0="4" v1="5" v2="8"/>
      <triangle index="7" v0="4" v1="7" v2="8"/>
    </cells>
  </mesh>
</dolfin>
```

Meshes stored in other data formats may be converted to the DOLFIN XML format using the command `dolfin-convert`, as explained in more detail below.

*Mesh entities.* Conceptually, a *mesh* (modeled by the class `Mesh`), consists of a collection of *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 as *edges*, entities of dimension 2 as *faces*. Entities of *codimension* 1 are referred to as *facets* and entities of codimension 0 as *cells*. These concepts are summarized in Figure 10.3 and Table 10.1. We note that a triangular mesh consists of vertices, edges and cells, and that the edges may alternatively be referred to as facets and the cells as faces. We further note that a tetrahedral mesh consists of vertices, edges, faces and cells, and that the faces may alternatively be referred to as facets. These concepts are implemented by the classes `MeshEntity`, `Vertex`, `Edge`, `Face`, `Facet` and `Cell`. These classes do not store any data. Instead, they are light-weight objects that provide views of the underlying mesh data. A `MeshEntity` may be created from a `Mesh`, a topological dimension and an index. The following code demonstrates how to create various entities on a mesh:

*C++ code*

```
MeshEntity entity(mesh, 0, 33); // vertex number 33
Vertex vertex(mesh, 33);        // vertex number 33
Cell cell(mesh, 25);            // cell number 25
```

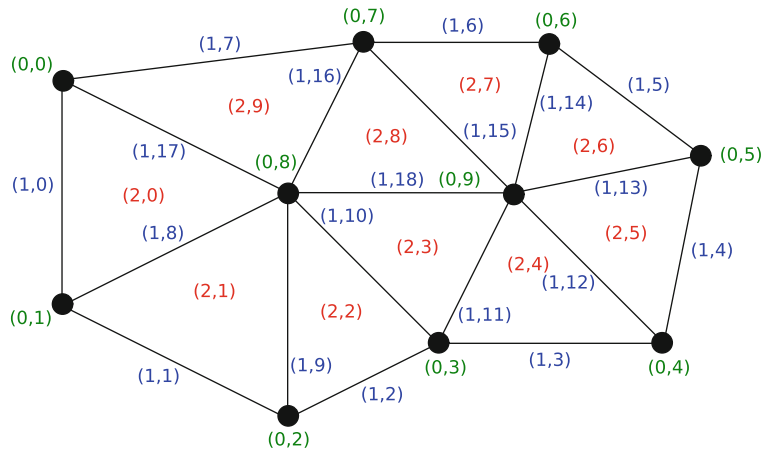


Figure 10.3: Each entity of a mesh is identified by a pair $(d, i)$ which specifies the topological dimension $d$ and a unique index $i$ for the entity within the set of entities of dimension $d$.

| Entity | Dimension | Codimension |
|---|---|---|
| Vertex | 0 | $D$ |
| Edge | 1 | $D-1$ |
| Face | 2 | $D-2$ |
| | | |
| Facet | $D-1$ | 1 |
| Cell | $D$ | 0 |

Table 10.1: Mesh entities and their dimensions/codimensions. The codimension of an entity is $D-d$ where $D$ is the maximal dimension and $d$ is the dimension.

*Python code*

```
entity = MeshEntity(mesh, 0, 33) # vertex number 33
vertex = Vertex(mesh, 33)        # vertex number 33
cell = Cell(mesh, 25)            # cell number 25
```

*Mesh topology and geometry.* The topology of a mesh is stored separately from its geometry. The topology of a mesh is a description of the relations between the various entities of the mesh, while the geometry describes how those entities are embedded in $\mathbb{R}^d$.

Users are rarely confronted with the `MeshTopology` and `MeshGeometry` classes directly since most algorithms on meshes can be expressed in terms of *mesh iterators*. However, users may sometimes need to access the dimension of a `Mesh`, which involves accessing either the `MeshTopology` or `MeshGeometry`, which are stored as part of the `Mesh`, as illustrated in the following code examples:

*C++ code*

```
uint gdim = mesh.topology().dim();
uint tdim = mesh.geometry().dim();
```

*Python code*

```
gdim = mesh.topology().dim()
tdim = mesh.geometry().dim()
```

It should be noted that the topological and geometric dimensions may differ. This is the case in particular for the boundary of a mesh, which is typically a mesh of topological dimension $D$ embedded in $\mathbb{R}^{D+1}$. That is, the geometry dimension is $D+1$.

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | – | × | – | × |
| 1 | × | × | – | – |
| 2 | – | – | – | – |
| 3 | × | × | – | × |

Table 10.2: DOLFIN computes the connectivity $d \to d'$ of a mesh for any pair $d, d' = 0, 1, \ldots, D$. The table indicates which connectivity pairs (indicated by $\times$) have been computed in order to compute the connectivity $1 \to 1$ (edge–edge connectivity) for a tetrahedral mesh.

*Mesh connectivity.* The topology of a `Mesh` is represented by the *connectivity* (incidence relations) of the mesh, which is a complete description of which entities of the mesh are connected to which entities. Such connectivity is stored in DOLFIN by the `MeshConnectivity` class. One such data set is stored as part of the class `MeshTopology` for each pair of topological dimensions $d \to d'$ for $d, d' = 0, 1, \ldots, D$, where $D$ is the topological dimension.

When a `Mesh` is created, a minimal `MeshTopology` is created. Only the connectivity from cells (dimension $D$) to vertices (dimension 0) is stored (`MeshConnectivity` $D \to 0$). When a certain connectivity is requested, such as for example the connectivity $1 \to 1$ (connectivity from edges to edges), DOLFIN automatically computes any other connectivities required for computing the requested connectivity. This is illustrated in Table 10.2, where we indicate which connectivities are required to compute the $1 \to 1$ connectivity. The following code demonstrates how to initialize various kinds of mesh connectivity for a tetrahedral mesh ($D = 3$):

*C++ code*

```
mesh.init(2);    // Compute faces
mesh.init(0, 0); // Compute vertex neighbors for each vertex
mesh.init(1, 1); // Compute edge neighbors for each edge
```

*Python code*

```
mesh.init(2)      # Compute faces
mesh.init(0, 0)   # Compute vertex neighbors for each vertex
mesh.init(1, 1)   # Compute edge neighbors for each edge
```

*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. Two vertices are considered as neighbors if they both belong to the same cell. For simplex meshes, this is equivalent to an edge connecting the two vertices.

*C++ code*

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

*Python code*

```
for c in cells(mesh):
    for v0 in vertices(c):
        for v1 in vertices(v0):
            print v1
```

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

*C++ code*

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

*Python code*

```
D = mesh.topology().dim()
for c in entities(mesh, D):
    for v0 in entities(c, 0):
        for v1 in entities(v0, 0):
            print v1
```

*Mesh functions.* A useful class for storing data associated with a `Mesh` is the `MeshFunction` class. This makes it simple to store, for example, material parameters, subdomain indicators, refinement markers on the `Cells` of a `Mesh` or boundary markers on the `Facets` of a `Mesh`. A `MeshFunction` is a discrete function that takes a value on each mesh entity of a given topological dimension $d$. The number of values stored in a `MeshFunction` is equal to the number of entities $n_d$ of dimension $d$. A `MeshFunction` is templated over the value type and may thus be used to store values of any type. For convenience, named `MeshFunctions` are provided by the classes `VertexFunction`, `EdgeFunction`, `FaceFunction`, `FacetFunction` and `CellFunction`. The following code illustrates how to create a pair of `MeshFunctions`, one for storing subdomain indicators on `Cells` and one for storing boundary markers on `Facets`:

*C++ code*

```
CellFunction<uint> sub_domains(mesh);
sub_domains.set_all(0);
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  Point p = cell.midpoint();
  if (p.x() > 0.5)
    sub_domains[cell] = 1;
}

FacetFunction<uint> boundary_markers(mesh);
boundary_markers.set_all(0);
for (FacetIterator facet(mesh); !facet.end(); ++facet)
{
  Point p = facet.midpoint();
  if (near(p.y(), 0.0) || near(p.y(), 1.0))
    boundary_markers[facet] = 1;
}
```

*Python code*

```
sub_domains = CellFunction("uint", mesh)
sub_domains.set_all(0)
for cell in cells(mesh):
    p = cell.midpoint()
    if p.x() > 0.5:
        sub_domains[cell] = 1

boundary_markers = FacetFunction("uint", mesh)
boundary_markers.set_all(0)
for facet in facets(mesh):
    p = facet.midpoint()
    if near(p.y(), 0.0) or near(p.y(), 1.0):
        boundary_markers[facet] = 1
```

*Mesh data.* The `MeshData` class provides a simple way to associate data with a `Mesh`. It allows arbitrary `MeshFunctions` (and other quantities) to be associated with a `Mesh`. The following code illustrates how to attach and retrieve a `MeshFunction` named `"sub_domains"` to/from a `Mesh`:

*C++ code*

```
MeshFunction<uint>* sub_domains = mesh.data().create_mesh_function("sub_domains");
sub_domains = mesh.data().mesh_function("sub_domains");
```

*Python code*

```
sub_domains = mesh.data().create_mesh_function("sub_domains")
sub_domains = mesh.data().mesh_function("sub_domains")
```

To list data associated with a given `Mesh`, issue the command `info(mesh.data(), true)` in C++ or `info(mesh.data(), True)` in Python.

*Mesh refinement.* A `Mesh` may be refined, by either uniform or local refinement, by calling the `refine` function, as illustrated in the code examples below.

*C++ code*

```
// Uniform refinement
mesh = refine(mesh);

// Local refinement
CellFunction<bool> cell_markers(mesh);
cell_markers.set_all(false);
Point origin(0.0, 0.0, 0.0);
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  Point p = cell.midpoint();
  if (p.distance(origin) < 0.1)
    cell_markers[cell] = true;
}
mesh = refine(mesh, cell_markers);
```

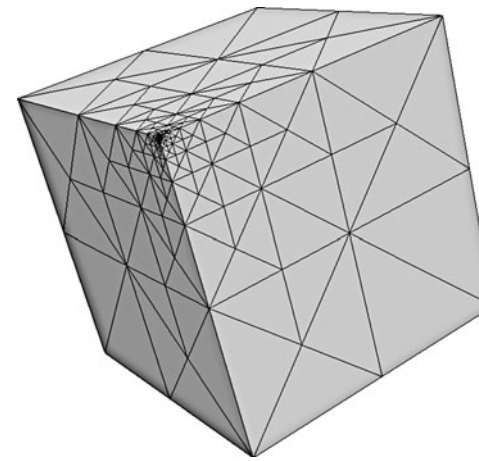

Figure 10.4: A locally refined mesh obtained by repeated marking of the cells close to one of the corners of the unit cube.

*Python code*

```
# Uniform refinement
mesh = refine(mesh)

# Local refinement
cell_markers = CellFunction("bool", mesh)
cell_markers.set_all(False)
origin = Point(0.0, 0.0, 0.0)
for cell in cells(mesh):
    p = cell.midpoint()
    if p.distance(origin) < 0.1:
        cell_markers[cell] = True
mesh = refine(mesh, cell_markers)
```

Currently, local refinement defaults to recursive refinement by edge bisection (Rivara, 1984, 1992). An example of a locally refined mesh obtained by a repeated marking of the cells close to one of the corners of the unit cube is shown in Figure 10.4.

*Parallel meshes.* When running a program in parallel on a distributed memory architecture (using MPI by invoking the program with the `mpirun` wrapper), DOLFIN automatically partitions and distributes meshes. Each process then stores a portion of the global mesh as a standard `Mesh` object. In addition, it stores auxiliary data needed for correctly computing local-to-global maps on each process and for communicating data to neighboring regions. Parallel computing with DOLFIN is discussed in Section 10.4.

### *10.3.3 Finite elements*

The concept of a finite element as discussed in Chapters 2 and 3 (the Ciarlet definition) is implemented by the DOLFIN `FiniteElement` class. This class is implemented differently in the C++ and Python interfaces.

The C++ implementation of the `FiniteElement` class relies on code generated by a form compiler such as FFC or SFC, which are discussed in Chapters 11 and 15, respectively. The class `FiniteElement` is essentially a wrapper class for the UFC class `ufc::finite_element`. A C++ `FiniteElement` provides all the functionality of a `ufc::finite_element`. Users of the DOLFIN C++ interface will typically not use the `FiniteElement` class directly, but it is an important building block for the `FunctionSpace` class, which is discussed below. However, users developing advanced algorithms

that require run-time evaluation of finite element basis function will need to familiarize themselves with the `FiniteElement` interface. For details, we refer to the DOLFIN Programmer's Reference.

The Python interface also provides a `FiniteElement` class. The Python `FiniteElement` class is imported directly from the UFL Python module (see Chapter 17). As such, it is just a label for a particular finite element that can be used to define variational problems. Variational problems are more conveniently defined in terms of the DOLFIN `FunctionSpace` class, so users of the Python interface are rarely confronted with the `FiniteElement` class. However, advanced users who wish to develop algorithms in Python that require functionality defined in the UFC interface, such as run-time evaluation of basis functions, can access such functionality by explicitly generating code from within the Python interface. This can be accomplished by a call to the DOLFIN `jit` function (just-in-time compilation), which takes as input a UFL `FiniteElement` and returns a pair containing a `ufc::finite_element` and a `ufc::dofmap`. The returned objects are created by first generating the corresponding C++ code, then compiling and wrapping that C++ code into a Python module. The returned objects are therefore directly usable from within Python.

The degrees of freedom of a `FiniteElement` can be plotted directly from the Python interface by a call to `plot(element)`. This will draw a picture of the shape of the finite element, along with a graphical representation of its degrees of freedom in accordance with the notation described in Chapter 3.

Table 10.3 lists the finite elements currently supported by DOLFIN (and the toolchain FIAT–UFL–FFC/SFC–UFC). A `FiniteElement` may be specified (from Python) using either its full name or its short symbol, as illustrated in the code example below:

*UFL code*

```
element = FiniteElement("Lagrange", tetrahedron, 5)
element = FiniteElement("CG", tetrahedron, 5)

element = FiniteElement("Brezzi-Douglas-Marini", triangle, 3)
element = FiniteElement("BDM", triangle, 3)

element = FiniteElement("Nedelec 1st kind H(curl)", tetrahedron, 2)
element = FiniteElement("N1curl", tetrahedron, 2)
```

| Name | Symbol |
|---|---|
| *Argyris* | *ARG* |
| *Arnold–Winther* | *AW* |
| Brezzi–Douglas–Marini | BDM |
| Crouzeix–Raviart | CR |
| Discontinuous Lagrange | DG |
| *Hermite* | *HER* |
| Lagrange | CG |
| *Mardal–Tai–Winther* | *MTW* |
| *Morley* | *MOR* |
| Nédélec 1st kind $H(\mathrm{curl})$ | N1curl |
| Nédélec 2nd kind $H(\mathrm{curl})$ | N2curl |
| Raviart–Thomas | RT |

Table 10.3: List of finite elements supported by DOLFIN 1.0. Elements in grey italics are partly supported in FEniCS but not throughout the entire toolchain.

### 10.3.4 Function spaces

The DOLFIN `FunctionSpace` class represents a finite element function space $V_h$, as defined in Chapter 2. The data of a `FunctionSpace` is represented in terms of a triplet consisting of a `Mesh`, a `DofMap` and a `FiniteElement`:

$$\texttt{FunctionSpace} = (\texttt{Mesh}, \texttt{DofMap}, \texttt{FiniteElement}).$$

The `Mesh` defines the computational domain and its discretization. The `DofMap` defines how the degrees of freedom of the function space are distributed. In particular, the `DofMap` provides the function `tabulate_dofs` which maps the local degrees of freedom on any given cell of the `Mesh` to global degrees of freedom. The `DofMap` plays a role in defining the global regularity of the finite element function space. The `FiniteElement` defines the local function space on any given cell of the `Mesh`. Note that if two or more `FunctionSpaces` are created on the same `Mesh`, that `Mesh` is shared between the two `FunctionSpaces`.

*Creating function spaces.* As for the `FiniteElement` class, `FunctionSpaces` are handled differently in the C++ and Python interfaces. In C++, the instantiation of a `FunctionSpace` relies on generated code. As an example, we consider here the creation of a `FunctionSpace` representing continuous piecewise linear Lagrange polynomials on triangles. First, the corresponding finite element must be defined in the UFL form language. We do this by entering the following code into a file named `Lagrange.ufl`:

*UFL code*

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

We may then generate C++ code using a form compiler such as FFC:

*Bash code*

```
ffc -l dolfin Lagrange.ufl
```

This generates a file named `Lagrange.h` that we may include in our C++ program to instantiate a `FunctionSpace` on a given `Mesh`:

*C++ code*

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

using namespace dolfin;

int main()
{
  UnitSquare mesh(8, 8);
  Lagrange::FunctionSpace V(mesh);

  ...
  return 0;
}
```

In typical applications, a `FunctionSpace` is not generated through a separate `.ufl` file, but is instead generated as part of the code generation for a variational problem.

From the Python interface, one may create a `FunctionSpace` directly, as illustrated by the following code which creates the same function space as the above example (piecewise linear Lagrange polynomials on triangles):

*Python code*

```
mesh = UnitSquare(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)
```

*Mixed spaces.* Mixed function spaces may be created from arbitrary combinations of function spaces. As an example, we consider here the creation of the *Taylor–Hood* function space for the discretization of the Stokes or incompressible Navier–Stokes equations. This mixed function space is the tensor product of a vector-valued continuous piecewise quadratic function space for the velocity field and a scalar continuous piecewise linear function space for the pressure field. This may be easily defined in either a UFL form file (for code generation and subsequent inclusion in a C++ program) or directly in a Python script as illustrated in the following code examples:

*UFL code*

```
V = VectorElement("Lagrange", triangle, 2)
Q = FiniteElement("Lagrange", triangle, 1)
W = V*Q
```

*Python code*

```
V = VectorFunctionSpace(mesh, "Lagrange", 2)
Q = FunctionSpace(mesh, "Lagrange", 1)
W = V*Q
```

DOLFIN allows the generation of arbitrarily nested mixed function spaces. A mixed function space can be used as a building block in the construction of a larger mixed space. When a mixed function space is created from more than two function spaces (nested on the same level), then one must use the `MixedElement` constructor (in UFL/C++) or the `MixedFunctionSpace` constructor (in Python). This is because Python will interpret the expression `V*Q*P` as `(V*Q)*P`, which will create a mixed function space consisting of two subspaces: the mixed space `V*Q` and the space `P`. If that is not the intention, one must instead define the mixed function space using `MixedElement([V, Q, P])` in UFL/C++ or `MixedFunctionSpace([V, Q, P])` in Python.

*Subspaces.* For a mixed function space, one may access its subspaces. These subspaces differ, in general, from the function spaces that were used to create the mixed space in their degree of freedom maps (`DofMap` objects). Subspaces are particularly useful for applying boundary conditions to components of a mixed element. We return to this issue below.

### 10.3.5 Functions

The `Function` class represents a finite element function $u_h$ in a finite element space $V_h$ as defined in Chapter 2:

$$u_h(x) = \sum_{j=1}^{N} U_j \phi_j(x), \tag{10.2}$$

where $U \in \mathbb{R}^N$ is the vector of degrees of freedom for the function $u_h$ and $\{\phi_j\}_{j=1}^N$ is a basis for $V_h$. A `Function` is represented in terms of a `FunctionSpace` and a `GenericVector`:

$$\texttt{Function} = (\texttt{FunctionSpace},\ \texttt{GenericVector}).$$

The `FunctionSpace` defines the function space $V_h$ and the `GenericVector` holds the vector $U$ of degrees of freedom; see Figure 10.5. When running in parallel on a distributed memory architecture, the `FunctionSpace` and the `GenericVector` are distributed across the processes.

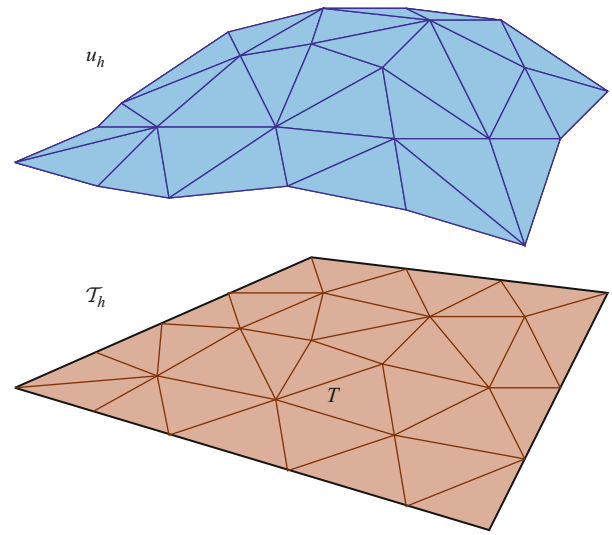


Figure 10.5: A piecewise linear finite element function $u_h$ on a mesh consisting of triangular elements. The vector of degrees of freedom $U$ is given by the values of $u_h$ at the mesh vertices.

*Creating functions.* `function!creation`

To create a `Function` on a `FunctionSpace`, one simply calls the constructor of the `Function` class with the `FunctionSpace` as the argument, as illustrated in the following code examples:

*C++ code*

```
Function u(V);
```

*Python code*

```
u = Function(V)
```

If two or more `Functions` are created on the same `FunctionSpace`, the `FunctionSpace` is shared between the `Functions`.

A `Function` is typically used to hold the computed solution to a partial differential equation. One may then obtain the degrees of freedom $U$ by solving a system of equations, as illustrated in the following code examples:

*C++ code*

```
Function u(V);
solve(A, u.vector(), b);
```

*Python code*

```
u = Function(V)
solve(A, u.vector(), b)
```

The process of assembling and solving a linear system is handled automatically by the classes `Linear/NonlinearVariationalSolver`, which will be discussed in more detail below.

*Function evaluation.* A `Function` may be evaluated at arbitrary points inside the computational domain[1]. The value of a `Function` is computed by first locating the cell of the mesh containing the

[1]One may also evaluate a `Function` outside of the computational domain by setting the global parameter value `"allow_extrapolation"` to true. This may sometimes be necessary when evaluating a `Function` on the boundary of a domain since round-off errors may result in points slightly outside of the domain.

given point, and then evaluating the linear combination of basis functions on that cell. Finding the cell exploits an efficient search tree algorithm that is implemented as part of CGAL.

The following code examples illustrate function evaluation in the C++ and Python interfaces for scalar- and vector-valued functions:

*C++ code*

```
# Evaluation of scalar function
double scalar = u(0.1, 0.2, 0.3);

# Evaluation of vector-valued function
Array<double> vector(3);
u(vector, 0.1, 0.2, 0.3);
```

*Python code*

```
# Evaluation of scalar function
scalar = u(0.1, 0.2, 0.3)

# Evaluation of vector-valued function
vector = u(0.1, 0.2, 0.3)
```

When running in parallel with a distributed mesh, functions can only be evaluated at points located in the portion of the mesh that is stored by the local process.

*Subfunctions.* For `Functions` constructed on a mixed `FunctionSpace`, subfunctions (components) of the `Function` can be accessed, for example to plot the solution components of a mixed system. Subfunctions may be accessed as either *shallow* or *deep copies*. By default, subfunctions are accessed as shallow copies, which means that the subfunctions share data with their parent functions. They provide *views* to the data of the parent function. Sometimes, it may also be desirable to access subfunctions as deep copies. A deep copied subfunction does not share its data (namely, the vector holding the degrees of freedom) with the parent `Function`. Both shallow and deep copies of `Function` objects are themselves `Function` objects and may (with some exceptions) be used as regular `Function` objects.

Creating shallow and deep copies of subfunctions is done differently in C++ and Python, as illustrated by the following code examples:

*C++ code*

```
Function w(W);

// Create shallow copies
Function& u = w[0];
Function& p = w[1];

// Create deep copies
Function uu = w[0];
Function pp = w[1];
```

*Python code*

```
w = Function(W)

# Create shallow copies
u, p = w.split()

# Create deep copies
uu, pp = w.split(deepcopy=True)
```

Note that component access, such as `w[0]`, from the Python interface does not create a new `Function` object as in the C++ interface. Instead, it creates a UFL expression that denotes a component of the original `Function`.

### 10.3.6 Expressions

The `Expression` class is closely related to the `Function` class in that it represents a function that can be evaluated on a finite element space. However, where a `Function` must be defined in terms of a vector of degrees of freedom, an `Expression` may be freely defined in terms of, for example, coordinate values, other geometric entities, or a table lookup.

An `Expression` may be defined in both C++ and Python by subclassing the `Expression` class and overloading the `eval` function, as illustrated in the following code examples which define the function $f(x, y) = \sin x \cos y$ as an `Expression`:

*C++ code*

```
class MyExpression : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0])*cos(x[1]);
  }
};

MyExpression f;
```

*Python code*

```
class MyExpression(Expression):
    def eval(self, values, x):
        values[0] = sin(x[0])*cos(x[1])

f = MyExpression()
```

For vector-valued (or tensor-valued) `Expressions`, one must also specify the value shape of the `Expression`. The following code examples demonstrate how to implement the vector-valued function $g(x, y) = (\sin x, \cos y)$. The value shape is defined slightly differently in C++ and Python.

*C++ code*

```
class MyExpression : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0]);
    values[1] = cos(x[1]);
  }

  uint value_rank() const
  {
    return 1;
  }

  uint value_dimension(uint i) const
  {
    return 2;
  }
};

MyExpression g;
```

*Python code*

```
class MyExpression(Expression):

    def eval(self, values, x):
        values[0] = sin(x[0])
        values[1] = cos(x[1])

    def value_shape(self):
        return (2,)

g = MyExpression()
```

The above *functor* construct for the definition of expressions is powerful and allows a user to define complex expressions, the evaluation of which may involve arbitrary operations as part of the `eval` function. For simple expressions like $f(x, y) = \sin x \cos y$ and $g(x, y) = (\sin x, \cos y)$, users of the Python interface may, alternatively, use a simpler syntax:

*Python code*

```
f = Expression("sin(x[0])*cos(x[1])")
g = Expression(("sin(x[0])", "cos(x[1])"))
```

The above code will automatically generate subclasses of the DOLFIN C++ `Expression` class that overload the `eval` function. This has the advantage of being more efficient, since the callback to the `eval` function takes place in C++ rather than in Python.

A feature that can be used to implement a time-dependent `Expression` in the Python interface is to use a variable name in an `Expression` string. For example, one may use the variable `t` to denote time:

*Python code*

```
h = Expression("t*sin(x[0])*cos(x[1])", t=0.0)
while t < T:
    h.t = t
    ...
    t += dt
```

The `t` variable has here been used to create a time-dependent `Expression`. Arbitrary variable names may be used as long as they do not conflict with the names of built-in functions, such as `sin` or `exp`.

In addition to the above examples, the Python interface allows the direct definition of (more complex) subclasses of the C++ `Expression` class by supplying C++ code for their definition. For more information, we refer to the DOLFIN Programmer's Reference.

### *10.3.7 Variational forms*

DOLFIN relies on the FEniCS toolchain FIAT–UFL–FFC/SFC–UFC for the evaluation of finite element variational forms. Variational forms expressed in the UFL form language (Chapter 17) are compiled using one of the form compilers FFC or SFC (Chapters 11 and 15), and the generated UFC code (Chapter 16) is used by DOLFIN to evaluate (assemble) variational forms.

The UFL form language allows a wide range of variational forms to be expressed in a language close to the mathematical notation, as exemplified by the following expressions defining (in part) the bilinear and linear forms for the discretization of a linear elastic problem:

*UFL code*

```
a = inner(sigma(u), epsilon(v))*dx
L = dot(f, v)*dx
```

This should be compared to the corresponding mathematical notation:

$$a(u, v) = \int_\Omega \sigma(u) : \epsilon(v) \, \mathrm{d}x, \tag{10.3}$$

$$L(v) = \int_\Omega f \cdot v \, \mathrm{d}x. \tag{10.4}$$

Here, $\epsilon(v) = (\operatorname{grad} v + (\operatorname{grad} v)^T)/2$ denotes the symmetric gradient and $\sigma(v) = 2\mu\,\epsilon(v) + \lambda \operatorname{tr} \epsilon(v) I$ is the stress tensor. For a detailed presentation of the UFL form language, we refer to Chapter 17.

The code generation process must be handled explicitly by users of the C++ interface by calling a form compiler on the command-line. To solve the linear elastic problem above for a specific choice of parameter values (the Lamé constants $\mu$ and $\lambda$), a user may enter the following code in a file named `Elasticity.ufl`[2]:

*UFL code*

```
V = VectorElement("Lagrange", tetrahedron, 1)

u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)

E  = 10.0
nu = 0.3

mu    = E / (2.0*(1.0 + nu))
lmbda = E*nu / ((1.0 + nu)*(1.0 - 2.0*nu))

def sigma(v):
    return 2.0*mu*sym(grad(v)) + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

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

This code may be compiled using a UFL/UFC compliant form compiler to generate UFC C++ code. For example, using FFC:

*Bash code*

```
ffc -l dolfin Elasticity.ufl
```

[2]Note that 'lambda' has been deliberately misspelled since it is a reserved keyword in Python.

This generates a C++ header file (including implementation) named `Elasticity.h` which may be included in a C++ program and used to instantiate the two forms `a` and `L`:

*C++ code*

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

using namespace dolfin;

int main()
{
  UnitSquare mesh(8, 8);
  Elasticity::FunctionSpace V(mesh);
  Elasticity::BilinearForm a(V, V);
  Elasticity::LinearForm L(V);
  MyExpression f; // code for the definition of MyExpression omitted
  L.f = f;

  return 0;
}
```

The instantiation of the forms involves the instantiation of the `FunctionSpace` on which the forms are defined. Any coefficients appearing in the definition of the forms (here the right-hand side `f`) must be attached after the creation of the forms.

Python users may rely on automated code generation, and define variational forms directly as part of a Python script:

*Python code*

```
from dolfin import *

mesh = UnitSquare(8, 8)
V = VectorFunctionSpace(mesh, "Lagrange", 1)

u = TrialFunction(V)
v = TestFunction(V)
f = MyExpression() # code emitted for the definition of f

E  = 10.0
nu = 0.3

mu    = E / (2.0*(1.0 + nu))
lmbda = E*nu / ((1.0 + nu)*(1.0 - 2.0*nu))

def sigma(v):
    return 2.0*mu*sym(grad(v)) + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

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

This script will trigger automatic code generation for the definition of the `FunctionSpace V`. Code generation of the two forms `a` and `L` is postponed until the point when the corresponding discrete operators (the matrix and vector) are assembled.

### *10.3.8 Finite element assembly*

A core functionality of DOLFIN is the assembly of finite element variational forms. Given a variational form (`a`), DOLFIN assembles the corresponding discrete operator (`A`). The assembly of the

discrete operator follows the general algorithm described in Chapter 6. The following code illustrates how to assemble a scalar (`m`), a vector (`b`) and a matrix (`A`) from a functional (`M`), a linear form (`L`) and a bilinear form (`a`), respectively:

*C++ code*

```
Vector b;
Matrix A;

double m = assemble(M);
assemble(b, L);
assemble(A, a);
```

*Python code*

```
m = assemble(M)
b = assemble(L)
A = assemble(a)
```

The assembly of variational forms from the Python interface automatically triggers code generation, compilation and linking at run-time. The generated code is automatically instantiated and sent to the DOLFIN C++ compiler. As a result, finite element assembly from the Python interface is equally efficient as assembly from the C++ interface, with only a small overhead for handling the automatic code generation. The generated code is cached for later reuse, hence repeated assembly of the same form or running the same program twice does not re-trigger code generation. Instead, the previously generated code is automatically loaded from cache.

DOLFIN provides a common assembly algorithm for the assembly of tensors of any rank (scalars, vectors, matrices, ...) for any form. This is possible since the assembly algorithm relies on the `GenericTensor` interface, portions of the assembly algorithm that depend on the variational form and its particular discretization are generated prior to assembly, and the mesh interface is dimension-independent. The assembly algorithm accepts a number of optional arguments that control whether the sparsity of the assembled tensor should be reset before assembly and whether the tensor should be zeroed before assembly. Arguments may also be supplied to specify subdomains of the `Mesh` if the form is defined over particular subdomains (using `dx(0)`, `dx(1)` etc.).

In addition to the `assemble` function, DOLFIN provides the `assemble_system` function which assembles a pair of forms consisting of a bilinear and a linear form and applies essential boundary conditions during the assembly process. The application of boundary conditions as part of the call to `assemble_system` preserves symmetry of the matrix being assembled (see Chapter 6).

The assembly algorithms have been parallelized for both distributed memory architectures (clusters) using MPI and shared memory architectures (multi-core) using OpenMP. This is discussed in more detail in Section 10.4.

### *10.3.9 Boundary conditions*

DOLFIN handles the application of both Neumann (natural) and Dirichlet (essential) boundary conditions.[3] Natural boundary conditions are usually applied via the variational statement of a problem, whereas essential boundary conditions are usually applied to the discrete system of equations.

[3] As noted in Chapter 2, Dirichlet boundary conditions may sometimes be *natural* and Neumann boundary conditions may sometimes be *essential*.

*Natural boundary conditions.* Natural boundary conditions typically appear as boundary terms as the result of integrating by parts a partial differential equation multiplied by a test function. As a simple example, we consider the linear elastic variational problem. The partial differential equation governing the displacement of an elastic body may be expressed as

$$\begin{aligned} -\operatorname{div}\sigma(u) &= f \quad \text{in } \Omega, \\ \sigma \cdot n &= g \quad \text{on } \Gamma_{\mathrm{N}} \subset \partial\Omega, \\ u &= u_0 \quad \text{on } \Gamma_{\mathrm{D}} \subset \partial\Omega, \end{aligned} \tag{10.5}$$

where $u$ is the unknown displacement field to be computed, $\sigma(u)$ is the stress tensor, $f$ is a given body force, $g$ is a given traction on a portion $\Gamma_{\mathrm{N}}$ of the boundary, and $u_0$ is a given displacement on a portion $\Gamma_{\mathrm{D}}$ of the boundary. Multiplying by a test function $v$ and integrating by parts, we obtain

$$\int_\Omega \sigma(u) : \epsilon(v)\,\mathrm{d}x - \int_{\partial\Omega} (\sigma \cdot n) \cdot v\,\mathrm{d}s = \int_\Omega f \cdot v\,\mathrm{d}x, \tag{10.6}$$

where we have used the symmetry of $\sigma(u)$ to replace $\operatorname{grad} v$ by the symmetric gradient $\epsilon(v)$. Since the displacement $u$ is known on the Dirichlet boundary $\Gamma_{\mathrm{D}}$, we let $v = 0$ on $\Gamma_{\mathrm{D}}$. Furthermore, we replace $\sigma \cdot n$ by the given traction $g$ on the remaining (Neumann) portion of the boundary $\Gamma_{\mathrm{N}}$ to obtain

$$\int_\Omega \sigma(u) : \epsilon(v)\,\mathrm{d}x = \int_\Omega f \cdot v\,\mathrm{d}x + \int_{\Gamma_{\mathrm{N}}} g \cdot v\,\mathrm{d}s. \tag{10.7}$$

The following code demonstrates how to implement this variational problem in the UFL form language, either as part of a `.ufl` file or as part of a Python script:

*UFL code*

```
a = inner(sigma(u), sym(grad(v)))*dx
L = dot(f, v)*dx + dot(g, v)*ds
```

To specify that the boundary integral `dot(g, v)*ds` should only be evaluated along the Neumann boundary $\Gamma_{\mathrm{N}}$, one must specify which part of the boundary is included in the `ds` integral. If there is only one Neumann boundary, then one may simply write the `ds` integral as an integral over the entire boundary, including the Dirichlet boundary as the test function `v` will be set to zero along the Dirichlet boundary.

In cases where there is more than one Neumann boundary condition, one must instead specify the Neumann boundary in terms of a `FacetFunction`. This `FacetFunction` must specify for each facet of the `Mesh` to which part of the boundary it belongs. For the current example, an appropriate strategy is to mark each facet on the Neumann boundary by `0` and all other facets (including facets internal to the domain) by `1`. This can be accomplished in a number of different ways. One simple way to do this is to use the program MeshBuilder and graphically mark the facets of the `Mesh`. Another option is through the DOLFIN class `SubDomain`. The following code illustrates how to mark all boundary facets to the left of $x = 0.5$ as the first Neumann boundary and all other boundary facets as the second Neumann boundary. Note the use of the `on_boundary` argument supplied by DOLFIN to the `inside` function. This argument informs whether a point is located on the boundary $\partial\Omega$ of $\Omega$, and this allows us to mark only facets that are on the boundary and to the left of $x = 0.5$. Also note the use of `DOLFIN_EPS` which makes sure that we include points that, as a result of finite precision arithmetic, may be located just to the right of $x = 0.5$.

*C++ code*

```
class NeumannBoundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  {
    return x[0] < 0.5 + DOLFIN_EPS && on_boundary;
  }
};

NeumannBoundary neumann_boundary;
FacetFunction<uint> exterior_facet_domains(mesh);
exterior_facet_domains.set_all(1);
neumann_boundary.mark(exterior_facet_domains, 0);
```

*Python code*

```
class NeumannBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < 0.5 + DOLFIN_EPS and on_boundary

neumann_boundary = NeumannBoundary()
exterior_facet_domains = FacetFunction("uint", mesh)
exterior_facet_domains.set_all(1)
neumann_boundary.mark(exterior_facet_domains, 0)
```

When combined with integrals defined using `ds(0)` and `ds(1)`, those integrals will correspond to integration over the domain boundary to the left of $x = 0.5$ and all facets to the right of $x = 0.5$, respectively.

Once the boundaries have been specified as a `FacetFunction`, that object can be used to define the corresponding domains of integration. This is done differently in C++ and Python. From C++, one must assign to the `ds` member variable of the corresponding forms:

*C++ code*

```
a.ds = exterior_facet_domains;
L.ds = exterior_facet_domains;
```

In addition to `exterior_facet_domains` specified in terms of the `ds` member variable, one may similarly specify `cell_domains` using the `dx` member variable and `interior_facet_domains` using the `dS` variable. Note that different forms may potentially use different definitions of their boundaries. From Python, one may simply connect the boundary definition to the corresponding measure by subscripting:

*Python code*

```
dss = ds[neumann_boundary]
a = ... + g*v*dss(0) + h*v*dss(1) + ...
```

The correct specification of boundaries is a common error source. For debugging the specification of boundary conditions, it can be helpful to plot the `FacetFunction` that specifies the boundary markers by writing the `FacetFunction` to a VTK file (see the file I/O section) or using the `plot` command. When using the `plot` command, the plot shows the facet values interpolated to the vertices of the `Mesh`. As a result, care must be taken to interpret the plot close to domain boundaries (corners) in this case. The issue is not present in the VTK output.

*Essential boundary conditions.* The application of essential boundary conditions is handled by the class `DirichletBC`. Using this class, one may specify a Dirichlet boundary condition in terms of a

`FunctionSpace`, a `Function` or an `Expression`, and a subdomain. The subdomain may be specified either in terms of a `SubDomain` object or in terms of a `FacetFunction`. A `DirichletBC` specifies that the solution should be equal to the given value on the given subdomain.

The following code examples illustrate how to define the Dirichlet condition $u(x) = u_0(x) = \sin x$ on the Dirichlet boundary $\Gamma_D$ (assumed here to be the part of the boundary to the right of $x = 0.5$) for the elasticity problem (10.5) using the `SubDomain` class. Alternatively, the subdomain may be specified using a `FacetFunction`.

*C++ code*

```
class DirichletValue : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0]);
  }
};

class DirichletBoundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  {
    return x[0] > 0.5 - DOLFIN_EPS && on_boundary;
  }
};

DirichletValue u_0;
DirichletBoundary Gamma_D;

DirichletBC bc(V, u_0, Gamma_D);
```

*Python code*

```
class DirichletValue(Expression):
    def eval(self, value, x):
        values[0] = sin(x[0])

class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] > 0.5 - DOLFIN_EPS and on_boundary

u_0 = DirichletValue()
Gamma_D = DirichletBoundary()

bc = DirichletBC(V, u_0, Gamma_D)
```

Python users may also use the following compact syntax:

*Python code*

```
u_0 = Expression("sin(x[0])")
bc = DirichletBC(V, u_0, "x[0] > 0.5 && on_boundary")
```

To speed up the application of Dirichlet boundary conditions, users of the Python interface may also use the function `compile_subdomains`. For details of this, we refer to the DOLFIN Programmer's Reference.

A Dirichlet boundary condition can be applied to a linear system or to a vector of degrees of freedom associated with a `Function`, as illustrated by the following code examples:

*C++ code*

```
bc.apply(A, b);
bc.apply(u.vector());
```

*Python code*

```
bc.apply(A, b)
bc.apply(u.vector())
```

The application of a Dirichlet boundary condition to a linear system will identify all degrees of freedom that should be set to the given value and modify the linear system such that its solution respects the boundary condition. This is accomplished by zeroing and inserting 1 on the diagonal of the rows of the matrix corresponding to Dirichlet values, and inserting the Dirichlet value in the corresponding entry of the right-hand side vector. This application of boundary conditions does not preserve symmetry. If symmetry is required, one may alternatively consider using the `assemble_system` function which applies Dirichlet boundary conditions symmetrically as part of the assembly process.

Multiple boundary conditions may be applied to a single system or vector. If two different boundary conditions are applied to the same degree of freedom, the last applied value will overwrite any previously set values.

### *10.3.10 Variational problems*

Variational problems (finite element discretizations of partial differential equations) can be easily solved in DOLFIN using the `solve` function. Both linear and nonlinear problems can be solved. A linear problem must be expressed in the following canonical form: find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall\, v \in \hat{V}. \tag{10.8}$$

A nonlinear problem must be expressed in the following canonical form: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}. \tag{10.9}$$

In the case of a linear variational problem specified in terms of a bilinear form `a` and a linear form `L`, the solution is computed by assembling the matrix `A` and vector `b` of the corresponding linear system, then applying boundary conditions to the system, and finally solving the linear system. In the case of a nonlinear variational problem specified in terms of a linear form `F` and a bilinear form `J` (the derivative or Jacobian of `F`), the solution is computed by Newton's method.

The code examples below demonstrate how to solve a linear variational problem specified in terms of a bilinear form `a`, a linear form `L` and a list of Dirichlet boundary conditions given as `DirichletBC` objects:

*C++ code*

```
std::vector<const BoundaryCondition*> bcs;
bcs.push_back(&bc0);
bcs.push_back(&bc1);
bcs.push_back(&bc2);

Function u(V);
solve(a == L, u, bcs);
```

*Python code*

```
bcs = [bc0, bc1, bc2]

u = Function(V)
solve(a == L, u, bcs=bcs)
```

To solve a nonlinear variational problem, one must supply a linear form `F` and, in the case of C++, its derivative `J`, which is a bilinear form. In Python, the derivative is computed automatically but may also be specified manually. In many cases, the derivative can be easily computed using the function `derivative`, either in a `.ufl` form file or as part of a Python script. We here demonstrate how a nonlinear problem may be solved using the Python interface. Nonlinear variational problems may be solved similarly in C++.

*Python code*

```
u  = Function(V)
v  = TestFunction(V)
F  = inner((1 + u**2)*grad(u), grad(v))*dx - f*v*dx

# Let DOLFIN compute Jacobian
solve(F == 0, u, bcs=bcs)

# Differentiate to get Jacobian
J = derivative(F, u)

# Supply Jacobian manually
solve(F == 0, u, bcs=bcs, J=J)
```

More advanced control over the solution process may be gained by using the classes `LinearVariational{Problem,Solver}` and `NonlinearVariational{Problem,Solver}`. Use of these classes is illustrated by the following code examples:

*Python code*

```
u = Function(V)
problem = LinearVariationalProblem(a, L, u, bcs=bcs)
solver = LinearVariationalSolver(problem)
solver.parameters["linear_solver"] = "gmres"
solver.parameters["preconditioner"] = "ilu"
solver.solve()
```

*Python code*

```
u = Function(V)
problem = NonlinearVariationalProblem(F, u, bcs=bcs, J=J)
solver = NonlinearVariationalSolver(problem)
solver.parameters["linear_solver"] = "gmres"
solver.parameters["preconditioner"] = "ilu"
solver.solve()
```

These classes may be used similarly from C++.

The solver classes provide a range of parameters that can be adjusted to control the solution process. For example, to view the list of available parameters for a `LinearVariationalSolver` or `NonlinearVariationalSolver`, issue the following commands:

*C++ code*

```
info(solver.parameters, true)
```

*Python code*

```
info(solver.parameters, True)
```

| Suffix | File format |
|---|---|
| `.xml` | DOLFIN XML format |
| `.ele` / `.node` | Triangle file format |
| `.mesh` | Medit format, generated by TetGen with option `-g` |
| `.msh` / `.gmsh` | Gmsh version 2.0 format |
| `.grid` | Diffpack tetrahedral grid format |
| `.inp` | Abaqus tetrahedral grid format |
| `.e` / `.exo` | Sandia Exodus II file format |
| `.ncdf` | ncdump'ed Exodus II file format |
| `.vrt`/`.cell` | Star-CD tetrahedral grid format |

Table 10.4: List of file formats supported by the `dolfin-convert` script.

### *10.3.11 File I/O and visualization*

*Preprocessing.* DOLFIN has capabilities for mesh generation only in the form of the built-in meshes `UnitSquare`, `UnitCube`, etc. External software must be used to generate more complicated meshes. To simplify this process, DOLFIN provides a simple script `dolfin-convert` to convert meshes from other formats to the DOLFIN XML format. Currently supported file formats are listed in Table 10.4. The following code illustrates how to convert a mesh from the Gmsh format (suffix `.msh` or `.gmsh`) to the DOLFIN XML format:

*Bash code*

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

Once a mesh has been converted to the DOLFIN XML file format, it can be read into a program, as illustrated by the following code examples:

*C++ code*

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

*Python code*

```
mesh = Mesh("mesh.xml")
```

*Postprocessing.* To visualize a solution (`Function`), a `Mesh` or a `MeshFunction`, the `plot` command[4] can be issued, from either C++ or Python:

*C++ code*

```
plot(u);
plot(mesh);
plot(mesh_function);
```

*Python code*

```
plot(u)
plot(mesh)
plot(mesh_function)
```

[4]The plot command requires a working installation of the `viper` Python module. Plotting finite elements requires access to FFC and the `soya` Python plotting module.

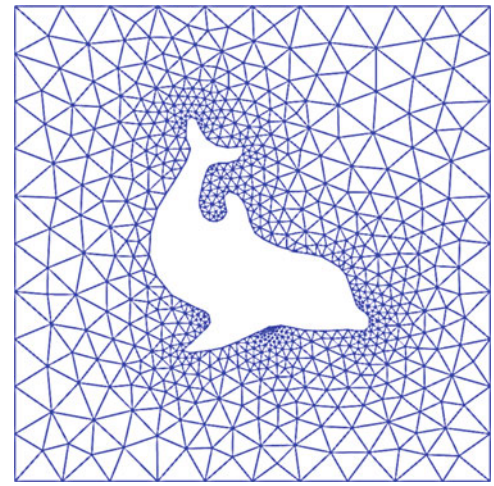

Figure 10.6: Plotting a mesh using the DOLFIN `plot` command, here the mesh `dolfin-1.xml.gz` distributed with DOLFIN.

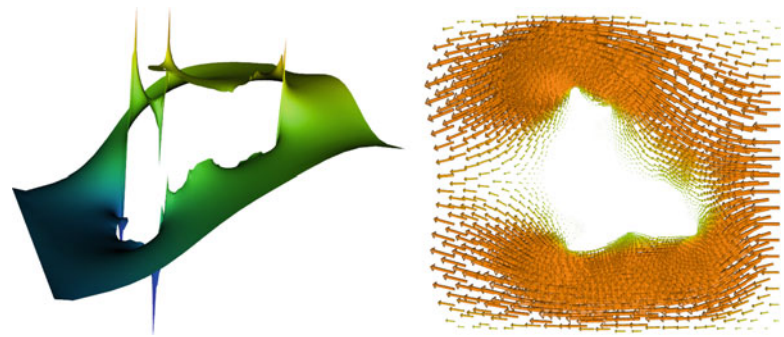

Figure 10.7: Plotting a scalar and a vector-valued function using the DOLFIN `plot` command, here the pressure (left) and velocity (right) from a solution of the Stokes equations on the mesh from Figure 10.6.

Example plots generated using the `plot` command are presented in Figures 10.6 and 10.7. From Python, one can also plot expressions and finite elements:

*Python code*

```
plot(grad(u))
plot(u*u)

element = FiniteElement("BDM", tetrahedron, 3)
plot(element)
```

To enable interaction with a plot window (rotate, zoom) from Python, call the function `interactive`, or add an optional argument `interactive=True` to the `plot` command.

The `plot` command provides rudimentary plotting, and advanced postprocessing is better handled by external software such as ParaView and MayaVi2. This is easily accomplished by storing the solution (a `Function` object) to file in PVD format (ParaView Data, an XML-based format). This can be done in both C++ and Python by writing to a file with the `.pvd` extension, as illustrated in the following code examples:

*C++ code*

```
File file("solution.pvd");
file << u;
```

*Python code*

```
file = File("solution.pvd")
file << u
```

The standard PVD format is ASCII based, hence the file size can become very large for large data sets. To use a compressed binary format, a string `"compressed"` can be used when creating a PVD-based `File` object:

*C++ code*

```
File file("solution.pvd", "compressed");
```

If multiple `Functions` are written to the same file (by repeated use of `<<`), then the data is interpreted as a time series, which may then be animated in ParaView or MayaVi2. Each frame of the time series is stored as a `.vtu` (VTK unstructured data) file, with references to these files stored in the `.pvd` file. When writing time-dependent data, it can be useful to store the time `t` of each snapshot. This is done as illustrated below:

*C++ code*

```
File file("solution.pvd", "compressed");
file << std::make_pair<const Function*, double>(&u, t);
```

*Python code*

```
file = File("solution.pvd", "compressed");
file << (u, t)
```

Storing the time is particularly useful when animating simulations that use a varying time step.

The PVD format supports parallel post-processing. When running in parallel, a single `.pvd` file is created and a `.vtu` file is created for the data on each partition. Results computed in parallel can be viewed seamlessly using ParaView.

*DOLFIN XML format.* DOLFIN XML is the native format of DOLFIN. An advantage of XML is that it is a robust and human-readable format. If the files are compressed, there is also little overhead in terms of file size compared to a binary format.

Many of the classes in DOLFIN can be written to and from DOLFIN XML files using the standard stream operators `<<` and `>>`, as illustrated in the following code examples:

*C++ code*

```
File vector_file("vector.xml");
vector_file << vector;
vector_file >> vector;

File mesh_file("mesh.xml");
mesh_file << mesh;
mesh_file >> mesh;

File parameters_file("parameters.xml");
parameters_file << parameters;
parameters_file >> parameters;
```

*Python code*

```
vector_file = File("vector.xml")
vector_file << vector
vector_file >> vector

mesh_file = File("mesh.xml")
mesh_file << mesh
mesh_file >> mesh

parameters_file = File("parameters.xml")
parameters_file << parameters
parameters_file >> parameters
```

One cannot read/write `Function` and `FunctionSpace` objects since the representation of a `FunctionSpace` (and thereby the representation of a `Function`) relies on generated code.

DOLFIN automatically handles reading and writing of gzipped XML files. Thus, one may save space by storing meshes and other data in gzipped XML files (with suffix `.xml.gz`).

*Time series.* For time-dependent problems, it may be useful to store a sequence of solutions or meshes in a format that enables fast reading/writing of data. For this purpose, DOLFIN provides the `TimeSeries` class. This enables the storage of a series of `Vectors` (of degrees of freedom) and/or `Meshes`. The following code illustrates how to store a series of `Vectors` and `Meshes` to a `TimeSeries`:

*C++ code*

```
TimeSeries time_series("simulation_data");

while (t < T)
{
  ...
  time_series.store(u.vector(), t);
  time_series.store(mesh, t);
  t += dt;
}
```

*Python code*

```
time_series = TimeSeries("simulation_data")

while t < T:
    ...
    time_series.store(u.vector(), t)
    time_series.store(mesh, t)
    t += dt
```

Data in a `TimeSeries` are stored in a binary format with one file for each stored dataset (`Vector` or `Mesh`) and a common index. Data may be retrieved from a `TimeSeries` by calling the `retrieve` member function as illustrated in the code examples below. If a dataset is not stored at the requested time, then the values are interpolated linearly for `Vectors`. For `Meshes`, the closest data point will be used.

*C++ code*

```
time_series.retrieve(u.vector(), t);
time_series.retrieve(mesh, t);
```

*Python code*

```
time_series.retrieve(u.vector(), t)
time_series.retrieve(mesh, t)
```

### 10.3.12 Logging / diagnostics

DOLFIN provides a simple interface for the 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.

*Printing messages.* Informational messages from DOLFIN are normally printed using the `info` command. This command takes a string argument and an optional list of variables to be formatted, much like the standard C `printf` command. Note that the `info` command automatically appends a newline to the given string. Alternatively, C++ users may use the `dolfin::cout` and `dolfin::endl` objects for C++ style formatting of messages as illustrated below.

*C++ code*

```
info("Assembling system of size %d x %d.", M, N);
cout << "Assembling system of size " << M << " x " << N << "." << endl;
```

*Python code*

```
info("Assembling system of size %d x %d." % (M, N))
```

The `info` command and the `dolfin::cout/endl` objects differ from the standard C `printf` command and the C++ `std::cout/endl` objects in that the output is directed into a special stream, the output of which may be redirected to destinations other than standard output. In particular, one may completely disable output from DOLFIN, or select the verbosity of printed messages, as explained below.

*Warnings and errors.* In addition to the `info` command, DOLFIN provides the commands `warning` and `error` that can be used to issue warnings and errors, respectively. These two commands work in much the same way as the `info` command. However, the `warning` command will prepend the given message with `"*** Warning: "` and the `error` command will raise an exception that can be caught, from both C++ and Python. Both commands will also print the message at a *log level* higher than messages printed using `info`.

*Setting the log level.* The DOLFIN log level determines which messages routed through the logging system will be printed. Only messages on a level higher than or equal to the current log level are printed. The log level of DOLFIN may be set using the function `set_log_level`. This function expects an integer value that specifies the log level. To simplify the specification of the log level, one may use one of a number of predefined log levels as listed in Table 10.5. The default log level is `INFO`. Log messages may be switched off entirely by calling the command `set_log_active(false)` from C++ and `set_log_active(False)` from Python. For technical reasons, the log level for debugging messages is named `DBG` in C++ and `DEBUG` in Python. This is summarized in Table 10.5.

| Log level | value |
|---|---|
| ERROR | 40 |
| WARNING | 30 |
| INFO | 20 |
| PROGRESS | 16 |
| DBG / DEBUG | 10 |

Table 10.5: Log levels in DOLFIN.

To print messages at an arbitrary log level, one may specify the log level to the `log` command, as illustrated in the code examples below.

*C++ code*

```
info("Test message");                       // will be printed
cout << "Test message" << endl;             // will be printed
log(DBG, "Test message");                   // will not be printed
log(15, "Test message");                    // will not be printed

set_log_level(DBG);
info("Test message");                       // will be printed
cout << "Test message" << endl;             // will be printed
log(DBG, "Test message");                   // will be printed
log(15, "Test message");                    // will be printed

set_log_level(WARNING);
info("Test message");                       // will not be printed
cout << "Test message" << endl;             // will not be printed
warning("Test message");                    // will be printed
std::cout << "Test message" << std::endl;   // will be printed!
```

*Python code*

```
info("Test message")                        # will be printed
log(DEBUG, "Test message")                  # will not be printed
log(15, "Test message")                     # will not be printed

set_log_level(DEBUG)
info("Test message")                        # will be printed
log(DEBUG, "Test message")                  # will be printed
log(15, "Test message")                     # will be printed

set_log_level(WARNING)
info("Test message")                        # will not be printed
warning("Test message")                     # will be printed
print "Test message"                        # will be printed!
```

*Printing objects.* Many of the standard DOLFIN objects can be printed using the `info` command, as illustrated in the code examples below.

*C++ code*

```
info(vector);
info(matrix);
info(solver);
info(mesh);
info(mesh_function);
info(function);
info(function_space);
info(parameters);
```

*Python code*

```
info(vector)
info(matrix)
info(solver)
info(mesh)
info(mesh_function)
info(function)
info(function_space)
info(parameters)
```

The above commands will print short informal messages. For example, the command `info(mesh)` may result in the following output:

*Generated code*

```
<Mesh of topological dimension 2 (triangles) with 25 vertices and 32 cells, ordered>
```

In the Python interface, the same short informal message can be printed by calling `print mesh`. To print more detailed data, one may set the verbosity argument of the `info` function to true (defaults to false), which will print a detailed summary of the object.

*C++ code*

```
info(mesh, true);
```

*Python code*

```
info(mesh, True)
```

The detailed output for some objects may be very lengthy.

*Tasks and progress bars.* In addition to basic commands for printing messages, DOLFIN provides a number of commands for organizing the diagnostic output from a simulation program. Two such commands are `begin` and `end`. These commands can be used to nest the output from a program; each call to `begin` increases the indentation level by one unit (two spaces), while each call to `end` decreases the indentation level by one unit.

Another way to provide feedback is via progress bars. DOLFIN provides the `Progress` class for this purpose. Although an effort has been made to minimize the overhead of updating the progress bar, it should be used with care. If only a small amount of work is performed in each iteration of a loop, the relative overhead of using a progress bar may be substantial. The code examples below illustrate the use of the `begin/end` commands and the progress bar.

*C++ code*

```
begin("Starting nonlinear iteration.");
info("Updating velocity.");
info("Updating pressure.");
info("Computing residual.");
end();

Progress p("Iterating over all cells.", mesh.num_cells());
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  ...
  p++;
}

Progress q("Time-stepping");
while (t < T)
{
  ...
  t += dt;
  q = t / T;
}
```

*Python code*

```
begin("Starting nonlinear iteration.")
info("Updating velocity.")
info("Updating pressure.")
info("Computing residual.")
end()

p = Progress("Iterating over all cells.", mesh.num_cells())
for cell in cells(mesh):
  ...
  p += 1

q = Progress("Time-stepping")
while t < T:
  ...
  t += dt
  q.update(t / T)
```

*Setting timers.* Timing can be accomplished using the `Timer` class. A `Timer` is automatically started when it is created, and automatically stopped when it goes out of scope. Creating a `Timer` at the start of a function is therefore a convenient way to time that function, as illustrated in the code examples below.

*C++ code*

```
void solve(const Matrix& A, Vector& x, const Vector& b)
{
  Timer timer("Linear solve");
  ...
}
```

*Python code*

```
def solve(A, b):
  timer = Timer("Linear solve")
  ...
  return x
```

One may explicitly call the `start` and `stop` member functions of a `Timer`. To directly access the value of a timer, the `value` member function can be called. A summary of the values of all timers created during the execution of a program can be printed by calling the `list_timings` function.

### *10.3.13 Parameters*

DOLFIN keeps a global database of parameters that control the behavior of its various components. Parameters are controlled via a uniform type-independent interface that allows the retrieval of parameter values, modification of parameter values, and the addition of new parameters to the database. Different components (classes) of DOLFIN also rely on parameters that are local to each instance of the class. This permits different parameter values to be set for different objects of a class.

Parameter values can be either integer-valued, real-valued (standard double), string-valued or boolean-valued. Parameter names must not contain spaces.

*Accessing parameters.* Global parameters can be accessed through the global variable `parameters`. The below code illustrates how to print the values of all parameters in the global parameter database, and how to access and change parameter values.

*C++ code*

```
info(parameters, True);
uint num_threads = parameters["num_threads"];
bool allow_extrapolation = parameters["allow_extrapolation"];
parameters["num_threads"] = 8;
parameters["allow_extrapolation"] = true;
```

*Python code*

```
info(parameters, True)
num_threads = parameters["num_threads"]
allow_extrapolation = parameters["allow_extrapolation"]
parameters["num_threads"] = 8
parameters["allow_extrapolation"] = True
```

Parameters that are local to specific components of DOLFIN can be controlled by accessing the member variable named `parameters`. The following code illustrates how to set some parameters for a Krylov solver:

*C++ code*

```
KrylovSolver solver;
solver.parameters["absolute_tolerance"] = 1e-6;
solver.parameters["report"] = true;
solver.parameters("gmres")["restart"] = 50;
solver.parameters("preconditioner")["reuse"] = true;
```

*Python code*

```
solver = KrylovSolver()
solver.parameters["absolute_tolerance"] = 1e-6
solver.parameters["report"] = True
solver.parameters["gmres"]["restart"] = 50
solver.parameters["preconditioner"]["reuse"] = True
```

The above example accesses the nested parameter databases `"gmres"` and `"preconditioner"`. DOLFIN parameters may be nested to arbitrary depths, which helps with organizing parameters into different categories. Note the subtle difference in accessing nested parameters in the two interfaces. In the C++ interface, nested parameters are accessed by brackets `("...")`, and in the Python interface are they accessed by square brackets `["..."]`. The parameters that are available for a certain component can be viewed by using the `info` function.

*Adding parameters.* Parameters can be added to an existing parameter database using the `add` member function which takes the name of the new parameter and its default value. It is also simple to create new parameter databases by creating a new instance of the `Parameters` class. The following code demonstrates how to create a new parameter database and adding to it a pair of integer-valued and floating-point valued parameters:

*C++ code*

```
Parameters parameters("my_parameters");
my_parameters.add("foo", 3);
my_parameters.add("bar", 0.1);
```

*Python code*

```
my_parameters = Parameters("my_parameters")
my_parameters.add("foo", 3)
my_parameters.add("bar", 0.1)
```

A parameter database resembles the `dict` class in the Python interface. A user can iterate over the `keys`, `values` and `items`:

*Python code*

```
for key, value in parameters.items():
    print key, value
```

A Python `dict` can also be used to update a Parameter database:

*Python code*

```
d = dict(num_threads=4, krylov_solver=dict(absolute_tolerance=1e-6))
parameters.update(d)
```

A parameter database can also be created in more compact way in the Python interface:

*Python code*

```
my_parameters = Parameters("my_parameters", foo=3, bar=0.1,
                           nested=Parameters("nested", baz=True))
```

*Parsing command-line parameters.* Command-line parameters may be parsed into the global parameter database or into any other parameter database. The following code illustrates how to parse command-line parameters in C++ and Python, and how to pass command-line parameters to the program:

*C++ code*

```
int main(int argc, char* argv[])
{
  ...
  parameters.parse(argc, argv);
  ...
}
```

*Python code*

```
parameters.parse()
```

*Bash code*

```
python myprogram.py --num_threads 8 --allow_extrapolation true
```

*Storing parameters to file.* It can be useful to store parameter values to file, for example to document which parameter values were used to run a simulation or to reuse a set of parameter values from a previous run. The following code illustrates how to write and then read back parameter values to/from a DOLFIN XML file:

*C++ code*

```
File file("parameters.xml");
file << parameters;
file >> parameters;
```

*Python code*

```
file = File("parameters.xml")
file << parameters
file >> parameters
```

At startup, DOLFIN automatically scans the current directory and the directory `.config/fenics` in the user's home directory (in that order) for a file named `dolfin_parameters.xml`. If found, these parameters are read into DOLFIN's global parameter database.

## 10.4 Implementation notes

In this section, we comment on specific aspects of the implementation of DOLFIN, including parallel computing, the generation of the Python interface, and just-in-time compilation.

### 10.4.1 Parallel computing

DOLFIN supports parallel computing on multi-core workstations through to massively parallel supercomputers. It is designed such that users can perform parallel simulations using the same code that is used for serial computations.

Two paradigms for parallel simulation are supported. The first paradigm is multithreading for shared memory machines. The second paradigm is fully distributed parallelization for distributed memory machines. For both paradigms, special preprocessing of a mesh is required. For multithreaded parallelization, a so-called coloring approach is used (see Figure 10.8a), and for distributed parallelization a mesh partitioning approach is used (see Figure 10.8b). Aspects of these two approaches are discussed below. It is also possible to combine the approaches, thereby yielding hybrid approaches to leverage the power of modern clusters of multi-core processors.

*Shared memory parallel computing.* Multithreaded assembly for finite element matrices and vectors on shared memory machines is supported using OpenMP. It is activated by setting the number of threads to use via the parameter system. For example, the code

*C++ code*

```
parameters["num_threads"] = 6;
```

instructs DOLFIN to use six threads in the assembly process. During assembly, DOLFIN loops over the cells or cell facets in a mesh, and computes local contributions to the global matrix or vector, which are then added to the global matrix or vector. When using multithreaded assembly, each thread is assigned a collection of cells or facets for which it is responsible. This is transparent to the user.

The use of multithreading requires design care to avoid race conditions, which occur if multiple threads attempt to write to the same memory location at the same time. Race conditions will typically result in unpredictable behavior of a program. To avoid race conditions during assembly, which

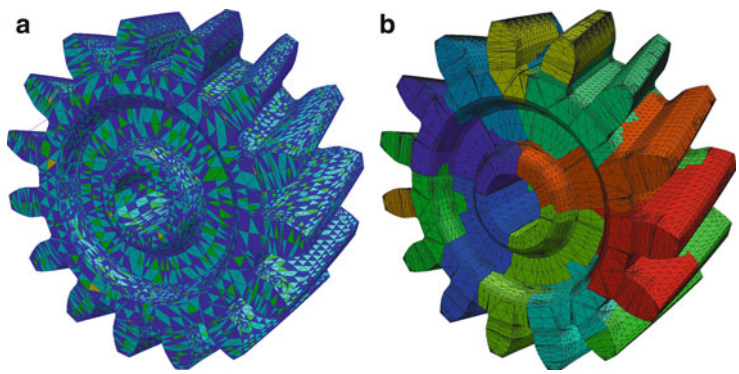


Figure 10.8: A mesh that is (a) colored based on facet connectivity such that cells that share a common facet have different colors and (b) partitioned into 12 parts, with each partition indicated by a color.

would occur if two threads were to add values to a global matrix or vector at almost the same time, DOLFIN uses a graph coloring approach. Before assembly, the mesh on a given process is 'colored' such that each cell is assigned a color (which in practice is an integer) and such that no two neighboring cells have the same color. The sense in which cells are neighbors for a given problem depends on the type of finite element being used. In most cases, cells that share a vertex are considered neighbors, but in other cases cells that share edges or facets may be considered neighbors. During assembly, cells are assembled by color. All cells of the first color are shared among the threads and assembled, and this is followed by the next color. Since cells of the same color are not neighbors, and therefore do not share entries in the global matrix or vector, race conditions will not occur during assembly. The coloring of a mesh is performed in DOLFIN using either the interface to the Boost Graph Library or the interface to Zoltan (which is part of the Trilinos project). Figure 10.8a shows a mesh that has been colored such that no two neighboring cells (in the sense of a shared facet) are of the same color.

Multithreaded support in third-party linear algebra libraries is limited at the present time, but is an area of active development. The LU solver PaStiX, which can be accessed via the PETSc linear algebra backend, supports multithreaded parallelism.

*Distributed memory parallel computing.* Fully distributed parallel computing is supported using the Message Passing Interface (MPI). To perform parallel simulations, DOLFIN should be compiled with MPI and a parallel linear algebra backend (such as PETSc or Trilinos) enabled. To execute a parallel simulation, a DOLFIN program should be launched using `mpirun` (the name of the program to launch MPI programs may differ on some computers). A C++ program using 16 processes can be executed using:

*Bash code*

```
mpirun -n 16 ./myprogram
```

and for Python:

*Bash code*

```
mpirun -n 16 python myprogram.py
```

DOLFIN supports fully distributed parallel meshes, which means that each processor has a copy of only the portion of the mesh for which it is responsible. This approach is scalable since no processor is required to hold a copy of the full mesh. An important step in a parallel simulation is the partitioning of the mesh. DOLFIN can perform mesh partitioning in parallel using the libraries ParMETIS and SCOTCH (Pellegrini). The library to be used for mesh partitioning can be specified via the parameter system, e.g., to use SCOTCH:

*C++ code*

```
parameters["mesh_partitioner"] = "SCOTCH";
```

or to use ParMETIS:

*Python code*

```
parameters["mesh_partitioner"] = "ParMETIS"
```

Figure 10.8b shows a mesh that has been partitioned in parallel into 12 domains. One process would take responsibility for each domain.

If a parallel program is launched using MPI and a parallel linear algebra backend is enabled, then linear algebra operations will be performed in parallel. In most applications, this will be transparent

to the user. Parallel output for postprocessing is supported through the PVD output format, and is used in the same way as for serial output. Each process writes an output file, and the single main output file points to the files produced by the different processes.

### 10.4.2 Implementation and generation of the Python interface

The DOLFIN C++ library is wrapped to Python using the Simplified Wrapper and Interface Generator SWIG (Beazley, 1996; SWIG); see Chapter 19 for more details. The wrapped C++ library is accessible in a Python module named `cpp` residing inside the main `dolfin` module of DOLFIN. This means that the compiled module, with all its functions and classes, can be accessed directly by:

*Python code*

```
from dolfin import cpp
Function = cpp.Function
assemble = cpp.assemble
```

The classes and functions in the `cpp` module have the same functionality as the corresponding classes and functions in the C++ interface. In addition to the wrapper layer automatically generated by SWIG, the DOLFIN Python interface relies on a number of components implemented directly in Python. Both are imported into the Python module named `dolfin`. In the following sections, the key customizations to the DOLFIN interface that facilitate this integration are presented. The Python interface also integrates well with the NumPy and SciPy toolkits, which is also discussed below.

### 10.4.3 UFL integration and just-in-time compilation

In the Python interface, the UFL form language has been integrated with the Python wrapped DOLFIN C++ module. When explaining the integration, we use in this section the notation `dolfin::Foo` or `dolfin::bar` to denote a C++ class or function in DOLFIN. The corresponding SWIG-wrapped classes or functions will be referred to as `cpp.Foo` and `cpp.bar`. A class in UFL will be referred to as `ufl.Foo` and a class in UFC as `ufc::foo` (note lower case). The Python classes and functions in the added Python layer on top of the wrapped C++ library, will be referred to as `dolfin.Foo` or `dolfin.bar`. The prefixes of the classes and functions are sometimes skipped for convenience. Most of the code snippets presented in this section are pseudo code. Their purpose is to illustrate the logic of a particular method or function. Parts of the actual code may be intentionally excluded. An interested reader can examine particular classes or functions in the code for a full understanding of the implementation.

*Construction of function spaces.* In the Python interface, `ufl.FiniteElement` and `dolfin::FunctionSpace` are integrated. The declaration of a `FunctionSpace` is similar to that of a `ufl.FiniteElement`, but instead of a cell type (for example, `triangle`) the `FunctionSpace` constructor takes a `cpp.Mesh` (`dolfin.Mesh`):

*Python code*

```
mesh = UnitSquare(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)
```

In the Python constructor of `FunctionSpace`, a `ufl.FiniteElement` is instantiated. The `FiniteElement` is passed to a just-in-time (JIT) compiler, which returns compiled and Python-wrapped `ufc` objects: a `ufc::finite_element` and a `ufc::dofmap`. These two objects, together with the mesh, are used

to instantiate a `cpp.FunctionSpace`. The following pseudo code illustrates the instantiation of a `FunctionSpace` from the Python interface:

*Python code*

```
class FunctionSpace(cpp.FunctionSpace):
    def __init__(self, mesh, family, degree):
        # Figure out the domain from the mesh topology
        if mesh.topology().dim() == 2:
            domain = ufl.triangle
        else:
            domain = ufl.tetrahedron

        # Create the UFL FiniteElement
        self.ufl_element = ufl.FiniteElement(family, domain, degree)

        # JIT compile and instantiate the UFC classes
        ufc_element, ufc_dofmap = jit(self.ufl_element)

        # Instantiate DOLFIN classes and finally the FunctionSpace
        dolfin_element = cpp.FiniteElement(ufc_element)
        dolfin_dofmap = cpp.DofMap(ufc_dofmap, mesh)
        cpp.FunctionSpace.__init__(self, mesh, dolfin_element, dolfin_dofmap)
```

*Constructing arguments (trial and test functions).* The `ufl.Argument` class (the base class of `ufl.TrialFunction` and `ufl.TestFunction`) is subclassed in the Python interface. Instead of using a `ufl.FiniteElement` to instantiate the classes, a DOLFIN `FunctionSpace` is used:

*Python code*

```
u = TrialFunction(V)
v = TestFunction(V)
```

The `ufl.Argument` base class is instantiated in the subclassed constructor by extracting the `ufl.FiniteElement` from the passed `FunctionSpace`, which is illustrated by the following pseudo code:

*Python code*

```
class Argument(ufl.Argument):
    def __init__(self, V, index=None):
        ufl.Argument.__init__(self, V.ufl_element, index)
        self.V = V
```

The `TrialFunction` and `TestFunction` are then defined using the subclassed `Argument` class:

*Python code*

```
def TrialFunction(V):
    return Argument(V, -1)

def TestFunction(V):
    return Argument(V, -2)
```

*Coefficients, functions and expressions.* When a UFL form is defined using a `Coefficient`, a user must associate with the form either a discrete finite element `Function` or a user-defined `Expression` before the form is assembled. In the C++ interface of DOLFIN, a user needs to explicitly carry out this association (`L.f = f`). In the Python interface of DOLFIN, the `ufl.Coefficient` class is combined

with the DOLFIN `Function` and `Expression` classes, and the association between the coefficient as a symbol in the form expression (`Coefficient`) and its value (`Function` or `Expression`) is automatic. A user can therefore assemble a form defined using instances of these combined classes directly:

*Python code*

```
class Source(Expression):
    def eval(self, values, x):
        values[0] = sin(x[0])

v = TestFunction(V)
f = Source()
L = f*v*dx
b = assemble(L)
```

The `Function` class in the Python interface inherits from both `ufl.Coefficient` and `cpp.Function`, as illustrated by the following pseudo code:

*Python code*

```
class Function(ufl.Coefficient, cpp.Function):
    def __init__(self, V):
        ufl.Coefficient.__init__(self, V.ufl_element)
        cpp.Function().__init__(self, V)
```

The actual constructor also includes logic to instantiate a `Function` from other objects. A more elaborate logic is also included to handle access to subfunctions.

A user-defined `Expression` can be created in two different ways: (i) as a pure Python `Expression`; or (ii) as a JIT compiled `Expression`. A pure Python `Expression` is an object instantiated from a subclass of `Expression` in Python. The `Source` class above is an example of this. Pseudo code for the constructor of the `Expression` class is similar to that for the `Function` class:

*Python code*

```
class Expression(ufl.Coefficient, cpp.Expression):
    def __init__(self, element=None):
        if element is None:
            element = auto_select_element(self.value_shape())
        ufl.Coefficient.__init__(self, element)
        cpp.Expression(element.value_shape())
```

If the `ufl.FiniteElement` is not defined by the user, DOLFIN will automatically choose an element using the `auto_select_element` function. This function takes the value shape of the `Expression` as argument. This has to be supplied by the user for vector- or tensor-valued `Expressions`, by overloading the `value_shape` method. The base class `cpp.Expression` is initialized using the value shape of the `ufl.FiniteElement`.

The actual code is considerably more complex than indicated above, as the same class, `Expression`, is used to handle both JIT compiled and pure Python `Expressions`. Also note that the actual subclass is eventually generated by a *metaclass* in Python, which makes it possible to include sanity checks for the declared subclass.

The `cpp.Expression` class is wrapped by a so-called *director class* in the SWIG-generated C++ layer. This means that the whole Python class is wrapped by a C++ subclass of `dolfin::Expression`. Each virtual method of the C++ base class is implemented by the SWIG-generated subclass in C++. These methods call the Python version of the method, which the user eventually implements by subclassing `cpp.Expression` in Python.

*Just-in-time compilation of expressions.* The performance of a pure Python `Expression` may be sub-optimal because of the callback from C++ to Python each time the `Expression` is evaluated. To circumvent this, a user can instead subclass the C++ version of `Expression` using a JIT compiled `Expression`. Because the subclass is implemented in C++, it will not involve any callbacks to Python, and can therefore be significantly faster than a pure Python `Expression`. A JIT compiled `Expression` is generated by passing a string of C++ code to the `Expression` constructor:

*Python code*

```
e = Expression("sin(x[0])")
```

The passed string is used to generate a subclass of `dolfin::Expression` in C++, where it is inlined into an overloaded `eval` method. The final code is JIT compiled and wrapped to Python using Instant (see Chapter 14). The generated Python class is then imported into Python. The class is not yet instantiated, as the final JIT compiled `Expression` also needs to inherit from `ufl.Coefficient`. To accomplish this, we dynamically create a class which inherits from both the generated class and `ufl.Coefficient`.

Classes in Python can be created during run-time by using the `type` function. The logic of creating a class and returning an instance of that class is handled in the `__new__` method of `dolfin.Expression`, as illustrated by the following pseudo code:

*Python code*

```
class Expression(object):
    def __new__(cls, cppcode=None):
        if cls.__name__ != "Expression":
            return object.__new__(cls)
        cpp_base = compile_expressions(cppcode)
        def __init__(self, cppcode):
            ...
        generated_class = type("CompiledExpression",
                               (Expression, ufl.Coefficient, cpp_base),
                               {"__init__": __init__})
        return generated_class()
```

The `__new__` method is called when a JIT compiled `Expression` is instantiated. However, it will also be called when a pure Python subclass of `Expression` is instantiated during initialization of the base-class. We handle the two different cases by checking the name of the instantiated class. If the name of the class is not `"Expression"`, then the call originates from the instantiation of a subclass of `Expression`. When a pure Python `Expression` is instantiated, like the `Source` instance in the code example above, the `__new__` method of `object` is called and the instantiated object is returned. In the other case, when a JIT compiled `Expression` is instantiated, we need to generate the JIT compiled base class from the passed Python string, as explained above. This is done by calling the function `compile_expressions`. Before `type` is called to generate the final class, an `__init__` method for the class is defined. This method initiates the new object by automatically selecting the element type and setting dimensions for the created `Expression`. This procedure is similar to what is done for the Python derived `Expression` class. Finally, we construct the new class which inherits the JIT compiled class and `ufl.Coefficient` by calling `type`.

The `type` function takes three arguments: the name of the class (`"CompiledExpression"`), the bases of the class (`Expression, ufl.Coefficient, cpp_base`), and a `dict` defining the interface (methods and attributes) of the class. The only new method or attribute we provide to the generated class is the `__init__` method. After the class is generated, we instantiate it and the object is returned to the user.

*Assembly of UFL forms.* The `assemble` function in the Python interface of DOLFIN enables a user to directly assemble a declared UFL form:

*Python code*

```
mesh = UnitSquare(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)
c = Expression("sin(x[0])")
a = c*dot(grad(u), grad(v))*dx
A = assemble(a)
```

The `assemble` function is a thin wrapper layer around the wrapped `cpp.assemble` function. The following pseudo code illustrates what happens in this layer:

*Python code*

```
def assemble(form, tensor=None, mesh=None):
    dolfin_form = Form(form)
    if tensor is None:
        tensor = create_tensor(dolfin_form.rank())
    if mesh is not None:
        dolfin_form.set_mesh(mesh)
    cpp.assemble(dolfin_form, tensor)
    return tensor
```

Here, `form` is a `ufl.Form`, which is used to generate a `dolfin.Form`, as explained below. In addition to the `form` argument, a user can choose to provide a tensor and/or a mesh. If a tensor is not provided, one will automatically be generated by the `create_tensor` function. The optional mesh is needed if the form does not contain any `Arguments`, or `Functions`; for example when a functional containing only `Expressions` is assembled. Note that the length of the above signature has been shortened. Other arguments to the `assemble` function exist but are skipped here for clarity.

The following pseudo code demonstrates what happens in the constructor of `dolfin.Form`, where the base class `cpp.Form` is initialized from a `ufl.Form`:

*Python code*

```
class Form(cpp.Form):
    def __init__(self, form):
        compiled_form, form_data = jit(form)
        function_spaces = extract_function_spaces(form_data)
        coefficients = extract_coefficients(form_data)
        cpp.Form.__init__(self, compiled_form, function_spaces, coefficients)
```

The `form` is first passed to the `dolfin.jit` function, which calls the registered form compiler to generate code and JIT compile it. There are presently two form compilers that can be chosen: `"ffc"` and `"sfc"` (see Chapters 11 and 15). Each one of these form compilers defines its own `jit` function, which eventually will receive the call. The form compiler can be chosen by setting:

*Python code*

```
parameters["form_compiler"]["name"] = "sfc"
```

The default form compiler is `"ffc"`. The `jit` function of the form compiler returns the JIT compiled `ufc::form` together with a `ufl.FormData` object. The latter is a data structure containing metadata

for the `ufl.form`, which is used to extract the function spaces and coefficients that are needed to instantiate a `cpp.Form`. The extraction of these data is handled by the `extract_function_spaces` and the `extract_coefficients` functions.

### *10.4.4 NumPy and SciPy integration*

The values of the `Matrix` and `Vector` classes in the Python interface of DOLFIN can be viewed as NumPy arrays. This is done by calling the `array` method of the vector or matrix:

*Python code*

```
A  = assemble(a)
AA = A.array()
```

Here, `A` is a matrix assembled from the form `a`. The NumPy array `AA` is a dense structure and all values are copied from the original data. The `array` function can be called on a distributed matrix or vector, in which case it will return the locally stored values.

*Direct access to linear algebra data.* Direct access to the underlying data is possible for the uBLAS and MTL4 linear algebra backends. A NumPy array view into the data will be returned by the method `data`:

*Python code*

```
parameters["linear_algebra_backend"] = "uBLAS"
b = assemble(L)
bb = b.data()
```

Here, `b` is a uBLAS vector and `bb` is a NumPy view into the data of `b`. Any changes to `bb` will directly affect `b`. A similar method exists for matrices:

*Python code*

```
parameters["linear_algebra_backend"] = "MTL4"
A = assemble(a)
rows, columns, values = A.data()
```

The data is returned in a compressed row storage format as the three NumPy arrays `rows`, `columns` and `values`. These are also views of the data that represent `A`. Any changes in `values` will directly result in a corresponding change in `A`.

*Sparse matrix and SciPy integration.* The `rows`, `columns` and `values` data structures can be used to instantiate a `csr_matrix` from the `scipy.sparse` module (Jones et al., 2009):

*Python code*

```
from scipy.sparse import csr_matrix
rows, columns, values = A.data()
csr = csr_matrix((values, columns, rows))
```

The `csr_matrix` can then be used with other Python modules that support sparse matrices, such as the `scipy.sparse` module and `pyamg`, which is an algebraic multigrid solver (Bell et al., 2011).

*Slicing vectors.* NumPy provides a convenient slicing interface for NumPy arrays. The Python interface of DOLFIN also provides such an interface for vectors (see Chapter 19 for details of the implementation). A slice can be used to access and set data in a vector:

*Python code*

```
# Create copy of vector
b_copy = b[:]

# Slice assignment (c can be a scalar, a DOLFIN vector or a NumPy array)
b[:] = c

# Set negative values to zero
b[b < 0] = 0

# Extract every second value
b2 = b[::2]
```

A difference between a NumPy slice and a slice of a DOLFIN vector is that a slice of a NumPy array provides a view into the original array, whereas in DOLFIN we provide a copy. A list/tuple of integers or a NumPy array can also be used to both access and set data in a vector:

*Python code*

```
b1 = b[[0, 4, 7, 10]]
b2 = b[array((0, 4, 7, 10))]
```

## 10.5 Historical notes

The first public version of DOLFIN, version 0.2.0, was released in 2002. At that time, DOLFIN was a self-contained C++ library with minimal external dependencies. All functionality was then implemented as part of DOLFIN itself, including linear algebra and finite element form evaluation. Although only piecewise linear elements were supported, DOLFIN provided rudimentary automated finite element assembly of variational forms. The form language was implemented by C++ operator overloading. For an overview of the development of the FEniCS form language and an example of the early form language implemented in DOLFIN, see Chapter 11.

Later, parts of the functionality of DOLFIN have been moved to either external libraries or other FEniCS components. In 2003, the FEniCS project was born and shortly after, with the release of version 0.5.0 in 2004, the form evaluation system in DOLFIN was replaced by an automated code generation system based on FFC and FIAT. In the following year, the linear algebra was replaced by wrappers for PETSc data structures and solvers. At this time, the DOLFIN Python interface (PyDOLFIN) was introduced. Since then, the Python interface has developed from a simple auto-generated wrapper layer for the DOLFIN C++ functionality to a mature problem-solving environment with support for just-in-time compilation of variational forms and integration with external Python modules like NumPy.

In 2006, the DOLFIN mesh data structures were simplified and reimplemented to improve efficiency and expand functionality. The new data structures were based on a light-weight object-oriented layer on top of an underlying data storage by plain contiguous C/C++ arrays and improved the efficiency by orders of magnitude over the old implementation, which was based on a fully object-oriented implementation with local storage of all mesh entities like cells and vertices. The first release of DOLFIN with the new mesh library was version 0.6.2.

In 2007, the UFC interface was introduced and the FFC form language was integrated with the DOLFIN Python interface. Just-in-time compilation was also introduced. The following year, the linear algebra interfaces of DOLFIN were redesigned to allow flexible handling of multiple linear algebra backends. In 2009, a major milestone was reached when parallel computing was introduced in DOLFIN.

Over the years, DOLFIN has undergone a large number of changes to its design, interface and implementation. However, since the release of DOLFIN 0.9.0, which introduced a redesign of the DOLFIN function classes based on the new function space abstraction, only minor changes have been made to the interface. Since the release of version 0.9.0, most work has gone into refining the interface, implementing missing functionality, fixing bugs and improving documentation, in anticipation of the first stable release of DOLFIN, version 1.0.

http://www.springer.com/978-3-642-23098-1

Automated Solution of Differential Equations by the Finite Element Method
The FEniCS Book
(Eds.)A. Logg; K.-A. Mardal; G. Wells
2012, XIII, 723 p. 346 illus., 52 in color., Hardcover
ISBN: 978-3-642-23098-1