# The FEniCS Manual

Excerpts from the FEniCS Book

With contributions from

Martin Sandve Alnæs

Johan Hake

Robert C. Kirby

Hans Petter Langtangen

Anders Logg

Garth N. Wells


Version October 31, 2011

# *Contents*

# *Preface*

The FEniCS Project set out in 2003 with an idea to automate the solution of mathematical models based on differential equations. Initially, the FEniCS Project consisted of two libraries: DOLFIN and FIAT. Since then, the project has grown and now consists of the core components DOLFIN, FFC, FIAT, Instant, UFC and UFL. Other FEniCS components and applications described in this book are SyFi/SFC, FErari, ASCoT, Unicorn, CBC.Block, CBC.RANS, CBC.Solve and DOLFWAVE.

This book is written by researchers and developers behind the FEniCS Project. The presentation spans mathematical background, software design and the use of FEniCS in applications. The mathematical framework is outlined in Part I, the implementation of central components is described in Part II, while Part III concerns a wide range of applications. New users of FEniCS may find the tutorial included as the opening chapter particularly useful.

Feedback on this book is welcome, and can be given at `https://launchpad.net/fenics-book`. Use the Launchpad system to file bug reports if you find errors in the text. For more information about the FEniCS Project, access to the software presented in this book, documentation, articles and presentations, visit the FEniCS Project web site at `http://fenicsproject.org`. Some of the chapters in this book are accompanied by supplementary material in the form of code examples. These code examples can be downloaded from `http://fenicsproject.org/book/`.

Anders Logg, Kent-Andre Mardal and Garth N. Wells
*Oslo and Cambridge, October 2011*

*This document ("The FEniCS Manual") contains excerpts from the book "Automated Solution of Differential Equations by the Finite Element Method" ("The FEniCS Book"). If you like this manual, buy the book.*

# 1 A FEniCS tutorial

By Hans Petter Langtangen

This chapter presents a FEniCS tutorial to get new users quickly up and running with solving differential equations. FEniCS can be programmed both in C++ and Python, but this tutorial focuses exclusively on Python programming since this is the simplest approach to exploring FEniCS for beginners and it does not compromise on performance. After having digested the examples in this tutorial, the reader should be able to learn more from the FEniCS documentation and from the other chapters in this book.

## 1.1 Fundamentals

FEniCS is a user-friendly tool for solving partial differential equations (PDEs). The goal of this tutorial is to get you started with FEniCS through a series of simple examples that demonstrate

- how to define the PDE problem in terms of a variational problem,
- how to define simple domains,
- how to deal with Dirichlet, Neumann, and Robin conditions,
- how to deal with variable coefficients,
- how to deal with domains built of several materials (subdomains),
- how to compute derived quantities like the flux vector field or a functional of the solution,
- how to quickly visualize the mesh, the solution, the flux, etc.,
- how to solve nonlinear PDEs in various ways,
- how to deal with time-dependent PDEs,
- how to set parameters governing solution methods for linear systems,
- how to create domains of more complex shape.

The mathematics of the illustrations is kept simple to better focus on FEniCS functionality and syntax. This means that we mostly use the Poisson equation and the time-dependent diffusion equation as model problems, often with input data adjusted such that we get a very simple solution that can be exactly reproduced by any standard finite element method over a uniform, structured mesh. This latter property greatly simplifies the verification of the implementations. Occasionally we insert a physically more relevant example to remind the reader that changing the PDE and boundary conditions to something more real might often be a trivial task.

FEniCS may seem to require a thorough understanding of the abstract mathematical version of the finite element method as well as familiarity with the Python programming language. Nevertheless, it turns out that many are able to pick up the fundamentals of finite elements *and* Python programming as they go along with this tutorial. Simply keep on reading and try out the examples. You will be amazed of how easy it is to solve PDEs with FEniCS!

Reading this tutorial obviously requires access to a machine where the FEniCS software is installed. Section 1.7.5 explains briefly how to install the necessary tools. All the examples discussed in the following are available as executable Python source code files in a directory tree.

### 1.1.1 The Poisson equation

Our first example regards the Poisson problem,

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_0 \quad \text{on } \partial\Omega. \end{aligned} \tag{1.1}$$

Here, $u = u(x)$ is the unknown function, $f = f(x)$ is a prescribed function, $\Delta$ is the Laplace operator (also often written as $\nabla^2$), $\Omega$ is the spatial domain, and $\partial\Omega$ is the boundary of $\Omega$. A stationary PDE like this, together with a complete set of boundary conditions, constitute a *boundary-value problem*, which must be precisely stated before it makes sense to start solving it with FEniCS.

In two space dimensions with coordinates $x$ and $y$, we can write out the Poisson equation (1.1) as

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y). \tag{1.2}$$

The unknown $u$ is now a function of two variables, $u(x, y)$, defined over a two-dimensional domain $\Omega$.

The Poisson equation (1.1) arises in numerous physical contexts, including heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, and water waves. Moreover, the equation appears in numerical splitting strategies of more complicated systems of PDEs, in particular the Navier–Stokes equations.

Solving a physical problem with FEniCS consists of the following steps:

1. Identify the PDE and its boundary conditions.
2. Reformulate the PDE problem as a variational problem.
3. Make a Python program where the formulas in the variational problem are coded, along with definitions of input data such as $f$, $u_0$, and a mesh for $\Omega$ in (1.1).
4. Add statements in the program for solving the variational problem, computing derived quantities such as $\nabla u$, and visualizing the results.

We shall now go through steps 2–4 in detail. The key feature of FEniCS is that steps 3 and 4 result in fairly short code, while most other software frameworks for PDEs require much more code and more technically difficult programming.

### 1.1.2 Variational formulation

FEniCS makes it easy to solve PDEs if finite elements are used for discretization in space and the problem is expressed as a *variational problem*. Readers who are not familiar with variational problems will get a brief introduction to the topic in this tutorial, and in the forthcoming chapter, but getting

and reading a proper book on the finite element method in addition is encouraged. Section 1.7.6 contains a list of some suitable books.

The core of the recipe for turning a PDE into a variational problem is to multiply the PDE by a function $v$, integrate the resulting equation over $\Omega$, and perform integration by parts of terms with second-order derivatives. The function $v$ which multiplies the PDE is in the mathematical finite element literature called a *test function*. The unknown function $u$ to be approximated is referred to as a *trial function*. The terms test and trial function are used in FEniCS programs too. Suitable function spaces must be specified for the test and trial functions. For standard PDEs arising in physics and mechanics such spaces are well known.

In the present case, we first multiply the Poisson equation by the test function $v$ and integrate:

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x. \tag{1.3}$$

Then we apply integration by parts to the integrand with second-order derivatives:

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s, \tag{1.4}$$

where $\partial u/\partial n$ is the derivative of $u$ in the outward normal direction on the boundary. The test function $v$ is required to vanish on the parts of the boundary where $u$ is known, which in the present problem implies that $v = 0$ on the whole boundary $\partial\Omega$. The second term on the right-hand side of (1.4) therefore vanishes. From (1.3) and (1.4) it follows that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x. \tag{1.5}$$

This equation is supposed to hold for all $v$ in some function space $\hat{V}$. The trial function $u$ lies in some (possibly different) function space $V$. We refer to (1.5) as the *weak form* of the original boundary-value problem (1.1).

The proper statement of our variational problem now goes as follows: find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \quad \forall\, v \in \hat{V}. \tag{1.6}$$

The trial and test spaces $V$ and $\hat{V}$ are in the present problem defined as

$$\begin{aligned} V &= \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}, \\ \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}. \end{aligned} \tag{1.7}$$

In short, $H^1(\Omega)$ is the mathematically well-known Sobolev space containing functions $v$ such that $v^2$ and $|\nabla v|^2$ have finite integrals over $\Omega$. The solution of the underlying PDE must lie in a function space where also the derivatives are continuous, but the Sobolev space $H^1(\Omega)$ allows functions with discontinuous derivatives. This weaker continuity requirement of $u$ in the variational statement (1.6), caused by the integration by parts, has great practical consequences when it comes to constructing finite elements.

To solve the Poisson equation numerically, we need to transform the continuous variational problem (1.6) to a discrete variational problem. This is done by introducing *finite-dimensional* test and trial spaces, often denoted as $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$. The discrete variational problem reads: find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{1.8}$$

The choice of $V_h$ and $\hat{V}_h$ follows directly from the kind of finite elements we want to apply in our problem. For example, choosing the well-known linear triangular element with three nodes implies that $V_h$ and $\hat{V}_h$ are the spaces of all piecewise linear functions over a mesh of triangles, where the functions in $\hat{V}_h$ are zero on the boundary and those in $V_h$ equal $u_0$ on the boundary.

The mathematics literature on variational problems writes $u_h$ for the solution of the discrete problem and $u$ for the solution of the continuous problem. To obtain (almost) a one-to-one relationship between the mathematical formulation of a problem and the corresponding FEniCS program, we shall use $u$ for the solution of the discrete problem and $u_e$ for the exact solution of the continuous problem, *if* we need to explicitly distinguish between the two. In most cases, we will introduce the PDE problem with $u$ as unknown, derive a variational equation $a(u,v) = L(v)$ with $u \in V$ and $v \in \hat{V}$, and then simply discretize the problem by saying that we choose finite-dimensional spaces for $V$ and $\hat{V}$. This restriction of $V$ implies that $u$ becomes a discrete finite element function. In practice this means that we turn our PDE problem into a continuous variational problem, create a mesh and specify an element type, and then let $V$ correspond to this mesh and element choice. Depending upon whether $V$ is infinite- or finite-dimensional, $u$ will be the exact or approximate solution.

It turns out to be convenient to introduce a unified notation for a linear weak form like (1.8):

$$a(u,v) = L(v). \tag{1.9}$$

In the present problem we have that

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.10}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x. \tag{1.11}$$

From the mathematics literature, $a(u,v)$ is known as a *bilinear form* and $L(v)$ as a *linear form*. We shall in every linear problem we solve identify the terms with the unknown $u$ and collect them in $a(u,v)$, and similarly collect all terms with only known functions in $L(v)$. The formulas for $a$ and $L$ are then coded directly in the program.

To summarize, before making a FEniCS program for solving a PDE, we must first perform two steps:

1. Turn the PDE problem into a discrete variational problem: find $u \in V$ such that

$$a(u,v) = L(v) \quad \forall\, v \in \hat{V}. \tag{1.12}$$

2. Specify the choice of spaces ($V$ and $\hat{V}$), which means specifying the mesh and type of finite elements.

### 1.1.3 Implementation

The test problem so far has a general domain $\Omega$ and general functions $u_0$ and $f$. For our first implementation we must decide on specific choices of $\Omega$, $u_0$, and $f$. It will be wise to construct a specific problem where we can easily check that the computed solution is correct. Let us start with specifying an exact solution

$$u_{\mathrm{e}}(x,y) = 1 + x^2 + 2y^2 \tag{1.13}$$

on some 2D domain. By inserting (1.13) in our Poisson problem, we find that $u_{\mathrm{e}}(x,y)$ is a solution if

$$f(x,y) = -6, \quad u_0(x,y) = u_{\mathrm{e}}(x,y) = 1 + x^2 + 2y^2,$$

regardless of the shape of the domain. We choose here, for simplicity, the domain to be the unit square,

$$\Omega = [0,1] \times [0,1].$$

The reason for specifying the solution (1.13) is that the finite element method, with a rectangular domain uniformly partitioned into linear triangular elements, will exactly reproduce a second-order polynomial at the vertices of the cells, regardless of the size of the elements. This property allows us to verify the implementation by comparing the computed solution, called $u$ in this document (except when setting up the PDE problem), with the exact solution, denoted by $u_e$: $u$ should equal $u$ to machine precision *at the nodes*. Test problems with this property will be frequently constructed throughout this tutorial.

A FEniCS program for solving the Poisson equation in 2D with the given choices of $u_0$, $f$, and $\Omega$ may look as follows:

*Python code*

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquare(6, 4)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define boundary conditions
u0 = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]")

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u)
plot(mesh)

# Dump solution to file in VTK format
file = File("poisson.pvd")
file << u

# Hold plot
interactive()
```

The complete code can be found in the file `d1_p2D.py` in the directory `stationary/poisson`.

We shall now dissect this FEniCS program in detail. The program is written in the Python programming language. You may either take a quick look at a Python tutorial [The Python Tutorial] to pick up the basics of Python if you are unfamiliar with the language, or you may learn enough Python as you go along with the examples in the present tutorial. The latter strategy has proven to work for many newcomers to FEniCS. Section 1.7.7 lists some relevant Python books.

The listed FEniCS program defines a finite element mesh, the discrete function spaces $V$ and $\hat{V}$ corresponding to this mesh and the element type, boundary conditions for $u$ (the function $u_0$), $a(u, v)$, and $L(v)$. Thereafter, the unknown trial function $u$ is computed. Then we can investigate $u$ visually or analyze the computed values.

The first line in the program,

*Python code*

```
from dolfin import *
```

imports the key classes `UnitSquare`, `FunctionSpace`, `Function`, and so forth, from the DOLFIN library. All FEniCS programs for solving PDEs by the finite element method normally start with this line. DOLFIN is a software library with efficient and convenient C++ classes for finite element computing, and `dolfin` is a Python package providing access to this C++ library from Python programs. You can think of FEniCS as an umbrella, or project name, for a set of computational components, where DOLFIN is one important component for writing finite element programs. The `dolfin` package applies other components in the FEniCS suite under the hood, but newcomers to FEniCS programming do not need to care about this.

The statement

*Python code*

```
mesh = UnitSquare(6, 4)
```

defines a uniform finite element mesh over the unit square $[0, 1] \times [0, 1]$. The mesh consists of *cells*, which are triangles with straight sides. The parameters 6 and 4 tell that the square is first divided into $6 \times 4$ rectangles, and then each rectangle is divided into two triangles. The total number of triangles then becomes 48. The total number of vertices in this mesh is $7 \cdot 5 = 35$. DOLFIN offers some classes for creating meshes over very simple geometries. For domains of more complicated shape one needs to use a separate *preprocessor* program to create the mesh (see Section 1.4). The FEniCS program will then read the mesh from file.

Having a mesh, we can define a discrete function space `V` over this mesh:

*Python code*

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

The second argument reflects the type of element, while the third argument is the degree of the basis functions on the element. The type of element is here `"Lagrange"`, implying the standard Lagrange family of elements (some FEniCS programs use `"CG"`, for Continuous Galerkin, as a synonym for `"Lagrange"`). With degree 1, we simply get the standard linear Lagrange element, which is a triangle with nodes at the three vertices. Some finite element practitioners refer to this element as the "linear triangle". The computed $u$ will be continuous and linearly varying in $x$ and $y$ over each cell in the mesh. Higher-degree polynomial approximations over each cell are trivially obtained by increasing the third parameter in `FunctionSpace`. Changing the second parameter to `"DG"` creates a function space for discontinuous Galerkin methods.

In mathematics, we distinguish between the trial and test spaces $V$ and $\hat{V}$. The only difference in the present problem is the boundary conditions. In FEniCS we do not specify the boundary conditions as part of the function space, so it is sufficient to work with one common space `V` for the test and trial functions in the program:

*Python code*

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

The next step is to specify the boundary condition: $u = u_0$ on $\partial\Omega$. This is done by

*Python code*

```
bc = DirichletBC(V, u0, u0_boundary)
```

where `u0` is an instance holding the $u_0$ values, and `u0_boundary` is a function (or object) describing whether a point lies on the boundary where $u$ is specified.

Boundary conditions of the type $u = u_0$ are known as *Dirichlet conditions*, and also as *essential boundary conditions* in a finite element context. Naturally, the name of the DOLFIN class holding the information about Dirichlet boundary conditions is `DirichletBC`.

The `u0` variable refers to an `Expression` object, which is used to represent a mathematical function. The typical construction is

*Python code*

```
u0 = Expression(formula)
```

where `formula` is a string containing the mathematical expression. This formula is written with C++ syntax (the expression is automatically turned into an efficient, compiled C++ function, see Section 1.7.3 and Chapter 3 for details on the syntax). The independent variables in the function expression are supposed to be available as a point vector `x`, where the first element `x[0]` corresponds to the $x$ coordinate, the second element `x[1]` to the $y$ coordinate, and (in a three-dimensional problem) `x[2]` to the $z$ coordinate. With our choice of $u_0(x, y) = 1 + x^2 + 2y^2$, the formula string must be written as `1 + x[0]*x[0] + 2*x[1]*x[1]`:

*Python code*

```
u0 = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]")
```

The information about where to apply the `u0` function as boundary condition is coded in a function `u0_boundary`:

*Python code*

```
def u0_boundary(x, on_boundary):
    return on_boundary
```

A function like `u0_boundary` for marking the boundary must return a boolean value: `True` if the given point `x` lies on the Dirichlet boundary and `False` otherwise. The argument `on_boundary` is supplied by DOLFIN and equals `True` if `x` is on the physical boundary of the mesh. In the present case, where we are supposed to return `True` for all points on the boundary, we can just return the supplied value of `on_boundary`. The `u0_boundary` function will be called for every discrete point in the mesh, which allows us to have boundaries where $u$ are known also inside the domain, if desired.

One can also omit the `on_boundary` argument, but in that case we need to test on the value of the coordinates in `x`:

*Python code*

```
def u0_boundary(x):
    return x[0] == 0 or x[1] == 0 or x[0] == 1 or x[1] == 1
```

As for the formula in `Expression` objects, `x` in the `u0_boundary` function represents a point in space with coordinates `x[0]`, `x[1]`, etc. Comparing floating-point values using an exact match test with `==` is not good programming practice, because small round-off errors in the computations of the `x` values could make a test `x[0] == 1` become false even though `x` lies on the boundary. A better test is to check for equality with a tolerance:

*Python code*

```
def u0_boundary(x):
    tol = 1E-15
    return abs(x[0]) < tol or \
           abs(x[1]) < tol or \
           abs(x[0] - 1) < tol or \
           abs(x[1] - 1) < tol
```

Before defining $a(u,v)$ and $L(v)$ we have to specify the $f$ function:

*Python code*

```
f = Expression("-6")
```

When $f$ is constant over the domain, `f` can be more efficiently represented as a `Constant` object:

*Python code*

```
f = Constant(-6.0)
```

Now we have all the objects we need in order to specify this problem's $a(u,v)$ and $L(v)$:

*Python code*

```
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx
```

In essence, these two lines specify the PDE to be solved. Note the very close correspondence between the Python syntax and the mathematical formulas $\nabla u \cdot \nabla v \,\mathrm{d}x$ and $fv\,\mathrm{d}x$. This is a key strength of FEniCS: the formulas in the variational formulation translate directly to very similar Python code, a feature that makes it easy to specify PDE problems with lots of PDEs and complicated terms in the equations. The language used to express weak forms is called UFL (Unified Form Language) and is an integral part of FEniCS.

Instead of `nabla_grad` we could also just have written `grad` in the examples in this tutorial. However, when taking gradients of vector fields, `grad` and `nabla_grad` differ. The latter is consistent with the tensor algebra commonly used to derive vector and tensor PDEs, where $\nabla$ acts as a vector operator, and therefore this author prefers to always use `nabla_grad`.

Having `a` and `L` defined, and information about essential (Dirichlet) boundary conditions in `bc`, we can compute the solution, a finite element function `u`, by

*Python code*

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

Some prefer to replace `a` and `L` by an `equation` variable, which is accomplished by this equivalent code:

*Python code*

```
equation = inner(nabla_grad(u), nabla_grad(v))*dx == f*v*dx
u = Function(V)
solve(equation, u, bc)
```

Note that we first defined the variable `u` as a `TrialFunction` and used it to represent the unknown in the form `a`. Thereafter, we redefined `u` to be a `Function` object representing the solution; that is, the computed finite element function $u$. This redefinition of the variable `u` is possible in Python and often done in FEniCS applications. The two types of objects that `u` refers to are equal from a mathematical point of view, and hence it is natural to use the same variable name for both objects. In a program,

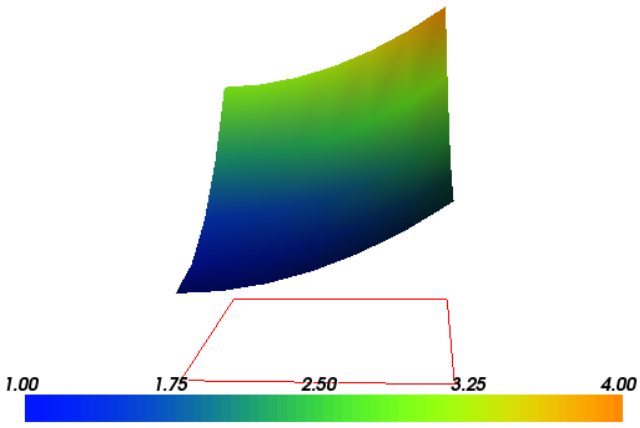



Figure 1.1: Plot of the solution in the first FEniCS example. (A bounding box around the mesh is added by pressing `o` in the plot window, and the mouse buttons are then used to rotate and move the plot, see Section 1.1.8.)

however, `TrialFunction` objects must always be used for the unknowns in the problem specification (the form `a`), while `Function` objects must be used for quantities that are computed (known).

The simplest way of quickly looking at `u` and the mesh is to say

*Python code*

```
plot(u)
plot(mesh)
interactive()
```

The `interactive()` call is necessary for the plot to remain on the screen. With the left, middle, and right mouse buttons you can rotate, translate, and zoom (respectively) the plotted surface to better examine what the solution looks like. Figures 1.1 and 1.2 display the resulting $u$ function and the finite element mesh, respectively.

It is also possible to dump the computed solution to file, e.g., in the VTK format:

*Python code*

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

The `poisson.pvd` file can now be loaded into any front-end to VTK, say ParaView or VisIt. The `plot` function is intended for quick examination of the solution during program development. More in-depth visual investigations of finite element solutions will normally benefit from using highly professional tools such as ParaView and VisIt.

The next three sections deal with some technicalities about specifying the solution method for linear systems (so that you can solve large problems) and examining array data from the computed solution (so that you can check that the program is correct). These technicalities are scattered around in forthcoming programs. However, the impatient reader who is more interested in seeing the previous program being adapted to a real physical problem, and play around with some interesting visualizations, can safely jump to Section 1.1.7. Information in the intermediate sections can be studied on demand.

### *1.1.4 Controlling the solution process*

Sparse LU decomposition (Gaussian elimination) is used by default to solve linear systems of equations in FEniCS programs. This is a very robust and recommended method for a few thousand unknowns

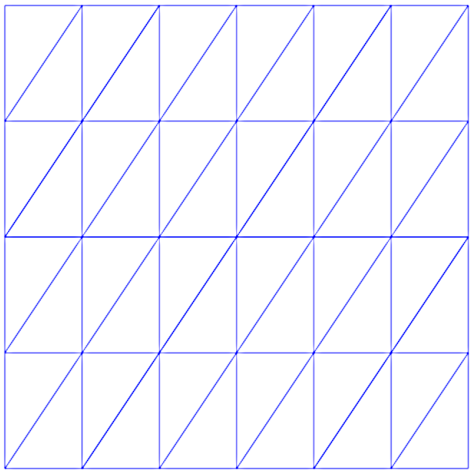


Figure 1.2: Plot of the mesh in the first FEniCS example.

in the equation system, and may hence be the method of choice in many 2D and smaller 3D problems. However, sparse LU decomposition becomes slow and memory demanding in large problems. This fact forces the use of iterative methods, which are faster and require much less memory.

Preconditioned Krylov solvers is a type of popular iterative methods that are easily accessible in FEniCS programs. The Poisson equation results in a symmetric, positive definite coefficient matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method. Incomplete LU factorization (ILU) is a popular and robust all-round preconditioner, so let us try the CG–ILU pair:

*Python code*

```
solve(a == L, u, bc)
      solver_parameters={"linear_solver": "cg",
                         "preconditioner": "ilu"})
# Alternative syntax
solve(a == L, u, bc,
      solver_parameters=dict(linear_solver="cg",
                             preconditioner="ilu"))
```

Section 1.7.4 lists the most popular choices of Krylov solvers and preconditioners available in FEniCS.

The actual CG and ILU implementations that are brought into action depends on the choice of linear algebra package. FEniCS interfaces several linear algebra packages, called *linear algebra backends* in FEniCS terminology. PETSc is the default choice if DOLFIN is compiled with PETSc, otherwise uBLAS. Epetra (Trilinos) and MTL4 are two other supported backends. Which backend to apply can be controlled by setting

*Python code*

```
parameters["linear_algebra_backend"] = backendname
```

where `backendname` is a string, either `"PETSc"`, `"uBLAS"`, `"Epetra"`, or `"MTL4"`. All these backends offer high-quality implementations of both iterative and direct solvers for linear systems of equations.

A common platform for FEniCS users is Ubuntu Linux. The FEniCS distribution for Ubuntu contains PETSc, making this package the default linear algebra backend. The default solver is sparse LU decomposition (`"lu"`), and the actual software that is called is then the sparse LU solver from UMFPACK (which PETSc has an interface to).

We will normally like to control the tolerance in the stopping criterion and the maximum number of iterations when running an iterative method. Such parameters can be set by accessing the *global*

*parameter database*, which is called `parameters` and behaves as a nested dictionary. Write

*Python code*

```
info(parameters, True)
```

to list all parameters and their default values in the database. The nesting of parameter sets is indicated through indentation in the output from `info`. According to this output, the relevant parameter set is named `"krylov_solver"`, and the parameters are set like this:

*Python code*

```
prm = parameters["krylov_solver"] # short form
prm["absolute_tolerance"] = 1E-10
prm["relative_tolerance"] = 1E-6
prm["maximum_iterations"] = 1000
```

Stopping criteria for Krylov solvers usually involve the norm of the residual, which must be smaller than the absolute tolerance parameter and smaller than the relative tolerance parameter times the initial residual.

To see the number of actual iterations to reach the stopping criterion, we can insert

*Python code*

```
set_log_level(PROGRESS)
# or
set_log_level(DEBUG)
```

A message with the equation system size, solver type, and number of iterations arises from specifying the argument `PROGRESS`, while `DEBUG` results in more information, including CPU time spent in the various parts of the matrix assembly and solve process.

The complete solution process with control of the solver parameters now contains the statements

*Python code*

```
prm = parameters["krylov_solver"] # short form
prm["absolute_tolerance"] = 1E-10
prm["relative_tolerance"] = 1E-6
prm["maximum_iterations"] = 1000
set_log_level(PROGRESS)

solve(a == L, u, bc,
      solver_parameters={"linear_solver": "cg",
                         "preconditioner": "ilu"})
```

The demo program `d2_p2D.py` in the `stationary/poisson` directory incorporates the above shown control of the linear solver and precnditioner, but is otherwise similar to the previous `d1_p2D.py` program.

We remark that default values for the global parameter database can be defined in an XML file, see the example file `dolfin_parameters.xml` in the directory `stationary/poisson`. If such a file is found in the directory where a FEniCS program is run, this file is read and used to initialize the `parameters` object. Otherwise, the file `.config/fenics/dolfin_parameters.xml` in the user's home directory is read, if it exists. The XML file can also be in gzip'ed form with the extension `.xml.gz`.

### *1.1.5 Linear variational problem and solver objects*

The `solve(a == L, u, bc)` call is just a compact syntax alternative to a slightly more comprehensive specification of the variational equation and the solution of the associated linear system. This

alternative syntax is used in a lot of FEniCS applications and will also be used later in this tutorial, so we show it already now:

*Python code*

```
u = Function(V)
problem = LinearVariationalProblem(a, L, u, bc)
solver  = LinearVariationalSolver(problem)
solver.solve()
```

Many objects have an attribute `parameters` corresponding to a parameter set in the global `parameters` database, but local to the object. Here, `solver.parameters` play that role. Setting the CG method with ILU preconditiong as solution method and specifying solver-specific parameters can be done like this:

*Python code*

```
solver.parameters["linear_solver"] = "cg"
solver.parameters["preconditioner"] = "ilu"
cg_prm = solver.parameters["krylov_solver"] # short form
cg_prm["absolute_tolerance"] = 1E-7
cg_prm["relative_tolerance"] = 1E-4
cg_prm["maximum_iterations"] = 1000
```

Calling `info(solver.parameters, True)` lists all the available parameter sets with default values for each parameter. Settings in the global `parameters` database are propagated to parameter sets in individual objects, with the possibility of being overwritten as done above.

The `d3_p2D.py` program modifies the `d2_p2D.py` file to incorporate objects for the variational problem and solver.

### 1.1.6 Examining the discrete solution

We know that, in the particular boundary-value problem of Section 1.1.3, the computed solution $u$ should equal the exact solution at the vertices of the cells. An important extension of our first program is therefore to examine the computed values of the solution, which is the focus of the present section.

A finite element function like $u$ is expressed as a linear combination of basis functions $\phi_j$, spanning the space $V$:

$$\sum_{j=1}^{N} U_j \phi_j. \tag{1.14}$$

By writing `solve(a == L, u, bc)` in the program, a linear system will be formed from $a$ and $L$, and this system is solved for the $U_1, \ldots, U_N$ values. The $U_1, \ldots, U_N$ values are known as *degrees of freedom* of $u$. For Lagrange elements (and many other element types) $U_k$ is simply the value of $u$ at the node with global number $k$. (The nodes and cell vertices coincide for linear Lagrange elements, while for higher-order elements there may be additional nodes at the facets and in the interior of cells.)

Having `u` represented as a `Function` object, we can either evaluate `u(x)` at any vertex `x` in the mesh, or we can grab all the values $U_j$ directly by

*Python code*

```
u_nodal_values = u.vector()
```

The result is a DOLFIN `Vector` object, which is basically an encapsulation of the vector object used in the linear algebra package that is used to solve the linear system arising from the variational problem. Since we program in Python it is convenient to convert the `Vector` object to a standard `numpy` array for further processing:

*Python code*

```
u_array = u_nodal_values.array()
```

With `numpy` arrays we can write "MATLAB-like" code to analyze the data. Indexing is done with square brackets: `u_array[i]`, where the index `i` always starts at `0`.

Mesh information can be gathered from the `mesh` object, e.g.,

- `mesh.coordinates()` returns the coordinates of the vertices as an $M \times d$ `numpy` array, $M$ being the number of vertices in the mesh and $d$ being the number of space dimensions,
- `mesh.num_cells()` returns the number of cells (triangles) in the mesh,
- `mesh.num_vertices()` returns the number of vertices in the mesh (with our choice of linear Lagrange elements this equals the number of nodes),
- `str(mesh)` returns a short "pretty print" description of the mesh, e.g.,

*Output*

```
<Mesh of topological dimension 2 (triangles) with
16 vertices and 18 cells, ordered>
```

and `print mesh` is actually the same as `print str(mesh)`.

All mesh objects are of type `Mesh` so typing the command `pydoc dolfin.Mesh` in a terminal window will give a list of methods[1] that can be called through any `Mesh` object. In fact, `pydoc dolfin.X` shows the documentation of any DOLFIN name `X`.

Writing out the solution on the screen can now be done by a simple loop:

*Python code*

```
coor = mesh.coordinates()
if mesh.num_vertices() == len(u_array):
    for i in range(mesh.num_vertices()):
        print 'u(%8g,%8g) = %g' % (coor[i][0], coor[i][1], u_array[i])
```

The beginning of the output looks like this:

*Output*

```
u(       0,       0) = 1
u(0.166667,       0) = 1.02778
u(0.333333,       0) = 1.11111
u(     0.5,       0) = 1.25
u(0.666667,       0) = 1.44444
u(0.833333,       0) = 1.69444
u(       1,       0) = 2
```

For Lagrange elements of degree higher than one, the vertices do not correspond to all the nodal points and the `if`-test fails.

For verification purposes we want to compare the values of the computed `u` at the nodes (given by `u_array`) with the exact solution `u0` evaluated at the nodes. The difference between the computed and exact solution should be less than a small tolerance at all the nodes. The `Expression` object `u0` can be evaluated at any point `x` by calling `u0(x)`. Specifically, `u0(coor[i])` returns the value of `u0` at the vertex or node with global number `i`. Alternatively, we can make a finite element field `u_e`, representing the exact solution, whose values at the nodes are given by the `u0` function. With

[1] A method in Python (and other languages supporting the class construct) is simply a function in a class.

mathematics, $u_e = \sum_{j=1}^{N} E_j \phi_j$, where $E_j = u_0(x_j, y_j)$, $(x_j, y_j)$ being the coordinates of node number $j$. This process is known as interpolation. FEniCS has a function for performing the operation:

*Python code*

```
u_e = interpolate(u0, V)
```

The maximum error can now be computed as

*Python code*

```
u_e_array = u_e.vector().array()
print "Max error:", numpy.abs(u_e_array - u_array).max()
```

The value of the error should be at the level of the machine precision ($10^{-16}$).

To demonstrate the use of point evaluations of `Function` objects, we write out the computed `u` at the center point of the domain and compare it with the exact solution:

*Python code*

```
center = (0.5, 0.5)
print "numerical u at the center point:",  u(center)
print "exact     u at the center point:", u0(center)
```

Trying a $3 \times 3$ mesh, the output from the previous snippet becomes

*Output*

```
numerical u at the center point: [ 1.83333333]
exact     u at the center point: [ 1.75]
```

The discrepancy is due to the fact that the center point is not a node in this particular mesh, but a point in the interior of a cell, and `u` varies linearly over the cell while `u0` is a quadratic function.

We have seen how to extract the nodal values in a `numpy` array. If desired, we can adjust the nodal values too. Say we want to normalize the solution such that the maximum value is 1. Then we must divide all $U_j$ values by $\max\{U_1, \ldots, U_N\}$. The following snippet performs the task:

*Python code*

```
max_u = u_array.max()
u_array /= max_u
u.vector()[:] = u_array
u.vector().set_local(u_array)  # alternative
print u.vector().array()
```

That is, we manipulate `u_array` as desired, and then we insert this array into `u`'s `Vector` object. The `/=` operator implies an in-place modification of the object on the left-hand side: all elements of the `u_array` are divided by the value `max_u`. Alternatively, one could write `u_array = u_array/max_u`, which implies creating a new array on the right-hand side and assigning this array to the name `u_array`.

A call like `u.vector().array()` returns a copy of the data in `u.vector()`. One must therefore never perform assignments like `u.vector.array()[:] = ...`, but instead extract the `numpy` array (that is, a copy), manipulate it, and insert it back with `u.vector()[:] =` or `u.set_local(...)`.

All the code in this subsection can be found in the file `d4_p2D.py` in the `stationary/poisson` directory.

### 1.1.7 Solving a real physical problem

Perhaps you are not particularly amazed by viewing the simple surface of $u$ in the test problem from Section 1.1.3. However, solving a real physical problem with a more interesting and amazing solution

on the screen is only a matter of specifying a more exciting domain, boundary condition, and/or right-hand side $f$.

One possible physical problem regards the deflection $D(x,y)$ of an elastic circular membrane with radius $R$, subject to a localized perpendicular pressure force, modeled as a Gaussian function. The appropriate PDE model is

$$-T\Delta D = p(x,y) \quad \text{in } \Omega = \{(x,y) \,|\, x^2 + y^2 \leqslant R\}, \tag{1.15}$$

with

$$p(x,y) = \frac{A}{2\pi\sigma}\exp\left(-\frac{1}{2}\left(\frac{x-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{y-y_0}{\sigma}\right)^2\right). \tag{1.16}$$

Here, $T$ is the tension in the membrane (constant), $p$ is the external pressure load, $A$ the amplitude of the pressure, $(x_0, y_0)$ the localization of the Gaussian pressure function, and $\sigma$ the "width" of this function. The boundary of the membrane has no deflection, implying $D = 0$ as boundary condition.

For scaling and verification it is convenient to simplify the problem to find an analytical solution. In the limit $\sigma \to \infty$, $p \to A/(2\pi\sigma)$, which allows us to integrate an axi–symmetric version of the equation in the radial coordinate $r \in [0, R]$ and obtain $D(r) = (r^2 - R^2)A/(8\pi\sigma T)$. This result gives a rough estimate of the characteristic size of the deflection: $|D(0)| = AR^2/(8\pi\sigma T)$, which can be used to scale the deflection. With $R$ as characteristic length scale, we can derive the equivalent dimensionless problem on the unit circle,

$$-\Delta w = f, \tag{1.17}$$

with $w = 0$ on the boundary and with

$$f(x,y) = 4\exp\left(-\frac{1}{2}\left(\frac{Rx-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{Ry-y_0}{\sigma}\right)^2\right). \tag{1.18}$$

For notational convenience we have dropped introducing new symbols for the scaled coordinates in (1.18). Now $D$ is related to $w$ through $D = AR^2w/(8\pi\sigma T)$.

Let us list the modifications of the `d1_p2D.py` program that are needed to solve this membrane problem:

1. Initialize $T$, $A$, $R$, $x_0$, $y_0$, and $\sigma$,
2. create a mesh over the unit circle,
3. make an expression object for the scaled pressure function $f$,
4. define the `a` and `L` formulas in the variational problem for $w$ and compute the solution,
5. plot the mesh, $w$, and $f$,
6. write out the maximum real deflection $D$,

Some suitable values of $T$, $A$, $R$, $x_0$, $y_0$, and $\sigma$ are

*Python code*

```
T = 10.0  # tension
A = 1.0   # pressure amplitude
R = 0.3   # radius of domain
theta = 0.2
x0 = 0.6*R*cos(theta)
y0 = 0.6*R*sin(theta)
sigma = 0.025
```

A mesh over the unit circle can be created by

*Python code*

```
mesh = UnitCircle(n)
```

where `n` is the typical number of elements in the radial direction.

The function $f$ is represented by an `Expression` object. There are many physical parameters in the formula for $f$ that enter the expression string and these parameters must have their values set by keyword arguments:

*Python code*

```
f = Expression("4*exp(-0.5*(pow((R*x[0] - x0)/sigma, 2)) "\
               "     - 0.5*(pow((R*x[1] - y0)/sigma, 2)))",
               R=R, x0=x0, y0=y0, sigma=sigma)
```

The coordinates in `Expression` objects *must* be a vector with indices 0, 1, and 2, and with the name `x`. Otherwise we are free to introduce names of parameters as long as these are given default values by keyword arguments. All the parameters initialized by keyword arguments can at any time have their values modified. For example, we may set

*Python code*

```
f.sigma = 50
f.x0 = 0.3
```

It would be of interest to visualize $f$ along with $w$ so that we can examine the pressure force and its response. We must then transform the formula (`Expression`) to a finite element function (`Function`). The most natural approach is to construct a finite element function whose degrees of freedom (values at the nodes in this case) are calculated from $f$. That is, we interpolate $f$ (see Section 1.1.6):

*Python code*

```
f = interpolate(f, V)
```

Calling `plot(f)` will produce a plot of $f$. Note that the assignment to `f` destroys the previous `Expression` object `f`, so if it is of interest to still have access to this object another name must be used for the `Function` object returned by `interpolate`.

We need some evidence that the program works, and to this end we may use the analytical solution listed above for the case $\sigma \to \infty$. In scaled coordinates the solution reads

$$w(x, y) = 1 - x^2 - y^2.$$

Practical values for an infinite $\sigma$ may be 50 or larger, and in such cases the program will report the maximum deviation between the computed $w$ and the (approximate) exact $w_e$.

Note that the variational formulation remains the same as in the program from Section 1.1.3, except that $u$ is replaced by $w$ and $u_0 = 0$. The final program is found in the file `membrane1.py`, located in the `stationary/poisson` directory, and also listed below. We have inserted capabilities for iterative solution methods and hence large meshes (Section 1.1.4), used objects for the variational problem and solver (Section 1.1.5), and made numerical comparison of the numerical and (approximate) analytical solution (Section 1.1.6).

*Python code*

```
from dolfin import *

# Set pressure function:
```

```
T = 10.0  # tension
A = 1.0   # pressure amplitude
R = 0.3   # radius of domain
theta = 0.2
x0 = 0.6*R*cos(theta)
y0 = 0.6*R*sin(theta)
sigma = 0.025
#sigma = 50  # large value for verification
n = 40   # approx no of elements in radial direction
mesh = UnitCircle(n)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define boundary condition w=0
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, Constant(0.0), boundary)

# Define variational problem
w = TrialFunction(V)
v = TestFunction(V)
a = inner(nabla_grad(w), nabla_grad(v))*dx
f = Expression("4*exp(-0.5*(pow((R*x[0] - x0)/sigma, 2)) "\
               "     - 0.5*(pow((R*x[1] - y0)/sigma, 2)))",
               R=R, x0=x0, y0=y0, sigma=sigma)
L = f*v*dx

# Compute solution
w = Function(V)
problem = LinearVariationalProblem(a, L, w, bc)
solver  = LinearVariationalSolver(problem)
solver.parameters["linear_solver"] = "cg"
solver.parameters["preconditioner"] = "ilu"
solver.solve()

# Plot scaled solution, mesh and pressure
plot(mesh, title="Mesh over scaled domain")
plot(w, title="Scaled deflection")
f = interpolate(f, V)
plot(f, title="Scaled pressure")

# Find maximum real deflection
max_w = w.vector().array().max()
max_D = A*max_w/(8*pi*sigma*T)
print "Maximum real deflection is", max_D

# Verification for "flat" pressure (large sigma)
if sigma >= 50:
    w_exact = Expression("1 - x[0]*x[0] - x[1]*x[1]")
    w_e = interpolate(w_exact, V)
    dev = numpy.abs(w_e.vector().array() - w.vector().array()).max()
    print 'sigma=%g: max deviation=%e' % dev

# Should be at the end
interactive()
```

Choosing a small width $\sigma$ (say 0.01) and a location $(x_0, y_0)$ toward the circular boundary (say $(0.6R\cos\theta, 0.6R\sin\theta)$ for any $\theta \in [0, 2\pi]$), may produce an exciting visual comparison of $w$ and $f$ that demonstrates the very smoothed elastic response to a peak force (or mathematically, the smoothing properties of the inverse of the Laplace operator). One needs to experiment with the mesh resolution

to get a smooth visual representation of $f$. You are strongly encouraged to play around with the plots and different mesh resolutions.

### 1.1.8 Quick visualization with VTK

As we go along with examples it is fun to play around with `plot` commands and visualize what is computed. This section explains some useful visualization features.

The `plot(u)` command launches a FEniCS component called Viper, which applies the VTK package to visualize finite element functions. Viper is not a full-fledged, easy-to-use front-end to VTK (like Mayavi2, ParaView, or VisIt), but rather a thin layer on top of VTK's Python interface, allowing us to quickly visualize a DOLFIN function or mesh, or data in plain Numerical Python arrays, within a Python program. Viper is ideal for debugging, teaching, and initial scientific investigations. The visualization can be interactive, or you can steer and automate it through program statements. More advanced and professional visualizations are usually better done with advanced tools like MayaVi2, ParaView, or VisIt.

We have made a program `membrane1v.py` for the membrane deflection problem in Section 1.1.7 and added various demonstrations of Viper capabilities. You are encouraged to play around with `membrane1v.py` and modify the code as you read about various features.

The `plot` function can take additional arguments, such as a title of the plot, or a specification of a wireframe plot (elevated mesh) instead of a colored surface plot:

*Python code*

```
plot(mesh, title="Finite element mesh")
plot(w, wireframe=True, title="solution")
```

The three mouse buttons can be used to rotate, translate, and zoom the surface. Pressing `h` in the plot window makes a printout of several key bindings that are available in such windows. For example, pressing `m` in the mesh plot window dumps the plot of the mesh to an Encapsulated PostScript (`.eps`) file, while pressing `i` saves the plot in PNG format. All file names are automatically generated as `simulationX.eps`, where `X` is a counter `0000`, `0001`, `0002`, etc., being increased every time a new plot file in that format is generated (the extension of PNG files is `.png` instead of `.eps`). Pressing `o` adds a red outline of a bounding box around the domain.

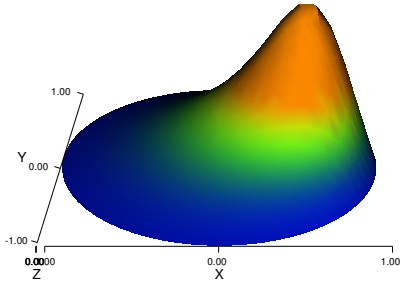
One can alternatively control the visualization from the program code directly. This is done through a `Viper` object returned from the `plot` command. Let us grab this object and use it to 1) tilt the camera $-65$ degrees in the latitude direction, 2) add $x$ and $y$ axes, 3) change the default name of the plot files, 4) change the color scale, and 5) write the plot to a PNG and an EPS file. Here is the code:

*Python code*

```
viz_w = plot(w,
             wireframe=False,
             title="Scaled membrane deflection",
             rescale=False,
             axes=True,              # include axes
             basename="deflection",  # default plotfile name
             )

viz_w.elevate(-65) # tilt camera -65 degrees (latitude dir)
viz_w.set_min_max(0, 0.5*max_w)  # color scale
viz_w.update(w)    # bring settings above into action
viz_w.write_png("deflection.png")
viz_w.write_ps("deflection", format="eps")
```

Figure 1.3: Plot of the deflection of a membrane.




The `format` argument in the latter line can also take the values `"ps"` for a standard PostScript file and `"pdf"` for a PDF file. Note the necessity of the `viz_w.update(w)` call – without it we will not see the effects of tilting the camera and changing the color scale. Figure 1.3 shows the resulting scalar surface.

### *1.1.9 Computing derivatives*

In Poisson and many other problems the gradient of the solution is of interest. The computation is in principle simple: since $u = \sum_{j=1}^{N} U_j \phi_j$, we have that

$$\nabla u = \sum_{j=1}^{N} U_j \nabla \phi_j. \tag{1.19}$$

Given the solution variable `u` in the program, its gradient is obtained by `grad(u)` or `nabla_grad(u)`. However, the gradient of a piecewise continuous finite element scalar field is a discontinuous vector field since the $\phi_j$ has discontinuous derivatives at the boundaries of the cells. For example, using Lagrange elements of degree 1, $u$ is linear over each cell, and the numerical $\nabla u$ becomes a piecewise constant vector field. On the contrary, the exact gradient is continuous. For visualization and data analysis purposes we often want the computed gradient to be a continuous vector field. Typically, we want each component of $\nabla u$ to be represented in the same way as $u$ itself. To this end, we can project the components of $\nabla u$ onto the same function space as we used for $u$. This means that we solve $w = \nabla u$ approximately by a finite element method. This process is known as *projection*. Looking at the component $\partial u/\partial x$ of the gradient, we project the (discrete) derivative $\sum_j U_j \partial \phi_j/\partial x$ onto a function space with basis $\phi_1, \phi_2, \ldots$ such that the derivative in this space is expressed by the standard sum $\sum_j \bar{U}_j \phi_j$, for suitable (new) coefficients $\bar{U}_j$.

The variational problem for $w$ reads: find $w \in V^{(\mathrm{g})}$ such that

$$a(w, v) = L(v) \quad \forall\, v \in \hat{V}^{(\mathrm{g})}, \tag{1.20}$$

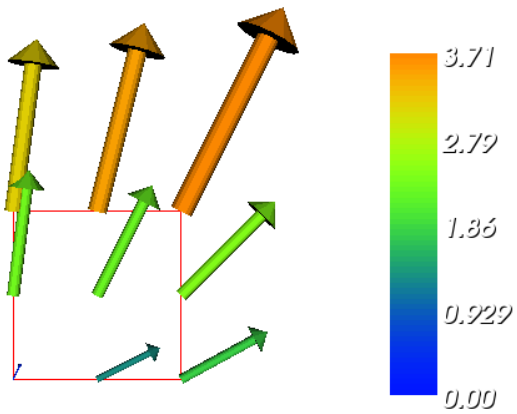



Figure 1.4: Example of visualizing the vector field $\nabla u$ by arrows at the nodes.

where

$$a(w, v) = \int_\Omega w \cdot v \, \mathrm{d}x, \tag{1.21}$$

$$L(v) = \int_\Omega \nabla u \cdot v \, \mathrm{d}x. \tag{1.22}$$

The function spaces $V^{(\mathrm{g})}$ and $\hat{V}^{(\mathrm{g})}$ (with the superscript g denoting "gradient") are vector versions of the function space for $u$, with boundary conditions removed (if $V$ is the space we used for $u$, with no restrictions on boundary values, $V^{(\mathrm{g})} = \hat{V}^{(\mathrm{g})} = [V]^d$, where $d$ is the number of space dimensions). For example, if we used piecewise linear functions on the mesh to approximate $u$, the variational problem for $w$ corresponds to approximating each component field of $w$ by piecewise linear functions.

The variational problem for the vector field $w$, called `grad_u` in the code, is easy to solve in FEniCS:

*Python code*

```
V_g = VectorFunctionSpace(mesh, "Lagrange", 1)
w = TrialFunction(V_g)
v = TestFunction(V_g)

a = inner(w, v)*dx
L = inner(grad(u), v)*dx
grad_u = Function(V_g)
solve(a == L, grad_u)

plot(grad_u, title="grad(u)")
```

The boundary condition argument to `solve` is dropped since there are no essential boundary conditions in this problem. The new thing is basically that we work with a `VectorFunctionSpace`, since the unknown is now a vector field, instead of the `FunctionSpace` object for scalar fields. Figure 1.4 shows an example of how Viper can visualize such a vector field.

The scalar component fields of the gradient can be extracted as separate fields and, e.g., visualized:

*Python code*

```
grad_u_x, grad_u_y = grad_u.split(deepcopy=True)  # extract components
plot(grad_u_x, title="x-component of grad(u)")
plot(grad_u_y, title="y-component of grad(u)")
```

The `deepcopy=True` argument signifies a *deep copy*, which is a general term in computer science implying that a copy of the data is returned. (The opposite, `deepcopy=False`, means a *shallow copy*, where the returned objects are just pointers to the original data.)

The `grad_u_x` and `grad_u_y` variables behave as `Function` objects. In particular, we can extract the underlying arrays of nodal values by

*Python code*

```
grad_u_x_array = grad_u_x.vector().array()
grad_u_y_array = grad_u_y.vector().array()
```

The degrees of freedom of the `grad_u` vector field can also be reached by

*Python code*

```
grad_u_array = grad_u.vector().array()
```

but this is a flat `numpy` array where the degrees of freedom for the $x$ component of the gradient is stored in the first part, then the degrees of freedom of the $y$ component, and so on.

The program `d5_p2D.py` extends the code `d4_p2D.py` from Section 1.1.6 with computations and visualizations of the gradient. Examining the arrays `grad_u_x_array` and `grad_u_y_array`, or looking at the plots of `grad_u_x` and `grad_u_y`, quickly reveals that the computed `grad_u` field does not equal the exact gradient $(2x, 4y)$ in this particular test problem where $u = 1 + x^2 + 2y^2$. There are inaccuracies at the boundaries, arising from the approximation problem for $w$. Increasing the mesh resolution shows, however, that the components of the gradient vary linearly as $2x$ and $4y$ in the interior of the mesh (as soon as we are one element away from the boundary). See Section 1.1.8 for illustrations of this phenomenon.

Projecting some function onto some space is a very common operation in finite element programs. The manual steps in this process have therefore been collected in a utility function `project(q, W)`, which returns the projection of some `Function` or `Expression` object named `q` onto the `FunctionSpace` or `VectorFunctionSpace` named `W`. Specifically, the previous code for projecting each component of `grad(u)` onto the same space that we use for `u`, can now be done by a one–line call:

*Python code*

```
grad_u = project(grad(u), VectorFunctionSpace(mesh, "Lagrange", 1))
```

The applications of projection are many, including turning discontinuous gradient fields into continuous ones, comparing higher- and lower-order function approximations, and transforming a higher-order finite element solution down to a piecewise linear field, which is required by many visualization packages.

### *1.1.10 A variable-coefficient Poisson problem*

Suppose we have a variable coefficient $p(x, y)$ in the Laplace operator, as in the boundary-value problem

$$\begin{aligned} -\nabla \cdot [p(x,y)\nabla u(x,y)] &= f(x,y) \quad \text{in } \Omega, \\ u(x,y) &= u_0(x,y) \quad \text{on } \partial\Omega. \end{aligned} \tag{1.23}$$

We shall quickly demonstrate that this simple extension of our model problem only requires an equally simple extension of the FEniCS program.

Let us continue to use our favorite solution $u(x, y) = 1 + x^2 + 2y^2$ and then prescribe $p(x, y) = x + y$. It follows that $u_0(x, y) = 1 + x^2 + 2y^2$ and $f(x, y) = -8x - 10y$.

What are the modifications we need to do in the `d4_p2D.py` program from Section 1.1.6?

1. `f` must be an `Expression` since it is no longer a constant,
2. a new `Expression` `p` must be defined for the variable coefficient,
3. the variational problem is slightly changed.

First we address the modified variational problem. Multiplying the PDE in (1.23) and integrating by parts now results in

$$\int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega} p\frac{\partial u}{\partial n}v\,\mathrm{d}s = \int_\Omega fv\,\mathrm{d}x. \tag{1.24}$$

The function spaces for $u$ and $v$ are the same as in Section 1.1.2, implying that the boundary integral vanishes since $v = 0$ on $\partial\Omega$ where we have Dirichlet conditions. The weak form $a(u,v) = L(v)$ then has

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x, \tag{1.25}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x. \tag{1.26}$$

In the code from Section 1.1.3 we must replace

*Python code*

```
a = inner(nabla_grad(u), nabla_grad(v))*dx
```

by

*Python code*

```
a = p*inner(nabla_grad(u), nabla_grad(v))*dx
```

The definitions of `p` and `f` read

*Python code*

```
p = Expression("x[0] + x[1]")
f = Expression("-8*x[0] - 10*x[1]")
```

No additional modifications are necessary. The complete code can be found in in the file `vcp2D.py` (variable-coefficient Poisson problem in 2D). You can run it and confirm that it recovers the exact $u$ at the nodes.

The flux $-p\nabla u$ may be of particular interest in variable-coefficient Poisson problems as it often has an interesting physical significance. As explained in Section 1.1.9, we normally want the piecewise discontinuous flux or gradient to be approximated by a continuous vector field, using the same elements as used for the numerical solution $u$. The approximation now consists of solving $w = -p\nabla u$ by a finite element method: find $w \in V^{(\mathrm{g})}$ such that

$$a(w,v) = L(v) \quad \forall\, v \in \hat{V}^{(\mathrm{g})}, \tag{1.27}$$

where

$$a(w,v) = \int_\Omega w\cdot v\,\mathrm{d}x, \tag{1.28}$$

$$L(v) = \int_\Omega (-p\nabla u)\cdot v\,\mathrm{d}x. \tag{1.29}$$

This problem is identical to the one in Section 1.1.9, except that $p$ enters the integral in $L$.

The relevant Python statements for computing the flux field take the form

*Python code*

```
V_g = VectorFunctionSpace(mesh, "Lagrange", 1)
w = TrialFunction(V_g)
v = TestFunction(V_g)

a = inner(w, v)*dx
L = inner(-p*grad(u), v)*dx
flux = Function(V_g)
solve(a == L, flux)
```

The following call to `project` is equivalent to the above statements:

*Python code*

```
flux = project(-p*nabla_grad(u),
               VectorFunctionSpace(mesh, "Lagrange", 1))
```

Plotting the flux vector field is naturally as easy as plotting the gradient in Section 1.1.9:

*Python code*

```
plot(flux, title="flux field")

flux_x, flux_y = flux.split(deepcopy=True)  # extract components
plot(flux_x, title="x-component of flux (-p*grad(u))")
plot(flux_y, title="y-component of flux (-p*grad(u))")
```

For data analysis of the nodal values of the flux field we can grab the underlying `numpy` arrays:

*Python code*

```
flux_x_array = flux_x.vector().array()
flux_y_array = flux_y.vector().array()
```

The program `vcp2D.py` contains in addition some plots, including a curve plot comparing `flux_x` and the exact counterpart along the line $y = 1/2$. The associated programming details related to this visualization are explained in Section 1.1.12.

### 1.1.11 Computing functionals

After the solution $u$ of a PDE is computed, we occasionally want to compute functionals of $u$, for example,

$$\frac{1}{2}||\nabla u||^2 \equiv \frac{1}{2}\int_\Omega \nabla u \cdot \nabla u \,\mathrm{d}x, \tag{1.30}$$

which often reflects some energy quantity. Another frequently occurring functional is the error

$$||u_\mathrm{e} - u|| = \left(\int_\Omega (u_\mathrm{e} - u)^2 \,\mathrm{d}x\right)^{1/2}, \tag{1.31}$$

where $u_\mathrm{e}$ is the exact solution. The error is of particular interest when studying convergence properties. Sometimes the interest concerns the flux out of a part $\Gamma$ of the boundary $\partial\Omega$,

$$F = -\int_\Gamma p\nabla u \cdot \,\mathrm{d}s, \tag{1.32}$$

where $n$ is an outward unit normal at $\Gamma$ and $p$ is a coefficient (see the problem in Section 1.1.10 for a specific example). All these functionals are easy to compute with FEniCS, and this section describes how it can be done.

*Energy functional.* The integrand of the energy functional (1.30) is described in the UFL language in the same manner as we describe weak forms:

*Python code*

```
energy = 0.5*inner(grad(u), grad(u))*dx
E = assemble(energy)
```

The `assemble` call performs the integration. It is possible to restrict the integration to subdomains, or parts of the boundary, by using a mesh function to mark the subdomains (this technique will be explained in Section 1.5.3). The program `membrane2.py` carries out the computation of the elastic energy

$$\frac{1}{2}||T\nabla D||^2 = \frac{1}{2}\left(\frac{AR}{8\pi\sigma}\right)^2 ||\nabla w||^2 \tag{1.33}$$

in the membrane problem from Section 1.1.7.

*Convergence estimation.* To illustrate error computations and convergence of finite element solutions, we modify the `d5_p2D.py` program from Section 1.1.9 and specify a more complicated solution,

$$u(x,y) = \sin(\omega\pi x)\sin(\omega\pi y) \tag{1.34}$$

on the unit square. This choice implies $f(x,y) = 2\omega^2\pi^2 u(x,y)$. With $\omega$ restricted to an integer it follows that $u_0 = 0$. We must define the appropriate boundary conditions, the exact solution, and the $f$ function in the code:

*Python code*

```
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, Constant(0.0), boundary)

omega = 1.0
u_e = Expression("sin(omega*pi*x[0])*sin(omega*pi*x[1])",
                 omega=omega)

f = 2*pi**2*omega**2*u_e
```

The computation of (1.31) can be done by

*Python code*

```
error = (u - u_e)**2*dx
E = sqrt(assemble(error))
```

Here, `u_e` will be interpolated onto the function space `V`. This implies that the exact solution used in the integral will vary linearly over the cells, and not as a sine function, if `V` corresponds to linear Lagrange elements. This situation may yield a smaller error `u - u_e` than what is actually true.

More accurate representation of the exact solution is easily achieved by interpolating the formula onto a space defined by higher-order elements, say of third degree:

*Python code*

```
Ve = FunctionSpace(mesh, "Lagrange", degree=3)
u_e_Ve = interpolate(u_e, Ve)
error = (u - u_e_Ve)**2*dx
E = sqrt(assemble(error))
```

To achieve complete mathematical control of which function space the computations are carried out in, we can explicitly interpolate u too:

*Python code*

```
u_Ve = interpolate(u, Ve)
error = (u_Ve - u_e_Ve)**2*dx
```

The square in the expression for `error` will be expanded and lead to a lot of terms that almost cancel when the error is small, with the potential of introducing significant round-off errors. The function `errornorm` is available for avoiding this effect by first interpolating `u` and `u_e` to a space with higher-order elements, then subtracting the degrees of freedom, and then performing the integration of the error field. The usage is simple:

*Python code*

```
E = errornorm(u_e, u, normtype="L2", degree=3)
```

It is illustrative to look at the short implementation of `errornorm`:

*Python code*

```
def errornorm(u_e, u, Ve):
    u_Ve = interpolate(u, Ve)
    u_e_Ve = interpolate(u_e, Ve)
    e_Ve = Function(Ve)
    # Subtract degrees of freedom for the error field
    e_Ve.vector()[:] = u_e_Ve.vector().array() - \
                       u_Ve.vector().array()
    error = e_Ve**2*dx
    return sqrt(assemble(error))
```

The `errornorm` procedure turns out to be identical to computing the expression `(u_e - u)**2*dx` directly in the present test case.

Sometimes it is of interest to compute the error of the gradient field: $||\nabla(u - u_e)||$ (often referred to as the $H^1$ seminorm of the error). Given the error field `e_Ve` above, we simply write

*Python code*

```
H1seminorm = sqrt(assemble(inner(grad(e_Ve), grad(e_Ve))*dx))
```

Finally, we remove all `plot` calls and printouts of $u$ values in the original program, and collect the computations in a function:

*Python code*

```
def compute(nx, ny, degree):
    mesh = UnitSquare(nx, ny)
    V = FunctionSpace(mesh, "Lagrange", degree=degree)
    ...
    Ve = FunctionSpace(mesh, "Lagrange", degree=5)
    E = errornorm(u_e, u, Ve)
    return E
```

Calling `compute` for finer and finer meshes enables us to study the convergence rate. Define the element size $h = 1/n$, where $n$ is the number of divisions in $x$ and $y$ direction (`nx=ny` in the code). We perform experiments with $h_0 > h_1 > h_2 \cdots$ and compute the corresponding errors $E_0, E_1, E_3$ and so forth. Assuming $E_i = Ch_i^r$ for unknown constants $C$ and $r$, we can compare two consecutive

experiments, $E_i = Ch_i^r$ and $E_{i-1} = Ch_{i-1}^r$, and solve for $r$:

$$r = \frac{\ln(E_i/E_{i-1})}{\ln(h_i/h_{i-1})}. \tag{1.35}$$

The $r$ values should approach the expected convergence rate `degree+1` as $i$ increases.

The procedure above can easily be turned into Python code:

*Python code*

```
import sys
degree = int(sys.argv[1])  # read degree as 1st command-line arg
h = []  # element sizes
E = []  # errors
for nx in [4, 8, 16, 32, 64, 128, 264]:
    h.append(1.0/nx)
    E.append(compute(nx, nx, degree))

# Convergence rates
from math import log as ln  # (log is a dolfin name too)
for i in range(1, len(E)):
    r = ln(E[i]/E[i-1])/ln(h[i]/h[i-1])
    print "h=%10.2E r=%.2f" % (h[i], r)
```

The resulting program has the name `d6_p2D.py` and computes error norms in various ways. Running this program for elements of first degree and $\omega = 1$ yields the output

*Output*

```
h=1.25E-01 E=3.25E-02 r=1.83
h=6.25E-02 E=8.37E-03 r=1.96
h=3.12E-02 E=2.11E-03 r=1.99
h=1.56E-02 E=5.29E-04 r=2.00
h=7.81E-03 E=1.32E-04 r=2.00
h=3.79E-03 E=3.11E-05 r=2.00
```

That is, we approach the expected second-order convergence of linear Lagrange elements as the meshes become sufficiently fine.

Running the program for second-degree elements results in the expected value $r = 3$,

*Output*

```
h=1.25E-01 E=5.66E-04 r=3.09
h=6.25E-02 E=6.93E-05 r=3.03
h=3.12E-02 E=8.62E-06 r=3.01
h=1.56E-02 E=1.08E-06 r=3.00
h=7.81E-03 E=1.34E-07 r=3.00
h=3.79E-03 E=1.53E-08 r=3.00
```

However, using `(u - u_e)**2` for the error computation, which implies interpolating `u_e` onto the same space as `u`, results in $r = 4$ (!). This is an example where it is important to interpolate `u_e` to a higher-order space (polynomials of degree 3 are sufficient here) to avoid computing a too optimistic convergence rate.

Running the program for third-degree elements results in the expected value $r = 4$:

*Output*

```
h=1.25E-01 r=4.09
h=6.25E-02 r=4.03
h=3.12E-02 r=4.01
h=1.56E-02 r=4.00
h=7.81E-03 r=4.00
```

Checking convergence rates is the next best method for verifying PDE codes (the best being exact recovery of a solution as in Section 1.1.6 and many other places in this tutorial).

*Flux functionals.* To compute flux integrals like (1.32) we need to define the $n$ vector, referred to as *facet normal* in FEniCS. If $\Gamma$ is the complete boundary we can perform the flux computation by

*Python code*

```
n = FacetNormal(mesh)
flux = -p*dot(nabla_grad(u), n)*ds
total_flux = assemble(flux)
```

Although `nabla_grad(u)` and `grad(u)` are interchangeable in the above expression when `u` is a scalar function, we have chosen to write `nabla_grad(u)` because this is the right expression if we generalize the underlying equation to a vector Laplace/Poisson PDE. With `grad(u)` we must in that case write `dot(n, grad(u))`.

It is possible to restrict the integration to a part of the boundary using a mesh function to mark the relevant part, as explained in Section 1.5.3. Assuming that the part corresponds to subdomain number `i`, the relevant form for the flux is `-p*dot(nabla_grad(u), n)*ds(i)`.

### 1.1.12 *Visualization of structured mesh data*

When finite element computations are done on a structured rectangular mesh, maybe with uniform partitioning, VTK-based tools for completely unstructured 2D/3D meshes are not required. Instead we can use visualization and data analysis tools for *structured data*. Such data typically appear in finite difference simulations and image analysis. Analysis and visualization of structured data are faster and easier than doing the same with data on unstructured meshes, and the collection of tools to choose among is much larger. We shall demonstrate the potential of such tools and how they allow for tailored and flexible visualization and data analysis.

A necessary first step is to transform our `mesh` object to an object representing a rectangle with equally-shaped *rectangular* cells. The Python package `scitools` has this type of structure, called a `UniformBoxGrid`. The second step is to transform the one-dimensional array of nodal values to a two-dimensional array holding the values at the corners of the cells in the structured grid. In such grids, we want to access a value by its $i$ and $j$ indices, $i$ counting cells in the $x$ direction, and $j$ counting cells in the $y$ direction. This transformation is in principle straightforward, yet it frequently leads to obscure indexing errors. The `BoxField` object in `scitools` takes conveniently care of the details of the transformation. With a `BoxField` defined on a `UniformBoxGrid` it is very easy to call up more standard plotting packages to visualize the solution along lines in the domain or as 2D contours or lifted surfaces.

Let us go back to the `vcp2D.py` code from Section 1.1.10 and map `u` onto a `BoxField` object:

*Python code*

```
import scitools.BoxField
u2 = u if u.ufl_element().degree() == 1 else \
     interpolate(u, FunctionSpace(mesh, "Lagrange", 1))
u_box = scitools.BoxField.dolfin_function2BoxField(
        u2, mesh, (nx,ny), uniform_mesh=True)
```

The function `dolfin_function2BoxField` can only work with finite element fields with *linear* (degree 1) elements, so for higher-degree elements we here simply interpolate the solution onto a mesh with linear elements. We could also interpolate/project onto a finer mesh in the higher-degree case. Such transformations to linear finite element fields are very often needed when calling up plotting packages

or data analysis tools. The `u.ufl_element()` method returns an object holding the element type, and this object has a method `degree()` for returning the element degree as an integer. The parameters `nx` and `ny` are the number of divisions in each space direction that were used when calling `UnitSquare` to make the `mesh` object. The result `u_box` is a `BoxField` object that supports "finite difference" indexing and an underlying grid suitable for `numpy` operations on 2D data. Also 1D and 3D meshes (with linear elements) can be turned into `BoxField` objects.

The ability to access a finite element field in the way one can access a finite difference-type of field is handy in many occasions, including visualization and data analysis. Here is an example of writing out the coordinates and the field value at a grid point with indices `i` and `j` (going from 0 to `nx` and `ny`, respectively, from lower left to upper right corner):

*Python code*

```
X = 0; Y = 1; Z = 0  # convenient indices

i = nx; j = ny   # upper right corner
print "u(%g,%g)=%g" % (u_box.grid.coor[X][i],
                      u_box.grid.coor[Y][j],
                      u_box.values[i,j])
```

For instance, the $x$ coordinates are reached by `u_box.grid.coor[X]`. The `grid` attribute is an instance of class `UniformBoxGrid`.

Many plotting programs can be used to visualize the data in `u_box`. Matplotlib is now a very popular plotting program in the Python world and could be used to make contour plots of `u_box`. However, other programs like Gnuplot, VTK, and MATLAB have better support for surface plots at the time of this writing. Our choice in this tutorial is to use the Python package `scitools.easyviz`, which offers a uniform MATLAB-like syntax as interface to various plotting packages such as Gnuplot, matplotlib, VTK, OpenDX, MATLAB, and others. With `scitools.easyviz` we write one set of statements, close to what one would do in MATLAB or Octave, and then it is easy to switch between different plotting programs, at a later stage, through a command-line option, a line in a configuration file, or an import statement in the program.

A contour plot is made by the following `scitools.easyviz` command:

*Python code*

```
import scitools.easyviz as ev
ev.contour(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
           5, clabels="on")
ev.title("Contour plot of u")
ev.savefig("u_contours.eps")

# or more compact syntax:
ev.contour(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
           5, clabels="on",
           savefig="u_contours.eps", title="Contour plot of u")
```

The resulting plot can be viewed in Figure 1.5a. The `contour` function needs arrays with the $x$ and $y$ coordinates expanded to 2D arrays (in the same way as demanded when making vectorized `numpy` calculations of arithmetic expressions over all grid points). The correctly expanded arrays are stored in `grid.coorv`. The above call to `contour` creates 5 equally spaced contour lines, and with `clabels="on"` the contour values can be seen in the plot.

Other functions for visualizing 2D scalar fields are `surf` and `mesh` as known from MATLAB:

*Python code*

```
import scitools.easyviz as ev
ev.figure()
ev.surf(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        shading="interp", colorbar="on",
        title="surf plot of u", savefig="u_surf.eps")

ev.figure()
ev.mesh(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        title="mesh plot of u", savefig="u_mesh.eps")
```

Figure 1.6 exemplifies the surfaces arising from the two plotting commands above. You can type `pydoc scitools.easyviz` in a terminal window to get a full tutorial. Note that `scitools.easyviz` offers function names like `plot` and `mesh`, which clash with `plot` from `dolfin` and the `mesh` variable in our programs. Therefore, we recommend the `ev` prefix.

A handy feature of `BoxField` is the ability to give a start point in the grid and a direction, and then extract the field and corresponding coordinates along the nearest grid line. In 3D fields one can also extract data in a plane. Say we want to plot $u$ along the line $y = 1/2$ in the grid. The grid points, `x`, and the $u$ values along this line, `uval`, are extracted by

*Python code*

```
start = (0, 0.5)
x, uval, y_fixed, snapped = u_box.gridline(start, direction=X)
```

The variable `snapped` is true if the line had to be snapped onto a grid line and in that case `y_fixed` holds the snapped (altered) $y$ value. Plotting $u$ versus the $x$ coordinate along this line, using `scitools.easyviz`, is now a matter of

*Python code*

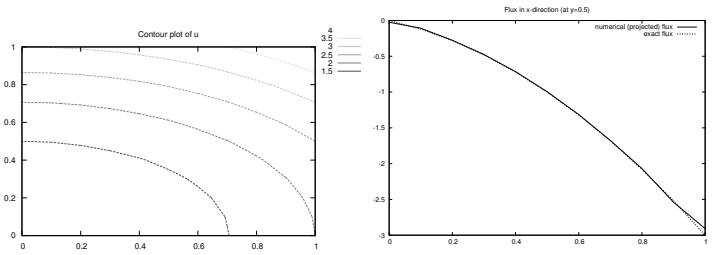
```
ev.figure()  # new plot window
ev.plot(x, uval, "r-")  # "r--: red solid line
ev.title("Solution")
ev.legend("finite element solution")

# or more compactly:
ev.plot(x, uval, "r-", title="Solution",
        legend="finite element solution")
```

A more exciting plot compares the projected numerical flux in $x$ direction along the line $y = 1/2$ with the exact flux:

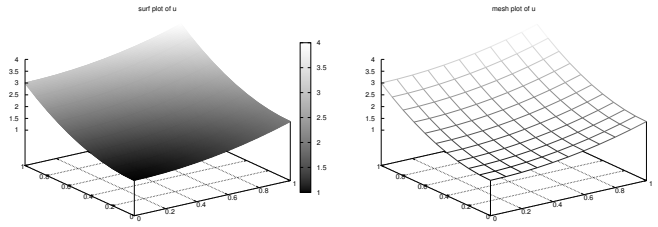
*Python code*

```
ev.figure()
flux2_x = flux_x if flux_x.ufl_element().degree() == 1 else \
          interpolate(flux_x, FunctionSpace(mesh, "Lagrange", 1))
flux_x_box = scitools.BoxField.dolfin_function2BoxField(
          flux2_x, mesh, (nx,ny), uniform_mesh=True)
x, fluxval, y_fixed, snapped = \
          flux_x_box.gridline(start, direction=X)
y = y_fixed
flux_x_exact = -(x + y)*2*x
ev.plot(x, fluxval, "r-",
        x, flux_x_exact, "b-",
        legend=("numerical (projected) flux", "exact flux"),
        title="Flux in x-direction (at y=%g)" % y_fixed,
        savefig="flux.eps")
```




(a) (b)

Figure 1.5: Examples of plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) contour plot of the solution; (b) curve plot of the exact flux $-p\partial u/\partial x$ against the corresponding projected numerical flux.




(a) (b)

Figure 1.6: Examples of plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) a surface plot of the solution; (b) lifted mesh plot of the solution.

As seen from Figure 1.5b, the numerical flux is accurate except in the elements closest to the boundaries.

The visualization constructions shown above and used to generate the figures are found in the program `vcp2D.py` in the `stationary/poisson` directory.

It should be easy with the information above to transform a finite element field over a uniform rectangular or box-shaped mesh to the corresponding `BoxField` object and perform MATLAB-style visualizations of the whole field or the field over planes or along lines through the domain. By the transformation to a regular grid we have some more flexibility than what Viper offers. However, we must remark that comprehensive tools like VisIt, MayaVi2, or ParaView also have the possibility for plotting fields along lines and extracting planes in 3D geometries, though usually with less degree of control compared to Gnuplot, MATLAB, and matplotlib.

### *1.1.13 Combining Dirichlet and Neumann conditions*

Let us make a slight extension of our two-dimensional Poisson problem from Section 1.1.1 and add a Neumann boundary condition. The domain is still the unit square, but now we set the Dirichlet condition $u = u_0$ at the left and right sides, $x = 0$ and $x = 1$, while the Neumann condition

$$-\frac{\partial u}{\partial n} = g \tag{1.36}$$

is applied to the remaining sides $y = 0$ and $y = 1$. The Neumann condition is also known as a *natural boundary condition* (in contrast to an essential boundary condition).

Let $\Gamma_{\mathrm{D}}$ and $\Gamma_{\mathrm{N}}$ denote the parts of $\partial\Omega$ where the Dirichlet and Neumann conditions apply,

respectively. The complete boundary-value problem can be written as

$$-\Delta u = f \text{ in } \Omega, \tag{1.37}$$

$$u = u_0 \text{ on } \Gamma_{\mathrm{D}}, \tag{1.38}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_{\mathrm{N}}. \tag{1.39}$$

Again we choose $u = 1 + x^2 + 2y^2$ as the exact solution and adjust $f$, $g$, and $u_0$ accordingly:

$$f = -6, \tag{1.40}$$

$$g = \begin{cases} -4, & y = 1 \\ 0, & y = 0 \end{cases} \tag{1.41}$$

$$u_0 = 1 + x^2 + 2y^2. \tag{1.42}$$

For ease of programming we may introduce a $g$ function defined over the whole of $\Omega$ such that $g$ takes on the right values at $y = 0$ and $y = 1$. One possible extension is

$$g(x, y) = -4y. \tag{1.43}$$

The first task is to derive the variational problem. This time we cannot omit the boundary term arising from the integration by parts, because $v$ is only zero on $\Gamma_{\mathrm{D}}$. We have

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s, \tag{1.44}$$

and since $v = 0$ on $\Gamma_{\mathrm{D}}$,

$$-\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s = -\int_{\Gamma_{\mathrm{N}}} \frac{\partial u}{\partial n} v \,\mathrm{d}s = \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s, \tag{1.45}$$

by applying the boundary condition on $\Gamma_{\mathrm{N}}$. The resulting weak form reads

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x. \tag{1.46}$$

Expressing (1.46) in the standard notation $a(u, v) = L(v)$ is straightforward with

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.47}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s. \tag{1.48}$$

How does the Neumann condition impact the implementation? The code in the file `d4_p2D.py` in the directory `stationary/poisson` remains almost the same. Only two adjustments are necessary:

1. The function describing the boundary where Dirichlet conditions apply must be modified.
2. The new boundary term must be added to the expression in `L`.

Step 1 can be coded as

*Python code*

```
def Dirichlet_boundary(x, on_boundary):
    if on_boundary:
```

```
        if x[0] == 0 or x[0] == 1:
            return True
        else:
            return False
    else:
        return False
```

A more compact implementation reads

*Python code*

```
def Dirichlet_boundary(x, on_boundary):
    return on_boundary and (x[0] == 0 or x[0] == 1)
```

As pointed out already in Section 1.1.3, testing for an exact match of real numbers is not good programming practice so we introduce a tolerance in the test:

*Python code*

```
def Dirichlet_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and \
           (abs(x[0]) < tol or abs(x[0] - 1) < tol)
```

The second adjustment of our program concerns the definition of `L`, where we have to add a boundary integral and a definition of the $g$ function to be integrated:

*Python code*

```
g = Expression("-4*x[1]")
L = f*v*dx - g*v*ds
```

The `ds` variable implies a boundary integral, while `dx` implies an integral over the domain $\Omega$. No more modifications are necessary.

The file `dn1_p2D.py` in the `stationary/poisson` directory implements this problem. Running the program verifies the implementation: $u$ equals the exact solution at all the nodes, regardless of how many elements we use.

### 1.1.14 Multiple Dirichlet conditions

The PDE problem from the previous section applies a function $u_0(x, y)$ for setting Dirichlet conditions at two parts of the boundary. Having a single function to set multiple Dirichlet conditions is seldom possible. The more general case is to have $m$ functions for setting Dirichlet conditions on $m$ parts of the boundary. The purpose of this section is to explain how such multiple conditions are treated in FEniCS programs.

Let us return to the case from Section 1.1.13 and define two separate functions for the two Dirichlet conditions:

$$-\Delta u = -6 \text{ in } \Omega, \tag{1.49}$$

$$u = u_L \text{ on } \Gamma_0, \tag{1.50}$$

$$u = u_R \text{ on } \Gamma_1, \tag{1.51}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_{\mathrm{N}}. \tag{1.52}$$

Here, $\Gamma_0$ is the boundary $x = 0$, while $\Gamma_1$ corresponds to the boundary $x = 1$. We have that $u_L = 1 + 2y^2$, $u_R = 2 + 2y^2$, and $g = -4y$. For the left boundary $\Gamma_0$ we define the usual triple of a

function for the boundary value, a function for defining the boundary of interest, and a `DirichletBC` object:

*Python code*

```
u_L = Expression("1 + 2*x[1]*x[1]")

def left_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and abs(x[0]) < tol

Gamma_0 = DirichletBC(V, u_L, left_boundary)
```

For the boundary $x = 1$ we define a similar code:

*Python code*

```
u_R = Expression("2 + 2*x[1]*x[1]")

def right_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and abs(x[0] - 1) < tol

Gamma_1 = DirichletBC(V, u_R, right_boundary)
```

The various essential conditions are then collected in a list and used in the solution process:

*Python code*

```
bcs = [Gamma_0, Gamma_1]
...
solve(a == L, u, bcs)
# or
problem = LinearVariationalProblem(a, L, u, bcs)
solver  = LinearVariationalSolver(problem)
solver.solve()
```

If the $u$ values are constant at a part of the boundary, we may use a simple `Constant` object instead of an `Expression` object.

The file `dn2_p2D.py` contains a complete program which demonstrates the constructions above. An extended example with multiple Neumann conditions would have been quite natural now, but this requires marking various parts of the boundary using the mesh function concept and is therefore left to Section 1.5.3.

### *1.1.15 A linear algebra formulation*

Given $a(u,v) = L(v)$, the discrete solution $u$ is computed by inserting $u = \sum_{j=1}^{N} U_j\phi_j$ into $a(u,v)$ and demanding $a(u,v) = L(v)$ to be fulfilled for $N$ test functions $\hat{\phi}_1, \ldots, \hat{\phi}_N$. This implies

$$\sum_{j=1}^{N} a(\phi_j, \hat{\phi}_i)U_j = L(\hat{\phi}_i), \quad i = 1, \ldots, N, \tag{1.53}$$

which is nothing but a linear system,

$$AU = b, \tag{1.54}$$

where the entries in $A$ and $b$ are given by

$$A_{ij} = a(\phi_j, \hat{\phi}_i), \\ b_i = L(\hat{\phi}_i). \tag{1.55}$$

The examples so far have specified the left- and right-hand side of the variational formulation and then asked FEniCS to assemble the linear system and solve it. An alternative to is explicitly call functions for assembling the coefficient matrix $A$ and the right-side vector $b$, and then solve the linear system $AU = b$ with respect to the $U$ vector. Instead of `solve(a == L, u, b)` we now write

*Python code*

```
A = assemble(a)
b = assemble(L)
bc.apply(A, b)
u = Function(V)
U = u.vector()
solve(A, U, b)
```

The variables `a` and `L` are as before; that is, `a` refers to the bilinear form involving a `TrialFunction` object (say `u`) and a `TestFunction` object (`v`), and `L` involves a `TestFunction` object (`v`). From `a` and `L`, the `assemble` function can compute the matrix elements $A_{i,j}$ and the vector elements $b_i$.

The matrix $A$ and vector $b$ are first assembled without incorporating essential (Dirichlet) boundary conditions. Thereafter, the `bc.apply(A, b)` call performs the necessary modifications to the linear system. When we have multiple Dirichlet conditions stored in a list `bcs`, as explained in Section 1.1.14, we must apply each condition in `bcs` to the system:

*Python code*

```
# bcs is a list of DirichletBC objects
for bc in bcs:
    bc.apply(A, b)
```

There is an alternative function `assemble_system` that can assemble the system and take boundary conditions into account in one call:

*Python code*

```
A, b = assemble_system(a, L, bcs)
```

The `assemble_system` function incorporates the boundary conditions in the element matrices and vectors, prior to assembly. The conditions are also incorporated in a symmetric way to preserve eventual symmetry of the coefficient matrix. With `bc.apply(A,b)` the matrix `A` is modified in an unsymmetric way.

Note that the solution `u` is, as before, a `Function` object. The degrees of freedom, $U = A^{-1}b$, are filled into `u`'s `Vector` object (`u.vector()`) by the `solve` function.

The object `A` is of type `Matrix`, while `b` and `u.vector()` are of type `Vector`. We may convert the matrix and vector data to `numpy` arrays by calling the `array()` method as shown before. If you wonder how essential boundary conditions are incorporated in the linear system, you can print out `A` and `b` before and after the `bc.apply(A, b)` call:

*Python code*

```
if mesh.num_cells() < 16:  # print for small meshes only
    print A.array()
    print b.array()
bc.apply(A, b)
```

```
if mesh.num_cells() < 16:
    print A.array()
    print b.array()
```

With access to the elements in `A` as a `numpy` array we can easily do computations on this matrix, such as computing the eigenvalues (using the `numpy.linalg.eig` function). We can alternatively dump `A` and `b` to file in MATLAB format and invoke MATLAB or Octave to analyze the linear system. Dumping the arrays `A` and `b` to MATLAB format is done by

*Python code*

```
import scipy.io
scipy.io.savemat("Ab.mat", {"A": A, "b": b})
```

Writing `load Ab.mat` in MATLAB or Octave will then make the variables `A` and `b` available for computations.

Matrix processing in Python or MATLAB/Octave is only feasible for small PDE problems since the `numpy` arrays or matrices in MATLAB file format are dense matrices. DOLFIN also has an interface to the eigensolver package SLEPc, which is a preferred tool for computing the eigenvalues of large, sparse matrices of the type encountered in PDE problems (see `demo/la/eigenvalue` in the DOLFIN source code tree for a demo).

A complete code where the linear system $AU = b$ is explicitly assembled and solved is found in the file `dn3_p2D.py` in the directory `stationary/poisson`. This code solves the same problem as in `dn2_p2D.py` (Section 1.1.14). For small linear systems, the program writes out `A` and `b` before and after incorporation of essential boundary conditions and illustrates the difference between `assemble` and `assemble_system`. The reader is encouraged to run the code for a $2 \times 1$ mesh (`UnitSquare(2, 1)` and study the output of `A`.

By default, `solve(A, U, b)` applies sparse LU decomposition as solver. Specification of an iterative solver and preconditioner is done through two optional arguments:

*Python code*

```
solve(A, U, b, "cg", "ilu")
```

Appropriate names of solvers and preconditioners are found in Section 1.7.4.

To control tolerances in the stopping criterion and the maximum number of iterations, one can explicitly form a `KrylovSolver` object and set items in its `parameters` attribute (see Section 1.1.5):

*Python code*

```
solver = KrylovSolver("cg", "ilu")
solver.parameters["absolute_tolerance"] = 1E-7
solver.parameters["relative_tolerance"] = 1E-4
solver.parameters["maximum_iterations"] = 1000
u = Function(V)
U = u.vector()
set_log_level(DEBUG)
solver.solve(A, U, b)
```

The program `dn4_p2D.py` is a modification of `dn3_p2D.py` illustrating this latter approach.

The choice of start vector for the iterations in a linear solver is often important. With the `solve(A, U, b)` function the start vector is the vector `U` we feed in for the solution. A start vector with random numbers in the interval $[-100, 100]$ can be computed as

*Python code*

```
n = u.vector().array().size
U = u.vector()
```

```
U[:] = numpy.random.uniform(-100, 100, n)
solver.parameters['nonzero_initial_guess'] = True
solver.solve(A, U, b)
```

Note that we must turn off the default behavior of setting the start vector ("initial guess") to zero. A random start vector is included in the `dn4_p2D.py` code.

Creating the linear system explicitly in a program can have some advantages in more advanced problem settings. For example, $A$ may be constant throughout a time-dependent simulation, so we can avoid recalculating $A$ at every time level and save a significant amount of simulation time. Sections 1.3.2 and 1.3.3 deal with this topic in detail.

### 1.1.16 Parameterizing the number of space dimensions

FEniCS makes it is easy to write a unified simulation code that can operate in 1D, 2D, and 3D. We will conveniently make use of this feature in forthcoming examples. As an appetizer, go back to the introductory program `d1_p2D.py` in the `stationary/poisson` directory and change the mesh construction from `UnitSquare(6, 4)` to `UnitCube(6, 4, 5)`. Now the domain is the unit cube with $6 \times 4 \times 5$ cells. Run the program and observe that we can solve a 3D problem without any other modifications (!). The visualization allows you rotate to the cube and observe the function values as colors on the boundary.

The forthcoming material introduces some convenient technicalities such that the same program can run in 1D, 2D, or 3D without any modifications. Consider the simple problem

$$u''(x) = 2 \text{ in } [0,1], \quad u(0) = 0,\ u(1) = 1, \tag{1.56}$$

with exact solution $u(x) = x^2$. Our aim is to formulate and solve this problem in a 2D and a 3D domain as well. We may generalize the domain $[0,1]$ to a box of any size in the $y$ and $z$ directions and pose homogeneous Neumann conditions $\partial u/\partial n = 0$ at all additional boundaries $y = \text{const}$ and $z = \text{const}$ to ensure that $u$ only varies with $x$. For example, let us choose a unit hypercube as domain: $\Omega = [0,1]^d$, where $d$ is the number of space dimensions. The generalized $d$-dimensional Poisson problem then reads

$$\begin{aligned} \Delta u &= 2 \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \Gamma_0, \\ u &= 1 \quad \text{on } \Gamma_1, \\ \frac{\partial u}{\partial n} &= 0 \quad \text{on } \partial\Omega \setminus (\Gamma_0 \cup \Gamma_1), \end{aligned} \tag{1.57}$$

where $\Gamma_0$ is the side of the hypercube where $x = 0$, and where $\Gamma_1$ is the side where $x = 1$.

Implementing (1.57) for any $d$ is no more complicated than solving a problem with a specific number of dimensions. The only non-trivial part of the code is actually to define the mesh. We use the command-line to provide user-input to the program. The first argument can be the degree of the polynomial in the finite element basis functions. Thereafter, we supply the cell divisions in the various spatial directions. The number of command-line arguments will then imply the number of space dimensions. For example, writing `3 10 3 4` on the command-line means that we want to approximate $u$ by piecewise polynomials of degree 3, and that the domain is a three-dimensional cube with $10 \times 3 \times 4$ divisions in the $x$, $y$, and $z$ directions, respectively. Each of the $10 \times 3 \times 4 = 120$ boxes will be divided into six tetrahedra. The Python code can be quite compact:

*Python code*

```
degree = int(sys.argv[1])
divisions = [int(arg) for arg in sys.argv[2:]]
d = len(divisions)
domain_type = [UnitInterval, UnitSquare, UnitCube]
```

```
mesh = domain_type[d-1](*divisions)
V = FunctionSpace(mesh, "Lagrange", degree)
```

First note that although `sys.argv[2:]` holds the divisions of the mesh, all elements of the list `sys.argv[2:]` are string objects, so we need to explicitly convert each element to an integer. The construction `domain_type[d-1]` will pick the right name of the object used to define the domain and generate the mesh. Moreover, the argument `*divisions` sends each component of the list `divisions` as a separate argument. For example, in a 2D problem where `divisions` has two elements, the statement

*Python code*

```
mesh = domain_type[d-1](*divisions)
```

is equivalent to

*Python code*

```
mesh = UnitSquare(divisions[0], divisions[1])
```

The next part of the program is to set up the boundary conditions. Since the Neumann conditions have $\partial u/\partial n = 0$ we can omit the boundary integral from the weak form. We then only need to take care of Dirichlet conditions at two sides:

*Python code*

```
tol = 1E-14   # tolerance for coordinate comparisons
def Dirichlet_boundary0(x, on_boundary):
    return on_boundary and abs(x[0]) < tol

def Dirichlet_boundary1(x, on_boundary):
    return on_boundary and abs(x[0] - 1) < tol

bc0 = DirichletBC(V, Constant(0), Dirichlet_boundary0)
bc1 = DirichletBC(V, Constant(1), Dirichlet_boundary1)
bcs = [bc0, bc1]
```

Note that this code is independent of the number of space dimensions. So are the statements defining and solving the variational problem:

*Python code*

```
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-2)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

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

The complete code is found in `paD.py` (Poisson problem in any–D).

If we want to parameterize the direction in which $u$ varies, say by the space direction number `e`, we only need to replace `x[0]` in the code by `x[e]`. The parameter `e` could be given as a second command-line argument. The reader is encouraged to perform this modification.

## 1.2 Nonlinear problems

Now we shall address how to solve nonlinear PDEs in FEniCS. Our sample PDE for implementation is taken as a nonlinear Poisson equation:

$$-\nabla \cdot (q(u)\nabla u) = f. \tag{1.58}$$

The coefficient $q(u)$ makes the equation nonlinear (unless $q(u)$ is constant in $u$).

To be able to easily verify our implementation, we choose the domain, $q(u)$, $f$, and the boundary conditions such that we have a simple, exact solution $u$. Let $\Omega$ be the unit hypercube $[0,1]^d$ in $d$ dimensions, $q(u) = (1+u)^m$, $f = 0$, $u = 0$ for $x_0 = 0$, $u = 1$ for $x_0 = 1$, and $\partial u/\partial n = 0$ at all other boundaries $x_i = 0$ and $x_i = 1$, $i = 1, \ldots, d-1$. The coordinates are now represented by the symbols $x_0, \ldots, x_{d-1}$. The exact solution is then

$$u(x_0, \ldots, x_d) = \left((2^{m+1}-1)x_0 + 1\right)^{1/(m+1)} - 1. \tag{1.59}$$

The variational formulation of our model problem reads: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}, \tag{1.60}$$

where

$$F(u; v) = \int_\Omega q(u)\nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.61}$$

and

$$\begin{aligned} \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } x_0 = 1\}, \\ V &= \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } v = 1 \text{ on } x_0 = 1\}. \end{aligned} \tag{1.62}$$

The discrete problem arises as usual by restricting $V$ and $\hat{V}$ to a pair of discrete spaces. As usual, we omit any subscript on discrete spaces and simply say $V$ and $\hat{V}$ are chosen finite dimensional according to some mesh and element type. The nonlinear problem then reads: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}, \tag{1.63}$$

with $u = \sum_{j=1}^N U_j \phi_j$. Since $F$ is a nonlinear function of $u$, (1.63) gives rise to a system of nonlinear algebraic equations. From now on the interest is only in the discrete problem, and as mentioned in Section 1.1.2, we simply write $u$ instead of $u_h$ to get a closer resemblance in notation between the mathematics and the Python code. When the exact solution needs to be distinguished, we denote it by $u_\mathrm{e}$.

FEniCS can be used in alternative ways for solving a nonlinear PDE problem. We shall in the following subsections go through four solution strategies: 1) a simple Picard-type iteration, 2) a Newton method at the algebraic level, 3) a Newton method at the PDE level, and 4) an automatic approach where FEniCS attacks the nonlinear variational problem directly. The "black box" strategy 4) is definitely the simplest one from a programmer's point of view, but the others give more control of the solution process for nonlinear equations (which also has some pedagogical advantages).

### 1.2.1 Picard iteration

Picard iteration is an easy way of handling nonlinear PDEs: we simply use a known, previous solution in the nonlinear terms so that these terms become linear in the unknown $u$. The strategy is also known

as the method of successive substitutions. For our particular problem, we use a known, previous solution in the coefficient $q(u)$. More precisely, given a solution $u^k$ from iteration $k$, we seek a new (hopefully improved) solution $u^{k+1}$ in iteration $k+1$ such that $u^{k+1}$ solves the *linear problem*

$$\nabla \cdot \left( q(u^k) \nabla u^{k+1} \right) = 0, \quad k = 0, 1, \ldots \tag{1.64}$$

The iterations require an initial guess $u^0$. The hope is that $u^k \to u$ as $k \to \infty$, and that $u^{k+1}$ is sufficiently close to the exact solution $u$ of the discrete problem after just a few iterations.

We can easily formulate a variational problem for $u^{k+1}$ from Equation (1.64). Equivalently, we can approximate $q(u)$ by $q(u^k)$ in (1.61) to obtain the same linear variational problem. In both cases, the problem consists of seeking $u^{k+1} \in V$ such that

$$\tilde{F}(u^{k+1}; v) = 0 \quad \forall\, v \in \hat{V}, \quad k = 0, 1, \ldots, \tag{1.65}$$

with

$$\tilde{F}(u^{k+1}; v) = \int_\Omega q(u^k) \nabla u^{k+1} \cdot \nabla v \, \mathrm{d}x. \tag{1.66}$$

Since this is a linear problem in the unknown $u^{k+1}$, we can equivalently use the formulation

$$a(u^{k+1}, v) = L(v), \tag{1.67}$$

with

$$a(u, v) = \int_\Omega q(u^k) \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{1.68}$$

$$L(v) = 0. \tag{1.69}$$

The iterations can be stopped when $\epsilon \equiv ||u^{k+1} - u^k|| < \text{tol}$, where tol is small, say $10^{-5}$, or when the number of iterations exceed some critical limit. The latter case will pick up divergence of the method or unacceptable slow convergence.

In the solution algorithm we only need to store $u^k$ and $u^{k+1}$, called `u_k` and `u` in the code below. The algorithm can then be expressed as follows:

*Python code*

```
def q(u):
    return (1+u)**m

# Define variational problem for Picard iteration
u = TrialFunction(V)
v = TestFunction(V)
u_k = interpolate(Constant(0.0), V)  # previous (known) u
a = inner(q(u_k)*nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = f*v*dx

# Picard iterations
u = Function(V)     # new unknown function
eps = 1.0           # error measure ||u-u_k||
tol = 1.0E-5        # tolerance
iter = 0            # iteration counter
maxiter = 25        # max no of iterations allowed
while eps > tol and iter < maxiter:
    iter += 1
    solve(a == L, u, bcs)
```

```
diff = u.vector().array() - u_k.vector().array()
eps = numpy.linalg.norm(diff, ord=numpy.Inf)
print "iter=%d: norm=%g" % (iter, eps)
u_k.assign(u)   # update for next iteration
```

We need to define the previous solution in the iterations, `u_k`, as a finite element function so that `u_k` can be updated with `u` at the end of the loop. We may create the initial `Function` `u_k` by interpolating an `Expression` or a `Constant` to the same vector space as `u` lives in (`V`).

In the code above we demonstrate how to use `numpy` functionality to compute the norm of the difference between the two most recent solutions. Here we apply the maximum norm ($\ell_\infty$ norm) on the difference of the solution vectors (`ord=1` and `ord=2` give the $\ell_1$ and $\ell_2$ vector norms – other norms are possible for `numpy` arrays, see `pydoc numpy.linalg.norm`).

The file `picard_np.py` (Picard iteration for a nonlinear Poisson problem) contains the complete code for this problem. The implementation is $d$ dimensional, with mesh construction and setting of Dirichlet conditions as explained in Section 1.1.16. For a $33 \times 33$ grid with $m = 2$ we need 9 iterations for convergence when the tolerance is $10^{-5}$.

### 1.2.2 A Newton method at the algebraic level

After having discretized our nonlinear PDE problem, we may use Newton's method to solve the system of nonlinear algebraic equations. From the continuous variational problem (1.60), the discrete version (1.63) results in a system of equations for the unknown parameters $U_1, \ldots, U_N$ (by inserting $u = \sum_{j=1}^N U_j\phi_j$ and $v = \hat{\phi}_i$ in (1.63)):

$$F_i(U_1, \ldots, U_N) \equiv \sum_{j=1}^N \int_\Omega \left( q\left(\sum_{\ell=1}^N U_\ell \phi_\ell\right) \nabla\phi_j U_j \right) \cdot \nabla\hat{\phi}_i \, \mathrm{d}x = 0, \quad i = 1, \ldots, N. \tag{1.70}$$

Newton's method for the system $F_i(U_1, \ldots, U_j) = 0$, $i = 1, \ldots, N$ can be formulated as

$$\sum_{j=1}^N \frac{\partial}{\partial U_j} F_i(U_1^k, \ldots, U_N^k)\delta U_j = -F_i(U_1^k, \ldots, U_N^k), \quad i = 1, \ldots, N, \tag{1.71}$$

$$U_j^{k+1} = U_j^k + \omega\delta U_j, \quad j = 1, \ldots, N, \tag{1.72}$$

where $\omega \in [0, 1]$ is a relaxation parameter, and $k$ is an iteration index. An initial guess $u^0$ must be provided to start the algorithm. The original Newton method has $\omega = 1$, but in problems where it is difficult to obtain convergence, so-called *under-relaxation* with $\omega < 1$ may help.

We need, in a program, to compute the Jacobian matrix $\partial F_i/\partial U_j$ and the right-hand side vector $-F_i$. Our present problem has $F_i$ given by (1.70). The derivative $\partial F_i/\partial U_j$ becomes

$$\int_\Omega \left[ q'(\sum_{\ell=1}^N U_\ell^k \phi_\ell)\phi_j \nabla(\sum_{j=1}^N U_j^k \phi_j) \cdot \nabla\hat{\phi}_i + q\left(\sum_{\ell=1}^N U_\ell^k \phi_\ell\right) \nabla\phi_j \cdot \nabla\hat{\phi}_i \right] \mathrm{d}x. \tag{1.73}$$

The following results were used to obtain (1.73):

$$\frac{\partial u}{\partial U_j} = \frac{\partial}{\partial U_j} \sum_{j=1}^N U_j\phi_j = \phi_j, \quad \frac{\partial}{\partial U_j}\nabla u = \nabla\phi_j, \quad \frac{\partial}{\partial U_j} q(u) = q'(u)\phi_j. \tag{1.74}$$

We can reformulate the Jacobian matrix in (1.73) by introducing the short notation $u^k = \sum_{j=1}^N U_j^k \phi_j$:

$$\frac{\partial F_i}{\partial U_j} = \int_\Omega \left[ q'(u^k)\phi_j \nabla u^k \cdot \nabla \hat{\phi}_i + q(u^k)\nabla \phi_j \cdot \nabla \hat{\phi}_i \right] \mathrm{d}x. \tag{1.75}$$

In order to make FEniCS compute this matrix, we need to formulate a corresponding variational problem. Looking at the linear system of equations in Newton's method,

$$\sum_{j=1}^N \frac{\partial F_i}{\partial U_j} \delta U_j = -F_i, \quad i = 1, \ldots, N,$$

we can introduce $v$ as a general test function replacing $\hat{\phi}_i$, and we can identify the unknown $\delta u = \sum_{j=1}^N \delta U_j \phi_j$. From the linear system we can now go "backwards" to construct the corresponding discrete weak form

$$\int_\Omega \left[ q'(u^k)\delta u \nabla u^k \cdot \nabla v + q(u^k)\nabla \delta u \cdot \nabla v \right] \mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \, \mathrm{d}x. \tag{1.76}$$

Equation (1.76) fits the standard form $a(\delta u, v) = L(v)$ with

$$a(\delta u, v) = \int_\Omega \left[ q'(u^k)\delta u \nabla u^k \cdot \nabla v + q(u^k)\nabla \delta u \cdot \nabla v \right] \mathrm{d}x \tag{1.77}$$

$$L(v) = -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \, \mathrm{d}x. \tag{1.78}$$

Note the important feature in Newton's method that the previous solution $u^k$ replaces $u$ in the formulas when computing the matrix $\partial F_i/\partial U_j$ and vector $F_i$ for the linear system in each Newton iteration.

We now turn to the implementation. To obtain a good initial guess $u^0$, we can solve a simplified, linear problem, typically with $q(u) = 1$, which yields the standard Laplace equation $\Delta u^0 = 0$. The recipe for solving this problem appears in Sections 1.1.2, 1.1.3, and 1.1.13. The code for computing $u^0$ becomes as follows:

*Python code*

```
tol = 1E-14
def left_boundary(x, on_boundary):
    return on_boundary and abs(x[0]) < tol

def right_boundary(x, on_boundary):
    return on_boundary and abs(x[0]-1) < tol

Gamma_0 = DirichletBC(V, Constant(0.0), left_boundary)
Gamma_1 = DirichletBC(V, Constant(1.0), right_boundary)
bcs = [Gamma_0, Gamma_1]

# Define variational problem for initial guess (q(u)=1, m=0)
u = TrialFunction(V)
v = TestFunction(V)
a = inner(nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = f*v*dx
A, b = assemble_system(a, L, bcs)
u_k = Function(V)
U_k = u_k.vector()
solve(A, U_k, b)
```

Here, `u_k` denotes the solution function for the previous iteration, so that the solution after each Newton iteration is `u = u_k + omega*du`. Initially, `u_k` is the initial guess we call $u^0$ in the mathematics.

The Dirichlet boundary conditions for the problem to be solved in each Newton iteration are somewhat different than the conditions for $u$. Assuming that $u^k$ fulfills the Dirichlet conditions for $u$, $\delta u$ must be zero at the boundaries where the Dirichlet conditions apply, in order for $u^{k+1} = u^k + \omega\delta u$ to fulfill the right Dirichlet values. We therefore define an additional list of Dirichlet boundary conditions objects for $\delta u$:

*Python code*

```
Gamma_0_du = DirichletBC(V, Constant(0), left_boundary)
Gamma_1_du = DirichletBC(V, Constant(0), right_boundary)
bcs_du = [Gamma_0_du, Gamma_1_du]
```

The nonlinear coefficient and its derivative must be defined before coding the weak form of the Newton system:

*Python code*

```
def q(u):
    return (1+u)**m

def Dq(u):
    return m*(1+u)**(m-1)

du = TrialFunction(V) # u = u_k + omega*du
a = inner(q(u_k)*nabla_grad(du), nabla_grad(v))*dx + \
    inner(Dq(u_k)*du*nabla_grad(u_k), nabla_grad(v))*dx
L = -inner(q(u_k)*nabla_grad(u_k), nabla_grad(v))*dx
```

The Newton iteration loop is very similar to the Picard iteration loop in Section 1.2.1:

*Python code*

```
du = Function(V)
u  = Function(V)  # u = u_k + omega*du
omega = 1.0       # relaxation parameter
eps = 1.0
tol = 1.0E-5
iter = 0
maxiter = 25
while eps > tol and iter < maxiter:
    iter += 1
    A, b = assemble_system(a, L, bcs_du)
    solve(A, du.vector(), b)
    eps = numpy.linalg.norm(du.vector().array(), ord=numpy.Inf)
    print "Norm:", eps
    u.vector()[:] = u_k.vector() + omega*du.vector()
    u_k.assign(u)
```

There are other ways of implementing the update of the solution as well:

*Python code*

```
u.assign(u_k)  # u = u_k
u.vector().axpy(omega, du.vector())

# or
u.vector()[:] += omega*du.vector()
```

The `axpy(a, y)` operation adds a scalar `a` times a `Vector` `y` to a `Vector` object. It is usually a fast operation calling up an optimized BLAS routine for the calculation.

Mesh construction for a $d$-dimensional problem with arbitrary degree of the Lagrange elements can be done as explained in Section 1.1.16. The complete program appears in the file `alg_newton_np.py`.

### *1.2.3 A Newton method at the PDE level*

Although Newton's method in PDE problems is normally formulated at the linear algebra level; that is, as a solution method for systems of nonlinear algebraic equations, we can also formulate the method at the PDE level. This approach yields a linearization of the PDEs before they are discretized. FEniCS users will probably find this technique simpler to apply than the more standard method of Section 1.2.2.

Given an approximation to the solution field, $u^k$, we seek a perturbation $\delta u$ so that

$$u^{k+1} = u^k + \delta u \tag{1.79}$$

fulfills the nonlinear PDE. However, the problem for $\delta u$ is still nonlinear and nothing is gained. The idea is therefore to assume that $\delta u$ is sufficiently small so that we can linearize the problem with respect to $\delta u$. Inserting $u^{k+1}$ in the PDE, linearizing the $q$ term as

$$q(u^{k+1}) = q(u^k) + q'(u^k)\delta u + \mathcal{O}((\delta u)^2) \approx q(u^k) + q'(u^k)\delta u, \tag{1.80}$$

and dropping other nonlinear terms in $\delta u$, we get

$$\nabla\cdot\left(q(u^k)\nabla u^k\right) + \nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = 0.$$

We may collect the terms with the unknown $\delta u$ on the left-hand side,

$$\nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = -\nabla\cdot\left(q(u^k)\nabla u^k\right), \tag{1.81}$$

The weak form of this PDE is derived by multiplying by a test function $v$ and integrating over $\Omega$, integrating the second-order derivatives by parts:

$$\int_\Omega \left(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\right)\,\mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x. \tag{1.82}$$

The variational problem reads: find $\delta u \in V$ such that $a(\delta u, v) = L(v)$ for all $v \in \hat{V}$, where

$$a(\delta u, v) = \int_\Omega \left(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\right)\,\mathrm{d}x, \tag{1.83}$$

$$L(v) = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x. \tag{1.84}$$

The function spaces $V$ and $\hat{V}$, being continuous or discrete, are as in the linear Poisson problem from Section 1.1.2.

We must provide some initial guess, e.g., the solution of the PDE with $q(u) = 1$. The corresponding weak form $a_0(u^0, v) = L_0(v)$ has

$$a_0(u, v) = \int_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x, \quad L(v) = 0. \tag{1.85}$$

Thereafter, we enter a loop and solve $a(\delta u, v) = L(v)$ for $\delta u$ and compute a new approximation $u^{k+1} = u^k + \delta u$. Note that $\delta u$ is a correction, so if $u^0$ satisfies the prescribed Dirichlet conditions on some part $\Gamma_\mathrm{D}$ of the boundary, we must demand $\delta u = 0$ on $\Gamma_\mathrm{D}$.

Looking at (1.83) and (1.84), we see that the variational form is the same as for the Newton method at the algebraic level in Section 1.2.2. Since Newton's method at the algebraic level required some "backward" construction of the underlying weak forms, FEniCS users may prefer Newton's method at the PDE level, which is more straightforward. There is seemingly no need for differentiations to derive a Jacobian matrix, but a mathematically equivalent derivation is done when nonlinear terms are linearized using the first two Taylor series terms and when products in the perturbation $\delta u$ are neglected.

The implementation is identical to the one in Section 1.2.2 and is found in the file `pde_newton_np.py`. The reader is encouraged to go through this code to be convinced that the present method actually ends up with the same program as needed for the Newton method at the linear algebra level in Section 1.2.2.

### 1.2.4 Solving the nonlinear variational problem directly

The previous hand-calculations and manual implementation of Picard or Newton methods can be automated by tools in FEniCS. In a nutshell, one can just write

*Python code*

```
problem = NonlinearVariationalProblem(F, u, bcs, J)
solver  = NonlinearVariationalSolver(problem)
solver.solve()
```

where `F` corresponds to the nonlinear form $F(u;v)$, `u` is the unknown `Function` object, `bcs` represents the essential boundary conditions (list of `DirichletBC` objects), and `J` is a variational form for the Jacobian of `F`.

Let us explain in detail how to use the built-in tools for nonlinear variational problems and their solution. The `F` form corresponding to (1.61) is straightforwardly defined as follows, assuming `q(u)` is coded as a Python function:

*Python code*

```
v  = TestFunction(V)
u_ = Function(V)  # the unknown
F  = inner(q(u_)*nabla_grad(u_), nabla_grad(v))*dx
```

Note here that `u_` is a `Function` (not a `TrialFunction`). An alternative and perhaps more intuitive formula for $F$ is to define $F(u;v)$ directly in terms of a trial function for $u$ and a test function for $v$, and then create the proper `F` by

*Python code*

```
u  = TrialFunction(V)
v  = TestFunction(V)
F  = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
u_ = Function(V)  # most recently computed solution
F  = action(F, u_)
```

The latter statement is equivalent to $F(u = u_-;v)$, where $u_-$ is an existing finite element function representing the most recently computed approximation to the solution.

The Jacobian or derivative $J$ (`J`) of $F$ (`F`) is formally the Gateaux derivative $DF(u^k;\delta u,v)$ of $F(u;v)$ at $u = u^k$ in the direction of $\delta u$. Technically, this Gateaux derivative is derived by computing

$$\lim_{\epsilon\to 0}\frac{d}{d\epsilon}F_i(u^k + \epsilon\delta u;v) \tag{1.86}$$

The $\delta u$ is now the trial function and $u^k$ is as usual the previous approximation to the solution $u$. We start with

$$\frac{d}{d\epsilon}\int_\Omega \nabla v\cdot\left(q(u^k+\epsilon\delta u)\nabla(u^k+\epsilon\delta u)\right)\,\mathrm{d}x \tag{1.87}$$

and obtain

$$\int_\Omega \nabla v\cdot\left[q'(u^k+\epsilon\delta u)\delta u\nabla(u^k+\epsilon\delta u)+q(u^k+\epsilon\delta u)\nabla\delta u\right]\,\mathrm{d}x, \tag{1.88}$$

which leads to

$$\int_\Omega \nabla v\cdot\left[q'(u^k)\delta u\nabla(u^k)+q(u^k)\nabla\delta u\right]\,\mathrm{d}x, \tag{1.89}$$

as $\epsilon\to 0$. This last expression is the Gateaux derivative of $F$. We may use $J$ or $a(\delta u,v)$ for this derivative, the latter having the advantage that we easily recognize the expression as a bilinear form. However, in the forthcoming code examples `J` is used as variable name for the Jacobian.

The specification of `J` goes as follows if `du` is the `TrialFunction`:

*Python code*

```
du = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u_)*nabla_grad(u_), nabla_grad(v))*dx

J = inner(q(u_)*nabla_grad(du), nabla_grad(v))*dx + \
    inner(Dq(u_)*du*nabla_grad(u_), nabla_grad(v))*dx
```

The alternative specification of `F`, with `u` as `TrialFunction`, leads to

*Python code*

```
u  = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
F  = action(F, u_)

J = inner(q(u_)*nabla_grad(u), nabla_grad(v))*dx + \
    inner(Dq(u_)*u*nabla_grad(u_), nabla_grad(v))*dx
```

The UFL language, used to specify weak forms, supports differentiation of forms. This feature facilitates automatic *symbolic* computation of the Jacobian `J` by calling the function `derivative` with `F`, the most recently computed solution (`Function`), and the unknown (`TrialFunction`) as parameters:

*Python code*

```
du = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u_)*nabla_grad(u_), nabla_grad(v))*dx

J  = derivative(F, u_, du)  # Gateaux derivative in dir. of du
```

or

*Python code*

```
u  = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
F  = action(F, u_)
```

```
J  = derivative(F, u_, u)   # Gateaux derivative in dir. of u
```

The `derivative` function is obviously very convenient in problems where differentiating `F` by hand implies lengthy calculations.

The preferred implementation of `F` and `J`, depending on whether `du` or `u` is the `TrialFunction` object, is a matter of personal taste. Derivation of the Gateaux derivative by hand, as shown above, is most naturally matched by an implementation where `du` is the `TrialFunction`, while use of automatic symbolic differentiation through the `derivative` function is most naturally matched with an implementation where `u` is the `TrialFunction`. We have implemented both approaches in two files: `vp1_np.py` with `u` as `TrialFunction`, and `vp2_np.py` with `du` as `TrialFunction`. Both files are located in the `stationary/nonlinear_poisson` directory. The first command-line argument determines if the Jacobian is to be automatically derived or computed from the hand-derived formula.

The following code defines the nonlinear variational problem and an associated solver based on Newton's method. We also demonstrate how key parameters in Newton's method can be set, as well as the choice of solver and preconditioner, and associated parameters, for the linear system occurring in the Newton iteration.

*Python code*

```
problem = NonlinearVariationalProblem(F, u_, bcs, J)
solver  = NonlinearVariationalSolver(problem)

prm = solver.parameters
prm["newton_solver"]["absolute_tolerance"] = 1E-8
prm["newton_solver"]["relative_tolerance"] = 1E-7
prm["newton_solver"]["maximum_iterations"] = 25
prm["newton_solver"]["relaxation_parameter"] = 1.0
if iterative_solver:
    prm["linear_solver"] = "gmres"
    prm["preconditioner"] = "ilu"
    prm["krylov_solver"]["absolute_tolerance"] = 1E-9
    prm["krylov_solver"]["relative_tolerance"] = 1E-7
    prm["krylov_solver"]["maximum_iterations"] = 1000
    prm["krylov_solver"]["gmres"]["restart"] = 40
    prm["krylov_solver"]["preconditioner"]["ilu"]["fill_level"] = 0
set_log_level(PROGRESS)

solver.solve()
```

A list of available parameters and their default values can as usual be printed by calling `info(prm, True)`. The `u_` we feed to the nonlinear variational problem object is filled with the solution by the call `solver.solve()`.

## 1.3 Time-dependent problems

The examples in Section 1.1 illustrate that solving linear, stationary PDE problems with the aid of FEniCS is easy and requires little programming. That is, FEniCS automates the spatial discretization by the finite element method. The solution of nonlinear problems, as we showed in Section 1.2, can also be automated (see Section 1.2.4), but many scientists will prefer to code the solution strategy of the nonlinear problem themselves and experiment with various combinations of strategies in difficult

problems. Time-dependent problems are somewhat similar in this respect: we have to add a time discretization scheme, which is often quite simple, making it natural to explicitly code the details of the scheme so that the programmer has full control. We shall explain how easily this is accomplished through examples.

### *1.3.1 A diffusion problem and its discretization*

Our time-dependent model problem for teaching purposes is naturally the simplest extension of the Poisson problem into the time domain; that is, the diffusion problem

$$\frac{\partial u}{\partial t} = \Delta u + f \text{ in } \Omega, \text{ for } t > 0, \tag{1.90}$$

$$u = u_0 \text{ on } \partial\Omega, \text{ for } t > 0, \tag{1.91}$$

$$u = I \text{ at } t = 0. \tag{1.92}$$

Here, $u$ varies with space and time, e.g., $u = u(x, y, t)$ if the spatial domain $\Omega$ is two-dimensional. The source function $f$ and the boundary values $u_0$ may also vary with space and time. The initial condition $I$ is a function of space only.

A straightforward approach to solving time-dependent PDEs by the finite element method is to first discretize the time derivative by a finite difference approximation, which yields a recursive set of stationary problems, and then turn each stationary problem into a variational formulation.

Let superscript $k$ denote a quantity at time $t_k$, where $k$ is an integer counting time levels. For example, $u^k$ means $u$ at time level $k$. A finite difference discretization in time first consists in sampling the PDE at some time level, say $k$:

$$\frac{\partial}{\partial t}u^k = \Delta u^k + f^k. \tag{1.93}$$

The time-derivative can be approximated by a finite difference. For simplicity and stability reasons we choose a simple backward difference:

$$\frac{\partial}{\partial t}u^k \approx \frac{u^k - u^{k-1}}{\mathrm{d}t}, \tag{1.94}$$

where $\mathrm{d}t$ is the time discretization parameter. Inserting (1.94) in (1.93) yields

$$\frac{u^k - u^{k-1}}{\mathrm{d}t} = \Delta u^k + f^k. \tag{1.95}$$

This is our time-discrete version of the diffusion PDE (1.90). Reordering (1.95) so that $u^k$ appears on the left-hand side only, shows that (1.95) is a recursive set of spatial (stationary) problems for $u^k$ (assuming $u^{k-1}$ is known from computations at the previous time level):

$$u^0 = I, \tag{1.96}$$

$$u^k - \mathrm{d}t\Delta u^k = u^{k-1} + \mathrm{d}t f^k, \quad k = 1, 2, \ldots \tag{1.97}$$

Given $I$, we can solve for $u^0$, $u^1$, $u^2$, and so on.

We use a finite element method to solve the equations (1.96) and (1.97). This requires turning the equations into weak forms. As usual, we multiply by a test function $v \in \hat{V}$ and integrate second-derivatives by parts. Introducing the symbol $u$ for $u^k$ (which is natural in the program too), the resulting weak form can be conveniently written in the standard notation: $a_0(u, v) = L_0(v)$ for (1.96)

and $a(u,v) = L(v)$ for (1.97), where

$$a_0(u,v) = \int_\Omega uv \,\mathrm{d}x, \tag{1.98}$$

$$L_0(v) = \int_\Omega Iv \,\mathrm{d}x, \tag{1.99}$$

$$a(u,v) = \int_\Omega (uv + \mathrm{d}t\nabla u \cdot \nabla v) \,\mathrm{d}x, \tag{1.100}$$

$$L(v) = \int_\Omega \left(u^{k-1} + \mathrm{d}t f^k\right) v \,\mathrm{d}x. \tag{1.101}$$

The continuous variational problem is to find $u^0 \in V$ such that $a_0(u^0, v) = L_0(v)$ holds for all $v \in \hat{V}$, and then find $u^k \in V$ such that $a(u^k, v) = L(v)$ for all $v \in \hat{V}$, $k = 1, 2, \ldots$.

Approximate solutions in space are found by restricting the functional spaces $V$ and $\hat{V}$ to finite-dimensional spaces, exactly as we have done in the Poisson problems. We shall use the symbol $u$ for the finite element approximation at time $t_k$. In case we need to distinguish this space-time discrete approximation from the exact solution of the continuous diffusion problem, we use $u_\mathrm{e}$ for the latter. By $u^{k-1}$ we mean, from now on, the finite element approximation of the solution at time $t_{k-1}$.

Note that the forms $a_0$ and $L_0$ are identical to the forms met in Section 1.1.9, except that the test and trial functions are now scalar fields and not vector fields. Instead of solving (1.96) by a finite element method; that is, projecting $I$ onto $V$ via the problem $a_0(u,v) = L_0(v)$, we could simply interpolate $u^0$ from $I$. That is, if $u^0 = \sum_{j=1}^N U_j^0 \phi_j$, we simply set $U_j = I(x_j, y_j)$, where $(x_j, y_j)$ are the coordinates of node number $j$. We refer to these two strategies as computing the initial condition by either projecting $I$ or interpolating $I$. Both operations are easy to compute through one statement, using either the `project` or `interpolate` function.

### 1.3.2 Implementation

Our program needs to perform the time stepping explicitly, but can rely on FEniCS to easily compute $a_0$, $L_0$, $a$, and $L$, and solve the linear systems for the unknowns. We realize that $a$ does not depend on time, which means that its associated matrix also will be time independent. Therefore, it is wise to explicitly create matrices and vectors as in Section 1.1.15. The matrix $A$ arising from $a$ can be computed prior to the time stepping, so that we only need to compute the right-hand side $b$, corresponding to $L$, in each pass in the time loop. Let us express the solution procedure in algorithmic form, writing $u$ for the unknown spatial function at the new time level ($u^k$) and $u_1$ for the spatial solution at one earlier time level ($u^{k-1}$):

define Dirichlet boundary condition ($u_0$, Dirichlet boundary, etc.)
if $u_1$ is to be computed by projecting $I$:
    define $a_0$ and $L_0$
    assemble matrix $M$ from $a_0$ and vector $b$ from $L_0$
    solve $MU = b$ and store $U$ in $u_1$
else: (interpolation)
    let $u_1$ interpolate $I$
define $a$ and $L$
assemble matrix $A$ from $a$
set some stopping time $T$
$t = \mathrm{d}t$
while $t \leqslant T$
    assemble vector $b$ from $L$

apply essential boundary conditions
solve $AU = b$ for $U$ and store in $u$
$t \leftarrow t + \mathrm{d}t$
$u_1 \leftarrow u$ (be ready for next step)

Before starting the coding, we shall construct a problem where it is easy to determine if the calculations are correct. The simple backward time difference is exact for linear functions, so we decide to have a linear variation in time. Combining a second-degree polynomial in space with a linear term in time,

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{1.102}$$

yields a function whose computed values at the nodes may be exact, regardless of the size of the elements and $\mathrm{d}t$, as long as the mesh is uniformly partitioned. Inserting (1.102) in the PDE problem (1.90), it follows that $u_0$ must be given as (1.102) and that $f(x,y,t) = \beta - 2 - 2\alpha$ and $I(x,y) = 1 + x^2 + \alpha y^2$.

A new programming issue is how to deal with functions that vary in space *and time*, such as the boundary condition $u_0$ given by (1.102). A natural solution is to apply an `Expression` object with time $t$ as a parameter, in addition to the parameters $\alpha$ and $\beta$ (see Section 1.1.7 for `Expression` objects with parameters):

*Python code*

```
alpha = 3; beta = 1.2
u0 = Expression("1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t",
                alpha=alpha, beta=beta, t=0)
```

The time parameter can later be updated by assigning values to `u0.t`.

The essential boundary conditions, along the whole boundary in this case, are set in the usual way,

*Python code*

```
def boundary(x, on_boundary):  # define the Dirichlet boundary
    return on_boundary

bc = DirichletBC(V, u0, boundary)
```

We shall use `u` for the unknown $u$ at the new time level and `u_1` for $u$ at the previous time level. The initial value of `u_1`, implied by the initial condition on $u$, can be computed by either projecting or interpolating $I$. The $I(x,y)$ function is available in the program through `u0`, as long as `u0.t` is zero. We can then do

*Python code*

```
u_1 = interpolate(u0, V)
# or
u_1 = project(u0, V)
```

Note that we could, as an equivalent alternative to using `project`, define $a_0$ and $L_0$ as we did in Section 1.1.9 and solve the associated variational problem. To actually recover (1.102) to machine precision, it is important not to compute the discrete initial condition by projecting $I$, but by interpolating $I$ so that the nodal values are exact at $t = 0$ (projection results in approximative values at the nodes).

The definition of $a$ and $L$ goes as follows:

*Python code*

```
dt = 0.3      # time step

u = TrialFunction(V)
```

```
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)

a = u*v*dx + dt*inner(nabla_grad(u), nabla_grad(v))*dx
L = (u_1 + dt*f)*v*dx

A = assemble(a)   # assemble only once, before the time stepping
```

Finally, we perform the time stepping in a loop:

*Python code*

```
u = Function(V)   # the unknown at a new time level
T = 2             # total simulation time
t = dt

while t <= T:
    b = assemble(L)
    u0.t = t
    bc.apply(A, b)
    solve(A, u.vector(), b)

    t += dt
    u_1.assign(u)
```

Observe that `u0.t` must be updated before the `bc.apply` statement, to enforce computation of Dirichlet conditions at the current time level.

The time loop above does not contain any comparison of the numerical and the exact solution, which we must include in order to verify the implementation. As in many previous examples, we compute the difference between the array of nodal values of `u` and the array of the interpolated exact solution. The following code is to be included inside the loop, after `u` is found:

*Python code*

```
u_e = interpolate(u0, V)
maxdiff = numpy.abs(u_e.vector().array()-u.vector().array()).max()
print "Max error, t=%.2f: %-10.3f" % (t, maxdiff)
```

The right-hand side vector `b` must obviously be recomputed at each time level. With the construction `b = assemble(L)`, a new vector for `b` is allocated in memory in every pass of the time loop. It would be much more memory friendly to reuse the storage of the `b` we already have. This is easily accomplished by

*Python code*

```
b = assemble(L, tensor=b)
```

That is, we send in our previous `b`, which is then filled with new values and returned from `assemble`. Now there will be only a single memory allocation of the right-hand side vector. Before the time loop we set `b = None` such that `b` is defined in the first call to `assemble`.

The complete program code for this time-dependent case is stored in the file `d1_d2D.py` in the directory `transient/diffusion`.

### 1.3.3 *Avoiding assembly*

The purpose of this section is to present a technique for speeding up FEniCS simulators for time-dependent problems where it is possible to perform all assembly operations prior to the time loop. There are two costly operations in the time loop: assembly of the right-hand side $b$ and solution of

the linear system via the `solve` call. The assembly process involves work proportional to the number of degrees of freedom $N$, while the solve operation has a work estimate of $\mathcal{O}(N^\alpha)$, for some $\alpha \geqslant 1$. As $N \to \infty$, the solve operation will dominate for $\alpha > 1$, but for the values of $N$ typically used on smaller computers, the assembly step may still represent a considerable part of the total work at each time level. Avoiding repeated assembly can therefore contribute to a significant speed-up of a finite element code in time-dependent problems.

To see how repeated assembly can be avoided, we look at the $L(v)$ form in (1.101), which in general varies with time through $u^{k-1}$, $f^k$, and possibly also with $\mathrm{d}t$ if the time step is adjusted during the simulation. The technique for avoiding repeated assembly consists in expanding the finite element functions in sums over the basis functions $\phi_i$, as explained in Section 1.1.15, to identify matrix-vector products that build up the complete system. We have $u^{k-1} = \sum_{j=1}^N U_j^{k-1}\phi_j$, and we can expand $f^k$ as $f^k = \sum_{j=1}^N F_j^k\phi_j$. Inserting these expressions in $L(v)$ and using $v = \hat{\phi}_i$ result in

$$\begin{aligned}\int_\Omega \left(u^{k-1} + \mathrm{d}t f^k\right) v \,\mathrm{d}x &= \int_\Omega \left(\sum_{j=1}^N U_j^{k-1}\phi_j + \mathrm{d}t \sum_{j=1}^N F_j^k\phi_j\right)\hat{\phi}_i \,\mathrm{d}x, \\ &= \sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) U_j^{k-1} + \mathrm{d}t \sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) F_j^k.\end{aligned} \tag{1.103}$$

Introducing $M_{ij} = \int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x$, we see that the last expression can be written

$$\sum_{j=1}^N M_{ij}U_j^{k-1} + \mathrm{d}t \sum_{j=1}^N M_{ij}F_j^k, \tag{1.104}$$

which is nothing but two matrix-vector products,

$$MU^{k-1} + \mathrm{d}t MF^k, \tag{1.105}$$

if $M$ is the matrix with entries $M_{ij}$ and

$$U^{k-1} = (U_1^{k-1}, \ldots, U_N^{k-1})^\top, \tag{1.106}$$

and

$$F^k = (F_1^k, \ldots, F_N^k)^\top. \tag{1.107}$$

We have immediate access to $U^{k-1}$ in the program since that is the vector in the `u_1` function. The $F^k$ vector can easily be computed by interpolating the prescribed $f$ function (at each time level if $f$ varies with time). Given $M$, $U^{k-1}$, and $F^k$, the right-hand side $b$ can be calculated as

$$b = MU^{k-1} + \mathrm{d}t MF^k. \tag{1.108}$$

That is, no assembly is necessary to compute $b$.

The coefficient matrix $A$ can also be split into two terms. We insert $v = \hat{\phi}_i$ and $u^k = \sum_{j=1}^N U_j^k\phi_j$ in the expression (1.100) to get

$$\sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) U_j^k + \mathrm{d}t \sum_{j=1}^N \left(\int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x\right) U_j^k, \tag{1.109}$$

which can be written as a sum of matrix-vector products,

$$MU^k + \mathrm{d}tKU^k = (M + \mathrm{d}tK)U^k, \tag{1.110}$$

if we identify the matrix $M$ with entries $M_{ij}$ as above and the matrix $K$ with entries

$$K_{ij} = \int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x. \tag{1.111}$$

The matrix $M$ is often called the "mass matrix" while "stiffness matrix" is a common nickname for $K$. The associated bilinear forms for these matrices, as we need them for the assembly process in a FEniCS program, become

$$a_K(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.112}$$

$$a_M(u,v) = \int_\Omega uv \,\mathrm{d}x. \tag{1.113}$$

The linear system at each time level, written as $AU^k = b$, can now be computed by first computing $M$ and $K$, and then forming $A = M + \mathrm{d}tK$ at $t = 0$, while $b$ is computed as $b = MU^{k-1} + \mathrm{d}tMF^k$ at each time level.

The following modifications are needed in the `d1_d2D.py` program from the previous section in order to implement the new strategy of avoiding assembly at each time level:

1. Define separate forms $a_M$ and $a_K$
2. Assemble $a_M$ to $M$ and $a_K$ to $K$
3. Compute $A = M + \mathrm{d}t\,K$
4. Define $f$ as an `Expression`
5. Interpolate the formula for $f$ to a finite element function $F^k$
6. Compute $b = MU^{k-1} + \mathrm{d}tMF^k$

The relevant code segments become

*Python code*

```
# 1.
a_K = inner(nabla_grad(u), nabla_grad(v))*dx
a_M = u*v*dx

# 2. and 3.
M = assemble(a_M)
K = assemble(a_K)
A = M + dt*K

# 4.
f = Expression("beta - 2 - 2*alpha", beta=beta, alpha=alpha)

# 5. and 6.
while t <= T:
    f_k = interpolate(f, V)
    F_k = fk.vector()
    b = M*u_1.vector() + dt*M*F_k
```

The complete program appears in the file `d2_d2D.py`.

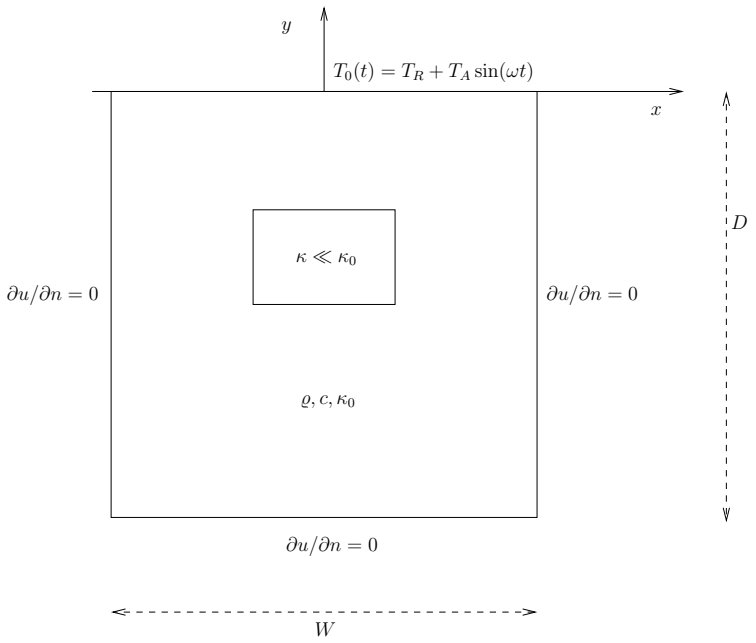



Figure 1.7: Sketch of a (2D) problem involving heating and cooling of the ground due to an oscillating surface temperature

### 1.3.4 A physical example

With the basic programming techniques for time-dependent problems from Sections 1.3.3–1.3.2 we are ready to attack more physically realistic examples. The next example concerns the question: How is the temperature in the ground affected by day and night variations at the earth's surface? We consider some box-shaped domain $\Omega$ in $d$ dimensions with coordinates $x_0, \ldots, x_{d-1}$ (the problem is meaningful in 1D, 2D, and 3D). At the top of the domain, $x_{d-1} = 0$, we have an oscillating temperature

$$T_0(t) = T_R + T_A \sin(\omega t), \tag{1.114}$$

where $T_R$ is some reference temperature, $T_A$ is the amplitude of the temperature variations at the surface, and $\omega$ is the frequency of the temperature oscillations. At all other boundaries we assume that the temperature does not change anymore when we move away from the boundary; that is, the normal derivative is zero. Initially, the temperature can be taken as $T_R$ everywhere. The heat conductivity properties of the soil in the ground may vary with space so we introduce a variable coefficient $\kappa$ reflecting this property. Figure 1.7 shows a sketch of the problem, with a small region where the heat conductivity is much lower.

The initial-boundary value problem for this problem reads

$$\varrho c \frac{\partial T}{\partial t} = \nabla \cdot (\kappa \nabla T) \text{ in } \Omega \times (0, t_{\text{stop}}], \tag{1.115}$$

$$T = T_0(t) \text{ on } \Gamma_0, \tag{1.116}$$

$$\frac{\partial T}{\partial n} = 0 \text{ on } \partial\Omega \backslash \Gamma_0, \tag{1.117}$$

$$T = T_R \text{ at } t = 0. \tag{1.118}$$

Here, $\varrho$ is the density of the soil, $c$ is the heat capacity, $\kappa$ is the thermal conductivity (heat conduction coefficient) in the soil, and $\Gamma_0$ is the surface boundary $x_{d-1} = 0$.

We use a $\theta$-scheme in time; that is, the evolution equation $\partial P/\partial t = Q(t)$ is discretized as

$$\frac{P^k - P^{k-1}}{\mathrm{d}t} = \theta Q^k + (1-\theta)Q^{k-1}, \tag{1.119}$$

where $\theta \in [0,1]$ is a weighting factor: $\theta = 1$ corresponds to the backward difference scheme, $\theta = 1/2$ to the Crank–Nicolson scheme, and $\theta = 0$ to a forward difference scheme. The $\theta$-scheme applied to our PDE results in

$$\varrho c \frac{T^k - T^{k-1}}{\mathrm{d}t} = \theta \nabla \cdot \left(\kappa \nabla T^k\right) + (1-\theta)\nabla \cdot \left(k \nabla T^{k-1}\right). \tag{1.120}$$

Bringing this time-discrete PDE into weak form follows the technique shown many times earlier in this tutorial. In the standard notation $a(T,v) = L(v)$ the weak form has

$$a(T,v) = \int_\Omega \left(\varrho c T v + \theta\, \mathrm{d}t \kappa \nabla T \cdot \nabla v\right)\, \mathrm{d}x, \tag{1.121}$$

$$L(v) = \int_\Omega \left(\varrho c T^{k-1} v - (1-\theta)\, \mathrm{d}t \kappa \nabla T^{k-1} \cdot \nabla v\right)\, \mathrm{d}x. \tag{1.122}$$

Observe that boundary integrals vanish because of the Neumann boundary conditions.

The size of a 3D box is taken as $W \times W \times D$, where $D$ is the depth and $W = D/2$ is the width. We give the degree of the basis functions at the command-line, then $D$, and then the divisions of the domain in the various directions. To make a box, rectangle, or interval of arbitrary (not unit) size, we have the DOLFIN classes `Box`, `Rectangle`, and `Interval` at our disposal. The mesh and the function space can be created by the following code:

*Python code*

```
degree = int(sys.argv[1])
D = float(sys.argv[2])
W = D/2.0
divisions = [int(arg) for arg in sys.argv[3:]]
d = len(divisions)  # no of space dimensions
if d == 1:
    mesh = Interval(divisions[0], -D, 0)
elif d == 2:
    mesh = Rectangle(-W/2, -D, W/2, 0, divisions[0], divisions[1])
elif d == 3:
    mesh = Box(-W/2, -W/2, -D, W/2, W/2, 0,
               divisions[0], divisions[1], divisions[2])
V = FunctionSpace(mesh, "Lagrange", degree)
```

The `Rectangle` and `Box` objects are defined by the coordinates of the "minimum" and "maximum" corners.

Setting Dirichlet conditions at the upper boundary can be done by

*Python code*

```
T_R = 0; T_A = 1.0; omega = 2*pi

T_0 = Expression("T_R + T_A*sin(omega*t)",
                 T_R=T_R, T_A=T_A, omega=omega, t=0.0)

def surface(x, on_boundary):
    return on_boundary and abs(x[d-1]) < 1E-14

bc = DirichletBC(V, T_0, surface)
```

The $\kappa$ function can be defined as a constant $\kappa_1$ inside the particular rectangular area with a special soil composition, as indicated in Figure 1.7. Outside this area $\kappa$ is a constant $\kappa_0$. The domain of the rectangular area is taken as

$$[-W/4, W/4] \times [-W/4, W/4] \times [-D/2, -D/2 + D/4]$$

in 3D, with $[-W/4, W/4] \times [-D/2, -D/2 + D/4]$ in 2D and $[-D/2, -D/2 + D/4]$ in 1D. Since we need some testing in the definition of the $\kappa(x)$ function, the most straightforward approach is to define a subclass of `Expression`, where we can use a full Python method instead of just a C++ string formula for specifying a function. The method that defines the function is called `eval`:

*Python code*

```
class Kappa(Function):
    def eval(self, value, x):
        """x: spatial point, value[0]: function value."""
        d = len(x)  # no of space dimensions
        material = 0  # 0: outside, 1: inside
        if d == 1:
            if -D/2. < x[d-1] < -D/2. + D/4.:
                material = 1
        elif d == 2:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4.:
                material = 1
        elif d == 3:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4. and -W/4. < x[1] < W/4.:
                material = 1
        value[0] = kappa_0 if material == 0 else kappa_1
```

The `eval` method gives great flexibility in defining functions, but a downside is that C++ calls up `eval` in Python for each point `x`, which is a slow process, and the number of calls is proportional to the number of nodes in the mesh. Function expressions in terms of strings are compiled to efficient C++ functions, being called from C++, so we should try to express functions as string expressions if possible. (The `eval` method can also be defined through C++ code, but this is more complicated and not covered here.) Using inline if-tests in C++, we can make string expressions for $\kappa$:

*Python code*

```
kappa_str = {}
kappa_str[1] = "x[0] > -D/2 && x[0] < -D/2 + D/4 ? kappa_1 : kappa_0"
kappa_str[2] = "x[0] > -W/4 && x[0] < W/4 "\
               "&& x[1] > -D/2 && x[1] < -D/2 + D/4 ? "\
               "kappa_1 : kappa_0"
kappa_str[3] = "x[0] > -W/4 && x[0] < W/4 "\
               "x[1] > -W/4 && x[1] < W/4 "\
               "&& x[2] > -D/2 && x[2] < -D/2 + D/4 ?"\
               "kappa_1 : kappa_0"

kappa = Expression(kappa_str[d],
                   D=D, W=W, kappa_0=kappa_0, kappa_1=kappa_1)
```

Let `T` denote the unknown spatial temperature function at the current time level, and let `T_1` be the corresponding function at one earlier time level. We are now ready to define the initial condition and the `a` and `L` forms of our problem:

*Python code*

```
T_1 = interpolate(Constant(T_R), V)

rho = 1
c = 1
period = 2*pi/omega
t_stop = 5*period
dt = period/20  # 20 time steps per period
theta = 1

T = TrialFunction(V)
v = TestFunction(V)
f = Constant(0)
a = rho*c*T*v*dx + theta*dt*kappa*\
    inner(nabla_grad(T), nabla_grad(v))*dx
L = (rho*c*T_prev*v + dt*f*v -
     (1-theta)*dt*kappa*inner(nabla_grad(T), nabla_grad(v)))*dx

A = assemble(a)
b = None  # variable used for memory savings in assemble calls
T = Function(V)   # unknown at the current time level
```

We could, alternatively, break `a` and `L` up in subexpressions and assemble a mass matrix and stiffness matrix, as exemplified in Section 1.3.3, to avoid assembly of `b` at every time level. This modification is straightforward and left as an exercise. The speed-up can be significant in 3D problems.

The time loop is very similar to what we have displayed in Section 1.3.2:

*Python code*

```
t = dt
while t <= t_stop:
    b = assemble(L, tensor=b)
    T_0.t = t
    bc.apply(A, b)
    solve(A, T.vector(), b)
    # visualization statements
    t += dt
    T_prev.assign(T)
```

The complete code in `sin_daD.py` contains several statements related to visualization and animation of the solution, both as a finite element field (`plot` calls) and as a curve in the vertical direction. The code also plots the exact analytical solution,

$$T(x,t) = T_R + T_A e^{ax}\sin(\omega t + ax), \quad a = \sqrt{\frac{\omega\varrho c}{2\kappa}}, \tag{1.123}$$

which is valid when $\kappa = \kappa_0 = \kappa_1$.

Implementing this analytical solution as a Python function taking scalars and numpy arrays as arguments requires a word of caution. A straightforward function like

*Python code*

```
def T_exact(x):
    a = sqrt(omega*rho*c/(2*kappa_0))
    return T_R + T_A*exp(a*x)*sin(omega*t + a*x)
```

will not work and result in an error message from UFL. The reason is that the names `exp` and `sin` are those imported by the `from dolfin import *` statement, and these names come from UFL and are

aimed at being used in variational forms. In the `T_exact` function where `x` may be a scalar or a `numpy` array, we therefore need to explicitly specify `numpy.exp` and `numpy.sin`:

*Python code*

```
def T_exact(x):
    a = sqrt(omega*rho*c/(2*kappa_0))
    return T_R + T_A*numpy.exp(a*x)*numpy.sin(omega*t + a*x)
```

The reader is encouraged to play around with the code and test out various parameter sets:

1. $T_R = 0$, $T_A = 1$, $\kappa_0 = \kappa_1 = 0.2$, $\varrho = c = 1$, $\omega = 2\pi$
2. $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.01$, $\varrho = c = 1$, $\omega = 2\pi$
3. $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.001$, $\varrho = c = 1$, $\omega = 2\pi$
4. $T_R = 10$ C, $T_A = 10$ C, $\kappa_0 = 2.3\ \mathrm{K^{-1}Ns^{-1}}$, $\kappa_1 = 100\ \mathrm{K^{-1}Ns^{-1}}$, $\varrho = 1500\ \mathrm{kg/m^3}$, $c = 1600\ \mathrm{Nm\,kg^{-1}K^{-1}}$, $\omega = 2\pi/24\ 1/\mathrm{h} = 7.27 \cdot 10^{-5}\ 1/\mathrm{s}$, $D = 1.5$ m
5. As above, but $\kappa_0 = 12.3\ \mathrm{K^{-1}Ns^{-1}}$ and $\kappa_1 = 10^4\ \mathrm{K^{-1}Ns^{-1}}$

Data set no. 4 is relevant for real temperature variations in the ground (not necessarily the large value of $\kappa_1$), while data set no. 5 exaggerates the effect of a large heat conduction contrast so that it becomes clearly visible in an animation.

## 1.4 Creating more complex domains

Up to now we have been very fond of the unit square as domain, which is an appropriate choice for initial versions of a PDE solver. The strength of the finite element method, however, is its ease of handling domains with complex shapes. This section shows some methods that can be used to create different types of domains and meshes.

Domains of complex shape must normally be constructed in separate preprocessor programs. Two relevant preprocessors are Triangle for 2D domains and NETGEN for 3D domains.

### 1.4.1 Built-in mesh generation tools

DOLFIN has a few tools for creating various types of meshes over domains with simple shape: `UnitInterval`, `UnitSquare`, `UnitCube`, `Interval`, `Rectangle`, `Box`, `UnitCircle`, and `UnitSphere`. Some of these names have been briefly met in previous sections. The hopefully self-explanatory code snippet below summarizes typical constructions of meshes with the aid of these tools:

*Python code*

```
# 1D domains
mesh = UnitInterval(20)     # 20 cells, 21 vertices
mesh = Interval(20, -1, 1)  # domain [-1,1]

# 2D domains (6x10 divisions, 120 cells, 77 vertices)
mesh = UnitSquare(6, 10)  # "right" diagonal is default
# The diagonals can be right, left or crossed
mesh = UnitSquare(6, 10, "left")
mesh = UnitSquare(6, 10, "crossed")

# Domain [0,3]x[0,2] with 6x10 divisions and left diagonals
mesh = Rectangle(0, 0, 3, 2, 6, 10, "left")
```

```
# 6x10x5 boxes in the unit cube, each box gets 6 tetrahedra:
mesh = UnitCube(6, 10, 5)

# Domain [-1,1]x[-1,0]x[-1,2] with 6x10x5 divisions
mesh = Box(-1, -1, -1, 1, 0, 2, 6, 10, 5)

# 10 divisions in radial directions
mesh = UnitCircle(10)
mesh = UnitSphere(10)
```

### 1.4.2 Transforming mesh coordinates

A mesh that is denser toward a boundary is often desired to increase accuracy in that region. Given a mesh with uniformly spaced coordinates $x_0, \ldots, x_{M-1}$ in $[a, b]$, the coordinate transformation $\xi = (x-a)/(b-a)$ maps $x$ onto $\xi \in [0,1]$. A new mapping $\eta = \xi^s$, for some $s > 1$, stretches the mesh toward $\xi = 0$ ($x = a$), while $\eta = \xi^{1/s}$ makes a stretching toward $\xi = 1$ ($x = b$). Mapping the $\eta \in [0,1]$ coordinates back to $[a, b]$ gives new, stretched $x$ coordinates,

$$\bar{x} = a + (b-a)\left((x-a)\,b-a\right)^s \tag{1.124}$$

toward $x = a$, or

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^{1/s} \tag{1.125}$$

toward $x = b$

One way of creating more complex geometries is to transform the vertex coordinates in a rectangular mesh according to some formula. Say we want to create a part of a hollow cylinder of $\Theta$ degrees, with inner radius $a$ and outer radius $b$. A standard mapping from polar coordinates to Cartesian coordinates can be used to generate the hollow cylinder. Given a rectangle in $(\bar{x}, \bar{y})$ space such that $a \leqslant \bar{x} \leqslant b$ and $0 \leqslant \bar{y} \leqslant 1$, the mapping

$$\hat{x} = \bar{x}\cos(\Theta\bar{y}), \quad \hat{y} = \bar{x}\sin(\Theta\bar{y}), \tag{1.126}$$

takes a point in the rectangular $(\bar{x}, \bar{y})$ geometry and maps it to a point $(\hat{x}, \hat{y})$ in a hollow cylinder.

The corresponding Python code for first stretching the mesh and then mapping it onto a hollow cylinder looks as follows:

*Python code*

```
Theta = pi/2
a, b = 1, 5.0
nr = 10  # divisions in r direction
nt = 20  # divisions in theta direction
mesh = Rectangle(a, 0, b, 1, nr, nt, "crossed")

# First make a denser mesh towards r=a
x = mesh.coordinates()[:,0]
y = mesh.coordinates()[:,1]
s = 1.3

def denser(x, y):
    return [a + (b-a)*((x-a)/(b-a))**s, y]

x_bar, y_bar = denser(x, y)
xy_bar_coor = numpy.array([x_bar, y_bar]).transpose()
mesh.coordinates()[:] = xy_bar_coor
```

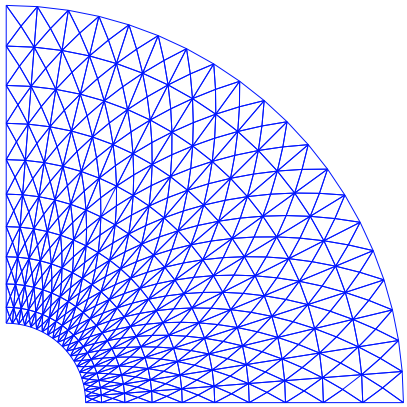


Figure 1.8: A hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.

```
plot(mesh, title="stretched mesh")

def cylinder(r, s):
    return [r*numpy.cos(Theta*s), r*numpy.sin(Theta*s)]

x_hat, y_hat = cylinder(x_bar, y_bar)
xy_hat_coor = numpy.array([x_hat, y_hat]).transpose()
mesh.coordinates()[:] = xy_hat_coor
plot(mesh, title="hollow cylinder")
interactive()
```

The result of calling `denser` and `cylinder` above is a list of two vectors, with the $x$ and $y$ coordinates, respectively. Turning this list into a `numpy` array object results in a $2 \times M$ array, $M$ being the number of vertices in the mesh. However, `mesh.coordinates()` is by a convention an $M \times 2$ array so we need to take the transpose. The resulting mesh is displayed in Figure 1.8.

Setting boundary conditions in meshes created from mappings like the one illustrated above is most conveniently done by using a mesh function to mark parts of the boundary. The marking is easiest to perform before the mesh is mapped since one can then conceptually work with the sides in a pure rectangle.

## 1.5 Handling domains with different materials

Solving PDEs in domains made up of different materials is a frequently encountered task. In FEniCS, these kind of problems are handled by defining subdomains inside the domain. The subdomains may represent the various materials. We can thereafter define material properties through functions, known in FEniCS as *mesh functions*, that are piecewise constant in each subdomain. A simple example with two materials (subdomains) in 2D will demonstrate the basic steps in the process.

### 1.5.1 Working with two subdomains

Suppose we want to solve

$$\nabla \cdot [k(x,y)\nabla u(x,y)] = 0, \tag{1.127}$$

in a domain $\Omega$ consisting of two subdomains where $k$ takes on a different value in each subdomain. For simplicity, yet without loss of generality, we choose for the current implementation the domain

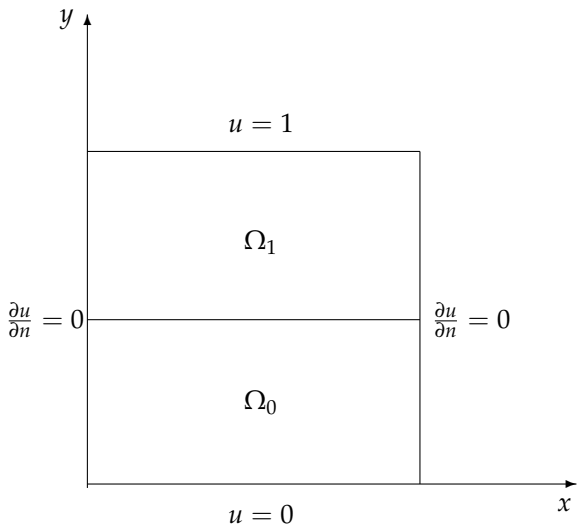



Figure 1.9: Sketch of a Poisson problem with a variable coefficient that is constant in each of the two subdomains $\Omega_0$ and $\Omega_1$.

$\Omega = [0,1] \times [0,1]$ and divide it into two equal subdomains, as depicted in Figure 1.9,

$$\Omega_0 = [0,1] \times [0,1/2], \quad \Omega_1 = [0,1] \times (1/2,1]. \tag{1.128}$$

We define $k(x,y) = k_0$ in $\Omega_0$ and $k(x,y) = k_1$ in $\Omega_1$, where $k_0 > 0$ and $k_1 > 0$ are given constants. As boundary conditions, we choose $u = 0$ at $y = 0$, $u = 1$ at $y = 1$, and $\partial u/\partial n = 0$ at $x = 0$ and $x = 1$. One can show that the exact solution is now given by

$$u(x,y) = \begin{cases} \frac{2yk_1}{k_0+k_1}, & y \leqslant 1/2 \\ \frac{(2y-1)k_0+k_1}{k_0+k_1}, & y \geqslant 1/2 \end{cases} \tag{1.129}$$

As long as the element boundaries coincide with the internal boundary $y = 1/2$, this piecewise linear solution should be exactly recovered by Lagrange elements of any degree. We use this property to verify the implementation.

Physically, the present problem may correspond to heat conduction, where the heat conduction in $\Omega_1$ is ten times more efficient than in $\Omega_0$. An alternative interpretation is flow in porous media with two geological layers, where the layers' ability to transport the fluid differs by a factor of 10.

### 1.5.2 Implementation

The new functionality in this subsection regards how to define the subdomains $\Omega_0$ and $\Omega_1$. For this purpose we need to use subclasses of class `SubDomain`, not only plain functions as we have used so far for specifying boundaries. Consider the boundary function

*Python code*

```
def boundary(x, on_boundary):
    tol = 1E-14
    return on_boundary and abs(x[0]) < tol
```

for defining the boundary $x = 0$. Instead of using such a stand-alone function, we can create an instance[2] of a subclass of `SubDomain`, which implements the `inside` method as an alternative to the

[2]The term *instance* means a Python object of a particular type (such as `SubDomain`, `Function`, `FunctionSpace`, etc.). Many use *instance* and *object* as interchangeable terms. In other computer programming languages one may also use the term *variable* for

`boundary` function:

*Python code*

```
class Boundary(SubDomain):
    def inside(x, on_boundary):
        tol = 1E-14
        return on_boundary and abs(x[0]) < tol

boundary = Boundary()
bc = DirichletBC(V, Constant(0), boundary)
```

A subclass of `SubDomain` with an `inside` method offers functionality for marking parts of the domain or the boundary. Now we need to define one class for the subdomain $\Omega_0$ where $y \leqslant 1/2$ and another for the subdomain $\Omega_1$ where $y \geqslant 1/2$:

*Python code*

```
class Omega0(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] <= 0.5 else False

class Omega1(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] >= 0.5 else False
```

Notice the use of `<=` and `>=` in both tests. For a cell to belong to, e.g., $\Omega_1$, the `inside` method must return `True` for all the vertices `x` of the cell. So to make the cells at the internal boundary $y = 1/2$ belong to $\Omega_1$, we need the test `x[1] >= 0.5`.

The next task is to use a `MeshFunction` to mark all cells in $\Omega_0$ with the subdomain number 0 and all cells in $\Omega_1$ with the subdomain number 1. Our convention is to number subdomains as $0, 1, 2, \ldots$.

A `MeshFunction` is a discrete function that can be evaluated at a set of so-called *mesh entities*. Examples of mesh entities are cells, facets, and vertices. A `MeshFunction` over cells is suitable to represent subdomains (materials), while a `MeshFunction` over facets is used to represent pieces of external or internal boundaries. Mesh functions over vertices can be used to describe continuous fields.

Since we need to define subdomains of $\Omega$ in the present example, we must make use of a `MeshFunction` over cells. The `MeshFunction` constructor is fed with three arguments: 1) the type of value: `"int"` for integers, `"uint"` for positive (unsigned) integers, `"double"` for real numbers, and `"bool"` for logical values; 2) a `Mesh` object, and 3) the topological dimension of the mesh entity in question: cells have topological dimension equal to the number of space dimensions in the PDE problem, and facets have one dimension lower. Alternatively, the constructor can take just a filename and initialize the `MeshFunction` from data in a file.

We start with creating a `MeshFunction` whose values are non-negative integers (`"uint"`) for numbering the subdomains. The mesh entities of interest are the cells, which have dimension 2 in a two-dimensional problem (1 in 1D, 3 in 3D). The appropriate code for defining the `MeshFunction` for two subdomains then reads

*Python code*

```
subdomains = MeshFunction("uint", mesh, 2)
# Mark subdomains with numbers 0 and 1
subdomain0 = Omega0()
subdomain0.mark(subdomains, 0)
subdomain1 = Omega1()
subdomain1.mark(subdomains, 1)
```

the same thing. We mostly use the well-known term *object* in this text.

Calling `subdomains.array()` returns a `numpy` array of the subdomain values. That is, `subdomain.array()[i]` is the subdomain value of cell number `i`. This array is used to look up the subdomain or material number of a specific element.

We need a function `k` that is constant in each subdomain $\Omega_0$ and $\Omega_1$. Since we want `k` to be a finite element function, it is natural to choose a space of functions that are constant over each element. The family of discontinuous Galerkin methods, in FEniCS denoted by `"DG"`, is suitable for this purpose. Since we want functions that are piecewise constant, the value of the degree parameter is zero:

*Python code*

```
V0 = FunctionSpace(mesh, "DG", 0)
k  = Function(V0)
```

To fill `k` with the right values in each element, we loop over all cells (the indices in `subdomain.array()`), extract the corresponding subdomain number of a cell, and assign the corresponding $k$ value to the `k.vector()` array:

*Python code*

```
k_values = [1.5, 50]  # values of k in the two subdomains
for cell_no in range(len(subdomains.array())):
    subdomain_no = subdomains.array()[cell_no]
    k.vector()[cell_no] = k_values[subdomain_no]
```

Long loops in Python are known to be slow, so for large meshes it is preferable to avoid such loops and instead use *vectorized code*. Normally this implies that the loop must be replaced by calls to functions from the `numpy` library that operate on complete arrays (in efficient C code). The functionality we want in the present case is to compute an array of the same size as `subdomain.array()`, but where the value `i` of an entry in `subdomain.array()` is replaced by `k_values[i]`. Such an operation is carried out by the `numpy` function `choose`:

*Python code*

```
help = numpy.asarray(subdomains.array(), dtype=numpy.int32)
k.vector()[:] = numpy.choose(help, k_values)
```

The `help` array is required since `choose` cannot work with `subdomain.array()` because this array has elements of type `uint32`. We must therefore transform this array to an array `help` with standard `int32` integers.

Having the `k` function ready for finite element computations, we can proceed in the normal manner with defining essential boundary conditions, as in Section 1.1.14, and the $a(u, v)$ and $L(v)$ forms, as in Section 1.1.10. All the details can be found in the file `mat2_p2D.py`.

### 1.5.3 Multiple Neumann, Robin, and Dirichlet conditions

Let us go back to the model problem from Section 1.1.14 where we had both Dirichlet and Neumann conditions. The term `v*g*ds` in the expression for `L` implies a boundary integral over the complete boundary, or in FEniCS terms, an integral over all exterior facets. However, the contributions from the parts of the boundary where we have Dirichlet conditions are erased when the linear system is modified by the Dirichlet conditions. We would like, from an efficiency point of view, to integrate `v*g*ds` only over the parts of the boundary where we actually have Neumann conditions. And more importantly, in other problems one may have different Neumann conditions or other conditions like the Robin type condition. With the mesh function concept we can mark different parts of the boundary and integrate over specific parts. The same concept can also be used to treat multiple Dirichlet conditions. The forthcoming text illustrates how this is done.

Essentially, we still stick to the model problem from Section 1.1.14, but replace the Neumann condition at $y = 0$ by a *Robin condition*[3]:

$$-\frac{\partial u}{\partial n} = p(u - q), \tag{1.130}$$

where $p$ and $q$ are specified functions. Since we have prescribed a simple solution in our model problem, $u = 1 + x^2 + 2y^2$, we adjust $p$ and $q$ such that the condition holds at $y = 0$. This implies that $q = 1 + x^2 + 2y^2$ and $p$ can be arbitrary (the normal derivative at $y = 0$: $\partial u/\partial n = -\partial u/\partial y = -4y = 0$).

Now we have four parts of the boundary: $\Gamma_{\mathrm{N}}$ which corresponds to the upper side $y = 1$, $\Gamma_R$ which corresponds to the lower part $y = 0$, $\Gamma_0$ which corresponds to the left part $x = 0$, and $\Gamma_1$ which corresponds to the right part $x = 1$. The complete boundary-value problem reads

$$-\Delta u = -6 \text{ in } \Omega, \tag{1.131}$$
$$u = u_L \text{ on } \Gamma_0, \tag{1.132}$$
$$u = u_R \text{ on } \Gamma_1, \tag{1.133}$$
$$-\frac{\partial u}{\partial n} = p(u - q) \text{ on } \Gamma_R, \tag{1.134}$$
$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_{\mathrm{N}}. \tag{1.135}$$

The involved prescribed functions are $u_L = 1 + 2y^2$, $u_R = 2 + 2y^2$, $q = 1 + x^2 + 2y^2$, $p$ is arbitrary, and $g = -4y$.

Integration by parts of $-\int_\Omega v\Delta u \,\mathrm{d}x$ becomes as usual

$$-\int_\Omega v\Delta u \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s. \tag{1.136}$$

The boundary integral vanishes on $\Gamma_0 \cup \Gamma_1$, and we split the parts over $\Gamma_{\mathrm{N}}$ and $\Gamma_R$ since we have different conditions at those parts:

$$-\int_{\partial\Omega} v\frac{\partial u}{\partial n} \,\mathrm{d}s = -\int_{\Gamma_{\mathrm{N}}} v\frac{\partial u}{\partial n} \,\mathrm{d}s - \int_{\Gamma_R} v\frac{\partial u}{\partial n} \,\mathrm{d}s = \int_{\Gamma_{\mathrm{N}}} vg \,\mathrm{d}s + \int_{\Gamma_R} vp(u - q) \,\mathrm{d}s. \tag{1.137}$$

The weak form then becomes

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s + \int_{\Gamma_R} p(u - q)v \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x, \tag{1.138}$$

We want to write this weak form in the standard notation $a(u, v) = L(v)$, which requires that we identify all integrals with *both* $u$ and $v$, and collect these in $a(u, v)$, while the remaining integrals with $v$ and not $u$ go into $L(v)$. The integral from the Robin condition must of this reason be split in two parts:

$$\int_{\Gamma_R} p(u - q)v \,\mathrm{d}s = \int_{\Gamma_R} puv \,\mathrm{d}s - \int_{\Gamma_R} pqv \,\mathrm{d}s. \tag{1.139}$$

[3]The Robin condition is most often used to model heat transfer to the surroundings and arise naturally from Newton's cooling law.

We then have

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_R} puv \,\mathrm{d}s, \tag{1.140}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_N} gv \,\mathrm{d}s + \int_{\Gamma_R} pqv \,\mathrm{d}s. \tag{1.141}$$

A natural starting point for implementation is the file `stationary/poisson/dn2_p2D.py`. The new aspects are

1. definition of a mesh function over the boundary,
2. marking each side as a subdomain, using the mesh function,
3. splitting a boundary integral into parts.

Task 1 makes use of the `MeshFunction` object, but contrary to Section 1.5.2, this is not a function over cells, but a function over cell facets. The topological dimension of cell facets is one lower than the cell interiors, so in a two-dimensional problem the dimension becomes 1. In general, the facet dimension is given as `mesh.topology().dim()-1`, which we use in the code for ease of direct reuse in other problems. The construction of a `MeshFunction` object to mark boundary parts now reads

*Python code*

```
boundary_parts = \
  MeshFunction("uint", mesh, mesh.topology().dim()-1)
```

As in Section 1.5.2 we use a subclass of `SubDomain` to identify the various parts of the mesh function. Problems with domains of more complicated geometries may set the mesh function for marking boundaries as part of the mesh generation. In our case, the $y = 0$ boundary can be marked by

*Python code*

```
class LowerRobinBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[1]) < tol

Gamma_R = LowerRobinBoundary()
Gamma_R.mark(boundary_parts, 0)
```

The code for the $y = 1$ boundary is similar and is seen in `dnr_p2D`.

The Dirichlet boundaries are marked similarly, using subdomain number 2 for $\Gamma_0$ and 3 for $\Gamma_1$:

*Python code*

```
class LeftBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[0]) < tol

Gamma_0 = LeftBoundary()
Gamma_0.mark(boundary_parts, 2)

class RightBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[0] - 1) < tol

Gamma_1 = RightBoundary()
Gamma_1.mark(boundary_parts, 3)
```

Specifying the `DirichletBC` objects may now make use of the mesh function (instead of a `SubDomain` subclass object) and an indicator for which subdomain each condition should be applied to:

*Python code*

```
u_L = Expression("1 + 2*x[1]*x[1]")
u_R = Expression("2 + 2*x[1]*x[1]")
bcs = [DirichletBC(V, u_L, boundary_parts, 2),
       DirichletBC(V, u_R, boundary_parts, 3)]
```

Some functions need to be defined before we can go on with the `a` and `L` of the variational problem:

*Python code*

```
g = Expression("-4*x[1]")
q = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]")
p = Constant(100)  # arbitrary function can go here
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
```

The new aspect of the variational problem is the two distinct boundary integrals. Having a mesh function over exterior cell facets (our `boundary_parts` object), where subdomains (boundary parts) are numbered as $0, 1, 2, \ldots$, the special symbol `ds(0)` implies integration over subdomain (part) 0, `ds(1)` denotes integration over subdomain (part) 1, and so on. The idea of multiple `ds`-type objects generalizes to volume integrals too: `dx(0)`, `dx(1)`, etc., are used to integrate over subdomain 0, 1, etc., inside $\Omega$.

The variational problem can be defined as

*Python code*

```
a = inner(nabla_grad(u), nabla_grad(v))*dx + p*u*v*ds(0)
L = f*v*dx - g*v*ds(1) + p*q*v*ds(0)
```

For the `ds(0)` and `ds(1)` symbols to work we must obviously connect them (or `a` and `L`) to the mesh function marking parts of the boundary. This is done by a certain keyword argument to the `assemble` function:

*Python code*

```
A = assemble(a, exterior_facet_domains=boundary_parts)
b = assemble(L, exterior_facet_domains=boundary_parts)
```

Then essential boundary conditions are enforced, and the system can be solved in the usual way:

*Python code*

```
for bc in bcs: bc.apply(A, b)
u = Function(V)
U = u.vector()
solve(A, U, b)
```

The complete code is in the `dnr_p2D.py` file in the `stationary/poisson` directory.

## 1.6 More examples

Many more topics could be treated in a FEniCS tutorial, e.g., how to solve systems of PDEs, how to work with mixed finite element methods, how to create more complicated meshes and mark boundaries, and how to create more advanced visualizations. However, to limit the size of this tutorial,

the examples end here. There are, fortunately, a rich set of FEniCS demos. The FEniCS documentation explains a collection of PDE solvers in detail: the Poisson equation, the mixed formulation for the Poisson equation, the Biharmonic equation, the equations of hyperelasticity, the Cahn-Hilliard equation, and the incompressible Navier–Stokes equations. Both Python and C++ versions of these solvers are explained. An eigenvalue solver is also documented. In the `dolfin/demo` directory of the DOLFIN source code tree you can find programs for these and many other examples, including the advection-diffusion equation, the equations of elastodynamics, a reaction-diffusion equation, various finite element methods for the Stokes problem, discontinuous Galerkin methods for the Poisson and advection-diffusion equations, and an eigenvalue problem arising from electromagnetic waveguide problem with Nédélec elements. There are also numerous demos on how to apply various functionality in FEniCS, e.g., mesh refinement and error control, moving meshes (for ALE methods), computing functionals over subsets of the mesh (such as lift and drag on bodies in flow), and creating separate subdomain meshes from a parent mesh.

The project CBC.Solve (`https://launchpad.net/cbc.solve`) offers more complete PDE solvers for the Navier–Stokes equations (Chapter 5), the equations of hyperelasticity (Chapter 5), fluid–structure interaction (Chapter 5), viscous mantle flow (Chapter 5), and the bidomain model of electrophysiology. Another project, CBC.RANS (`https://launchpad.net/cbc.rans`), offers an environment for very flexible and easy implementation of Navier–Stokes solvers and turbulence [Mortensen et al., 2011b,a]. For example, CBC.RANS contains an elliptic relaxation model for turbulent flow involving 18 nonlinear PDEs. FEniCS proved to be an ideal environment for implementing such complicated PDE models. The easy construction of systems of nonlinear PDEs in CBC.RANS has been further generalized to simplify the implementation of large systems of nonlinear PDEs in general. The functionality is found in the CBC.PDESys package (`https://launchpad.net/cbcpdesys`).

## 1.7 Miscellaneous topics

### 1.7.1 Glossary

Below we explain some key terms used in this tutorial.

FEniCS: name of a software suite composed of many individual software components (see `fenicsproject.org`). Some components are DOLFIN and Viper, explicitly referred to in this tutorial. Others are FFC and FIAT, heavily used by the programs appearing in this tutorial, but never explicitly used from the programs.

DOLFIN: a FEniCS component, more precisely a C++ library, with a Python interface, for performing important actions in finite element programs. DOLFIN makes use of many other FEniCS components and many external software packages.

Viper: a FEniCS component for quick visualization of finite element meshes and solutions.

UFL: a FEniCS component implementing the *unified form language* for specifying finite element forms in FEniCS programs. The definition of the forms, typically called `a` and `L` in this tutorial, must have legal UFL syntax. The same applies to the definition of functionals (see Section 1.1.11).

Class (Python): a programming construction for creating objects containing a set of variables and functions. Most types of FEniCS objects are defined through the class concept.

Instance (Python): an object of a particular type, where the type is implemented as a class. For instance, `mesh = UnitInterval(10)` creates an instance of class `UnitInterval`, which is reached by the name `mesh`. (Class `UnitInterval` is actually just an interface to a corresponding C++ class in the DOLFIN C++ library.)

Class method (Python): a function in a class, reached by dot notation: `instance_name.method_name`.

`self` parameter (Python): required first parameter in class methods, representing a particular object of the class. Used in method definitions, but never in calls to a method. For example, if `method(self, x)` is the definition of `method` in a class `Y`, `method` is called as `y.method(x)`, where `y` is an instance of class `Y`. In a call like `y.method(x)`, `method` is invoked with `self=y`.

Class attribute (Python): a variable in a class, reached by dot notation: `instance_name.attribute_name`.

### *1.7.2 Overview of objects and functions*

Most classes in FEniCS have an explanation of the purpose and usage that can be seen by using the general documentation command `pydoc` for Python objects. You can type

*Output*

```
pydoc dolfin.X
```

to look up documentation of a Python class `X` from the DOLFIN library (`X` can be `UnitSquare`, `Function`, `FunctionSpace`, etc.). Below is an overview of the most important classes and functions in FEniCS programs, in the order they typically appear within programs.

`UnitSquare(nx, ny)`: generate mesh over the unit square $[0,1] \times [0,1]$ using `nx` divisions in $x$ direction and `ny` divisions in $y$ direction. Each of the `nx*ny` squares are divided into two cells of triangular shape.

`UnitInterval`, `UnitCube`, `UnitCircle`, `UnitSphere`, `Interval`, `Rectangle`, and `Box`: generate mesh over domains of simple geometric shape, see Section 1.4.

`FunctionSpace(mesh, element_type, degree)`: a function space defined over a mesh, with a given element type (e.g., `"Lagrange"` or `"DG"`), with basis functions as polynomials of a specified degree.

`Expression(formula, p1=v1, p2=v2, ...)`: a scalar- or vector-valued function, given as a mathematical expression `formula` (string) written in C++ syntax. The spatial coordinates in the expression are named `x[0]`, `x[1]`, and `x[2]`, while time and other physical parameters can be represented as symbols `p1`, `p2`, etc., with corresponding values `v1`, `v2`, etc., initialized through keyword arguments. These parameters become attributes, whose values can be modified when desired.

`Function(V)`: a scalar- or vector-valued finite element field in the function space `V`. If `V` is a `FunctionSpace` object, `Function(V)` becomes a scalar field, and with `V` as a `VectorFunctionSpace` object, `Function(V)` becomes a vector field.

`SubDomain`: class for defining a subdomain, either a part of the boundary, an internal boundary, or a part of the domain. The programmer must subclass `SubDomain` and implement the `inside(self, x, on_boundary)` function (see Section 1.1.3) for telling whether a point `x` is inside the subdomain or not.

`Mesh`: class for representing a finite element mesh, consisting of cells, vertices, and optionally faces, edges, and facets.

`MeshFunction`: tool for marking parts of the domain or the boundary. Used for variable coefficients ("material properties", see Section 1.5.1) or for boundary conditions (see Section 1.5.3).

`DirichletBC(V, value, where)`: specification of Dirichlet (essential) boundary conditions via a function space `V`, a function `value(x)` for computing the value of the condition at a point `x`, and a specification `where` of the boundary, either as a `SubDomain` subclass instance, a plain function, or as a `MeshFunction` instance. In the latter case, a 4th argument is provided to describe which subdomain number that describes the relevant boundary.

`TrialFunction(V)`: define a trial function on a space `V` to be used in a variational form to represent the unknown in a finite element problem.

`TestFunction(V)`: define a test function on a space `V` to be used in a variational form.

`assemble(X)`: assemble a matrix, a right-hand side, or a functional, given a from `X` written with UFL syntax.

`assemble_system(a, L, bcs)`: assemble the matrix and the right-hand side from a bilinear (`a`) and linear (`L`) form written with UFL syntax. The `bcs` parameter holds one or more `DirichletBC` objects.

`LinearVariationalProblem(a, L, u, bcs)`: define a variational problem, given a bilinear (`a`) and linear (`L`) form, written with UFL syntax, and one or more `DirichletBC` objects stored in `bcs`.

`LinearVariationalSolver(problem)`: create solver object for a a linear variational problem object (`problem`).

`solve(A, U, b)`: solve a linear system with `A` as coefficient matrix (`Matrix` object), `U` as unknown (`Vector` object), and `b` as right-hand side (`Vector` object). Usually, `U = u.vector()`, where `u` is a `Function` object representing the unknown finite element function of the problem, while `A` and `b` are computed by calls to `assemble` or `assemble_system`.

`plot(q)`: quick visualization of a mesh, function, or mesh function `q`, using the Viper component in FEniCS.

`interpolate(func, V)`: interpolate a formula or finite element function `func` onto the function space `V`.

`project(func, V)`: project a formula or finite element function `func` onto the function space `V`.

### 1.7.3 User-defined functions

When defining a function in terms of a mathematical expression inside a string formula, e.g.,

*Python code*

```
myfunc = Expression("sin(x[0])*cos(x[1])")
```

the expression contained in the first argument will be turned into a C++ function and compiled to gain efficiency. Therefore, the syntax used in the expression must be valid C++ syntax. Most Python syntax for mathematical expressions are also valid C++ syntax, but power expressions make an exception: `p**a` must be written as `pow(p,a)` in C++ (this is also an alternative Python syntax). The following mathematical functions can be used directly in C++ expressions when defining `Expression` objects: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`, `exp`, `frexp`, `ldexp`, `log`, `log10`, `modf`, `pow`, `sqrt`, `ceil`, `fabs`, `floor`, and `fmod`. Moreover, the number $\pi$ is available as the symbol `pi`. All the listed functions are taken from the `cmath` C++ header file, and one may hence consult documentation of `cmath` for more information on the various functions.

### 1.7.4 Linear solvers and preconditioners

The following solution methods for linear systems can be accessed in FEniCS programs:

| Name | Method |
|---|---|
| `"lu"` | sparse LU factorization (Gaussian elim.) |
| `"cholesky"` | sparse Cholesky factorization |
| `"cg"` | Conjugate gradient method |
| `"gmres"` | Generalized minimal residual method |
| `"bicgstab"` | Biconjugate gradient stabilized method |
| `"minres"` | Minimal residual method |
| `"tfqmr"` | Transpose-free quasi-minimal residual method |
| `"richardson"` | Richardson method |

Possible choices of preconditioners include

| Name | Method |
|---|---|
| `"none"` | No preconditioner |
| `"ilu"` | Incomplete LU factorization |
| `"icc"` | Incomplete Cholesky factorization |
| `"jacobi"` | Jacobi iteration |
| `"bjacobi"` | Block Jacobi iteration |
| `"sor"` | Successive over-relaxation |
| `"amg"` | Algebraic multigrid (BoomerAMG or ML) |
| `"additive_schwarz"` | Additive Schwarz |
| `"hypre_amg"` | Hypre algebraic multigrid (BoomerAMG) |
| `"hypre_euclid"` | Hypre parallel incomplete LU factorization |
| `"hypre_parasails"` | Hypre parallel sparse approximate inverse |
| `"ml_amg"` | ML algebraic multigrid |

Many of the choices listed above are only offered by a specific backend, so setting the backend appropriately is necessary for being able to choose a desired linear solver or preconditioner.

An up-to-date list of the available solvers and preconditioners in FEniCS can be produced by

*Python code*

```
list_linear_solver_methods()
list_krylov_solver_preconditioners()
```

### *1.7.5 Installing FEniCS*

The FEniCS software components are available for Linux, Windows and Mac OS X platforms. Detailed information on how to get FEniCS running on such machines are available at the `fenicsproject.org` website. Here are just some quick descriptions and recommendations by the author.

To make the installation of FEniCS as painless and reliable as possible, the reader is strongly recommended to use Ubuntu Linux[4]. Any standard PC can easily be equipped with Ubuntu Linux, which may live side by side with either Windows or Mac OS X or another Linux installation. Basically, you download Ubuntu from `http://www.ubuntu.com/getubuntu/download`, burn the file on a CD or copy it to a memory stick, reboot the machine with the CD or memory stick, and answer some usually straightforward questions (if necessary). On Windows, Wubi is a tool that automatically installs Ubuntu on the machine. Just give a user name and password for the Ubuntu installation, and Wubi performs the rest. The graphical user interface (GUI) of Ubuntu is quite similar to both Windows 7 and Mac OS X, but to be efficient when doing science with FEniCS this author recommends to run programs in a terminal window and write them in a text editor like Emacs or Vim. You can employ integrated development environment such as Eclipse, but intensive FEniCS developers and users tend to find terminal windows and plain text editors more user friendly.

Instead of making it possible to boot your machine with the Linux Ubuntu operating system, you can run Ubuntu in a separate window in your existing operation system. There are several solutions to chose among: the free *VirtualBox* and *VMWare Player*, or the commercial tools *VMWare Fusion* and *Parallels* (just search for the names to download the programs).

Once the Ubuntu window is up and running, FEniCS is painlessly installed by

[4]Even though Mac users now can get FEniCS by a one-click install, I recommend using Ubuntu on Mac, unless you have high Unix competence and much experience with compiling and linking C++ libraries on Mac OS X.

*Bash code*

```
sudo apt-get install fenics
```

Sometimes the FEniCS software in a standard Ubuntu installation lacks some recent features and bug fixes. Visiting the detailed download page on `fenicsproject.org` and copying a few Unix commands is all you have to do to install a newer version of the software.

### *1.7.6 Books on the finite element method*

There are a large number of books on the finite element method. The books typically fall in either of two categories: the abstract mathematical version of the method and the engineering "structural analysis" formulation. FEniCS builds heavily on concepts in the abstract mathematical exposition. An easy-to-read book, which provides a good general background for using FEniCS, is Gockenbach [2006]. The book Donea and Huerta [2003] has a similar style, but aims at readers with interest in fluid flow problems. Hughes [1987] is also highly recommended, especially for those interested in solid mechanics and heat transfer applications.

Readers with background in the engineering "structural analysis" version of the finite element method may find Bickford [1994] as an attractive bridge over to the abstract mathematical formulation that FEniCS builds upon. Those who have a weak background in differential equations in general should consult a more fundamental book, and Eriksson et al. [1996] is a very good choice. On the other hand, FEniCS users with a strong background in mathematics and interest in the mathematical properties of the finite element method, will appreciate the texts Brenner and Scott [2008], Braess [2007], Ern and Guermond [2004], Quarteroni and Valli [2008], or Ciarlet [2002].

### *1.7.7 Books on Python*

Two very popular introductory books on Python are "Learning Python" [Lutz, 2007] and "Practical Python" [Hetland, 2002]. More advanced and comprehensive books include "Programming Python" [Lutz, 2006], and "Python Cookbook" [Martelli and Ascher, 2005] and "Python in a Nutshell" [Martelli, 2006]. The web page `http://wiki.python.org/moin/PythonBooks` lists numerous additional books. Very few texts teach Python in a mathematical and numerical context, but the references Langtangen [2008, 2011], Kiusalaas [2009] are exceptions.

*Acknowledgments.* The author is very thankful to Johan Hake, Anders Logg, Kent-Andre Mardal, and Kristian Valen-Sendstad for promptly answering all my questions about FEniCS functionality and for implementing all my requests. I will in particular thank Professor Douglas Arnold for very valuable feedback on the text. Øystein Sørensen pointed out a lot of typos and contributed with many helpful comments. Many errors and typos were also reported by Mauricio Angeles, Ida Drøsdal, Hans Ekkehard Plesser, and Marie Rognes. Ekkehard Ellmann as well as two anonymous reviewers provided a series of suggestions and improvements.

# 2 The finite element method

By Robert C. Kirby and Anders Logg

The finite element method has emerged as a universal method for the solution of differential equations. Much of the success of the finite element method can be attributed to its generality and elegance, allowing a wide range of differential equations from all areas of science to be analyzed and solved within a common framework. Another contributing factor to the success of the finite element method is the flexibility of formulation, allowing the properties of the discretization to be controlled by the choice of approximating finite element spaces.

In this chapter, we review the finite element method and summarize some basic concepts and notation used throughout this book. In the coming chapters, we discuss these concepts in more detail, with a particular focus on the implementation and automation of the finite element method as part of the FEniCS Project.

## 2.1 A simple model problem

In 1813, Siméon Denis Poisson published in *Bulletin de la société philomatique* his famous equation as a correction of an equation published earlier by Pierre-Simon Laplace. Poisson's equation is a second-order partial differential equation stating that the negative Laplacian $-\Delta u$ of some unknown field $u = u(x)$ is equal to a given function $f = f(x)$ on a domain $\Omega \subset \mathbb{R}^d$, possibly amended by a set of boundary conditions for the solution $u$ on the boundary $\partial\Omega$ of $\Omega$:

$$\begin{aligned} -\Delta u &= f \quad && \text{in } \Omega, \\ u &= u_0 && \text{on } \Gamma_{\mathrm{D}} \subset \partial\Omega, \\ -\partial_n u &= g && \text{on } \Gamma_{\mathrm{N}} \subset \partial\Omega. \end{aligned} \tag{2.1}$$

The Dirichlet boundary condition $u = u_0$ signifies a prescribed value for the unknown $u$ on a subset $\Gamma_{\mathrm{D}}$ of the boundary, and the Neumann boundary condition $-\partial_n u = g$ signifies a prescribed value for the (negative) normal derivative of $u$ on the remaining boundary $\Gamma_{\mathrm{N}} = \partial\Omega \setminus \Gamma_{\mathrm{D}}$. Poisson's equation is a simple model for gravity, electromagnetism, heat transfer, fluid flow, and many other physical processes. It also appears as the basic building block in a large number of more complex physical models, including the Navier–Stokes equations which we return to in Chapters 5, 5, 5, 5, 5, 5, 5 and 5.

To derive Poisson's equation (2.1), we may consider a model for the temperature $u$ in a body occupying a domain $\Omega$ subject to a heat source $f$. Letting $\sigma = \sigma(x)$ denote heat flux, it follows by conservation of energy that the outflow of energy over the boundary $\partial\omega$ of any test volume $\omega \subset \Omega$ must be balanced by the energy emitted by the heat source $f$:

$$\int_{\partial\omega} \sigma \cdot n \, \mathrm{d}s = \int_{\omega} f \, \mathrm{d}x. \tag{2.2}$$

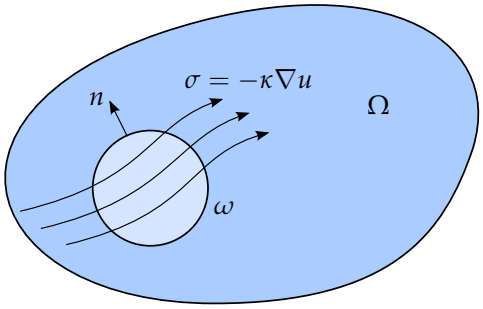



Figure 2.1: Poisson's equation is a simple consequence of balance of energy in an arbitrary test volume $\omega \subset \Omega$.

Integrating by parts, we find that

$$\int_\omega \nabla \cdot \sigma \, \mathrm{d}x = \int_\omega f \, \mathrm{d}x. \tag{2.3}$$

Since (2.3) holds for all test volumes $\omega \subset \Omega$, it follows that $\nabla \cdot \sigma = f$ throughout $\Omega$ (with suitable regularity assumptions on $\sigma$ and $f$). If we now make the assumption that the heat flux $\sigma$ is proportional to the negative gradient of the temperature $u$ (Fourier's law),

$$\sigma = -\kappa \nabla u, \tag{2.4}$$

we arrive at the following system of equations:

$$\begin{aligned} \nabla \cdot \sigma &= f \quad \text{in } \Omega, \\ \sigma + \nabla u &= 0 \quad \text{in } \Omega, \end{aligned} \tag{2.5}$$

where we have assumed that the heat conductivity is $\kappa = 1$. Replacing $\sigma$ in the first of these equations by $-\nabla u$, we arrive at Poisson's equation (2.1). Note that one may as well arrive at the system of first-order equations (2.5) by introducing $\sigma = -\nabla u$ as an auxiliary variable in the second-order equation (2.1). We also note that the Dirichlet and Neumann boundary conditions in (2.1) correspond to prescribed values for the temperature and heat flux, respectively.

## 2.2 Finite element discretization

### 2.2.1 Discretizing Poisson's equation

To discretize Poisson's equation (2.1) by the finite element method, we first multiply by a test function $v$ and integrate by parts to obtain

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x - \int_{\partial\Omega} \partial_n u \, v \, \mathrm{d}s = \int_\Omega f v \, \mathrm{d}x. \tag{2.6}$$

Letting the test function $v$ vanish on the Dirichlet boundary $\Gamma_\mathrm{D}$ where the solution $u$ is known, we arrive at the following classical variational problem: find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x - \int_{\Gamma_\mathrm{N}} g v \, \mathrm{d}s \quad \forall\, v \in \hat{V}. \tag{2.7}$$

The test space $\hat{V}$ is defined by

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_\mathrm{D}\}, \tag{2.8}$$

and the trial space $V$ contains members of $\hat{V}$ shifted by the Dirichlet condition:

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \Gamma_D\}. \tag{2.9}$$

We may now discretize Poisson's equation by restricting the variational problem (2.7) to a pair of discrete spaces: find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_N} gv \,\mathrm{d}s \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{2.10}$$

We note here that the Dirichlet condition $u = u_0$ on $\Gamma_D$ enters directly into the definition of the trial space $V_h$ (it is an *essential* boundary condition), whereas the Neumann condition $-\partial_n u = g$ on $\Gamma_N$ enters into the variational problem (it is a *natural* boundary condition).

To solve the discrete variational problem (2.10), we must construct a suitable pair of discrete trial and test spaces $V_h$ and $\hat{V}_h$. We return to this issue below, but assume for now that we have a basis $\{\phi_j\}_{j=1}^N$ for $V_h$ and a basis $\{\hat{\phi}_i\}_{i=1}^N$ for $\hat{V}_h$. Here, $N$ denotes the dimension of the spaces $V_h$ and $\hat{V}_h$. We may then make an Ansatz for $u_h$ in terms of the basis functions of the trial space,

$$u_h(x) = \sum_{j=1}^N U_j \phi_j(x), \tag{2.11}$$

where $U \in \mathbb{R}^N$ is the vector of degrees of freedom to be computed. Inserting this into (2.10) and varying the test function $v$ over the basis functions of the discrete test space $\hat{V}_h$, we obtain

$$\sum_{j=1}^N U_j \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \,\mathrm{d}x = \int_\Omega f\hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_N} g\hat{\phi}_i \,\mathrm{d}s, \quad i = 1, 2, \ldots, N. \tag{2.12}$$

We may thus compute the finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ by solving the linear system

$$AU = b, \tag{2.13}$$

where

$$\begin{aligned} A_{ij} &= \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \,\mathrm{d}x, \\ b_i &= \int_\Omega f\hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_N} g\hat{\phi}_i \,\mathrm{d}s. \end{aligned} \tag{2.14}$$

### 2.2.2 Discretizing the first-order system

We may similarly discretize the first-order system (2.5) by multiplying the first equation by a test function $v$ and the second equation by a test function $\tau$. Summing up and integrating by parts, we find that

$$\int_\Omega (\nabla \cdot \sigma)\, v + \sigma \cdot \tau - u \nabla \cdot \tau \,\mathrm{d}x + \int_{\partial\Omega} u\tau \cdot n \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x \quad \forall\, (v, \tau) \in \hat{V}. \tag{2.15}$$

The normal flux $\sigma \cdot n = g$ is known on the Neumann boundary $\Gamma_N$ so we may take $\tau \cdot n = 0$ on $\Gamma_N$. Inserting the value for $u$ on the Dirichlet boundary $\Gamma_D$, we arrive at the following variational problem: find $(u, \sigma) \in V$ such that

$$\int_\Omega (\nabla \cdot \sigma)\, v + \sigma \cdot \tau - u \nabla \cdot \tau \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_D} u_0 \tau \cdot n \,\mathrm{d}s \quad \forall\, (v, \tau) \in \hat{V}. \tag{2.16}$$

A suitable choice of trial and test spaces is

$$\begin{aligned} V &= \{(v,\tau) : v \in L^2(\Omega), \tau \in H(\mathrm{div},\Omega), \tau \cdot n = g \text{ on } \Gamma_{\mathrm{N}}\}, \\ \hat{V} &= \{(v,\tau) : v \in L^2(\Omega), \tau \in H(\mathrm{div},\Omega), \tau \cdot n = 0 \text{ on } \Gamma_{\mathrm{N}}\}. \end{aligned} \tag{2.17}$$

Note that the variational problem (2.16) differs from the variational problem (2.7) in that the Dirichlet condition $u = u_0$ on $\Gamma_{\mathrm{D}}$ enters into the variational formulation (it is now a natural boundary condition), whereas the Neumann condition $\sigma \cdot n = g$ on $\Gamma_{\mathrm{N}}$ enters into the definition of the trial space $V$ (it is now an essential boundary condition).

As above, we restrict the variational problem to a pair of discrete trial and test spaces $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$ and make an Ansatz for the finite element solution of the form

$$(u_h, \sigma_h) = \sum_{j=1}^{N} U_j (\phi_j, \psi_j), \tag{2.18}$$

where $\{(\phi_j, \psi_j)\}_{j=1}^N$ is a basis for the trial space $V_h$. Typically, either $\phi_j$ or $\psi_j$ will vanish, so that the basis is really the tensor product of a basis for the $L^2$ space with a basis for the $H(\mathrm{div})$ space. We thus obtain a linear system for the degrees of freedom $U \in \mathbb{R}^N$ by solving a linear system $AU = b$, where now

$$\begin{aligned} A_{ij} &= \int_\Omega (\nabla \cdot \psi_j)\, \hat{\phi}_i + \psi_j \cdot \hat{\psi}_i - \phi_j \nabla \cdot \hat{\psi}_i \,\mathrm{d}x, \\ b_i &= \int_\Omega f \hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_{\mathrm{D}}} u_0\, \hat{\psi}_i \cdot n \,\mathrm{d}s. \end{aligned} \tag{2.19}$$

The finite element discretization (2.19) is an example of a *mixed method*. Such formulations require some care in selecting spaces that discretize the different function spaces, here $L^2$ and $H(\mathrm{div})$, in a compatible way. Stable discretizations must satisfy the so-called *inf–sup* or Ladyzhenskaya–Babuška–Brezzi (LBB) condition(s). This theory explains why many of the finite element spaces for mixed methods seem complicated compared to those for standard methods. In Chapter 5 below, we give several examples of such finite element spaces.

## 2.3 Finite element abstract formalism

### 2.3.1 Linear problems

We saw above that the finite element solution of Poisson's equation (2.1) or (2.5) can be obtained by restricting an infinite-dimensional (continuous) variational problem to a finite-dimensional (discrete) variational problem and solving a linear system.

To formalize this, we consider a general linear variational problem written in the following canonical form: find $u \in V$ such that

$$a(u,v) = L(v) \quad \forall\, v \in \hat{V}, \tag{2.20}$$

where $V$ is the trial space and $\hat{V}$ is the test space. We thus express the variational problem in terms of a *bilinear form* $a$ and a *linear form* (functional) $L$:

$$\begin{aligned} a &: V \times \hat{V} \to \mathbb{R}, \\ L &: \hat{V} \to \mathbb{R}. \end{aligned} \tag{2.21}$$

As above, we discretize the variational problem (2.20) by restricting to a pair of discrete trial and test spaces: find $u_h \in V_h \subset V$ such that

$$a(u_h, v) = L(v) \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{2.22}$$

To solve the discrete variational problem (2.22), we make an Ansatz of the form

$$u_h = \sum_{j=1}^{N} U_j \phi_j, \tag{2.23}$$

and take $v = \hat{\phi}_i$ for $i = 1, 2, \ldots, N$. As before, $\{\phi_j\}_{j=1}^N$ is a basis for the discrete trial space $V_h$ and $\{\hat{\phi}_i\}_{i=1}^N$ is a basis for the discrete test space $\hat{V}_h$. It follows that

$$\sum_{j=1}^{N} U_j\, a(\phi_j, \hat{\phi}_i) = L(\hat{\phi}_i), \quad i = 1, 2, \ldots, N. \tag{2.24}$$

The degrees of freedom $U$ of the finite element solution $u_h$ may then be computed by solving a linear system $AU = b$, where

$$\begin{aligned} A_{ij} &= a(\phi_j, \hat{\phi}_i), \quad i, j = 1, 2, \ldots, N, \\ b_i &= L(\hat{\phi}_i). \end{aligned} \tag{2.25}$$

### *2.3.2 Nonlinear problems*

We also consider nonlinear variational problems written in the following canonical form: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}, \tag{2.26}$$

where now $F : V \times \hat{V} \to \mathbb{R}$ is a *semilinear* form, linear in the argument(s) subsequent to the semicolon. As above, we discretize the variational problem (2.26) by restricting to a pair of discrete trial and test spaces: find $u_h \in V_h \subset V$ such that

$$F(u_h; v) = 0 \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{2.27}$$

The finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ may then be computed by solving a nonlinear system of equations,

$$b(U) = 0, \tag{2.28}$$

where $b : \mathbb{R}^N \to \mathbb{R}^N$ and

$$b_i(U) = F(u_h; \hat{\phi}_i), \quad i = 1, 2, \ldots, N. \tag{2.29}$$

To solve the nonlinear system (2.28) by Newton's method or some variant of Newton's method, we compute the Jacobian $A = b'$. We note that if the semilinear form $F$ is differentiable in $u$, then the entries of the Jacobian $A$ are given by

$$A_{ij}(u_h) = \frac{\partial b_i(U)}{\partial U_j} = \frac{\partial}{\partial U_j} F(u_h; \hat{\phi}_i) = F'(u_h; \hat{\phi}_i)\, \frac{\partial u_h}{\partial U_j} = F'(u_h; \hat{\phi}_i)\, \phi_j \equiv F'(u_h; \phi_j, \hat{\phi}_i). \tag{2.30}$$

In each Newton iteration, we must then evaluate (assemble) the matrix $A$ and the vector $b$, and update the solution vector $U$ by

$$U^{k+1} = U^k - \delta U^k, \quad k = 0, 1, \ldots, \tag{2.31}$$

where $\delta U^k$ solves the linear system

$$A(u_h^k)\,\delta U^k = b(u_h^k). \tag{2.32}$$

We note that for each fixed $u_h$, $a = F'(u_h;\cdot,\cdot)$ is a bilinear form and $L = F(u_h;\cdot)$ is a linear form. In each Newton iteration, we thus solve a linear variational problem of the canonical form (2.20): find $\delta u \in V_{h,0}$ such that

$$F'(u_h;\delta u, v) = F(u_h;v) \quad \forall\, v \in \hat{V}_h, \tag{2.33}$$

where $V_{h,0} = \{v - w : v, w \in V_h\}$. Discretizing (2.33) as in Section 2.3.1, we recover the linear system (2.32).

**Example 2.1 (Nonlinear Poisson equation)** *As an example, consider the following nonlinear Poisson equation:*

$$\begin{aligned} -\nabla\cdot((1+u)\nabla u) &= f \quad \text{in } \Omega,\\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{2.34}$$

*Multiplying (2.34) with a test function $v$ and integrating by parts, we obtain*

$$\int_\Omega ((1+u)\nabla u)\cdot\nabla v\,\mathrm{d}x = \int_\Omega fv\,\mathrm{d}x, \tag{2.35}$$

*which is a nonlinear variational problem of the form (2.26), with*

$$F(u;v) = \int_\Omega ((1+u)\nabla u)\cdot\nabla v\,\mathrm{d}x - \int_\Omega fv\,\mathrm{d}x. \tag{2.36}$$

*Linearizing the semilinear form $F$ around $u = u_h$, we obtain*

$$F'(u_h;\delta u, v) = \int_\Omega (\delta u\nabla u_h)\cdot\nabla v\,\mathrm{d}x + \int_\Omega ((1+u_h)\nabla\delta u)\cdot\nabla v\,\mathrm{d}x. \tag{2.37}$$

*We may thus compute the entries of the Jacobian matrix $A(u_h)$ by*

$$A_{ij}(u_h) = F'(u_h;\phi_j,\hat{\phi}_i) = \int_\Omega (\phi_j\nabla u_h)\cdot\nabla\hat{\phi}_i\,\mathrm{d}x + \int_\Omega ((1+u_h)\nabla\phi_j)\cdot\nabla\hat{\phi}_i\,\mathrm{d}x. \tag{2.38}$$

## 2.4 Finite element function spaces

In the above discussion, we assumed that we could construct discrete subspaces $V_h \subset V$ of infinite-dimensional function spaces. A central aspect of the finite element method is the construction of such subspaces by patching together local function spaces defined by a set of *finite elements*. We here give a general overview of the construction of finite element function spaces and return in Chapters 5 and 5 to the construction of specific function spaces as subsets of $H^1$, $H(\mathrm{curl})$, $H(\mathrm{div})$ and $L^2$.

### 2.4.1 The mesh

To define $V_h$, we first partition the domain $\Omega$ into a finite set of cells $\mathcal{T}_h = \{T\}$ with disjoint interiors such that

$$\cup_{T\in\mathcal{T}_h} T = \Omega. \tag{2.39}$$

Together, these cells form a *mesh* of the domain $\Omega$. The cells are typically simple polygonal shapes like intervals, triangles, quadrilaterals, tetrahedra or hexahedra as shown in Figure 2.2. But other shapes

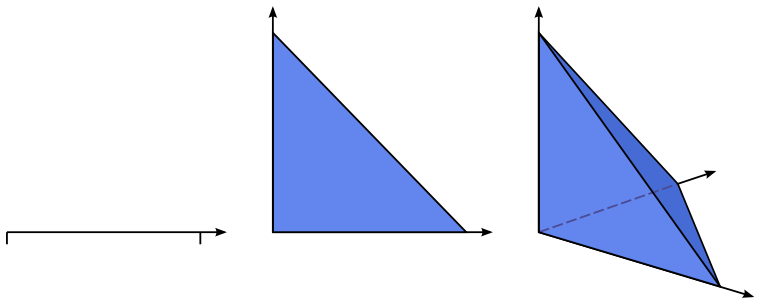


Figure 2.2: Examples of finite element cells in one, two and three space dimensions.

are possible, in particular curved cells to capture the boundary of a non-polygonal domain correctly.

### 2.4.2 The finite element definition

Once a domain $\Omega$ has been partitioned into cells, one may define a local function space $\mathcal{V}$ on each cell $T$ and use these local function spaces to build the global function space $V_h$. A cell $T$ together with a local function space $\mathcal{V}$ and a set of rules for describing the functions in $\mathcal{V}$ is called a *finite element*. This definition was first formalized by Ciarlet [1976] and it remains the standard formulation today [Brenner and Scott, 2008]. The formal definition reads as follows: a finite element is a triple $(T, \mathcal{V}, \mathcal{L})$, where

- the domain $T$ is a bounded, closed subset of $\mathbb{R}^d$ (for $d = 1, 2, 3, \dots$) with nonempty interior and piecewise smooth boundary;
- the space $\mathcal{V} = \mathcal{V}(T)$ is a finite dimensional function space on $T$ of dimension $n$;
- the set of degrees of freedom (nodes) $\mathcal{L} = \{\ell_1, \ell_2, \dots, \ell_n\}$ is a basis for the dual space $\mathcal{V}'$; that is, the space of bounded linear functionals on $\mathcal{V}$.

As an example, consider the standard linear Lagrange finite element on the triangle in Figure 2.3. The cell $T$ is given by the triangle and the space $\mathcal{V}$ is given by the space of first degree polynomials on $T$ (a space of dimension three). As a basis for $\mathcal{V}'$, we may take point evaluation at the three vertices of $T$; that is,

$$\begin{aligned} \ell_i &: \mathcal{V} \to \mathbb{R}, \\ \ell_i(v) &= v(x^i), \end{aligned} \tag{2.40}$$

for $i = 1, 2, 3$ where $x^i$ is the coordinate of the $i$th vertex. To check that this is indeed a finite element, we need to verify that $\mathcal{L}$ is a basis for $\mathcal{V}'$. This is equivalent to the unisolvence of $\mathcal{L}$; that is, if $v \in \mathcal{V}$ and $\ell_i(v) = 0$ for all $\ell_i$, then $v = 0$ [Brenner and Scott, 2008]. For the linear Lagrange triangle, we note that if $v$ is zero at each vertex, then $v$ must be zero everywhere, since a plane is uniquely determined by its values at three non-collinear points. It follows that the linear Lagrange triangle is indeed a finite element. In general, determining the unisolvence of $\mathcal{L}$ may be non-trivial.

### 2.4.3 The nodal basis

Expressing finite element solutions in $V_h$ in terms of basis functions for the local function spaces $\mathcal{V}$ may be greatly simplified by introducing a *nodal basis* for $\mathcal{V}$. A nodal basis $\{\phi_i\}_{i=1}^n$ for $\mathcal{V}$ is a basis for

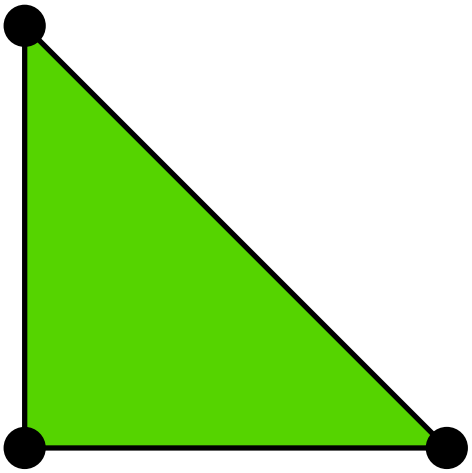


Figure 2.3: The degrees of freedom of the linear Lagrange (Courant) triangle are given by point evaluation at the three vertices of the triangle.

$\mathcal{V}$ that satisfies

$$\ell_i(\phi_j) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n. \tag{2.41}$$

It follows that any $v \in \mathcal{V}$ may be expressed by

$$v = \sum_{i=1}^{n} \ell_i(v)\phi_i. \tag{2.42}$$

In particular, any function $v$ in $\mathcal{V}$ for the linear Lagrange triangle is given by $v = \sum_{i=1}^{3} v(x^i)\phi_i$. In other words, the expansion coefficients of any function $v$ may be obtained by evaluating the linear functionals in $\mathcal{L}$ at $v$. We shall therefore interchangeably refer to both the expansion coefficients $U$ of $u_h$ and the linear functionals of $\mathcal{L}$ as the *degrees of freedom*.

**Example 2.2 (Nodal basis for the linear Lagrange simplices)** *The nodal basis for the linear Lagrange interval with vertices at $x^1 = 0$ and $x^2 = 1$ is given by*

$$\phi_1(x) = 1 - x, \quad \phi_2(x) = x. \tag{2.43}$$

*The nodal basis for the linear Lagrange triangle with vertices at $x^1 = (0, 0)$, $x^2 = (1, 0)$ and $x^3 = (0, 1)$ is given by*

$$\phi_1(x) = 1 - x_1 - x_2, \quad \phi_2(x) = x_1, \quad \phi_3(x) = x_2. \tag{2.44}$$

*The nodal basis for the linear Lagrange tetrahedron with vertices at $x^1 = (0, 0, 0)$, $x^2 = (1, 0, 0)$, $x^3 = (0, 1, 0)$ and $x^4 = (0, 0, 1)$ is given by*

$$\begin{aligned} \phi_1(x) &= 1 - x_1 - x_2 - x_3, & \phi_2(x) &= x_1, \\ \phi_3(x) &= x_2, & \phi_4(x) &= x_3. \end{aligned} \tag{2.45}$$

For any finite element $(T, \mathcal{V}, \mathcal{L})$, the nodal basis may be computed by solving a linear system of size $n \times n$. To see this, let $\{\psi_i\}_{i=1}^n$ be any basis (the *prime* basis) for $\mathcal{V}$. Such a basis is easy to construct if $\mathcal{V}$ is a full polynomial space or may otherwise be computed by a singular-value decomposition or a Gram–Schmidt procedure; see Kirby [2004]. We may then make an Ansatz for the nodal basis in terms of the prime basis:

$$\phi_j = \sum_{k=1}^{n} \alpha_{jk}\psi_k, \quad j = 1, 2, \ldots, n. \tag{2.46}$$

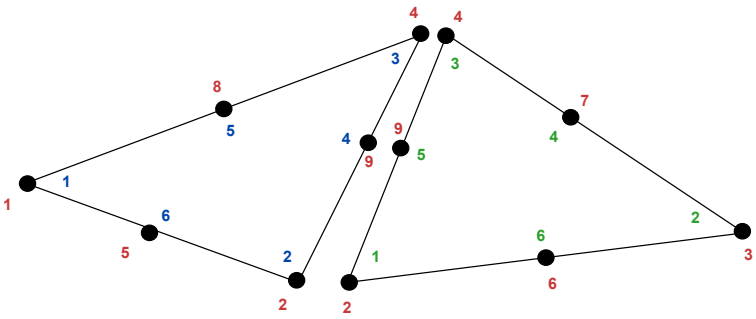



Figure 2.4: Local-to-global mapping for a simple mesh consisting of two triangles. The six local degrees of freedom of the left triangle ($T$) are mapped to the global degrees of freedom $\iota_T(i) = 1, 2, 4, 9, 8, 5$ for $i = 1, 2, \ldots, 6$, and the six local degrees of freedom of the right triangle ($T'$) are mapped to $\iota_{T'}(i) = 2, 3, 4, 7, 9, 6$ for $i = 1, 2, \ldots, 6$.

Inserting this into (2.41), we find that

$$\sum_{k=1}^{n} \alpha_{jk} \ell_i(\psi_k) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n. \tag{2.47}$$

In other words, the coefficients $\alpha$ expanding the nodal basis functions in the prime basis may be computed by solving the linear system

$$B\alpha^\top = I, \tag{2.48}$$

where $B_{ij} = \ell_i(\psi_j)$.

### 2.4.4 *The local-to-global mapping*

Now, to define a global function space $V_h = \text{span}\{\phi_i\}_{i=1}^N$ on $\Omega$ from a given set $\{(T, \mathcal{V}_T, \mathcal{L}_T)\}_{T \in \mathcal{T}_h}$ of finite elements, we also need to specify how the local function spaces are patched together. We do this by specifying for each cell $T \in \mathcal{T}_h$ a *local-to-global mapping*:

$$\iota_T : [1, n_T] \to [1, N]. \tag{2.49}$$

This mapping specifies how the local degrees of freedom $\mathcal{L}_T = \{\ell_i^T\}_{i=1}^{n_T}$ are mapped to global degrees of freedom $\mathcal{L} = \{\ell_i\}_{i=1}^N$. More precisely, the global degrees of freedom are defined by

$$\ell_{\iota_T(i)}(v) = \ell_i^T(v|_T), \quad i = 1, 2, \ldots, n_T, \tag{2.50}$$

for any $v \in V_h$. Thus, each local degree of freedom $\ell_i^T \in \mathcal{L}_T$ corresponds to a global degree of freedom $\ell_{\iota_T(i)} \in \mathcal{L}$ determined by the local-to-global mapping $\iota_T$. As we shall see, the local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space $V_h$.

For standard continuous piecewise linears, one may define the local-to-global mapping by simply mapping each local vertex number $i$ for $i = 1, 2, 3$ to the corresponding global vertex number $\iota_T(i)$. For continuous piecewise quadratics, one can base the local-to-global mapping on global vertex and edge numbers as illustrated in Figure 2.4 for a simple mesh consisting of two triangles.

### 2.4.5 *The global function space*

One may now define the global function space $V_h$ as the set of functions on $\Omega$ satisfying the following pair of conditions. We first require that

$$v|_T \in \mathcal{V}_T \quad \forall\, T \in \mathcal{T}_h; \tag{2.51}$$

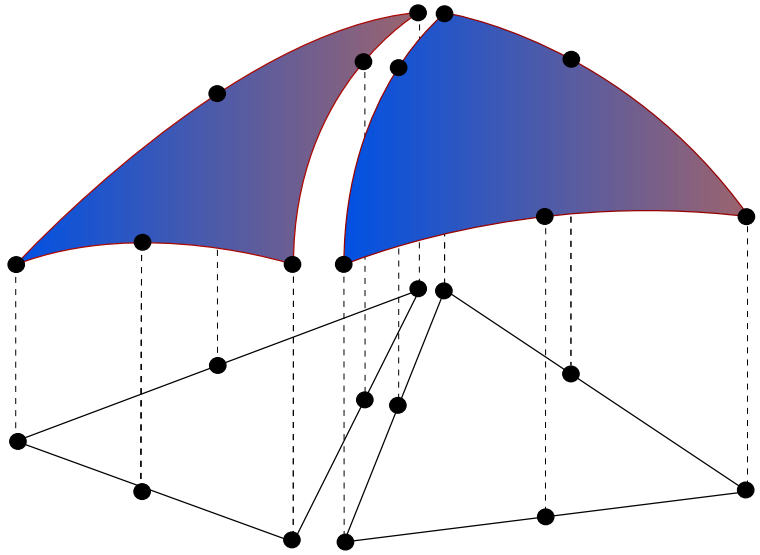


Figure 2.5: Patching together a pair of quadratic local function spaces on a pair of cells $(T, T')$ to form a global continuous piecewise quadratic function space on $\Omega = T \cup T'$.

that is, the restriction of $v$ to each cell $T$ lies in the local function space $\mathcal{V}_T$. Second, we require that for any pair of cells $(T, T') \in \mathcal{T}_h \times \mathcal{T}_h$ and any pair $(i, i') \in [1, n_T] \times [1, n_{T'}]$ satisfying

$$\iota_T(i) = \iota_{T'}(i'), \tag{2.52}$$

it holds that

$$\ell_i^T(v|_T) = \ell_{i'}^{T'}(v|_{T'}). \tag{2.53}$$

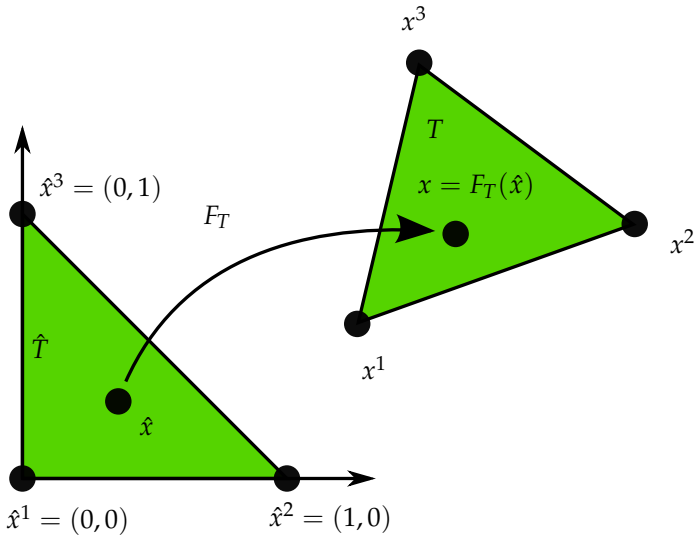
In other words, if two local degrees of freedom $\ell_i^T$ and $\ell_{i'}^{T'}$ are mapped to the same global degree of freedom, then they must agree for each function $v \in V_h$. Here, $v|_T$ denotes (the continuous extension of the) restriction of $v$ to the interior of $T$. This is illustrated in Figure 2.5 for the space of continuous piecewise quadratics obtained by patching together two quadratic Lagrange triangles.

Note that by this construction, the functions in $V_h$ are undefined on cell boundaries, unless the constraints (2.53) force the functions in $V_h$ to be continuous on cell boundaries. However, this is usually not a problem, since we can perform all operations on the restrictions of functions to the local cells.

The local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space $V_h$. For the linear Lagrange triangle, choosing the degrees of freedom as point evaluation at the vertices ensures that all functions in $V_h$ must be continuous at the two vertices of the common edge of any pair of adjacent triangles, and therefore along the entire common edge. It follows that the functions in $V_h$ are continuous throughout the domain $\Omega$. As a consequence, the space of piecewise linears generated by the Lagrange triangle is $H^1$-conforming; that is, $V_h \subset H^1(\Omega)$.

One may also consider degrees of freedom defined by point evaluation at the midpoint of each edge. This is the so-called Crouzeix–Raviart triangle. The corresponding global Crouzeix–Raviart space $V_h$ is consequently continuous only at edge midpoints. The Crouzeix–Raviart triangle is an example of an $H^1$-*nonconforming* element; that is, the function space $V_h$ constructed from a set of Crouzeix–Raviart elements is not a subspace of $H^1$. Other choices of degrees of freedom may ensure continuity of normal components, like for the $H(\mathrm{div})$-conforming Brezzi–Douglas–Marini elements, or tangential components, as for the $H(\mathrm{curl})$-conforming Nédélec elements. In Chapter 5, other examples of elements are given which ensure different kinds of continuity by the choice of degrees of freedom

Figure 2.6: The (affine) map $F_T$ from a reference cell $\hat{T}$ to a cell $T \in \mathcal{T}_h$.




and local-to-global mapping.

### 2.4.6 *The mapping from the reference element*

As we have seen, the global function space $V_h$ may be described by a mesh $\mathcal{T}_h$, a set of finite elements $\{(T, \mathcal{V}_T, \mathcal{L}_T)\}_{T \in \mathcal{T}_h}$ and a set of local-to-global mappings $\{\iota_T\}_{T \in \mathcal{T}_h}$. We may simplify this description further by introducing a *reference finite element* $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$, where $\hat{\mathcal{L}} = \{\hat{\ell}_1, \hat{\ell}_2, \ldots, \hat{\ell}_{\hat{n}}\}$, and a set of invertible mappings $\{F_T\}_{T \in \mathcal{T}_h}$ that map the reference cell $\hat{T}$ to the cells of the mesh:

$$T = F_T(\hat{T}) \quad \forall\, T \in \mathcal{T}_h. \tag{2.54}$$

This is illustrated in Figure 2.6. Note that $\hat{T}$ is generally not part of the mesh.

For function spaces discretizing $H^1$ as in (2.7), the mapping $F_T$ is typically *affine*; that is, $F_T$ can be written in the form $F_T(\hat{x}) = A_T\hat{x} + b_T$ for some matrix $A_T \in \mathbb{R}^{d \times d}$ and some vector $b_T \in \mathbb{R}^d$, or else *isoparametric*, in which case the components of $F_T$ are functions in $\hat{\mathcal{V}}$. For function spaces discretizing $H(\mathrm{div})$ like in (2.16) or $H(\mathrm{curl})$, the appropriate mappings are the contravariant and covariant Piola mappings which preserve normal and tangential components, respectively; see Rognes et al. [2009]. For simplicity, we restrict the following discussion to the case when $F_T$ is affine or isoparametric.

For each cell $T \in \mathcal{T}_h$, the mapping $F_T$ generates a function space on $T$ given by

$$\mathcal{V}_T = \{v : v = \hat{v} \circ F_T^{-1}, \quad \hat{v} \in \hat{\mathcal{V}}\}; \tag{2.55}$$

that is, each function $v = v(x)$ may be expressed as $v(x) = \hat{v}(F_T^{-1}(x)) = \hat{v} \circ F_T^{-1}(x)$ for some $\hat{v} \in \hat{\mathcal{V}}$.

The mapping $F_T$ also generates a set of degrees of freedom $\mathcal{L}_T$ on $\mathcal{V}_T$ given by

$$\mathcal{L}_T = \{\ell_i : \ell_i(v) = \hat{\ell}_i(v \circ F_T), \quad i = 1, 2, \ldots, \hat{n}\}. \tag{2.56}$$

The mappings $\{F_T\}_{T \in \mathcal{T}_h}$ thus generate from the reference finite element $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$ a set of finite elements $\{(T, \mathcal{V}_T, \mathcal{L}_T)\}_{T \in \mathcal{T}_h}$ given by

$$\begin{aligned} T &= F_T(\hat{T}), \\ \mathcal{V}_T &= \{v : v = \hat{v} \circ F_T^{-1}, \quad \hat{v} \in \hat{\mathcal{V}}\}, \\ \mathcal{L}_T &= \{\ell_i : \ell_i(v) = \hat{\ell}_i(v \circ F_T), \quad i = 1, 2, \ldots, \hat{n} = n_T\}. \end{aligned} \tag{2.57}$$

By this construction, we also obtain the nodal basis functions $\{\phi_i^T\}_{i=1}^{n_T}$ on $T$ from a set of nodal basis functions $\{\hat{\phi}_i\}_{i=1}^{\hat{n}}$ on the reference element satisfying $\hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}$. To see this, we let $\phi_i^T = \hat{\phi}_i \circ F_T^{-1}$ for $i = 1, 2, \ldots, n_T$ and find that

$$\ell_i^T(\phi_j^T) = \hat{\ell}_i(\phi_j^T \circ F_T) = \hat{\ell}_i(\hat{\phi}_j \circ F_T^{-1} \circ F_T) = \hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}, \tag{2.58}$$

so $\{\phi_i^T\}_{i=1}^{n_T}$ is a nodal basis for $\mathcal{V}_T$.

We may therefore define the function space $V_h$ by specifying a mesh $\mathcal{T}_h$, a reference finite element $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$, a set of local-to-global mappings $\{\iota_T\}_{T \in \mathcal{T}_h}$ and a set of mappings $\{F_T\}_{T \in \mathcal{T}_h}$ from the reference cell $\hat{T}$. Note that in general, the mappings need not be of the same type for all cells $T$ and not all finite elements need to be generated from the same reference finite element. In particular, one could employ a different (higher-degree) isoparametric mapping for cells on a curved boundary.

The above construction is valid for so-called affine-equivalent elements [Brenner and Scott, 2008] like the family of $H^1$-conforming Lagrange finite elements. A similar construction is possible for $H(\mathrm{div})$- and $H(\mathrm{curl})$-conforming elements, like the Brezzi–Douglas–Marini and Nédélec elements, where an appropriate Piola mapping must be used to map the basis functions (while an affine map may still be used to map the geometry). However, not all finite elements may be generated from a reference finite element using this simple construction. For example, this construction fails for the family of Hermite finite elements [Ciarlet, 2002, Brenner and Scott, 2008].

## 2.5 Finite element solvers

Finite elements provide a powerful methodology for discretizing differential equations, but solving the resulting algebraic systems also presents a challenge, even for linear systems. Good solvers must handle the sparsity and possible ill-conditioning of the algebraic system, and also scale well on parallel computers. The linear solve is a fundamental operation not only in linear problems, but also within each iteration of a nonlinear solve via Newton's method, an eigenvalue solve, or time-stepping.

A classical approach that has been revived recently is direct solution, based on Gaussian elimination. Thanks to techniques enabling parallel scalability and recognizing block structure, packages such as UMFPACK [Davis, 2004] and SuperLU [Li, 2005] have made direct methods competitive for quite large problems.

The 1970s and 1980s saw the advent of modern iterative methods. These grew out of classical iterative methods such as relaxation methods and the conjugate gradient iteration of Hestenes and Stiefel [1952]. These techniques can use much less memory than direct methods and are easier to parallelize.

Multigrid methods [Brandt, 1977, Wesseling, 1992] use relaxation techniques on a hierarchy of meshes to solve elliptic equations, typically for symmetric problems, in nearly linear time. However, they require a hierarchy of meshes that may not always be available. This motivated the introduction of *algebraic* multigrid methods (AMG) that mimic mesh coarsening, working only on the matrix entries. Successful AMG distributions include the Hypre package [Falgout and Yang, 2002] and the ML package distributed as part of Trilinos [Heroux et al., 2005].

Krylov methods such as conjugate gradients and GMRES [Saad and Schultz, 1986] generate a sequence of approximations converging to the solution of the linear system. These methods are based only on the matrix–vector product. The performance of these methods is significantly improved by use of *preconditioners*, which transform the linear system

$$AU = b \tag{2.59}$$

into

$$P^{-1}AU = P^{-1}b, \tag{2.60}$$

which is known as left preconditioning. The preconditioner $P^{-1}$ may also be applied from the right by recognizing that $AU = (AP^{-1})(PU)$ and solving the modified system for the matrix $AP^{-1}$, followed by an additional solve to obtain $U$ from the solution $PU$. To ensure good convergence, the preconditioner $P^{-1}$ should be a good approximation of $A^{-1}$. Some preconditioners are strictly algebraic, meaning they only use information available from the entries of $A$. Classical relaxation methods such as Gauss–Seidel may be used as preconditioners, as can so-called incomplete factorizations [Manteuffel, 1980, Axelsson, 1986, Saad, 1994]. Multigrid, whether geometric or algebraic, also can serve as a powerful preconditioner. Other kinds of preconditioners require special knowledge about the differential equation being solved and may require new matrices modeling related physical processes. Such methods are sometimes called *physics-based* preconditioners. An automated system, such as FEniCS, provides an interesting opportunity to assist with the development and implementation of these powerful but less widely used methods.

Fortunately, many of the methods discussed here are included in modern libraries such as PETSc [Balay et al., 2004] and Trilinos [Heroux et al., 2005]. FEniCS typically interacts with the solvers discussed here through these packages and so mainly need to be aware of the various methods at a high level, such as when the various methods are appropriate and how to access them.

## 2.6 Finite element error estimation and adaptivity

The error $e = u - u_h$ in a computed finite element solution $u_h$ approximating the exact solution $u$ of (2.20) may be estimated either *a priori* or *a posteriori*.

*A priori* error estimates express the error in terms of the regularity of the exact (unknown) solution and may give useful information about the order of convergence of a finite element method. *A posteriori* error estimates express the error in terms of computable quantities like the residual and (possibly) the solution of an auxiliary dual problem, as described below.

### 2.6.1 A priori *error analysis*

We consider the linear variational problem (2.20). We first assume that the bilinear form $a$ and the linear form $L$ are continuous (bounded); that is, there exists a constant $C > 0$ such that

$$\begin{aligned} a(v, w) &\leqslant C\|v\|_V\|w\|_V, \\ L(v) &\leqslant C\|v\|_V, \end{aligned} \tag{2.61}$$

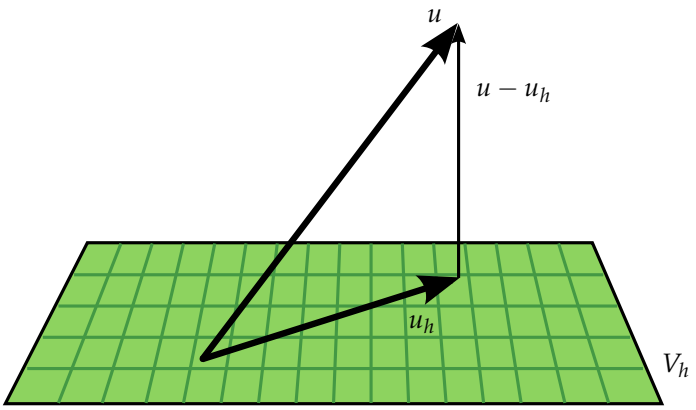



Figure 2.7: If the bilinear form $a$ is symmetric, then the finite element solution $u_h \in V_h \subset V$ is the $a$-projection of $u \in V$ onto the subspace $V_h$ and is consequently the best possible approximation of $u$ in the subspace $V_h$ (in the norm defined by the bilinear form $a$). This follows by the Galerkin orthogonality $\langle u - u_h, v\rangle_a \equiv a(u - u_h, v) = 0$ for all $v \in V_h$.

for all $v, w \in V$. For simplicity, we assume in this section that $V = \hat{V}$ is a Hilbert space. For (2.1), this corresponds to the case of homogeneous Dirichlet boundary conditions and $V = H_0^1(\Omega)$. Extensions to the general case $V \neq \hat{V}$ are possible; see for example Oden and Demkowicz [1996]. We further assume that the bilinear form $a$ is coercive ($V$-elliptic); that is, there exists a constant $\alpha > 0$ such that

$$a(v, v) \geqslant \alpha \|v\|_V^2, \tag{2.62}$$

for all $v \in V$. It then follows by the Lax–Milgram theorem [Lax and Milgram, 1954] that there exists a unique solution $u \in V$ to the variational problem (2.20).

To derive an *a priori* error estimate for the approximate solution $u_h$ defined by the discrete variational problem (2.22), we first note that

$$a(u - u_h, v) = a(u, v) - a(u_h, v) = L(v) - L(v) = 0 \tag{2.63}$$

for all $v \in V_h \subset V$ (the Galerkin orthogonality). By the coercivity and continuity of the bilinear form $a$, we find that

$$\begin{aligned} \alpha \|u - u_h\|_V^2 &\leqslant a(u - u_h, u - u_h) = a(u - u_h, u - v) + a(u_h - u, v - u_h) \\ &= a(u - u_h, u - v) \leqslant C \|u - u_h\|_V \, \|u - v\|_V. \end{aligned} \tag{2.64}$$

for all $v \in V_h$. It follows that

$$\|u - u_h\|_V \leqslant \frac{C}{\alpha} \|u - v\|_V \quad \forall\, v \in V_h. \tag{2.65}$$

The estimate (2.65) is referred to as Cea's lemma. We note that when the bilinear form $a$ is symmetric, it is also an inner product. We may then take $\|v\|_V = \sqrt{a(v, v)}$ and $C = \alpha = 1$. In this case, $u_h$ is the $a$-projection onto $V_h$ and Cea's lemma states that

$$\|u - u_h\|_V \leqslant \|u - v\|_V \quad \forall\, v \in V_h; \tag{2.66}$$

that is, $u_h$ is the best possible solution of the variational problem (2.20) in the subspace $V_h$. This is illustrated in Figure 2.7.

Cea's lemma together with a suitable interpolation estimate now yields an *a priori* error estimate

for $u_h$. By choosing $v = \pi_h u$, where $\pi_h : V \to V_h$ is an interpolation operator into $V_h$, we find that

$$\|u - u_h\|_V \leqslant \frac{C}{\alpha}\|u - \pi_h u\|_V \leqslant \frac{CC_i}{\alpha}\|h^p D^{q+1} u\|_{L^2}, \tag{2.67}$$

where $C_i$ is an interpolation constant and the values of $p$ and $q$ depend on the accuracy of interpolation and the definition of $\|\cdot\|_V$. For the solution of Poisson's equation in $V = H_0^1$, we have $C = \alpha = 1$ and $p = q = 1$.

### 2.6.2 A posteriori *error analysis*

*Energy norm error estimates.* The continuity and coercivity of the bilinear form $a$ also allow the derivation of an *a posteriori* error estimate. This type of error estimate is obtained by relating the size of the error to the size of the (weak) residual $r : \hat{V} \to \mathbb{R}$ defined by

$$r(v) = L(v) - a(u_h, v). \tag{2.68}$$

Note that the weak residual is formally related to the *strong residual* $R \in \hat{V}'$ by $r(v) = \langle R, v\rangle$ for all $v \in \hat{V}$.

We first note that the $V$-norm of the error $e = u - u_h$ is equivalent to the $V'$-norm of the residual $r$. To see this, note that by the continuity of the bilinear form $a$, we have

$$r(v) = L(v) - a(u_h, v) = a(u, v) - a(u_h, v) = a(u - u_h, v) \leqslant C\|u - u_h\|_V \, \|v\|_V. \tag{2.69}$$

Furthermore, by coercivity, we find that

$$\alpha\|u - u_h\|_V^2 \leqslant a(u - u_h, u - u_h) = a(u, u - u_h) - a(u_h, u - u_h) = L(u - u_h) - a(u_h, u - u_h) = r(u - u_h). \tag{2.70}$$

It follows that

$$\alpha\|u - u_h\|_V \leqslant \|r\|_{V'} \leqslant C\|u - u_h\|_V, \tag{2.71}$$

where $\|r\|_{V'} = \sup_{v\in V, v\neq 0} r(v)/\|v\|_V$.

The estimates (2.67) and (2.71) are sometimes referred to as *energy norm* error estimates. This is the case when the bilinear form $a$ is symmetric and thus defines an inner product. One may then take $\|v\|_V = \sqrt{a(v, v)}$ and $C = \alpha = 1$. In this case, it follows that

$$\eta \equiv \|e\|_V = \|r\|_{V'}. \tag{2.72}$$

The term energy norm refers to $a(v, v)$ corresponding to physical energy in many applications.

*Goal-oriented error estimates.* The classical *a priori* and *a posteriori* error estimates (2.67) and (2.71) relate the $V$-norm of the error $e = u - u_h$ to the regularity of the exact solution $u$ and the residual $r = L(v) - a(u_h, v)$ of the finite element solution $u_h$, respectively. However, in applications it is often necessary to control the error in a certain *output functional* $\mathcal{M} : V \to \mathbb{R}$ of the computed solution to within some given tolerance $\epsilon > 0$. Typical functionals are average values of the computed solution, such as the lift or drag of an object immersed in a flow field. In these situations, one would ideally like to choose the finite element space $V_h \subset V$ such that the finite element solution $u_h$ satisfies

$$\eta \equiv |\mathcal{M}(u) - \mathcal{M}(u_h)| \leqslant \epsilon \tag{2.73}$$

with minimal computational work. We assume here that both the output functional and the variational problem are linear, but the analysis may be easily extended to the full nonlinear case [Eriksson

et al.1995, Becker and Rannacher 2001].

To estimate the error in the output functional $\mathcal{M}$, we introduce an auxiliary *dual* problem: find $z \in V^*$ such that

$$a^*(z, v) = \mathcal{M}(v) \quad \forall\, v \in \hat{V}^*. \tag{2.74}$$

We note here that the functional $\mathcal{M}$ enters as data in the dual problem. The dual (adjoint) bilinear form $a^* : V^* \times \hat{V}^* \to \mathbb{R}$ is defined by

$$a^*(v, w) = a(w, v) \quad \forall\, (v, w) \in V^* \times \hat{V}^*. \tag{2.75}$$

The dual trial and test spaces are given by

$$\begin{aligned} V^* &= \hat{V}, \\ \hat{V}^* &= V_0 = \{v - w : v, w \in V\}; \end{aligned} \tag{2.76}$$

that is, the dual trial space is the primal test space and the dual test space is the primal trial space modulo boundary conditions. In particular, if $V = u_0 + \hat{V}$ then $V^* = \hat{V}^* = \hat{V}$, and both the dual test and trial functions vanish at Dirichlet boundaries. The definition of the dual problem leads us to the following representation of the error:

$$\begin{aligned} \mathcal{M}(u) - \mathcal{M}(u_h) &= \mathcal{M}(u - u_h) \\ &= a^*(z, u - u_h) \\ &= a(u - u_h, z) \\ &= L(z) - a(u_h, z) \\ &= r(z). \end{aligned} \tag{2.77}$$

We find that the error is exactly represented by the residual of the dual solution:

$$\mathcal{M}(u) - \mathcal{M}(u_h) = r(z). \tag{2.78}$$

### 2.6.3 Adaptivity

As seen above, one may estimate the error in a computed finite element solution $u_h$ in the $V$-norm or an output functional by estimating the size of the residual $r$. This may be done in several different ways. The estimate typically involves integration by parts to recover the strong element-wise residual of the original PDE, possibly in combination with the solution of local problems over cells or patches of cells. In the case of the standard piecewise linear finite element approximation of Poisson's equation (2.1), one may obtain the following estimate:

$$\|u - u_h\|_V \equiv \|\nabla e\|_{L^2} \leqslant C \left( \sum_{T \in \mathcal{T}_h} h_T^2 \|R\|_T^2 + h_T \|[\partial_n u_h]\|_{\partial T}^2 \right)^{1/2}, \tag{2.79}$$

where $R|_T = f|_T + \Delta u_h|_T$ is the strong residual, $h_T$ denotes the mesh size (the diameter of the smallest circumscribed sphere around each cell $T$) and $[\partial_n u_h]$ denotes the jump of the normal derivative across mesh facets. For a derivation of this estimate, see for example Elman et al. [2005]. Letting $\eta_T^2 = h_T^2 \|R\|_T^2 + h_T \|[\partial_n u_h]\|_{\partial T}^2$, we obtain the estimate

$$\|u - u_h\|_V \leqslant \eta_h \equiv C \left( \sum_T \eta_T^2 \right)^{1/2}. \tag{2.80}$$

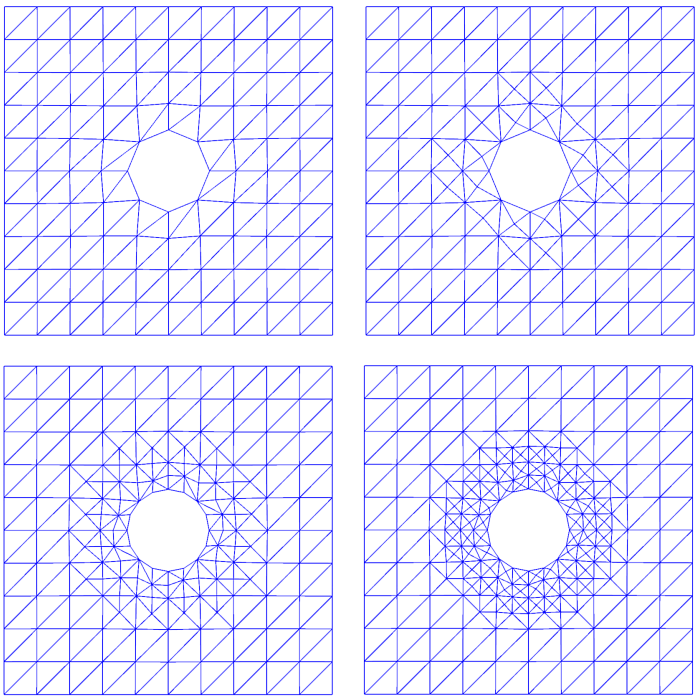


Figure 2.8: A sequence of adaptively refined meshes obtained by successive refinement of an original coarse mesh.

An adaptive algorithm seeks to determine a mesh size $h = h(x)$ such that $\eta_h \leqslant \epsilon$. Starting from an initial coarse mesh, the mesh is successively refined in those cells where the error indicator $\eta_T$ is large. Several strategies are available, such as refining the top fraction of all cells where $\eta_T$ is large, say the first 20% of all cells ordered by the size of $\eta_T$. Other strategies include refining all cells where $\eta_T$ is above a certain fraction of $\max_{T \in \mathcal{T}_h} \eta_T$, or refining a top fraction of all cells such that the sum of their error indicators account for a significant fraction of $\eta_h$ (so-called *Dörfler marking* [Dörfler, 1996]).

Once the mesh has been refined, a new solution and new error indicators can be computed. The process is then repeated until either $\eta_h \leqslant \epsilon$ (the stopping criterion) or the available resources (CPU time and memory) have been exhausted. The adaptive algorithm yields a sequence of successively refined meshes as illustrated in Figure 2.8. For time-dependent problems, an adaptive algorithm needs to decide both on the local mesh size and the size of the (local) time step as functions of space *and* time. Ideally, the error estimate $\eta_h$ is close to the actual error, as measured by the *efficiency index* $\eta_h / \eta$ which should be close to and bounded below by one.

## 2.7 *Automating the finite element method*

The FEniCS Project seeks to automate the solution of differential equations. This is a formidable task, but it may be approached by an automation of the finite element method. In particular, this automation relies on the following key steps:

(i) automation of discretization,

(ii) automation of discrete solution,

(iii) automation of error control.

Since its inception in 2003, the FEniCS Project has been concerned mainly with the automation of discretization, resulting in the development of the form compilers FFC and SyFi/SFC, the code generation interface UFC, the form language UFL, and a generic assembler implemented as part of DOLFIN. As a result, variational problems for a large class of partial differential equations may now be automatically discretized by the finite element method using FEniCS. For the automation of discrete solution; that is, the solution of linear and nonlinear systems arising from the automated discretization of variational problems, interfaces to state-of-the-art libraries for linear algebra have been implemented as part of DOLFIN. Ongoing work is now seeking to automate error control by automated error estimation and adaptivity. In the following chapters, we return to specific aspects of the automation of the finite element method developed as part of the FEniCS Project. The mathematical methodology behind the FEniCS Project has also been described in a number of scientific works. For further reading, we refer to Logg [2007], Logg and Wells [2010], Kirby [2004], Kirby and Logg [2006], Alnæs et al. [2009], Alnæs and Mardal [2010], Kirby et al. [2005, 2006], Kirby and Logg [2007, 2008], Kirby and Scott [2007], Kirby [2006], Ølgaard et al. [2008], Rognes et al. [2009], Ølgaard and Wells [2010], Logg [2009].

## 2.8 Historical notes

In 1915, Boris Grigoryevich Galerkin formulated a general method for solving differential equations [Galerkin, 1915]. A similar approach was presented sometime earlier by Bubnov. Galerkin's method, or the Bubnov–Galerkin method, was originally formulated with global polynomials and goes back to the variational principles of Leibniz, Euler, Lagrange, Dirichlet, Hamilton, Castigliano [Castigliano, 1879], Rayleigh [Rayleigh, 1870] and Ritz [Ritz, 1908]. Galerkin's method with piecewise polynomial spaces $(V_h, \hat{V}_h)$ is known as the *finite element method*. The finite element method was introduced by engineers for structural analysis in the 1950s and was independently proposed by Courant [Courant, 1943]. The exploitation of the finite element method among engineers and mathematicians exploded in the 1960s. Since then, the machinery of the finite element method has been expanded and refined into a comprehensive framework for the design and analysis of numerical methods for differential equations; see Zienkiewicz et al. [2005], Strang and Fix [1973], Ciarlet [1976], Becker et al. [1981], Hughes [1987], Brenner and Scott [2008]. Recently, the quest for compatible (stable) discretizations of mixed variational problems has led to the development of finite element exterior calculus [Arnold et al., 2006].

Work on *a posteriori* error analysis of finite element methods dates back to the pioneering work of Babuška and Rheinboldt [1978]. Important references include the works by Bank and Weiser [1985], Zienkiewicz and Zhu [1987], Eriksson and Johnson [1991, 1995a], Eriksson and Johnson, 1995b,c], Eriksson et al. [1998], Ainsworth and Oden [1993] and the reviews papers [Eriksson et al., 1995, Verfürth, 1994, 1999, Ainsworth and Oden, 2000, Becker and Rannacher, 2001].

# 3 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].

## 3.1 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 3.1 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 4) are passed to the form compiler FFC (Chapter 5) or SFC (Chapter 5) to generate UFC code (Chapter 5), 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 5), the just-in-time compilation utility Instant (Chapter 5) and the optional optimizing backend FErari (Chapter 5). 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].

## 3.2 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,

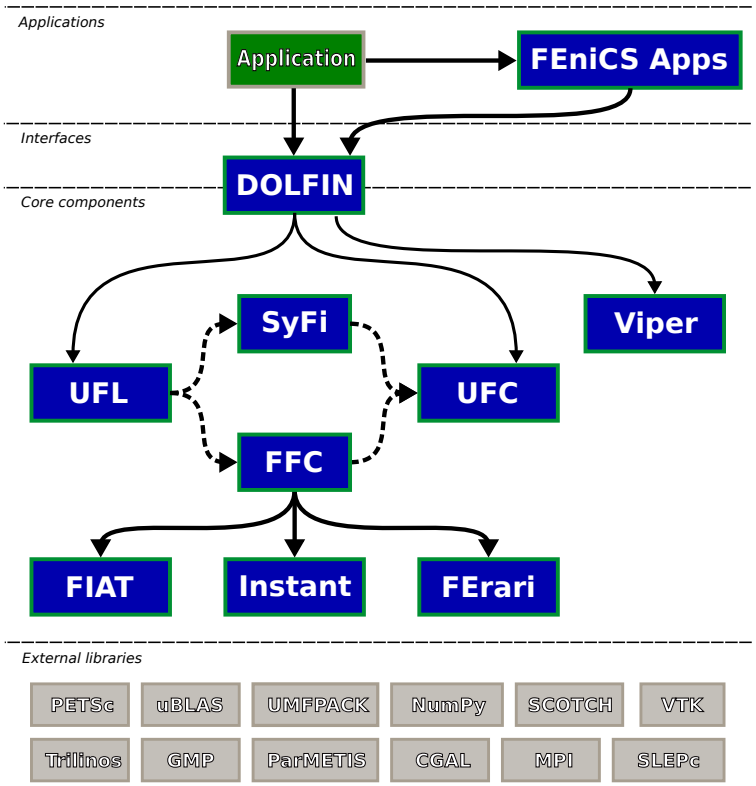



Figure 3.1: DOLFIN functions as the main user interface of FEniCS and handles the communication between the various components of FEniCS and external software. Solid lines indicate dependencies and dashed lines indicate data flow.

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.

### 3.2.1 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 3.2). 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.

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

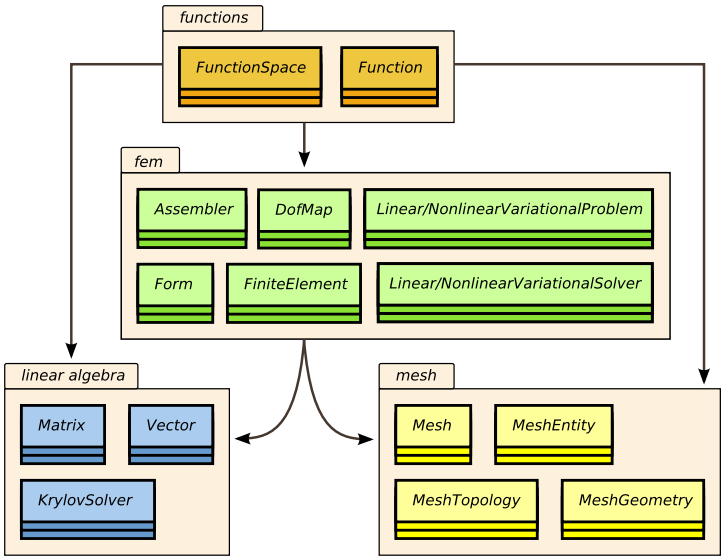



Figure 3.2: Schematic overview of some of the most important components and classes of DOLFIN. Arrows indicate dependencies.

issue in Section 3.3.

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;
}
```

### 3.2.2 *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 *
```

## 3.3 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].

### 3.3.1 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

$$F(x) = 0, \tag{3.1}$$

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.

### 3.3.2 *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">
```

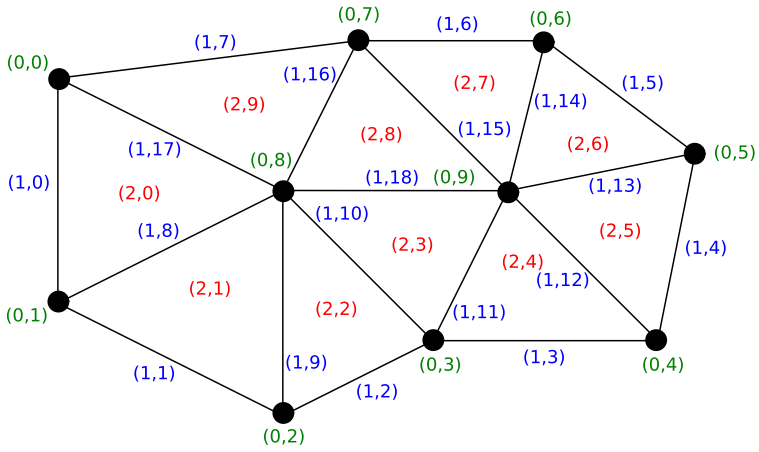



Figure 3.3: Each entity of a mesh is identified by a pair $(d, i)$ which specifies the topological dimension $d$ and a unique index $i$ for the entity within the set of entities of dimension $d$.

```
  <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 3.3 and Table 3.1. 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

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

Table 3.1: 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.

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
```

*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 $D+1$.

*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

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

Table 3.2: 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.

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 3.2, 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);
```

*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)
```

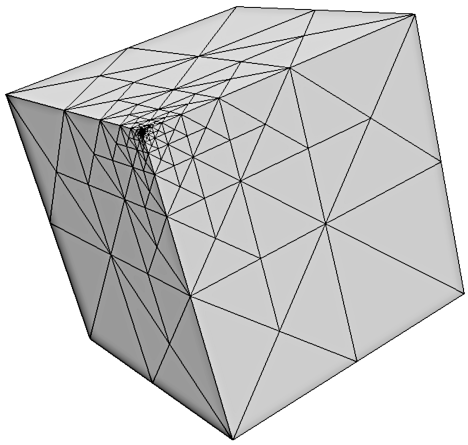


Figure 3.4: A locally refined mesh obtained by repeated marking of the cells close to one of the corners of the unit cube.

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 3.4.

*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 3.4.

### 3.3.3 *Finite elements*

The concept of a finite element as discussed in Chapters 2 and 5 (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 5 and 5, 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 4). 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

| 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 3.3: List of finite elements supported by DOLFIN 1.0. Elements in grey italics are partly supported in FEniCS but not throughout the entire toolchain.

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 5.

Table 3.3 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)
```

### 3.3.4 *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.

### 3.3.5 Functions

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

$$u_h(x) = \sum_{j=1}^{N} U_j \phi_j(x), \tag{3.2}$$

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`:

`Function` = (`FunctionSpace`, `GenericVector`).

The `FunctionSpace` defines the function space $V_h$ and the `GenericVector` holds the vector $U$ of degrees of freedom; see Figure 3.5. When running in parallel on a distributed memory architecture, the `FunctionSpace` and the `GenericVector` are distributed across the processes.

*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)
```

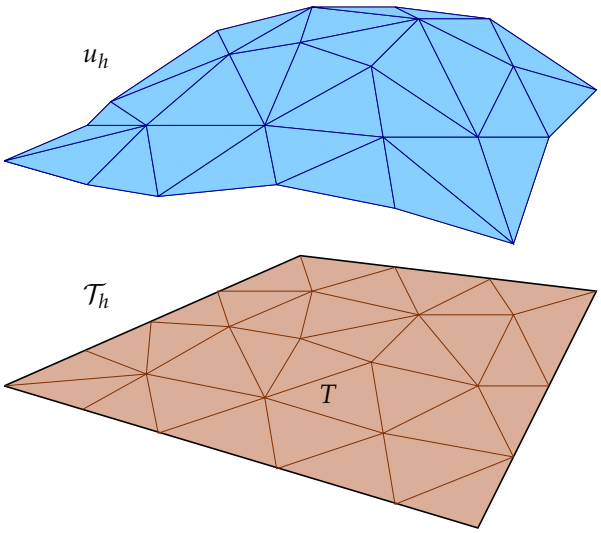



Figure 3.5: A piecewise linear finite element function $u_h$ on a mesh consisting of triangular elements. The vector of degrees of freedom $U$ is given by the values of $u_h$ at the mesh vertices.

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 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);
```

[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.

*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`.

### 3.3.6 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.

### 3.3.7 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 4) are compiled using one of the form compilers FFC or SFC (Chapters 5 and 5), and the generated UFC code (Chapter 5) 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:

$$a(u, v) = \int_\Omega \sigma(u) : \epsilon(v) \, \mathrm{d}x, \tag{3.3}$$

$$L(v) = \int_\Omega f \cdot v \, \mathrm{d}x. \tag{3.4}$$

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 4.

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
```

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:

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

*Python code*

```
from dolfin import *

mesh = UnitSquare(8, 8)
V = VectorElement(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.

### 3.3.8 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 5. 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 5).

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 3.4.

### 3.3.9 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.

*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

$$\begin{aligned} -\operatorname{div}\sigma(u) &= f \quad \text{in } \Omega, \\ \sigma \cdot n &= g \quad \text{on } \Gamma_{\mathrm{N}} \subset \partial\Omega, \\ u &= u_0 \quad \text{on } \Gamma_{\mathrm{D}} \subset \partial\Omega, \end{aligned} \tag{3.5}$$

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

$$\int_\Omega \sigma(u) : \epsilon(v)\,\mathrm{d}x - \int_{\partial\Omega} (\sigma \cdot n) \cdot v\,\mathrm{d}s = \int_\Omega f \cdot v\,\mathrm{d}x, \tag{3.6}$$

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

$$\int_\Omega \sigma(u) : \epsilon(v)\,\mathrm{d}x = \int_\Omega f \cdot v\,\mathrm{d}x + \int_{\Gamma_{\mathrm{N}}} g \cdot v\,\mathrm{d}s. \tag{3.7}$$

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:

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

*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 $\Gamma_N$, 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 (3.5) 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.

### 3.3.10 *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

$$a(u, v) = L(v) \quad \forall\, v \in \hat{V}. \tag{3.8}$$

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

$$F(u; v) = 0 \quad \forall v \in \hat{V}. \tag{3.9}$$

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)
```

### 3.3.11 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 3.4. 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")
```

| 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 3.4: List of file formats supported by the `dolfin-convert` script.

*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)
```

Example plots generated using the `plot` command are presented in Figures 3.6 and 3.7. 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;
```

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

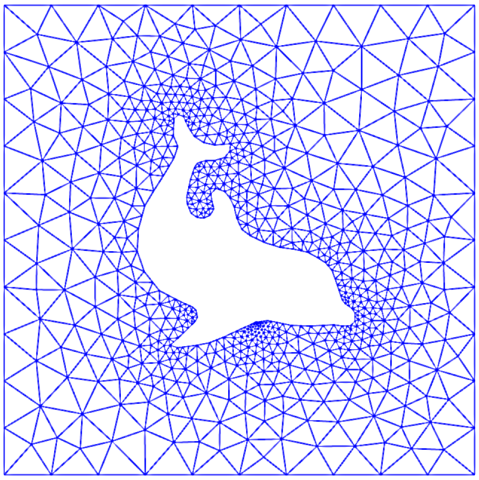
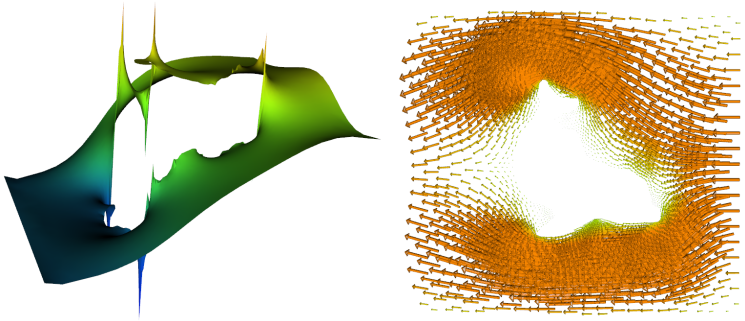


Figure 3.7: Plotting a scalar and a vector-valued function using the DOLFIN `plot` command, here the pressure (left) and velocity (right) from a solution of the Stokes equations on the mesh from Figure 3.6.



*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)
```

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

Table 3.5: Log levels in DOLFIN.

### 3.3.12 *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 3.5. 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 3.5.

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.

### 3.3.13 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
```

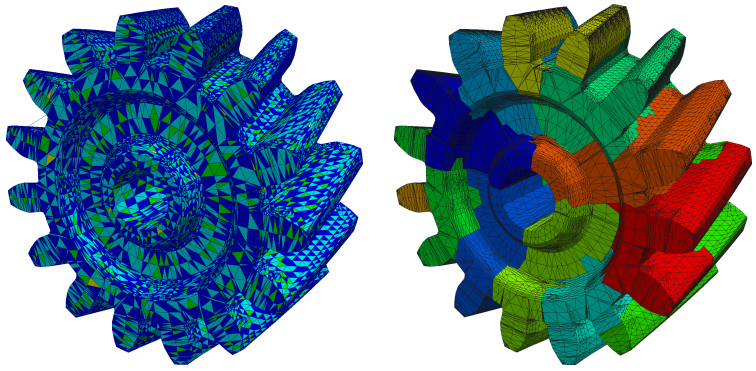
*Storing parameters to file.* It can be useful to store parameter values to file, for example to document which parameter values were used to run a simulation or to reuse a set of parameter values from a previous run. The following code illustrates how to write and then read back parameter values to/from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

*C++ code*

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

*Python code*

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



(a) (b)

Figure 3.8: 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.

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.

## 3.4 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.

### 3.4.1 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 3.8a), and for distributed parallelization a mesh partitioning approach is used (see Figure 3.8b). 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 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 3.8a 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 3.8b 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.

### 3.4.2 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 5 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.

### 3.4.3 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 5). 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 5 and 5). 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.

### 3.4.4 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 5 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))]
```

## 3.5 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 5.

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.

# 4 *UFL: a finite element form language*

By Martin Sandve Alnæs

The Unified Form Language – UFL [Alnæs and Logg, 2009] – is a domain specific language for the declaration of finite element discretizations of variational forms and functionals. More precisely, the language defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation.

The FEniCS project provides a framework for building applications for solving partial differential equations (PDEs). UFL is one of the core components of this framework. It defines the language you *express* your PDEs in. It is the input language and front-end of the form compilers FFC and SFC, which are covered in Chapter 5 and Chapter 5. The UFL implementation also provides algorithms that the form compilers can use to simplify the compilation process. The output from these form compilers is C++ [Stroustrup, 1997] code that conforms to the UFC specification, which is explained in Chapter 5. This code can be used with the C++/Python library DOLFIN, which is covered in Chapter 3, to efficiently assemble linear systems and compute solutions to PDEs. Note that this chapter does not cover how to actually solve equations defined in UFL. See Chapter 1 for a tutorial on how to use the complete FEniCS framework to solve equations.

This chapter is intended both for the FEniCS user who wants to learn how to express her equations, and for other FEniCS developers and technical users who want to know how UFL works on the inside. Therefore, the sections of this chapter are organized with an increasing amount of technical details. Sections 4.1–4.5 give an overview of the language as seen by the end-user and is intended for all audiences. Sections 4.6–4.9 explain the design of the implementation and dive into some implementation details. Many details of the language have to be omitted in a text such as this, and we refer to the UFL manual [Alnæs and Logg, 2009] for a more thorough description. Note that this chapter refers to UFL version 1.0.0, and both the user interface and the implementation may change in future versions.

Starting with a brief overview, we mention the main design goals for UFL and show an example implementation of a non-trivial PDE in Section 4.1. Next, we look at how to define finite element spaces in Section 4.2, followed by the overall structure of forms and their declaration in Section 4.3. The main part of the language is concerned with defining expressions from a set of data types and operators, which are discussed in Section 4.4. Operators applying to entire forms are the topic of Section 4.5.

The technical part of the chapter begins with Section 4.6 which discusses the representation of expressions. Building on the notation and data structures defined there, how to compute derivatives is discussed in Section 4.7. Some central internal algorithms and key issues in their implementation are discussed in Section 4.8. Implementation details, some of which are specific to the programming language Python [van Rossum et al.], are the topic of Section 4.9. Finally, Section 4.10 discusses future prospects of the UFL project.

#### 4.0.1 Related work

The combination of domain specific languages and symbolic computing with finite element methods has been pursued from other angles in several other projects. Sundance [Long, 2003, 2004b,a] implements a symbolic engine directly in C++ to define variational forms, and has support for automatic differentiation. The Life [Prud'homme, 2006b,a] project uses a domain specific language embedded in C++, based on expression template techniques to specify variational forms. SfePy [Cimrman et al., 2008] uses SymPy as a symbolic engine, extending it with finite element methods. GetDP [Dular and Geuzaine, 2005] is another project using a domain specific language for variational forms. The Mathematica package AceGen [Korelc, 1997, 2002] uses the symbolic capabilities of Mathematica to generate efficient code for finite element methods. All these packages have in common a focus on high level descriptions of partial differential equations to achieve higher human efficiency in the development of simulation software.

UFL almost resembles a library for symbolic computing, but its scope, goals and priorities are different from generic symbolic computing projects such as GiNaC [Bauer et al., 2002], Swiginac [Skavhaug and Čertík, 2009] and SymPy [Čertík et al., 2009]. Intended as a domain specific language and form compiler frontend, UFL is not suitable for large scale symbolic computing.

## 4.1 Overview

#### 4.1.1 Design goals

UFL is a unification, refinement and reimplementation of the form languages used in previous versions of FFC and SFC. The development of this language has been motivated by several factors, the most important being:

- A richer form language, especially for expressing nonlinear PDEs.
- Automatic differentiation of expressions and forms.
- Improving the performance of the form compiler technology to handle more complicated equations efficiently.

UFL fulfills all these requirements, and by this it represents a major step forward in the capabilities of the FEniCS project.

Tensor algebra and index notation support is modeled after the FFC form language and generalized further. Several nonlinear operators and functions which only SFC supported before have been included in the language. Differentiation of expressions and forms has become an integrated part of the language, and is much easier to use than the way these features were implemented in SFC before. In summary, UFL combines the best of FFC and SFC in one unified form language and adds additional capabilities.

The efficiency of code generated by the new generation of form compilers based on UFL has been verified to match previous form compiler benchmarks [Alnæs and Mardal, 2010, Ølgaard and Wells, 2010]. The form compilation process is now fast enough to blend into the regular application build process. Complicated forms that previously required too much memory to compile, or took tens of minutes or even hours to compile, now compiles in seconds with both SFC and FFC.

#### 4.1.2 Motivational example

One major motivating example during the initial development of UFL has been the equations for elasticity with large deformations. In particular, models of biological tissue use complicated

hyperelastic constitutive laws with anisotropies and strong nonlinearities. To implement these equations with FEniCS, all three design goals listed above had to be addressed. Below, one version of the hyperelasticity equations and their corresponding UFL implementation is shown. Keep in mind that this is only intended as an illustration of the close correspondence between the form language and the natural formulation of the equations. The meaning of these equations is not necessary for the reader to understand. Chapter 5 covers nonlinear elasticity in more detail. Note that many other examples are distributed together with UFL.

In the formulation of the hyperelasticity equations presented here, the unknown function is the displacement vector field $u$. The material coefficients $c_1$ and $c_2$ are scalar constants. The second Piola-Kirchoff stress tensor $S$ is computed from the strain energy function $W(C)$. $W$ defines the constitutive law, here a simple Mooney-Rivlin law. The equations relating the displacement and stresses read:

$$\begin{aligned} F &= I + \operatorname{grad} u, \\ C &= F^\top F, \\ I_C &= \operatorname{tr}(C), \\ II_C &= \frac{1}{2}(\operatorname{tr}(C)^2 - \operatorname{tr}(CC)), \\ W &= c_1(I_C - 3) + c_2(II_C - 3), \\ S &= 2\frac{\partial W}{\partial C}. \end{aligned} \tag{4.1}$$

For simplicity in this example, we ignore external body and boundary forces and assume a quasi-stationary situation, leading to the following mechanics problem. Find $u$ such that

$$\operatorname{div}(FS) = 0, \quad \text{in } \mathrm{d}x, \tag{4.2}$$

$$u = u_0, \quad \text{on } \mathrm{d}s. \tag{4.3}$$

The finite element method is presented in Chapter 2, so we will only very briefly cover the steps we take here. First we multiply Equation (4.2) with a test function $\phi \in V$, then integrate over the domain $\Omega$, and integrate by parts. The nonlinear variational problem then reads: find $u \in V$ such that

$$L(u;\phi) = \int_\Omega FS : \operatorname{grad}\phi \,\mathrm{d}x = 0 \quad \forall\, \phi \in V. \tag{4.4}$$

Here we have omitted the coefficients $c_1$ and $c_2$ for brevity. Approximating the displacement field as $u \approx u_h = \sum_k u_k \psi_k$, where $\psi_k \in V_h \approx V$ are trial functions, and using Newtons's method to solve the nonlinear equations, we end up with a system of equations to solve

$$\sum_{k=1}^{|V_h|} \frac{\partial L(u_h;\phi)}{\partial u_k} \Delta u_k = -L(u_h;\phi) \quad \forall\, \phi \in V_h. \tag{4.5}$$

A bilinear form $a(u;\psi,\phi)$ corresponding to the left-hand side of Equation (4.5) can be computed automatically by UFL, such that

$$a(u_h;\psi_k,\phi) = \frac{\partial L(u_h;\phi)}{\partial u_k} \quad k = 1,\ldots,|V_h|. \tag{4.6}$$

Figure 4.1 shows an implementation of equations (4.1), (4.4) and (4.6) in UFL. Notice the close

*UFL code*

```
# Finite element spaces
cell = tetrahedron
element = VectorElement("Lagrange", cell, 1)

# Form arguments
phi0 = TestFunction(element)
phi1 = TrialFunction(element)
u = Coefficient(element)
c1 = Constant(cell)
c2 = Constant(cell)

# Deformation gradient Fij = dXi/dxj
I = Identity(cell.d)
F = I + grad(u)

# Right Cauchy-Green strain tensor C with invariants
C = variable(F.T*F)
I_C = tr(C)
II_C = (I_C**2 - tr(C*C))/2

# Mooney-Rivlin constitutive law
W = c1*(I_C-3) + c2*(II_C-3)

# Second Piola-Kirchoff stress tensor
S = 2*diff(W, C)

# Weak forms
L = inner(F*S, grad(phi0))*dx
a = derivative(L, u, phi1)
```

Figure 4.1: UFL implementation of hyperelasticity equations with a Mooney-Rivlin material law.

relation between the mathematical notation and the UFL source code. In particular, note the automated differentiation of both the constitutive law and the residual equation. The operator `diff` can be applied to expressions to differentiate w.r.t designated variables such as `C` here, while the operator `derivative` can be applied to entire forms to differentiate w.r.t. each coefficient of a discrete function such as `u`. The combination of these features allows a new material law to be implemented by simply changing $W$, the rest is automatic. In the following sections, the notation, definitions and operators used in this implementation will be explained.

## 4.2 Defining finite element spaces

A polygonal cell is defined in UFL by a basic shape, and is declared by

*UFL code*

```
cell = Cell(shapestring)
```

UFL defines a set of valid polygonal cell shapes: “interval”, “triangle”, “tetrahedron”, “quadrilateral”, and “hexahedron”. `Cell` objects of all shapes are predefined and can be used instead by writing

*UFL code*

```
cell = tetrahedron
```

In the rest of this chapter, a variable name `cell` will be used where any cell is a valid argument, to make the examples dimension independent wherever possible.

UFL defines syntax for *declaring* finite element spaces, but does not know anything about the actual polynomial basis or degrees of freedom. The polynomial basis is selected implicitly by choosing among predefined basic element families and providing a polynomial degree, but UFL only assumes that there *exists* a basis with a fixed ordering for each finite element space $V_h$; that is,

$$V_h = \text{span}\left\{\phi_j\right\}_{j=1}^{n}. \tag{4.7}$$

Basic scalar elements can be combined to form vector elements or tensor elements, and elements can easily be combined in arbitrary mixed element hierarchies.

The set of predefined[1] element family names in UFL includes "Lagrange" (short name "CG"), representing scalar Lagrange finite elements (continuous piecewise polynomial functions), "Discontinuous Lagrange" (short name "DG"), representing scalar discontinuous Lagrange finite elements (discontinuous piecewise polynomial functions), and a range of other families that can be found in the manual. Each family name has an associated short name for convenience. To print all valid families to screen from Python, call `show_elements()`.

The syntax for declaring elements is best explained with some examples.

*UFL code*

```
cell = tetrahedron

P = FiniteElement("Lagrange", cell, 1)
V = VectorElement("Lagrange", cell, 2)
T = TensorElement("DG", cell, 0, symmetry=True)

TH = V*P
ME = MixedElement(T, V, P)
```

In the first line a polygonal cell is selected from the set of predefined cells. Then a scalar linear Lagrange element `P` is declared, as well as a quadratic vector Lagrange element `V`. Next a symmetric rank 2 tensor element `T` is defined, which is also piecewise constant on each cell. The code proceeds to declare a mixed element `TH`, which combines the quadratic vector element `V` and the linear scalar element `P`. This element is known as the Taylor-Hood element. Finally another mixed element with three subelements is declared. Note that writing `T*V*P` would not result in a mixed element with three direct subelements, but rather `MixedElement(MixedElement(T, V), P)`.

## 4.3 *Defining forms*

Consider Poisson's equation with two different boundary conditions on $\partial\Omega_0$ and $\partial\Omega_1$,

$$a(w;u,v) = \int_\Omega w \,\text{grad}\, u \cdot \text{grad}\, v \,\mathrm{d}x, \tag{4.8}$$

$$L(f,g,h;v) = \int_\Omega fv \,\mathrm{d}x + \int_{\partial\Omega_0} g^2 v \,\mathrm{d}s + \int_{\partial\Omega_1} hv \,\mathrm{d}s. \tag{4.9}$$

These forms can be expressed in UFL as

*UFL code*

```
a = w*dot(grad(u), grad(v))*dx
L = f*v*dx + g**2*v*ds(0) + h*v*ds(1)
```

[1] Form compilers can register additional element families.

where multiplication by the measures `dx`, `ds(0)` and `ds(1)` represent the integrals $\int_{\Omega_0}(\cdot)\,\mathrm{d}x$, $\int_{\partial\Omega_0}(\cdot)\,\mathrm{d}s$, and $\int_{\partial\Omega_1}(\cdot)\,\mathrm{d}s$ respectively.

Forms expressed in UFL are intended for finite element discretization followed by compilation to efficient code for computing the element tensor. Considering the above example, the bilinear form $a$ with one coefficient function $w$ is assumed to be evaluated at a later point with a range of basis functions and the coefficient function fixed, that is

$$V_h^1 = \operatorname{span}\left\{\phi_k^1\right\}, \quad V_h^2 = \operatorname{span}\left\{\phi_k^2\right\}, \quad V_h^3 = \operatorname{span}\left\{\phi_k^3\right\}, \tag{4.10}$$

$$w = \sum_{k=1}^{|V_h^3|} w_k \phi_k^3, \quad \{w_k\} \text{ given}, \tag{4.11}$$

$$A_{ij} = a(w; \phi_i^1, \phi_j^2), \quad i = 1, \ldots, |V_h^1|, \quad j = 1, \ldots, |V_h^2|. \tag{4.12}$$

In general, UFL is designed to express forms of the following generalized form:

$$a(w^1, \ldots, w^n; \phi^1, \ldots, \phi^r) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \,\mathrm{d}x + \sum_{k=1}^{n_e} \int_{\partial\Omega_k} I_k^e \,\mathrm{d}s + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i \,\mathrm{d}S. \tag{4.13}$$

Most of this chapter deals with ways to define the integrand expressions $I_k^c$, $I_k^e$ and $I_k^i$. The rest of the notation will be explained below.

The form arguments are divided in two groups, the basis functions $\phi^1, \ldots, \phi^r$ and the coefficient functions $w^1, \ldots, w^n$. All $\{\phi^k\}$ and $\{w^k\}$ are functions in some discrete function space with a basis. Note that the actual basis functions $\{\phi_j^k\}$ and the coefficients $\{w_k\}$ are never known to UFL, but we assume that the ordering of the basis for each finite element space is fixed. A fixed ordering only matters when differentiating forms, explained in Section 4.7.

Each term of a valid form expression must be a scalar-valued expression integrated exactly once, and they must be linear in $\{\phi^k\}$. Any term may have nonlinear dependencies on coefficient functions. A form with one or two basis function arguments ($r = 1, 2$) is called a linear or bilinear form respectively, ignoring its dependency on coefficient functions. These will be assembled to vectors and matrices when used in an application. A form depending only on coefficient functions ($r = 0$) is called a functional, since it will be assembled to a real number. Multilinear forms where $r > 2$ are supported but not as commonly used.

The entire domain is denoted $\Omega$, the external boundary is denoted $\partial\Omega$, while the set of interior facets of the triangulation is denoted $\Gamma$. Subdomains are marked with a suffix, e.g., $\Omega_k \subset \Omega$. As mentioned above, integration is expressed by multiplication with a measure, and UFL defines the measures `dx`, `ds` and `dS`. In summary, there are three kinds of integrals with corresponding UFL representations

- $\int_{\Omega_k}(\cdot)\,\mathrm{d}x \leftrightarrow$ `(·)*dx(k)`, called a *cell integral*,
- $\int_{\partial\Omega_k}(\cdot)\,\mathrm{d}s \leftrightarrow$ `(·)*ds(k)`, called an *exterior facet integral*,
- $\int_{\Gamma_k}(\cdot)\,\mathrm{d}S \leftrightarrow$ `(·)*dS(k)`, called an *interior facet integral*,

Defining a different quadrature order for each term in a form can be achieved by attaching metadata to measure objects, e.g.,

*UFL code*

```
dx02 = dx(0, { "integration_order": 2 })
dx14 = dx(1, { "integration_order": 4 })
```

```
dx12 = dx(1, { "integration_order": 2 })
L = f*v*dx02 + g*v*dx14 + h*v*dx12
```

Metadata can also be used to override other form compiler specific options separately for each term. For more details on this feature see the manuals of UFL and the form compilers.

## 4.4 Defining expressions

Most of UFL deals with how to declare expressions such as the integrand expressions in Equation 4.13. The most basic expressions are terminal values, which do not depend on other expressions. Other expressions are called operators, which are discussed in sections 4.4.2–4.4.5.

Terminal value types in UFL include form arguments (which is the topic of Section 4.4.1), geometric quantities, and literal constants. Among the literal constants are scalar integer and floating point values, as well as the $d$ by $d$ identity matrix `I = Identity(d)`. To get unit vectors, simply use rows or columns of the identity matrix, e.g., `e0 = I[0,:]`. Similarly, `I[i,j]` represents the Kronecker delta function $\delta_{ij}$ (see Section 4.4.2 for details on index notation). Available geometric values are the spatial coordinates $x \leftrightarrow$ `cell.x` and the facet normal $n \leftrightarrow$ `cell.n`. The geometric dimension is available as `cell.d`.

### 4.4.1 Form arguments

Basis functions and coefficient functions are represented by `Argument` and `Coefficient` respectively. The ordering of the arguments to a form is decided by the order in which the form arguments were declared in the UFL code. Each basis function argument represents any function in the basis of its finite element space

$$\phi^j \in \{\phi_k^j\}, \quad V_h^j = \text{span}\left\{\phi_k^j\right\}. \tag{4.14}$$

with the intention that the form is later evaluated for all $\phi_k$ such as in Equation (4.12). Each coefficient function $w$ represents a discrete function in some finite element space $V_h$; it is usually a sum of basis functions $\phi_k \in V_h$ with coefficients $w_k$

$$w = \sum_{k=1}^{|V_h|} w_k \phi_k. \tag{4.15}$$

The exception is coefficient functions that can only be evaluated point-wise, which are declared with a finite element with family "Quadrature". Basis functions are declared for an arbitrary element as in the following manner:

*UFL code*

```
phi = Argument(element)
v = TestFunction(element)
u = TrialFunction(element)
```

By using `TestFunction` and `TrialFunction` in declarations instead of `Argument` you can ignore their relative ordering. The only time `Argument` is needed is for forms of arity $r > 2$.

Coefficient functions are declared similarly for an arbitrary element, and shorthand notation exists for declaring constant coefficients:

*UFL code*

```
w = Coefficient(element)
```

```
c = Constant(cell)
v = VectorConstant(cell)
M = TensorConstant(cell)
```

If a form argument $u$ in a mixed finite element space $V_h = V_h^0 \times V_h^1$ is desired, but the form is more easily expressed using subfunctions $u_0 \in V_h^0$ and $u_1 \in V_h^1$, you can split the mixed function or basis function into its subfunctions in a generic way using `split`:

*UFL code*

```
V = V0*V1
u = Coefficient(V)
u0, u1 = split(u)
```

The `split` function can handle arbitrary mixed elements. Alternatively, a handy shorthand notation for argument declaration followed by `split` is

*UFL code*

```
v0, v1 = TestFunctions(V)
u0, u1 = TrialFunctions(V)
f0, f1 = Coefficients(V)
```

### 4.4.2 Index notation

UFL allows working with tensor expressions of arbitrary rank, using both tensor algebra and index notation. A basic familiarity with tensor algebra and index notation is assumed. The focus here is on how index notation is expressed in UFL.

Assuming a standard orthonormal Euclidean basis $\langle e_k \rangle_{k=1}^d$ for $\mathbb{R}^d$, a vector can be expressed with its scalar components in this basis. Tensors of rank two can be expressed using their scalar components in a dyadic basis $\{e_i \otimes e_j\}_{i,j=1}^d$. Arbitrary rank tensors can be expressed the same way, as illustrated here.

$$v = \sum_{k=1}^{d} v_k e_k, \tag{4.16}$$

$$A = \sum_{i=1}^{d} \sum_{j=1}^{d} A_{ij} e_i \otimes e_j, \tag{4.17}$$

$$\mathcal{C} = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k} C_{ijk} e_i \otimes e_j \otimes e_k. \tag{4.18}$$

Here, $v$, $A$ and $\mathcal{C}$ are rank 1, 2 and 3 tensors respectively. Indices are called *free* if they have no assigned value, such as $i$ in $v_i$, and *fixed* if they have a fixed value such as 1 in $v_1$. An expression with free indices represents any expression you can get by assigning fixed values to the indices. The expression $A_{ij}$ is scalar valued, and represents any component $(i, j)$ of the tensor $A$ in the Euclidean basis. When working on paper, it is easy to switch between tensor notation ($A$) and index notation ($A_{ij}$) with the knowledge that the tensor and its components are different representations of the same physical quantity. In a programming language, we must express the operations mapping from tensor to scalar components and back explicitly. Mapping from a tensor to its components, for a rank 2 tensor defined as

$$A_{ij} = A : (e_i \otimes e_j) \tag{4.19}$$

is accomplished using indexing with the notation `A[i,j]`. Defining a tensor $A$ from component values $A_{ij}$ is defined as

$$A = A_{ij} e_i \otimes e_j, \tag{4.20}$$

and is accomplished using the function `as_tensor(Aij, (i,j))`. To illustrate, consider the outer product of two vectors $A = u \otimes v = u_i v_j e_i \otimes e_j$, and the corresponding scalar components $A_{ij}$. One way to implement this is

*UFL code*

```
A = outer(u, v)
Aij = A[i, j]
```

Alternatively, the components of $A$ can be expressed directly using index notation, such as $A_{ij} = u_i v_j$. $A_{ij}$ can then be mapped to $A$ in the following manner:

*UFL code*

```
Aij = v[j]*u[i]
A = as_tensor(Aij, (i, j))
```

These two pairs of lines are mathematically equivalent, and the result of either pair is that the variable `A` represents the tensor $A$ and the variable `Aij` represents the tensor $A_{ij}$. Note that free indices have no ordering, so their order of appearance in the expression `v[j]*u[i]` is insignificant. Instead of `as_tensor`, the specialized functions `as_vector` and `as_matrix` can be used. Although a rank two tensor was used for the examples above, the mappings generalize to arbitrary rank tensors.

When indexing expressions, fixed indices can also be used such as in `A[0,1]` which represents a single scalar component. Fixed indices can also be mixed with free indices such as in `A[0,i]`. In addition, slices can be used in place of an index. An example of using slices is `A[0,:]` which is a vector expression that represents row 0 of `A`. To create new indices, you can either make a single one or make several at once:

*UFL code*

```
i = Index()
j, k, l = indices(3)
```

A set of indices `i`, `j`, `k`, `l` and `p`, `q`, `r`, `s` are predefined, and these should suffice for most applications.

If your components are not represented as an expression with free indices, but as separate unrelated scalar expressions, you can build a tensor from them using `as_tensor` and its peers. As an example, lets define a 2D rotation matrix and rotate a vector expression by $\frac{\pi}{2}$:

*UFL code*

```
th = pi/2
A = as_matrix([[ cos(th), -sin(th)],
               [ sin(th),  cos(th)]])
u = A*v
```

When indices are repeated in a term, summation over those indices is implied in accordance with the Einstein convention. In particular, indices can be repeated when indexing a tensor of rank two or higher (`A[i,i]`), when differentiating an expression with a free index (`v[i].dx(i)`), or when multiplying two expressions with shared free indices (`u[i]*v[i]`).

$$A_{ii} \equiv \sum_i A_{ii}, \qquad v_i u_i \equiv \sum_i v_i u_i, \qquad v_{i,i} \equiv \sum_i v_{i,i}. \tag{4.21}$$

An expression `Aij = A[i,j]` is represented internally using the `Indexed` class. `Aij` will reference `A`, keeping the representation of the original tensor expression `A` unchanged. Implicit summation is represented explicitly in the expression tree using the class `IndexSum`. Many algorithms become easier to implement with this explicit representation, since e.g. a `Product` instance can never implicitly represent a sum. More details on representation classes are found in Section 4.6.

### 4.4.3 Algebraic operators and functions

UFL defines a comprehensive set of operators that can be used for composing expressions. The elementary algebraic operators `+`, `-`, `*`, `/` can be used between most UFL expressions with a few limitations. Division requires a scalar expression with no free indices in the denominator. The operands to a sum must have the same shape and set of free indices.

The multiplication operator $*$ is valid between two scalars, a scalar and any tensor, a matrix and a vector, and two matrices. Other products could have been defined, but for clarity we use tensor algebra operators and index notation for those rare cases. A product of two expressions with shared free indices implies summation over those indices, see Section 4.4.2 for more about index notation.

Three often used operators are `dot(a, b)`, `inner(a, b)`, and `outer(a, b)`. The dot product of two tensors of arbitrary rank is the sum over the last index of the first tensor and the first index of the second tensor. Some examples are

$$v \cdot u = v_i u_i, \tag{4.22}$$
$$A \cdot u = A_{ij} u_j e_i, \tag{4.23}$$
$$A \cdot B = A_{ik} B_{kj} e_i e_j, \tag{4.24}$$
$$\mathcal{C} \cdot A = C_{ijk} A_{kl} e_i e_j e_l. \tag{4.25}$$

The inner product is the sum over all indices, for example

$$v : u = v_i u_i, \tag{4.26}$$
$$A : B = A_{ij} B_{ij}, \tag{4.27}$$
$$\mathcal{C} : \mathcal{D} = C_{ijkl} D_{ijkl}. \tag{4.28}$$

Some examples of the outer product are

$$v \otimes u = v_i u_j e_i e_j, \tag{4.29}$$
$$A \otimes u = A_{ij} u_k e_i e_j e_k, \tag{4.30}$$
$$A \otimes B = A_{ij} B_{kl} e_i e_j e_k e_l \tag{4.31}$$

Other common tensor algebra operators are `cross(u,v)`, `transpose(A)` (or `A.T`), `tr(A)`, `det(A)`, `inv(A)`, `cofac(A)`, `dev(A)`, `skew(A)`, and `sym(A)`. Most of these tensor algebra operators expect tensors without free indices. The detailed definitions of these operators are found in the manual.

A set of common elementary functions operating on scalar expressions without free indices are included, in particular `abs(f)`, `pow(f, g)`, `sqrt(f)`, `exp(f)`, `ln(f)`, `cos(f)`, `sin(f)`, `tan(f)`, `acos(f)`, `asin(f)`, `atan(f)`, and `sign(f)`. Any operator taking scalar arguments can be applied element-wise to tensors using e.g. `elem_op(sin, A)`.

### 4.4.4 Differential operators

UFL implements derivatives w.r.t. three different kinds of variables. The most used kind is spatial

derivatives. Expressions can also be differentiated w.r.t. arbitrary user defined variables. And the final kind of derivatives are derivatives of a form or functional w.r.t. the coefficients of a discrete function; that is, a `Coefficient` or `Constant`. Form derivatives are explained in Section 4.5.1.

Note that derivatives are not computed immediately when declared. A discussion of how derivatives are computed is found in Section 4.7.

*Spatial derivatives* Basic spatial derivatives $\frac{\partial f}{\partial x_i}$ can be expressed in two equivalent ways:

*UFL code*

```
df = Dx(f, i)
df = f.dx(i)
```

Here, `df` represents the derivative of `f` in the spatial direction $x_i$. The index `i` can either be an integer, representing differentiation in one fixed spatial direction $x_i$, or an `Index`, representing differentiation in the direction of a free index. The notation `f.dx(i)` is intended to mirror the index notation $f_{,i}$, which is shorthand for $\frac{\partial f}{\partial x_i}$. Repeated indices imply summation, such that the divergence of a vector valued expression `v` can be written $v_{i,i}$, or `v[i].dx(i)`.

Several common compound spatial derivative operators are defined, namely the gradient, divergence, and curl (rot) operators. These operators are named `grad`, `div`, `nabla_grad`, `nabla_div`, `curl` and `rot` (rot is a synonym for curl). Be aware that there are two common ways to define the gradient and divergence, and UFL supports both.

Let $s$ be a scalar expression, $v$ be a vector expression, and $M$ be a tensor expression of rank $r$. In UFL, the operator `grad` is then defined explicitly as

$$(\text{grad}(s))_i = s_{,i}, \tag{4.32}$$
$$(\text{grad}(v))_{ij} = v_{i,j}, \tag{4.33}$$
$$(\text{grad}(M))_{i_1 \ldots i_r k} = M_{i_1 \ldots i_r, k}, \tag{4.34}$$

and the operator `div` is correspondingly defined as

$$\text{div}(v) = v_{i,i}, \tag{4.35}$$
$$(\text{div}(M))_{i_1 \ldots i_{r-1}} = M_{i_1 \ldots i_r, i_r}. \tag{4.36}$$

In contrast, the `nabla_*` operators are defined in terms of the $\nabla$ operator

$$\nabla \equiv e_k \frac{\partial}{\partial x_k}. \tag{4.37}$$

The operator `nabla_grad` is the outer product of $\nabla$ with its operand:

$$(\nabla s)_i = s_{,i}, \tag{4.38}$$
$$(\nabla v)_{ij} = v_{j,i}, \tag{4.39}$$
$$(\nabla M)_{k,i_1 \ldots i_r} = M_{i_1 \ldots i_r, k}. \tag{4.40}$$

Similarly, the operator `nabla_div` is the dot product of $\nabla$ with its operand:

$$\nabla \cdot v = v_{i,i}, \tag{4.41}$$
$$(\nabla \cdot M)_{i_2 \ldots i_r} = M_{i_1 \ldots i_r, i_1}. \tag{4.42}$$

Thinking in terms of value shape, the `grad` operator appends an axis to the end of the shape of its operand, while the `nabla_grad` operator prepends an axis. For gradients of scalars, the result is the same. Correspondingly, the `div` operator sums over the last index of its operand, while the `nabla_div` operator sums over the first index of its operand. For the divergence of vectors, the result is the same.

For the operators `curl` and `rot` there is no difference between the two traditions. For 3D vector expressions, the curl can be defined in terms of the nabla operator and the cross product:

$$\text{curl}(v) \equiv \nabla \times v = e_0(v_{2,1} - v_{1,2}) - e_1(v_{2,0} - v_{0,2}) + e_2(v_{1,0} - v_{0,1}) \tag{4.43}$$

For 2D vector and scalar expressions the definitions are:

$$\text{curl}(v) \equiv v_{1,0} - v_{0,1}, \tag{4.44}$$
$$\text{curl}(f) \equiv f_{,1}e_0 - f_{,0}e_1. \tag{4.45}$$

*User defined variables* The second kind of differentiation variables are user-defined variables, which can represent arbitrary expressions. Automating derivatives w.r.t. arbitrary quantities is useful for several tasks, from differentiation of material laws to computing sensitivities. An arbitrary expression $g$ can be assigned to a variable $v$. An expression $f$ defined as a function of $v$ can be differentiated $f$ w.r.t. $v$:

$$v = g, \tag{4.46}$$
$$f = f(v), \tag{4.47}$$
$$h(v) = \frac{\partial f(v)}{\partial v}. \tag{4.48}$$

Setting $g = sin(x_0)$ and $f = e^{v^2}$, gives $h = 2ve^{v^2} = 2\sin(x_0)e^{\sin^2(x_0)}$, which can be implemented as follows:

*UFL code*

```
g = sin(cell.x[0])
v = variable(g)
f = exp(v**2)
h = diff(f, v)
```

Try running this code in a Python session and print the expressions. The result is

*Python code*

```
>> print v
var0(sin((x)[0]))
>> print h
d/d[var0(sin((x)[0]))] (exp((var0(sin((x)[0]))) ** 2))
```

Note that the variable has a label "`var0`", and that `h` still represents the abstract derivative. Section 4.7 explains how derivatives are computed.

### 4.4.5 Other operators

A few operators are provided for the implementation of discontinuous Galerkin methods. The basic concept is restricting an expression to the positive or negative side of an interior facet, which is expressed simply as `v("+")` or `v("-")` respectively. On top of this, the operators `avg` and `jump` are

implemented, defined as

$$\text{avg}(v) = \frac{1}{2}(v^+ + v^-), \tag{4.49}$$

$$\text{jump}(v) = v^+ - v^-. \tag{4.50}$$

These operators can only be used when integrating over the interior facets (`*dS`).

The only control flow construct included in UFL is conditional expressions. A conditional expression takes on one of two values depending on the result of a boolean logic expression. The syntax for this is

*UFL code*

```
f = conditional(condition, true_value, false_value)
```

which is interpreted as

$$f = \begin{cases} t, & \text{if condition is true,} \\ f, & \text{otherwise.} \end{cases} \tag{4.51}$$

The condition can be one of

- `lt(a, b)` $\leftrightarrow (a < b)$
- `le(a, b)` $\leftrightarrow (a \leq b)$
- `eq(a, b)` $\leftrightarrow (a = b)$
- `And(P, Q)` $\leftrightarrow (P \wedge Q)$
- `Not(P)` $\leftrightarrow (\neg P)$
- `gt(a, b)` $\leftrightarrow (a > b)$
- `ge(a, b)` $\leftrightarrow (a \geq b)$
- `ne(a, b)` $\leftrightarrow (a \neq b)$
- `Or(P, Q)` $\leftrightarrow (P \vee Q)$

## 4.5 *Form operators*

Once you have defined some forms, there are several ways to compute related forms from them. While operators in the previous section are used to define expressions, the operators discussed in this section are applied to forms, producing new forms. Form operators can both make form definitions more compact and reduce the chances of bugs since changes in the original form will propagate to forms computed from it automatically. These form operators can be combined arbitrarily; given a semi-linear form only a few lines are needed to compute the action of the adjoint of the Jacobi. Since these computations are done prior to processing by the form compilers, there is no overhead at run-time.

### 4.5.1 *Differentiating forms*

The form operator `derivative` declares the derivative of a form w.r.t. coefficients of a discrete function (`Coefficient`). This functionality can be used for example to linearize your nonlinear residual equation (linear form) automatically for use with the Newton-Raphson method. It can also be applied multiple times, which is useful to derive a linear system from a convex functional, in order to find the function that minimizes the functional. For non-trivial equations such expressions can be tedious to calculate by hand. Other areas in which this feature can be useful include optimal control and inverse methods, as well as sensitivity analysis.

In its simplest form, the declaration of the derivative of a form `L` w.r.t. the coefficients of a function `w` reads

*UFL code*

```
a = derivative(L, w, u)
```

The form `a` depends on an additional basis function argument `u`, which must be in the same finite element space as the function `w`. If the last argument is omitted, a new basis function argument is created.

Let us step through an example of how to apply `derivative` twice to a functional to derive a linear system. In the following, $V_h$ is a finite element space with some basis , $w$ is a function in $V_h$, and $f = f(w)$ is a functional we want to minimize. Derived from $f(w)$ is a linear form $F(w;v)$, and a bilinear form $J(w;u,v)$.

$$V_h = \text{span}\left\{\phi_k\right\}, \tag{4.52}$$

$$w(x) = \sum_{k=1}^{|V_h|} w_k \phi_k(x), \tag{4.53}$$

$$f : V_h \to \mathbb{R}, \tag{4.54}$$

$$F(w;\phi_i) = \frac{\partial f(w)}{\partial w_i}, \quad i = 1, \ldots, |V_h|, \tag{4.55}$$

$$J(w;\phi_j,\phi) = \frac{\partial F(w;\phi)}{\partial w_j}, \quad j = 1, \ldots, |V_h|, \quad \phi \in V_h. \tag{4.56}$$

For a concrete functional $f(w) = \int_\Omega \frac{1}{2} w^2 \, \mathrm{d}x$, we can implement this as

*UFL code*

```
v = TestFunction(element)
u = TrialFunction(element)
w = Coefficient(element)
f = 0.5*w**2*dx
F = derivative(f, w, v)
J = derivative(F, w, u)
```

This code declares two forms `F` and `J`. The linear form `F` represents the standard load vector `w*v*dx` and the bilinear form `J` represents the mass matrix `u*v*dx`.

Derivatives can also be defined w.r.t. coefficients of a function in a mixed finite element space. Consider the Harmonic map equations derived from the functional

$$f(x,\lambda) = \int_\Omega \text{grad}\, x : \text{grad}\, x + \lambda x \cdot x \, \mathrm{d}x, \tag{4.57}$$

where $x$ is a function in a vector finite element space $V_h^d$ and $\lambda$ is a function in a scalar finite element space $V_h$. The linear and bilinear forms derived from the functional in Equation 4.57 have basis function arguments in the mixed space $V_h^d \times V_h$. The implementation of these forms with automatic linearization reads

*UFL code*

```
Vx = VectorElement("Lagrange", triangle, 1)
Vy = FiniteElement("Lagrange", triangle, 1)
u = Coefficient(Vx*Vy)
x, y = split(u)
```

```
f = inner(grad(x), grad(x))*dx + y*dot(x,x)*dx
F = derivative(f, u)
J = derivative(F, u)
```

Note that the functional is expressed in terms of the subfunctions `x` and `y`, while the argument to `derivative` must be the single mixed function `u`. In this example the basis function arguments to `derivative` are omitted and thus provided automatically in the right function spaces.

Note that in computing derivatives of forms, we have assumed that

$$\frac{\partial}{\partial w_k}\int_\Omega I\,\mathrm{d}x = \int_\Omega \frac{\partial}{\partial w_k} I\,\mathrm{d}x, \tag{4.58}$$

or in particular that the domain $\Omega$ is independent of $w$. Also, any coefficients other than $w$ are assumed independent of $w$. Furthermore, note that there is no restriction on the choice of element in this framework, in particular arbitrary mixed elements are supported.

### 4.5.2 Adjoint

Another form operator is the adjoint $a^*$ of a bilinear form $a$, defined as $a^*(v,u) = a(u,v)$, which is equivalent to taking the transpose of the assembled sparse matrix. In UFL this is implemented simply by swapping the order of the test and trial functions, and can be written using the `adjoint` form operator. (Note that this is not the most generic definition of the adjoint of an operator). An example of its use on an anisotropic diffusion term looks like

*UFL code*

```
V = VectorElement("Lagrange", cell, 1)
T = TensorElement("Lagrange", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)
M = Coefficient(T)
a = M[i,j]*u[k].dx(j)*v[k].dx(i)*dx
astar = adjoint(a)
```

which corresponds to (with $u \in U$ and $v \in V$)

$$a(M;u,v) = \int_\Omega M_{ij}u_{k,j}v_{k,i}\,\mathrm{d}x, \tag{4.59}$$

$$a^*(M;v,u) = \int_\Omega M_{ij}u_{k,j}v_{k,i}\,\mathrm{d}x = a(M;u,v). \tag{4.60}$$

This automatic transformation is particularly useful if we need the adjoint of nonsymmetric bilinear forms computed using `derivative`, since the explicit expressions for $a$ are not at hand. Several of the form operators below are most useful when used in conjunction with `derivative`.

### 4.5.3 Replacing functions

Evaluating a form with new definitions of form arguments can be done by replacing terminal objects with other values. Lets say you have defined a form `L` that depends on some functions `f` and `g`. You can then specialize the form by replacing these functions with other functions or fixed values, such as

$$L(f,g;v) = \int_\Omega (f^2/(2g))v\,\mathrm{d}x, \tag{4.61}$$

$$L_2(f,g;v) = L(g,3;v) = \int_\Omega (g^2/6)v\,\mathrm{d}x. \tag{4.62}$$

This feature is implemented with `replace`, as illustrated in this case:

*UFL code*

```
V = FiniteElement("Lagrange", cell, 1)
v = TestFunction(V)
f = Coefficient(V)
g = Coefficient(V)
L = f**2 / (2*g)*v*dx
L2 = replace(L, { f: g, g: 3})
L3 = g**2 / 6*v*dx
```

Here `L2` and `L3` represent exactly the same form. Since they depend only on `g`, the code generated for these forms can be more efficient.

### 4.5.4 *Action*

In some applications the matrix is not needed explicitly, only the action of the matrix on a vector. Assembling the resulting vector directly can be much more efficient than assembling the sparse matrix and then performing the matrix-vector multiplication. Assume `a` is a bilinear form and `w` is a `Coefficient` defined on the same finite element as the trial function in `a`. Let $A$ denote the sparse matrix that can be assembled from `a`. Then you can assemble the action of $A$ on a vector directly by defining a linear form $L$ representing the action of a bilinear form $a$ on a function $w$. The notation for this is simply `L = action(a, w)`, or even shorter `L = a*w`.

### 4.5.5 *Splitting a system*

If you prefer to write your PDEs with all terms on one side such as

$$a(u, v) - L(v) = 0, \tag{4.63}$$

you can declare forms with both linear and bilinear terms and split the equations into $a$ and $L$ afterwards. A simple example is

*UFL code*

```
V = FiniteElement("Lagrange", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)
pde = u*v*dx - f*v*dx
a, L = system(pde)
```

Here `system` is used to split the PDE into its bilinear and linear parts. Alternatively, `lhs` and `rhs` can be used to obtain the two parts separately. Make note of the resulting sign of the linear part, which corresponds to moving $L$ to the right-hand side in Equation (4.63).

### 4.5.6 *Computing the sensitivity of a function*

If you have found the solution $u$ to Equation (4.63), and $u$ depends on some constant scalar value $c$, you can compute the sensitivity of $u$ w.r.t. changes in $c$. If $u$ is represented by a coefficient vector $x$ that is the solution to the algebraic linear system $Ax = b$, the coefficients of $\frac{\partial u}{\partial c}$ are $\frac{\partial x}{\partial c}$. Applying $\frac{\partial}{\partial c}$ to $Ax = b$ and using the chain rule, we can write

$$A\frac{\partial x}{\partial c} = \frac{\partial b}{\partial c} - \frac{\partial A}{\partial c}x, \tag{4.64}$$

and thus $\frac{\partial x}{\partial c}$ can be found by solving the same algebraic linear system used to compute $x$, only with a different right-hand side. The linear form corresponding to the right-hand side of Equation (4.64) can be written

*UFL code*

```
u = Coefficient(element)
sL = diff(L, c) - action(diff(a, c), u)
```

or you can use the equivalent form transformation

*UFL code*

```
sL = sensitivity_rhs(a, u, L, c)
```

Note that the solution `u` must be represented by a `Coefficient`, while $u$ in $a(u, v)$ is represented by a `Argument`.

## 4.6 *Expression representation*

From a high level view, UFL is all about defining forms. Each form contains one or more scalar integrand expressions, but the form representation is largely disconnected from the representation of the integrand expressions. Indeed, most of the complexity of the UFL implementation is related to expressing, representing, and manipulating expressions. The rest of this chapter will focus on expression representations and algorithms operating on them. These topics will be of little interest to the average user of UFL, and more directed towards developers and curious technically oriented users.

To reason about expression algorithms without the burden of implementation details, we need an abstract notation for the structure of an expression. UFL expressions are representations of programs, and the notation should allow us to see this connection. Below we will discuss the properties of expressions both in terms of this abstract notation, and related to specific implementation details.

### 4.6.1 *The structure of an expression*

The most basic expressions, which have no dependencies on other expressions, are called *terminal expressions*. Other expressions result from applying some operator to one or more existing expressions. Consider an arbitrary (non-terminal) expression $z$. This expression depends on a set of terminal expressions $\{t_i\}$, and is computed using a set of operators $\{f_i\}$. If each subexpression of $z$ is labeled with an integer, an abstract program can be written to compute $z$ by computing a sequence of subexpressions $\langle y_i \rangle_{i=1}^{n}$ and setting $z = y_n$. Algorithm 1 shows such a program.

**Algorithm 1** Program to compute an expression $z$.

1: **for** $i \leftarrow 1, \ldots, m$ **do**
2: $\quad y_i := t_i =$ terminal expression
3: **end for**
4: **for** $i \leftarrow m+1, \ldots, n$ **do**
5: $\quad y_i := f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
6: **end for**
7: $z := y_n$

Each terminal expression $t_i$ is a literal constant or input argument to the program. This includes coefficients, basis functions, and geometric quantities. A non-terminal subexpression $y_i$ is the result of

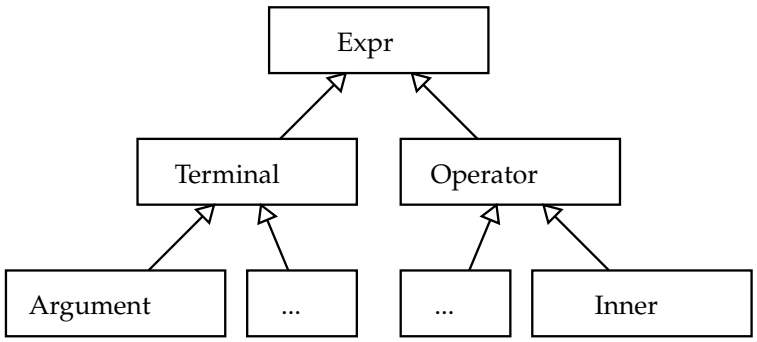



Figure 4.2: Expression class hierarchy.

applying an operator $f_i$ to a sequence of previously computed expressions $\langle y_j \rangle_{j \in \mathfrak{I}_i}$, where $\mathfrak{I}_i$ is an ordered sequence of expression labels. Note that the order in which subexpressions must be computed to produce the same value of $z$ is not unique. For correctness we only require $j < i \; \forall \, j \in \mathfrak{I}_i$, such that all dependencies of a subexpression $y_i$ has been computed before $y_i$. In particular, all terminals are numbered first in this abstract algorithm for notational convenience only.

The program to compute $z$ can be represented as a graph, where each expression $y_i$ corresponds to a graph vertex. There is a directed graph edge $e = (i, j)$ from $y_i$ to $y_j$ if $j \in \mathfrak{I}_i$, that is if $y_i$ depends on the value of $y_j$. More formally, the graph $G$ representing the computation of $z$ consists of a set of vertices $V$ and a set of edges $E$ defined by:

$$G = (V, E), \tag{4.65}$$

$$V = \langle v_i \rangle_{i=1}^{n} = \langle y_i \rangle_{i=1}^{n}, \tag{4.66}$$

$$E = \{e_k\} = \bigcup_{i=1}^{n} \{(i, j) \; \forall \, j \in \mathfrak{I}_i\}. \tag{4.67}$$

This graph is clearly directed, since dependencies have a direction. It is acyclic, since an expression can only be constructed from existing expressions. Thus a UFL expression can be represented by a directed acyclic graph (DAG). There are two ways this DAG can be represented in UFL. While defining expressions, a linked representation called the expression tree is built. Technically this is still a DAG since vertices can be reused in multiple subexpressions, but the representation emphasizes the tree like structure of the DAG. The other representation is called the computational graph, which closely mirrors the definition of $G$ above. This representation is mostly useful for form compilers. The details of these two DAG representations will be explained below. They both share the representation of a vertex in the graph as an expression object, which will be explained next.

### 4.6.2 Expression objects

Recall from Algorithm 1 that non-terminals are expressions $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$. The operator $f_i$ is represented by the class of the expression object, while the expression $y_i$ is represented by the instance of this class. In the UFL implementation, each expression object is an instance of some subclass of `Expr`. The class `Expr` is the superclass of a hierarchy containing all terminal expression types and operator types supported by UFL. `Expr` has two direct subclasses, `Terminal` and `Operator`, which divides the expression type hierarchy in two, as illustrated in Figure 4.2.

All expression objects are considered immutable; once constructed an expression object will never be modified. Manipulating an expression should always result in a new object being created. The immutable property ensures that expression objects can be reused and shared between expressions without side effects in other parts of a program. This both reduces memory usage, avoids needless

copying of objects, and simplifies recognition of common subexpressions.

Calling `e.operands()` on an `Expr` object `e` representing $y_i$ returns a tuple with expression objects representing $\langle y_j \rangle_{j \in \mathfrak{I}_i}$. Note that this also applies to terminals where there are no outgoing edges and `t.operands()` returns an empty tuple. Instead of modifying the operands of an expression object, a new expression object of the same type can be constructed with modified operands using `e.reconstruct(operands)`, where `operands` is a tuple of expression objects. If the operands are the same this function returns the original object, allowing many algorithms to save memory without additional complications. The invariant `e.reconstruct(e.operands()) == e` should always hold.

### *4.6.3 Expression properties*

In Section 4.4.2 the tensor algebra and index notation capabilities of UFL was discussed. Expressions can be scalar or tensor-valued, with arbitrary rank and shape. Therefore, each expression object `e` has a value shape `e.shape()`, which is a tuple of integers with the dimensions in each tensor axis. Scalar expressions have shape `()`. Another important property is the set of free indices in an expression, obtained as a tuple using `e.free_indices()`. Although the free indices have no ordering, they are represented with a tuple of `Index` instances for simplicity. Thus the ordering within the tuple carries no meaning.

UFL expressions are referentially transparent with some exceptions. Referential transparency means that a subexpression can be replaced by another representation of its value without changing the meaning of the expression. A key point here is that the value of an expression in this context includes the tensor shape and set of free indices. Another important point is that the derivative of a function $f(v)$ in a point, $f'(v)|_{v=g}$, depends on function values in the vicinity of $v = g$. The effect of this dependency is that operator types matter when differentiating, not only the current value of the differentiation variable. In particular, a `Variable` cannot be replaced by the expression it represents, because `diff` depends on the `Variable` instance and not the expression it has the value of. Similarly, replacing a `Coefficient` with some value will change the meaning of an expression that contains derivatives w.r.t. function coefficients.

The following example illustrate the issue with `Variable` and `diff`.

*UFL code*

```
e = 0
v = variable(e)
f = sin(v)
g = diff(f, v)
```

Here `v` is a variable that takes on the value 0, but `sin(v)` cannot be simplified to 0 since the derivative of `f` then would be 0. The correct result here is `g = cos(v)`. Printing `f` and `g` gives the strings `sin(var1(0))` and `d/d[var1(0)] (sin(var1(0)))`. Try just setting `v = e` and see how `f` and `g` becomes zero.

### *4.6.4 Tree representation*

The expression tree does not have a separate data structure. It is merely a way of viewing the structure of an expression. Any expression object `e` can be seen as the root of a tree, where `e.operands()` returns its children. If some of the children are equal, they will appear as many times as they appear in the expression. Thus it is easy to traverse the tree nodes; that is, $v_i$ in the DAG, but eventual reuse of subexpressions is not directly visible. Edges in the DAG does not appear explicitly, and the list of vertices can only be obtained by traversing the tree recursively and selecting unique objects.

An expression tree for the stiffness term $\operatorname{grad} u : \operatorname{grad} v$ is illustrated in Figure 4.3. The terminals $u$ and $v$ have no children, and the term $\operatorname{grad} u$ is itself represented by a tree with two nodes. Each time

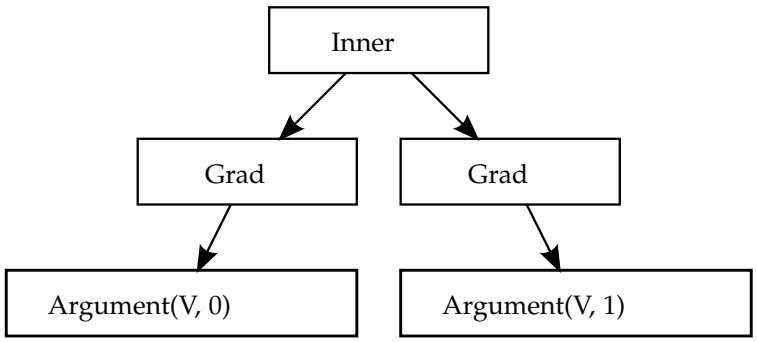



Figure 4.3: Expression tree for grad $u$ : grad $v$.

an operator is applied to some expressions, it will return a new tree root that references its operands. Note that the user will apply the functions `grad` and `inner` in her use of the language, while the names `Grad`, `Inner` and `Argument` in this figure are the names of the `Expr` subclasses used in UFL to represent the expression objects. In other words, taking the gradient of an expression with `grad(u)` gives an expression representation `Grad(u)`, and `inner(a, b)` gives an expression representation `Inner(a, b)`. This separation of language and representation is merely a design choice in the implementation of UFL.

### 4.6.5 Graph representation

When viewing an expression as a tree, the lists of all unique vertices and edges are not directly available. Representing the DAG more directly allows many algorithms to be simplified or optimized. UFL includes tools to build an array based representation of the DAG, the *computational graph*, from any expression. The computational graph $G = V, E$ is a data structure based on flat arrays, directly mirroring the definition of the graph in equations (4.65)–(4.67). This representation gives direct access to dependencies between subexpressions, and allows easy iteration over unique vertices. The graph is constructed easily with the lines:

*Python code*

```
from ufl.algorithms import Graph
G = Graph(expression)
V, E = G
```

One array (Python list) `V` is used to store the unique vertices $\langle v_i \rangle_{i=1}^{n}$ of the DAG. For each vertex $v_i$ an expression node $y_i$ is stored to represent it. Thus the expression tree for each vertex is also directly available, since each expression node is the root of its own expression tree. The edges are stored in an array `E` with integer tuples `(i,j)` representing an edge from $v_i$ to $v_j$; that is, $v_j$ is an operand of $v_i$. The vertex list in the graph is built using a postordering from a depth first traversal, which guarantees that the vertices are topologically sorted such that $j < i \; \forall \, j \in \mathfrak{I}_i$.

Let us look at an example of a computational graph. The following code defines a simple expression and then prints the vertices and edges of its graph.

*Python code*

```
from ufl import *
cell = triangle
V = FiniteElement("Lagrange", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)
c = Constant(cell)
f = Coefficient(V)
```

```
e = c*f**2*u*v

from ufl.algorithms import Graph, partition
G = Graph(e)
V, E, = G

print "str(e) = %s\n" % str(e)
print "\n".join("V[%d] = %s" % (i, v) for (i, v) in enumerate(V)), "\n"
print "\n".join("E[%d] = %s" % (i, e) for (i, e) in enumerate(E)), "\n"
```

An excerpt of the program output is shown here:

*Generated code*

```
V[0] = v_{-2}
...
V[7] = v_{-1} * c_0 * w_1 ** 2
V[8] = v_{-2} * v_{-1} * c_0 * w_1 ** 2
...
E[6] = (8, 0)
E[7] = (8, 7)
```

The two last edges shown here represent the dependencies of vertex 8 on vertex 7 and 0, since $v_8 = v_0 v_7$. Run the code to see the full output of this code. Try changing the expression and see what the graph looks like.

From the edges $E$, related arrays can be computed efficiently; in particular the vertex indices of dependencies of a vertex $v_i$ in both directions are useful:

$$\begin{aligned} V_{out} &= \langle \mathfrak{I}_i \rangle_{i=1}^{n}, \\ V_{in} &= \left\langle \{j | i \in \mathfrak{I}_j\} \right\rangle_{i=1}^{n} \end{aligned} \tag{4.68}$$

These arrays can be easily constructed for any expression:

*Python code*

```
Vin = G.Vin()
Vout = G.Vout()
```

Similar functions exist for obtaining indices into $E$ for all incoming and outgoing edges. A nice property of the computational graph built by UFL is that no two vertices will represent the same identical expression. During graph building, subexpressions are inserted in a hash map (Python dictionary) to achieve this. Some expression classes sort their arguments uniquely such that e.g. `a*b` and `b*a` will become the same vertex in the graph.

Free indices in expression nodes can complicate the interpretation of the linearized graph when implementing some algorithms, because an expression object with free indices represents not one value but a set of values, one for each permutation of the values its free indices can have. One solution to this can be to apply `expand_indices` before constructing the graph, which will replace all expressions with free indices with equivalent expressions with explicit fixed indices. Note however that free indices cannot be regained after expansion. See Section 4.8.3 for more about this transformation.

### *4.6.6 Partitioning*

UFL is intended as a front-end for form compilers. Since the end goal is generation of code from expressions, some utilities are provided for the code generation process. In principle, correct code can be generated for an expression from its computational graph simply by iterating over the vertices and generating code for each operation separately, basically mirroring Algorithm 1. However, a good

form compiler should be able to produce better code. UFL provides utilities for partitioning the computational graph into subgraphs (partitions) based on dependencies of subexpressions, which enables quadrature based form compilers to easily place subexpressions inside the right sets of loops. The function `partition` implements this feature. Each partition is represented by a simple array of vertex indices, and each partition is labeled with a set of dependencies. By default, this set of dependencies use the strings `x`, `c`, and `v%d` to denote dependencies on spatial coordinates, cell specific quantities, and form arguments (not coefficients) respectively.

The following example code partitions the graph built above, and prints vertices in groups based on their dependencies.

*Python code*

```
partitions, keys = partition(G)
for deps in sorted(partitions.keys()):
    P = partitions[deps]
    print "The following depends on", tuple(deps)
    for i in sorted(P):
        print "V[%d] = %s" % (i, V[i])
```

The output text from the program is included below. Notice that the literal constant 2 has no dependencies. Expressions in this partition can always be precomputed at compile-time. The `Constant c_0` depends on data which varies for each cell, represented by `c` in the dependency set, but not on spatial coordinates, so it can be placed outside the quadrature loop. The `Function w_1` and expressions depending on it depends in addition on the spatial coordinates, represented by `x`, and therefore needs to be computed for each quadrature point. Expressions depending on only the test or trial function are marked with `v%d` where the number is the internal counter used by UFL to distinguish between arguments. Note that test and trial functions are here marked as depending on the spatial coordinates, but not on cell dependent quantities. This is only true for finite elements defined on a local reference element, in which case the basis functions can be precomputed in each quadrature point. The actual run-time dependencies of a basis function in a finite element space is unknown to UFL, which is why the partition function takes an optional multifunction argument such that the form compiler writer can provide more accurate dependencies. We refer to the implementation of `partition` for such implementation details.

*Generated code*

```
The following depends on ()
V[4] = 2
The following depends on ("c",)
V[2] = c_0
The following depends on ("x", "c")
V[3] = w_1
V[5] = w_1 ** 2
V[6] = c_0 * w_1 ** 2
The following depends on ("x", "v-1")
V[1] = v_{-1}
The following depends on ("x", "c", "v-1")
V[7] = v_{-1} * c_0 * w_1 ** 2
The following depends on ("x", "v-2")
V[0] = v_{-2}
The following depends on ("x", "c", "v-2", "v-1")
V[8] = v_{-2} * v_{-1} * c_0 * w_1 ** 2
```

## *4.7 Computing derivatives*

When any kind of derivative expression is declared by the end-user of the form language, an expression object is constructed to represent it, but nothing is computed. The type of this expression object is a subclass of `Derivative`. Before low level code can be generated from the derivative expression, some kind of algorithm to evaluate derivatives must be applied, since differential operators are not available natively in low level languages such as C++. Computing exact derivatives is important, which rules out approximations by divided differences. Several alternative algorithms exist for computing exact derivatives. All relevant algorithms are based on the chain rule combined with differentiation rules for each expression object type. The main differences between the algorithms are in the extent of which subexpressions are reused, and in the way subexpressions are accumulated.

Mixing derivative computation into the code generation strategy of each form compiler would lead to a significant duplication of implementation effort. To separate concerns and keep the code manageable, differentiation is implemented as part of UFL in such a way that the form compilers are independent of the differentiation strategy chosen in UFL. Therefore, it is advantageous to use the same representation for the evaluated derivative expressions as for any other expression. Before expressions are interpreted by a form compiler, differential operators should be evaluated such that the only operators left are non-differential operators. An exception is made for spatial derivatives of terminals which are unknown to UFL because they are provided by the form compilers.

Below, the differences and similarities between some of the simplest algorithms are discussed. After the algorithm currently implemented in UFL has been explained, extensions to tensor and index notation and higher order derivatives are discussed. Finally, the section is closed with some remarks about the differentiation rules for terminal expressions.

### *4.7.1 Approaches to computing derivatives*

Algorithms for computing derivatives are designed with different end goals in mind. Symbolic Differentiation (SD) takes as input a single symbolic expression and produces a new symbolic expression for its derivative. Automatic Differentiation (AD) takes as input a program to compute a function and produces a new program to compute the derivative of the function. Several variants of AD algorithms exist, the two most common being Forward Mode AD and Reverse Mode AD [Griewank, 1989]. More advanced algorithms exist, and is an active research topic. A UFL expression is a symbolic expression, represented by an expression tree. But the expression tree is a directed acyclic graph that represents a program to evaluate said expression. Thus it seems the line between SD and AD becomes less distinct in this context.

Naively applied, SD can result in huge expressions, which can both require a lot of memory during the computation and be highly inefficient if written to code directly. However, some illustrations of the inefficiency of symbolic differentiation, such as in Griewank [1989], are based on computing closed form expressions of derivatives in some stand-alone computer algebra system (CAS). Copying the resulting large expressions directly into a computer code can lead to very inefficient code. The compiler may not be able to detect common subexpressions, in particular if simplification and rewriting rules in the CAS has changed the structure of subexpressions with a potential for reuse.

In general, AD is capable of handling algorithms that SD can not. A tool for applying AD to a generic source code must handle many complications such as subroutines, global variables, arbitrary loops and branches [Bischof et al., 1992, 2002, Giering and Kaminski, 1998]. Since the support for program flow constructs in UFL is very limited, the AD implementation in UFL will not run into such complications. In Section 4.7.2 the similarity between SD and forward mode AD in the context of UFL is explained in more detail.

### 4.7.2 Forward mode automatic differentiation

Recall Algorithm 1, which represents a program for computing an expression $z$ from a set of terminal values $\{t_i\}$ and a set of elementary operations $\{f_i\}$. Assume for a moment that there are no differential operators among $\{f_i\}$. The algorithm can then be extended to compute the derivative $\frac{dz}{dv}$, where $v$ represents a differentiation variable of any kind. This extension gives Algorithm 2.

**Algorithm 2** Forward mode AD on Algorithm 1.

1: **for** $i \leftarrow 1, \ldots, m$ **do**
2: $\quad y_i := t_i$
3: $\quad \frac{dy_i}{dv} := \frac{dt_i}{dv}$
4: **end for**
5: **for** $i \leftarrow m+1, \ldots, n$ **do**
6: $\quad y_i := f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
7: $\quad \frac{dy_i}{dv} := \sum_{k \in \mathfrak{I}_i} \frac{\partial f_i}{\partial y_k} \frac{dy_k}{dv}$
8: **end for**
9: $z := y_n$
10: $\frac{dz}{dv} := \frac{dy_n}{dv}$

This way of extending a program to simultaneously compute the expression $z$ and its derivative $\frac{dz}{dv}$ is called forward mode automatic differentiation (AD). By renaming $y_i$ and $\frac{dy_i}{dv}$ to a new sequence of values $\langle \hat{y}_j \rangle_{j=1}^{\hat{n}}$, Algorithm 2 can be rewritten as shown in Algorithm 3, which is isomorphic to Algorithm 1 (they have exactly the same structure).

**Algorithm 3** Program to compute $\frac{dz}{dv}$ produced by forward mode AD

1: **for** $i \leftarrow 1, \ldots, \hat{m}$ **do**
2: $\quad \hat{y}_i := \hat{t}_i$
3: **end for**
4: **for** $i \leftarrow \hat{m}+1, \ldots, \hat{n}$ **do**
5: $\quad \hat{y}_i := \hat{f}_i(\langle \hat{y}_j \rangle_{j \in \hat{\mathfrak{I}}_i})$
6: **end for**
7: $\frac{dz}{dv} := \hat{y}_{\hat{n}}$

Since the program in Algorithm 1 can be represented as a DAG, and Algorithm 3 is isomorphic to Algorithm 1, the program in Algorithm 3 can also be represented as a DAG. Thus a program to compute $\frac{dz}{dv}$ can be represented by an expression tree built from terminal values and non-differential operators.

The currently implemented algorithm for computing derivatives in UFL follows forward mode AD closely. Since the result is a new expression tree, the algorithm can also be called symbolic differentiation. In this context, the differences between the two are implementation details. To ensure that we can reuse expressions properly, simplification rules in UFL avoids modifying the operands of an operator. Naturally repeated patterns in the expression can therefore be detected easily by the form compilers. Efficient common subexpression elimination can then be implemented by placing subexpressions in a hash map. However, there are simplifications such as $0 * f \to 0$ and $1 * f \to f$, called constant folding, which simplify the result of the differentiation algorithm automatically as it is being constructed. These simplifications are crucial for the memory use during derivative computations, and the performance of the resulting program.

### *4.7.3 Extensions to tensors and indexed expressions*

So far we have not considered derivatives of non-scalar expression and expressions with free indices. This issue does not affect the overall algorithms, but it does affect the local derivative rules for each expression type.

Consider the expression `diff(A, B)` with `A` and `B` matrix expressions. The meaning of derivatives of tensors w.r.t. to tensors is easily defined via index notation, which is heavily used within the differentiation rules:

$$\frac{dA}{dB} = \frac{dA_{ij}}{dB_{kl}} e_i \otimes e_j \otimes e_k \otimes e_l \tag{4.69}$$

Derivatives of subexpressions are frequently evaluated to literal constants. For indexed expressions, it is important that free indices are propagated correctly with the derivatives. Therefore, differentiated expressions will some times include literal constants annotated with free indices.

There is one rare and tricky corner case when an index sum binds an index $i$ such as in $(v_i v_i)$ and the derivative w.r.t. $x_i$ is attempted. The simplest example of this is the expression $(v_i v_i)_{,j}$, which has one free index $j$. If $j$ is replaced by $i$, the expression can still be well defined, but you would never write $(v_i v_i)_{,i}$ manually. If the expression in the parenthesis is defined in a variable `e = v[i]*v[i]`, the expression `e.dx(i)` looks innocent. However, this will cause problems as derivatives (including the index $i$) are propagated up to terminals. If this case is encountered in the current implementation of UFL, it will be detected and an error message will be triggered. To work around the problem, simply use different index instances. In a future version of UFL, this case may be handled by relabeling indices to change any expression $(\sum_i e_i)_{,i}$ into $(\sum_j e_j)_{,i}$.

### *4.7.4 Higher order derivatives*

A simple forward mode AD implementation such as Algorithm 2 only considers one differentiation variable. Higher order or nested differential operators must also be supported, with any combination of differentiation variables. A simple example illustrating such an expression can be

$$a = \frac{d}{dx}\left(\frac{d}{dx} f(x) + 2\frac{d}{dy} g(x,y)\right). \tag{4.70}$$

Considerations for implementations of nested derivatives in a functional[2] framework have been explored in several papers [Karczmarczuk, 2001, Pearlmutter and Siskind, 2007, Siskind and Pearlmutter, 2008].

In the current UFL implementation this is solved in a different fashion. Considering Equation (4.70), the approach is simply to compute the innermost derivatives $\frac{d}{dx} f(x)$ and $\frac{d}{dy} g(x,y)$ first, and then computing the outer derivatives. This approach is possible because the result of a derivative computation is represented as an expression tree just as any other expression. Mainly this approach was chosen because it is simple to implement and easy to verify. Whether other approaches are faster has not been investigated. Furthermore, alternative AD algorithms such as reverse mode can be experimented with in the future without concern for nested derivatives in the first implementations.

An outer controller function `apply_ad` handles the application of a single variable AD routine to an expression with possibly nested derivatives. The AD routine is a function accepting a derivative expression node and returning an expression where the single variable derivative has been computed.

[2] Functional as in functional languages.

*Python code*

```
def apply_ad(e, ad_routine):
    if isinstance(e, Terminal):
        return e
    ops = [apply_ad(o, ad_routine) for o in e.operands()]
    e = e.reconstruct(*ops)
    if isinstance(e, Derivative):
        e = ad_routine(e)
    return e
```

Figure 4.4: Simple implementation of recursive `apply_ad` procedure.

This routine can be an implementation of Algorithm 3. The result of `apply_ad` is mathematically equivalent to the input, but with no derivative expression nodes left[3].

The function `apply_ad` works by traversing the tree recursively in post-order, discovering subtrees where the root represents a derivative, and applying the provided AD routine to the derivative subtree. Since the children of the derivative node has already been visited by `apply_ad`, they are guaranteed to be free of derivative expression nodes and the AD routine only needs to handle the case discussed above with algorithms 2 and 3.

The complexity of the `ad_routine` should be $O(n)$, with $n$ being the size of the expression tree. The size of the derivative expression is proportional to the original expression. If there are $d$ derivative expression nodes in the expression tree, the complexity of this algorithm is $O(dn)$, since `ad_routine` is applied to subexpressions $d$ times. As a result the worst case complexity of `apply_ad` is $O(n^2)$, but in practice $d \ll n$. A recursive implementation of this algorithm is shown in Figure 4.4.

### 4.7.5 Basic differentiation rules

To implement the algorithm descriptions above, we must implement differentiation rules for all expression node types. Derivatives of operators can be implemented as generic rules independent of the differentiation variable, and these are well known and not mentioned here. Derivatives of terminals depend on the differentiation variable type. Derivatives of literal constants are of course always zero, and only spatial derivatives of geometric quantities are nonzero. Since form arguments are unknown to UFL (they are provided externally by the form compilers), their spatial derivatives ($\frac{\partial \phi^k}{\partial x_i}$ and $\frac{\partial w^k}{\partial x_i}$) are considered input arguments as well. In all derivative computations, the assumption is made that form coefficients have no dependencies on the differentiation variable. Two more cases needs explaining, the user defined variables and derivatives w.r.t. the coefficients of a `Coefficient`.

If $v$ is a `Variable`, then we define $\frac{dt}{dv} \equiv 0$ for any terminal $t$. If $v$ is scalar valued then $\frac{dv}{dv} \equiv 1$. Furthermore, if $V$ is a tensor valued `Variable`, its derivative w.r.t. itself is

$$\frac{dV}{dV} = \frac{dV_{ij}}{dV_{kl}} e_i \otimes e_j \otimes e_k \otimes e_l = \delta_{ik}\delta_{jl} e_i \otimes e_j \otimes e_k \otimes e_l. \tag{4.71}$$

In addition, the derivative of a variable w.r.t. something else than itself equals the derivative of the expression it represents:

$$v = g, \tag{4.72}$$

$$\frac{dv}{dz} = \frac{dg}{dz}. \tag{4.73}$$

Finally, we consider the operator `derivative`, which represents differentiation w.r.t. all coefficients

[3] Except direct spatial derivatives of form arguments, but that is an implementation detail.

$\{w_k\}$ of a function $w$. Consider an object `element` which represents a finite element space $V_h$ with a basis $\{\phi_k\}$. Next consider form arguments defined in this space:

*UFL code*

```
v = Argument(element)
w = Coefficient(element)
```

The Argument instance `v` represents any $v \in \{\phi_k\}$, while the `Coefficient` instance `w` represents the sum

$$w = \sum_k w_k \phi_k(x). \tag{4.74}$$

The derivative of `w` w.r.t. any $w_k$ is the corresponding basis function in $V_h$,

$$\frac{\partial w}{\partial w_k} = \phi_k, \quad k = 1, \ldots, |V_h|, \tag{4.75}$$

$$\tag{4.76}$$

which can be represented by `v`, since

$$v \in \langle \phi_k \rangle_{k=1}^{|V_h|} = \left\langle \frac{\partial w}{\partial w_k} \right\rangle_{k=1}^{|V_h|}. \tag{4.77}$$

Note that `v` should be a basis function instance that has not already been used in the form.

## 4.8 Algorithms

In this section, some central algorithms and key implementation issues are discussed, much of which relates to the Python programming language. Thus, this section is mainly intended for developers and others who need to relate to UFL on a technical level. Python users may also find some of the techniques here interesting.

### 4.8.1 Effective tree traversal in Python

Applying some action to all nodes in a tree is naturally expressed using recursion:

*Python code*

```
def walk(expression, pre_action, post_action):
    pre_action(expression)
    for o in expression.operands():
        walk(o)
    post_action(expression)
```

This implementation simultaneously covers pre-order traversal, where each node is visited before its children, and post-order traversal, where each node is visited after its children.

A more "pythonic" way to implement iteration over a collection of nodes is using generators. A minimal implementation of this could be

*Python code*

```
def post_traversal(root):
    for o in root.operands():
        yield post_traversal(o)
    yield root
```

which then enables the natural Python syntax for iteration over expression nodes:

*Python code*

```
for e in post_traversal(expression):
    post_action(e)
```

For efficiency, the actual implementation of `post_traversal` in UFL is not using recursion. Function calls are very expensive in Python, which makes the non-recursive implementation an order of magnitude faster than the above.

### 4.8.2 *Type based function dispatch in Python*

A common task in both symbolic computing and compiler implementation is the selection of some operation based on the type of an expression node. For a selected few operations, this is done using overloading of functions in the subclasses of `Expr`, but this is not suitable for all operations. In many cases type-specific operations are better implemented together in the algorithm instead of distributed across class definitions. This implementation pattern is called the Visitor pattern [Gamma et al., 1995]. The implementation in UFL is somewhat different from the patterns used in a statically typed language such as C++.

One way to implement type based operation selection is to use a *type switch*, which is a sequence of if-tests as shown here:

*Python code*

```
def operation(expression):
    if isinstance(expression, IntValue):
        return int_operation(expression)
    elif isinstance(expression, Sum):
        return sum_operation(expression)
    # etc.
```

There are several problems with this approach, one of which is efficiency when there are many types to check. A type based function dispatch mechanism with efficiency independent of the number of types is implemented as an alternative through the class `MultiFunction`. The underlying mechanism is a dictionary lookup (which is $O(1)$) based on the type of the input argument, followed by a call to the function found in the dictionary. The lookup table is built in the `MultiFunction` constructor only once. Functions to insert in the table are discovered automatically using the introspection capabilities of Python.

A multifunction is declared as a subclass of `MultiFunction`. For each type that should be handled particularly, a member function is declared in the subclass. The `Expr` classes use the `CamelCaps` naming convention, which is automatically converted to `underscore_notation` for corresponding function names, such as `IndexSum` and `index_sum`. If a handler function is not declared for a type, the closest superclass handler function is used instead. Note that the `MultiFunction` implementation is specialized to types in the `Expr` class hierarchy. The declaration and use of a multifunction is illustrated in this example code:

*Python code*

```
class ExampleFunction(MultiFunction):
    def __init__(self):
        MultiFunction.__init__(self)

    def terminal(self, expression):
        return "Got a Terminal subtype %s." % type(expression)
```

```
    def operator(self, expression):
        return "Got an Operator subtype %s." % type(expression)

    def argument(self, expression):
        return "Got an Argument."

    def sum(self, expression):
        return "Got a Sum."

m = ExampleFunction()

cell = triangle
element = FiniteElement("Lagrange", cell, 1)
x = cell.x
print m(Argument(element))
print m(x)
print m(x[0] + x[1])
print m(x[0] * x[1])
```

Note that `argument` and `sum` will handle instances of the exact types `Argument` and `Sum`, while `terminal` and `operator` will handle the types `SpatialCoordinate` and `Product` since they have no specific handlers.

### 4.8.3 *Implementing expression transformations*

Many transformations of expressions can be implemented recursively with some type-specific operation applied to each expression node. Examples of operations are converting an expression node to a string representation, to an expression representation using an symbolic external library, or to a UFL representation with some different properties. A simple variant of this pattern can be implemented using a multifunction to represent the type-specific operation:

*Python code*

```
def apply(e, multifunction):
    ops = [apply(o, multifunction) for o in e.operands()]
    return multifunction(e, *ops)
```

The basic idea is as follows. Given an expression node `e`, begin with applying the transformation to each child node. Then return the result of some operation specialized according to the type of `e`, using the already transformed children as input.

The `Transformer` class implements this pattern. Defining a new algorithm using this pattern involves declaring a `Transformer` subclass, and implementing the type specific operations as member functions of this class just as with `MultiFunction`. The difference is that member functions take one additional argument for each operand of the expression node. The transformed child nodes are supplied as these additional arguments. The following code replaces terminal objects with objects found in a dictionary `mapping`, and reconstructs operators with the transformed expression trees. The algorithm is applied to an expression by calling the function `visit`, named after the similar Visitor pattern.

*Python code*

```
class Replacer(Transformer):
    def __init__(self, mapping):
        Transformer.__init__(self)
        self.mapping = mapping

    def operator(self, e, *ops):
        return e.reconstruct(*ops)
```

```
    def terminal(self, e):
        return self.mapping.get(e, e)

f = Constant(triangle)
r = Replacer({f: f**2})
g = r.visit(2*f)
```

After running this code the result is $g = 2f^2$. The actual implementation of the `replace` function is similar to this code.

In some cases, child nodes should not be visited before their parent node. This distinction is easily expressed using `Transformer`, simply by omitting the member function arguments for the transformed operands. See the source code for many examples of algorithms using this pattern.

### 4.8.4 *Important transformations*

There are many ways in which expression representations can be manipulated. Here, we describe three particularly important transformations. Note that each of these algorithms removes some abstractions, and hence may remove some opportunities for analysis or optimization. To demonstrate their effect, each transformation will be applied below to the expression

$$a = \operatorname{grad}(fu) \cdot \operatorname{grad} v. \tag{4.78}$$

At the end of the section, some example code is given to demonstrate more representation details.

Some operators in UFL are termed "compound" operators, meaning they can be represented by other more elementary operators. Try defining an expression `a = dot(grad(f*u), grad(v))`, and print `repr(a)`. As you will see, the representation of `a` is `Dot(Grad(Product(f, u)), Grad(v))`, with some more details in place of `f`, `u` and `v`. By representing the gradient directly with a high level type `Grad` instead of more low level types, the input expressions are easier to recognize in the representation, and rendering of expressions to for example LaTeX format can show the original compound operators as written by the end-user. However, since many algorithms must implement actions for each operator type, the function `expand_compounds` is used to replace all expression nodes of "compound" types with equivalent expressions using basic types. When this operation is applied to the input forms from the user, algorithms in both UFL and the form compilers can still be written purely in terms of more basic operators. Expanding the compound expressions from Equation (4.78) results in the expression

$$a_c = \sum_i \frac{\partial v}{\partial x_i} \frac{\partial (uf)}{\partial x_i}. \tag{4.79}$$

Another important transformation is `expand_derivatives`, which applies automatic differentiation to expressions, recursively and for all kinds of derivatives. The end result is that most derivatives are evaluated, and the only derivative operator types left in the expression tree applies to terminals. The precondition for this algorithm is that `expand_compounds` has been applied. Expanding the derivatives in $a_c$ from Equation (4.79) gives us

$$a_d = \sum_i \frac{\partial v}{\partial x_i} (u \frac{\partial f}{\partial x_i} + f \frac{\partial u}{\partial x_i}). \tag{4.80}$$

Index notation and the `IndexSum` expression node type complicate interpretation of an expression tree somewhat, in particular in expressions with nested index sums. Since expressions with free indices will take on multiple values, each expression object represents not only one value but a

set of values. The transformation `expand_indices` then comes in handy. The precondition for this algorithm is that `expand_compounds` and `expand_derivatives` have been applied. The postcondition of this algorithm is that there are no free indices left in the expression. Expanding the indices in Equation (4.80) finally gives

$$a_i = \frac{\partial v}{\partial x_0}(u\frac{\partial f}{\partial x_0} + f\frac{\partial u}{\partial x_0}) + \frac{\partial v}{\partial x_1}(u\frac{\partial f}{\partial x_1} + f\frac{\partial u}{\partial x_1}). \tag{4.81}$$

We started with the higher level concepts gradient and dot product in Equation (4.78), and ended with only scalar addition, multiplication, and partial derivatives of the form arguments. A form compiler will typically start with $a_d$ or $a_i$, insert values for the argument derivatives, apply some other transformations, before finally generating code.

Some example code to play around with should help in understanding what these algorithms do at the expression representation level. Since the printed output from this code is a bit lengthy, only key aspects of the output is repeated below. Copy this code to a python file or run it in a python interpreter to see the full output.

*Python code*

```
from ufl import *
V = FiniteElement("Lagrange", triangle, 1)
u = TestFunction(V)
v = TrialFunction(V)
f = Coefficient(V)

# Note no *dx! This is an expression, not a form.
a = dot(grad(f*u), grad(v))

from ufl.algorithms import *
ac = expand_compounds(a)
ad = expand_derivatives(ac)
ai = expand_indices(ad)
print "\na: ", str(a),  "\n", tree_format(a)
print "\nac:", str(ac), "\n", tree_format(ac)
print "\nad:", str(ad), "\n", tree_format(ad)
print "\nai:", str(ai), "\n", tree_format(ai)
```

The print output showing `a` is (with the details of the finite element object cut away for shorter lines):

*Output*

```
a:  (grad(v_{-2} * w_0)) . (grad(v_{-1}))
Dot
(
    Grad
        Product
        (
            Argument(FiniteElement(...), -2)
            Coefficient(FiniteElement(...), 0)
        )
    Grad
        Argument(FiniteElement(...), -1)
)
```

The arguments labeled -1 and -2 refer to $v$ and $u$ respectively.

In `ac`, the `Dot` product has been expanded to an `IndexSum` of a `Product` with two `Indexed` operands:

*Output*

```
IndexSum
(
    Product
    (
        Indexed
        (
            ...
            MultiIndex((Index(10),), {Index(10): 2})
        )
        Indexed
        (
            ...
            MultiIndex((Index(10),), {Index(10): 2})
        )
    )
    MultiIndex((Index(10),), {Index(10): 2})
)
```

The somewhat complex looking expression `MultiIndex((Index(10),), {Index(10): 2})` can be read simply as "index named $i_{10}$, bound to an axis with dimension 2".

Zooming in to one of the `...` lines above, the representation of grad$(fu)$ must still keep the vector shape after being transformed to more basic expressions, which is why the `SpatialDerivative` object is wrapped in a `ComponentTensor` object:

*Output*

```
ComponentTensor
(
    SpatialDerivative
    (
        Product
        (
            u
            f
        )
        MultiIndex((Index(8),), {Index(8): 2})
    )
    MultiIndex((Index(8),), {Index(8): 2})
)
```

A common pattern occurs in the algorithmically expanded expressions:

*Output*

```
Indexed
(
    ComponentTensor
    (
        ...
        MultiIndex((Index(8),), {Index(8): 2})
    )
    MultiIndex((Index(10),), {Index(10): 2})
)
```

This pattern acts as a relabeling of the index objects, renaming $i_8$ from inside ... to $i_{10}$ on the outside. When looking at the print of `ad`, the result of the chain rule ($(fu)' = uf' + fu'$) can be seen as the `Sum` of two `Product` objects.

*Output*

```
Sum
(
    Product
    (
        u
        SpatialDerivative
        (
            f
            MultiIndex((Index(8),), {Index(8): 2})
        )
    )
    Product
    (
        f
        SpatialDerivative
        (
            u
            MultiIndex((Index(8),), {Index(8): 2})
        )
    )
)
```

Finally after index expansion in `ai` (not shown here), no free `Index` objects are left, but instead a lot of `FixedIndex` objects can be seen in the print of `ai`. Looking through the full output from the example code above is strongly encouraged if you want a good understanding of the three transformations shown here.

### 4.8.5 *Evaluating expressions*

Even though UFL expressions are intended to be compiled by form compilers, it can be useful to evaluate them to floating point values directly. In particular, this makes testing and debugging of UFL much easier, and is used extensively in the unit tests. To evaluate an UFL expression, values of form arguments and geometric quantities must be specified. Expressions depending only on spatial coordinates can be evaluated by passing a tuple with the coordinates to the call operator. The following code which can be copied directly into an interactive Python session shows the syntax:

*Python code*

```
from ufl import *
cell = triangle
x = cell.x
e = x[0] + x[1]
print e((0.5, 0.7)) # prints 1.2
```

Other terminals can be specified using a dictionary that maps from terminal instances to values. This code extends the above code with a mapping:

*Python code*

```
c = Constant(cell)
e = c*(x[0] + x[1])
print e((0.5, 0.7), { c: 10 }) # prints 12.0
```

If functions and basis functions depend on the spatial coordinates, the mapping can specify a Python callable instead of a literal constant. The callable must take the spatial coordinates as input and return a floating point value. If the function being mapped is a vector function, the callable must return a tuple of values instead. These extensions can be seen in the following code:

*Python code*

```
element = VectorElement("Lagrange", triangle, 1)
c = Constant(triangle)
f = Coefficient(element)
e = c*(f[0] + f[1])
def fh(x):
    return (x[0], x[1])
print e((0.5, 0.7), { c: 10, f: fh }) # prints 12.0
```

To use expression evaluation for validating that the derivative computations are correct, spatial derivatives of form arguments can also be specified. The callable must then take a second argument which is called with a tuple of integers specifying the spatial directions in which to differentiate. A final example code computing $g^2 + g_{,0}^2 + g_{,1}^2$ for $g = x_0 x_1$ is shown below.

*Python code*

```
element = FiniteElement("Lagrange", triangle, 1)
g = Coefficient(element)
e = g**2 + g.dx(0)**2 + g.dx(1)**2
def gh(x, der=()):
    if der == ():   return x[0]*x[1]
    if der == (0,): return x[1]
    if der == (1,): return x[0]
print e((2, 3), { g: gh }) # prints 49
```

### 4.8.6 *Viewing expressions*

Expressions can be formatted in various ways for inspection, which is particularly useful while debugging. The Python built in string conversion operator `str(e)` provides a compact human readable string. If you type `print e` in an interactive Python session, `str(e)` is shown. Another Python built in string operator is `repr(e)`. UFL implements `repr` correctly such that `e == eval(repr(e))` for any expression `e`. The string `repr(e)` reflects all the exact representation types used in an expression, and can therefore be useful for debugging. Another formatting function is `tree_format(e)`, which produces an indented multi-line string that shows the tree structure of an expression clearly, as opposed to `repr` which can return quite long and hard to read strings. Information about formatting of expressions as LaTeX and the dot graph visualization format can be found in the manual.

## 4.9 *Implementation issues*

### 4.9.1 *Python as a basis for a domain specific language*

Many of the implementation details detailed in this section are influenced by the initial choice of implementing UFL as an embedded language in Python. Therefore some words about why Python is suitable for this, and why not, are appropriate here.

Python provides a simple syntax that is often said to be close to pseudo-code. This is a good starting point for a domain specific language. Object orientation and operator overloading is well supported, and this is fundamental to the design of UFL. The functional programming features of Python (such as generator expressions) are useful in the implementation of algorithms and form compilers. The built-in data structures `list`, `dict` and `set` play a central role in fast implementations of scalable algorithms.

There is one problem with operator overloading in Python, and that is the comparison operators. The problem stems from the fact that `__eq__` or `__cmp__` are used by the built-in data structures dictionary and set to compare keys, meaning that `a == b` must return a boolean value for `Expr` to be

used as keys. The result is that `__eq__` can not be overloaded to return some `Expr` type representation such as `Equals(a, b)` for later processing by form compilers. The other problem is that `and` and `or` cannot be overloaded, and therefore cannot be used in `conditional` expressions. There are good reasons for these design choices in Python. This conflict is the reason for the somewhat non-intuitive design of the comparison operators in UFL.

### *4.9.2 Ensuring unique form signatures*

The form compilers need to compute a unique signature of each form for use in a cache system to avoid recompilations. A convenient way to define a signature is using `repr(form)`, since the definition of this in Python is `eval(repr(form)) == form`. Therefore `__repr__` is implemented for all `Expr` subclasses.

Some forms are mathematically equivalent even though their representation is not exactly the same. UFL does not use a truly canonical form for its expressions, but takes some measures to ensure that trivially equivalent forms are recognized as such.

Some of the types in the `Expr` class hierarchy (subclasses of `Counted`), has a global counter to identify the order in which they were created. This counter is used by form arguments (both `Argument` and `Coefficient`) to identify their relative ordering in the argument list of the form. Other counted types are `Index` and `Label`, which only use the counter as a unique identifier. Algorithms are implemented for renumbering of all `Counted` types such that all counts start from 0.

In addition, some operator types such as `Sum` and `Product` maintains a sorted list of operands such that `a+b` and `b+a` are both represented as `Sum(a, b)`. This operand sorting is intentionally independent of the numbering of indices because that would not be stable. The reason for this instability is that the result of algorithms for renumbering indices depends on the order of operands. The operand sorting and renumberings combined ensure that the signature of equal forms will stay the same. Note that the representation, and thus the signature, of a form may change with versions of UFL. The following line prints the signature of a form with `expand_derivatives` and renumbering applied.

*Python code*

```
print repr(preprocess(myform).preprocessed_form)
```

### *4.9.3 Efficiency considerations*

By writing UFL in Python, we clearly do not put peak performance as a first priority. If the form compilation process can blend into the application build process, the performance is sufficient. We do, however, care about scaling performance to handle complicated equations efficiently, and therefore about the asymptotic complexity of the algorithms we use.

To write clear and efficient algorithms in Python, it is important to use the built in data structures correctly. These data structures include in particular `list`, `dict` and `set`. CPython [van Rossum et al.], the reference implementation of Python, implements the data structure `list` as an array, which means append, and pop, and random read or write access are all O(1) operations. Random insertion, however, is O(n). Both `dict` and `set` are implemented as hash maps, the latter simply with no value associated with the keys. In a hash map, random read, write, insertion and deletion of items are all $O(1)$ operations, as long as the key types implement `__hash__` and `__eq__` efficiently. The dictionary data structure is used extensively by the Python language, and therefore particular attention has been given to make it efficient [Kuchling, 2007]. Thus to enjoy efficient use of these containers, all `Expr` subclasses must implement these two special functions efficiently. Such considerations have been important for making the UFL implementation perform efficiently.

## 4.10 Conclusions and future directions

Many additional features can be introduced to UFL. Which features are added will depend on the needs of FEniCS users and developers. Some features can be implemented in UFL alone, but most features will require updates to other parts of the FEniCS project. Thus the future directions for UFL is closely linked to the development of the FEniCS project as a whole.

Improvements to finite element declarations is likely easy to do in UFL. The added complexity will mostly be in the form compilers. Among the current suggestions are space-time elements and time derivatives. Additional geometry mappings and finite element spaces with non-uniform cell types are also possible extensions.

Additional operators can be added to make the language more expressive. Some operators are easy to add because their implementation only affects a small part of the code. More compound operators that can be expressed using elementary operations is easy to add. Additional special functions are easy to add as well, as long as their derivatives are known. Other features may require more thorough design considerations, such as support for complex numbers which will affect large parts of the code.

User friendly notation and support for rapid development are core values in the design of UFL. Having a notation close to the mathematical abstractions allows expression of particular ideas more easily, which can reduce the probability of bugs in user code. However, the notion of metaprogramming and code generation adds another layer of abstraction which can make understanding the framework more difficult for end-users. Good error checking everywhere is therefore very important, to detect user errors as close as possible to the user input. Improvements to the error messages, documentation, and unit test suite will always be helpful, to avoid frequently repeated errors and misunderstandings among new users.

To support the form compiler projects, algorithms and utilities for generating better code more efficiently could be included in UFL. Such algorithms should probably be limited to algorithms such as general transformations of expression graphs which can be useful independently of form compiler specific approaches. In this area, more work on alternative automatic differentiation algorithms [Forth et al., 2004, Tadjouddine, 2008] can be useful.

To summarize, UFL is a central component in the FEniCS framework, where it provides a rich form language, automatic differentiation, and a building block for efficient form compilers. These are useful features in rapid development of applications for efficiently solving partial differential equations. UFL provides the user interface to Automation of Discretization that is the core feature of FEniCS, and adds Automation of Linearization to the framework. With these features, UFL has brought FEniCS one step closer to its overall goal Automation of Mathematical Modeling.

## 4.11 Acknowledgements

This work has been supported by the Norwegian Research Council (grant 162730) and Simula Research Laboratory. I wish to thank everyone who has helped improving UFL with suggestions and testing, in particular Anders Logg, Kristian Ølgaard, Garth Wells, Harish Narayanan and Marie Rognes. In addition to the two anonymous referees, both Kent-André Mardal and Marie Rognes performed critical reviews which greatly improved this chapter.

# 5 *The FEniCS book*

For further details, the reader is referred to the FEniCS book: *Automated solution of differential equations by the finite element method*.

# *References*


M. Ainsworth and J. T. Oden. A unified approach to *a posteriori* error estimation using element residual methods. *Numerische Mathematik*, 65(1):23–50, 1993. URL `http://dx.doi.org/10.1007/BF01385738`.

M. Ainsworth and J. T. Oden. A Posteriori *Error Estimation in Finite Element Analysis*. Wiley and Sons, New York, 2000.

M. S. Alnæs and A. Logg. *UFL Specification and User Manual*, 2009. URL `http://fenicsproject.org`.

M. S. Alnæs and K.-A. Mardal. On the efficiency of symbolic computations combined with code generation for finite element methods. *ACM Trans. Math. Softw.*, 37:6:1–6:26, 2010. URL `http://dx.doi.org/10.1145/1644001.1644007`.

M. S. Alnæs, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen. Unified framework for finite element assembly. *International Journal of Computational Science and Engineering*, 2009. Accepted for publication. Preprint: `http://simula.no/research/scientific/publications/Simula.SC.96`.

D. N. Arnold, R. S. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numerica*, 15:1–155, 2006. URL `http://dx.doi.org/10.1017/S0962492906210018`.

O. Axelsson. A general incomplete block-matrix factorization method. *Linear Algebra and its Applications*, 74:179–190, 1986. ISSN 0024-3795.

I. Babuška and W. C. Rheinboldt. A posteriori error estimates for the finite element method. *Int. J. Numer. Meth. Engrg.*, pages 1597–1615, 1978.

S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

R. E. Bank and A. Weiser. Some a posteriori error estimators for elliptic partial differential equations. *Mathematics of Computation*, pages 283–301, 1985.

C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation*, 33(1):1–1, 2002. URL `http://dx.doi.org/10.1006/jsco.2001.0494`.

D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop*, volume 4, Berkeley, 1996. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1267498.1267513`.

E. B. Becker, G. F. Carey, and J. T. Oden. *Finite Elements: An Introduction*. Prentice–Hall, Englewood–Cliffs, 1981.

R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001.

W. N. Bell, L. N. Olson, and J. Schroder. PyAMG: Algebraic multigrid solvers in Python, 2011. URL `http://www.pyamg.org`.

W. B. Bickford. *A First Course in the Finite Element Method*. Irwin, 1994.

C. Bischof, A. Carle, G. Corliss, and A. Griewank. ADIFOR: Automatic differentiation in a source translator environment. In *Papers from the international symposium on Symbolic and algebraic computation*, pages 294–302, Berkeley, California, 1992. URL `http://dx.doi.org/10.1145/143242.143335`.

C. H. Bischof, P. D. Hovland, and B. Norris. Implementation of automatic differentiation tools. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 98–107, Portland, Oregon, 2002. URL `http://dx.doi.org/10.1145/503032.503047`.

D. Braess. *Finite Elements*. Cambridge University Press, third edition, 2007. ISBN 978-0-521-70518-9.

A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, pages 333–390, 1977.

S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, third edition, 2008. URL `http://dx.doi.org/10.1007/978-0-387-75934-0`.

C. A. P. Castigliano. *Théorie de l'équilibre des systèmes élastiques et ses applications*. A.F. Negro ed., Torino, 1879.

CGAL. Software package. URL `http://www.cgal.org`.

P. G. Ciarlet. *Numerical Analysis of the Finite Element Method*. Les Presses de l'Universite de Montreal, 1976.

P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*, volume 40 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2002.

R. Cimrman et al. SfePy: Simple Finite Elements in Python, 2008. URL `http://sfepy.org`.

R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bull. Amer. Math. Soc.*, pages 1–23, 1943.

T. A. Davis. Algorithm 832: Umfpack v4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004. URL `http://dx.doi.org/10.1145/992200.992206`.

J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. Wiley, 2003.

W. Dörfler. A convergent adaptive algorithm for Poisson's equation. *SIAM Journal on Numerical Analysis*, 33(3):1106–1124, 1996.

P. Dular and C. Geuzaine. *GetDP Reference Manual*, 2005.

H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, 2005.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems III: Time steps variable in space. *in preparation*.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems I: A linear model problem. *SIAM J. Numer. Anal.*, 28, No. 1:43–77, 1991.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems II: Optimal error estimates in $ l_\ infty l_2$ and$ l_\ infty l_\ infty$. *SIAM Journal on Numerical Analysis*, 32:706, 1995a.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems IV: Nonlinear problems. *SIAM Journal on Numerical Analysis*, pages 1729–1749, 1995b.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems V: Long-time integration. *SIAM J. Numer. Anal.*, 32:1750–1763, 1995c.

K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. Introduction to adaptive methods for differential equations. *Acta Numerica*, 4:105–158, 1995.

K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

K. Eriksson, C. Johnson, and S. Larsson. Adaptive finite element methods for parabolic problems VI: Analytic semigroups. *SIAM J. Numer. Anal.*, 35:1315–1325, 1998.

A. Ern and J.-L. Guermond. *Theory and Practice of Finite Elements*, volume 159 of *Applied Mathematical Sciences*. Springer-Verlag, 2004.

R. D. Falgout and U. M. Yang. Hypre: A library of high performance preconditioners. In *Proceedings of the International Conference on Computational Science-Part III*, pages 632–641. Springer-Verlag, 2002.

S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Trans. Math. Softw.*, 30(3): 266–299, 2004. URL `http://dx.doi.org/10.1145/1024074.1024076`.

B. G. Galerkin. Series solution of some problems in elastic equilibrium of rods and plates. *Vestnik inzhenerov i tekhnikov*, 19:897–908, 1915.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Softw.*, 24(4): 437–474, 1998. URL `http://dx.doi.org/10.1145/293686.293695`.

M. S. Gockenbach. *Understanding and Implementing the Finite Element Method*. Society for Industrial and Applied Mathematics, 2006.

A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3): 397–423, 2005. URL `http://dx.doi.org/10.1145/1089014.1089021`.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand*, 49(6):409–436, 1952.

M. L. Hetland. *Practical Python*. APress, 2002.

T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2009. URL `http://scipy.org`.

J. Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, 2001. URL `http://dx.doi.org/10.1023/A:1011501232197`.

R. C. Kirby. Algorithm 839: FIAT, a new paradigm for computing finite element basis functions. *ACM Trans. Math. Softw.*, 30:502–516, 2004. URL `http://dx.doi.org/10.1145/1039813.1039820`.

R. C. Kirby. Optimizing FIAT with level 3 BLAS. *ACM Trans. Math. Softw.*, 32:223–235, 2006. URL `http://dx.doi.org/10.1145/1141885.1141889`.

R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32:417–444, 2006. URL `http://dx.doi.org/10.1145/1163641.1163644`.

R. C. Kirby and A. Logg. Efficient compilation of a class of variational forms. *ACM Transactions on Mathematical Software*, 33(3), 2007.

R. C. Kirby and A. Logg. Benchmarking domain-specific compiler optimizations for variational forms. *ACM Transactions on Mathematical Software*, 35(2):1–18, 2008. URL `http://dx.doi.org/10.1145/1377612.1377614`.

R. C. Kirby and L. R. Scott. Geometric optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 29:827–841, 2007. URL `http://dx.doi.org/10.1137/060660722`.

R. C. Kirby, M. G. Knepley, A. Logg, and L. R. Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(6):741–758, 2005. ISSN 1064-8275.

R. C. Kirby, A. Logg, L. R. Scott, and A. R. Terrel. Topological optimization of the evaluation of finite element matrices. *SIAM Journal on Scientific Computing*, 28(1):224–240, 2006. URL `http://dx.doi.org/10.1137/050635547`.

J. Kiusalaas. *Numerical Methods in Engineering with Python*. Cambridge University Press, second edition, 2009.

J. Korelc. Automatic generation of finite-element code by simultaneous optimization of expressions. *Theoretical Computer Science*, 187(1-2):231–248, 1997. URL `http://dx.doi.org/10.1016/S0304-3975(97)00067-4`.

J. Korelc. Multi-language and multi-environment generation of nonlinear finite element codes. *Engineering with Computers*, 18(4):312–327, 2002. URL `http://dx.doi.org/10.1007/s003660200028`.

A. Kuchling. Python's dictionary implementation: Being all things to all people. In A. Oram and G. Wilson, editors, *Beautiful Code*, chapter 18, pages 293–302. O'Reilly, 2007. URL `http://portal.acm.org/citation.cfm?id=1407867`.

H. P. Langtangen. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, third edition, 2008. URL `http://dx.doi.org/10.1007/978-3-540-73916-6`.

H. P. Langtangen. *A Primer on Scientific Programming with Python*, volume 6 of *Texts in Computational Science and Engineering*. Springer, second edition, 2011. URL `http://dx.doi.org/10.1007/978-3-642-18366-9`.

P. D. Lax and A. N. Milgram. Parabolic equations. *Annals of Mathematics Studies*, 33:167–190, 1954.

X. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software*, 31(3):302–325, 2005.

A. Logg. Automating the finite element method. *Arch. Comput. Methods Eng.*, 14(2):93–138, 2007.

A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4(4):283–295, 2009.

A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software*, 32(2):1–28, 2010. URL `http://dx.doi.org/10.1145/1731022.1731030`.

K. Long. Sundance, a rapid prototyping tool for parallel PDE-constrained optimization. In *Large-Scale PDE-Constrained Optimization*, Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2003.

K. Long. Sundance 2.0 tutorial. Technical Report TR–2004–09, Sandia National Laboratories, 2004a.

K. Long. Efficient discretization and differentiation of partial differential equations through automatic functional differentiation, 2004b. http://www.autodiff.org/ad04/abstracts/Long.pdf.

M. Lutz. *Programming Python*. O'Reilly, third edition, 2006.

M. Lutz. *Learning Python*. O'Reilly, third edition, 2007.

T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation*, 34(150):473–497, 1980.

A. Martelli. *Python in a Nutshell*. O'Reilly, second edition, 2006.

A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, second edition, 2005.

MayaVi2. Software package. URL `http://code.enthought.com/projects/mayavi/`.

MeshBuilder. Software package. URL `http://launchpad.net/meshbuilder`.

M. Mortensen, H. P. Langtangen, and J. Myre. cbc.rans – a new flexible, programmable software framework for computational fluid dynamics. In H. I. Andersson and B. Skallerud, editors, *Sixth National Conference on Computational Mechanics (MekIT'11)*. Tapir, 2011a.

M. Mortensen, H. P. Langtangen, and G. N. Wells. A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier-Stokes equations. *Advances in Water Resources*, 2011b. URL `http://dx.doi.org/10.1016/j.advwatres.2011.02.013`.

MTL4. Software package. URL `http://www.mtl4.org`.

J. T. Oden and L. Demkowicz. *Applied Functional Analysis*. CRC press, 1996.

K. B. Ølgaard and G. N. Wells. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software*, 37(1):8:1–8:23, 2010. URL http://dx.doi.org/10.1145/1644001.1644009.

K. B. Ølgaard, A. Logg, and G. N. Wells. Automated code generation for discontinuous Galerkin methods. *SIAM J. Sci. Comput.*, 31(2):849–864, 2008. URL http://dx.doi.org//10.1137/070710032.

ParaView. Software package. URL http://www.paraview.org.

ParMETIS. Software package. URL http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

PaStiX. Software package. URL http://pastix.gforge.inria.fr/.

B. A. Pearlmutter and J. M. Siskind. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–160, Nice, France, 2007. URL http://dx.doi.org/10.1145/1190216.1190242.

F. Pellegrini. Scotch. URL http://www.labri.fr/perso/pelegrin/scotch.

PETSc. software package. URL http://www.anl.gov/petsc.

C. Prud'homme. Life: Overview of a unified c++ implementation of the finite and spectral element methods in 1d, 2d and 3d. In *Workshop On State-Of-The-Art In Scientific And Parallel Computing*, Lecture Notes in Computer Science, page 10. Springer-Verlag, dec 2006a.

C. Prud'homme. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Sci. Program.*, 14(2):81–110, 2006b. URL http://portal.acm.org/citation.cfm?id=1376891.1376895.

A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*, volume 23 of *Springer Series in Computational Mathematics*. Springer Verlag, 2008. URL http://dx.doi.org/10.1007/978-3-540-85268-1.

Rayleigh. On the theory of resonance. *Trans. Roy. Soc.*, A161:77–118, 1870.

W. Ritz. Über eine neue Methode zur Lösung gewisser Variationsprobleme der mathematischen Physik. *J. reine angew. Math.*, 135:1–61, 1908.

M.-C. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613, 1984. URL http://dx.doi.org/10.1137/0721042.

M.-C. Rivara. Local modification of meshes for adaptive and/or multigrid finite-element methods. *Journal of Computational and Applied Mathematics*, 36(1):78–89, 1992.

M. E. Rognes, R. C. Kirby, and A. Logg. Efficient assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming finite elements. *SIAM Journal on Scientific Computing*, 31(6):4130–4151, 2009.

Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

J. M. Siskind and B. A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher Order Symbol. Comput.*, 21(4):361–376, 2008. URL http://dx.doi.org/10.1007/s10990-008-9037-1.

O. Skavhaug and O. Čertík. Swiginac Python interface to GiNaC, 2009. URL `http://swiginac.berlios.de/`.

G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*. Prentice-Hall, Englewood Cliffs, 1973.

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, July 1997.

SWIG. Software package. URL `http://www.swig.org`.

E. M. Tadjouddine. Vertex-ordering algorithms for automatic differentiation of computer codes. *The Computer Journal*, 51(6):688–699, Nov. 2008. URL `http://dx.doi.org/10.1093/comjnl/bxm115`.

The Python Tutorial. The python tutorial. `http://docs.python.org/tutorial/`.

Trilinos. software package. URL `http://trilinos.sandia.gov/`.

uBLAS. Software package. URL `http://www.boost.org/libs/numeric/ublas/doc/`.

G. van Rossum et al. *Python*. URL `http://www.python.org/`.

O. Čertík et al. *SymPy*, 2009. URL `http://docs.sympy.org`.

R. Verfürth. *A posteriori* error estimation and adaptive mesh-refinement techniques. In *Proceedings of the Fifth International Conference on Computational and Applied Mathematics*, pages 67–83. Amsterdam, Elsevier Science Publishers, 1994.

R. Verfürth. A review of a posteriori error estimation techniques for elasticity problems. *Computer Methods in Applied Mechanics and Engineering*, 176(1-4):419–440, 1999.

Viper. Software package. URL `https://launchpad.net/fenics-viper`.

P. Wesseling. *An Introduction to Multigrid Methods*. Wiley & Sons, 1992.

O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptive procedure for practical engineerng analysis. *International Journal for Numerical Methods in Engineering*, 24(2), 1987.

O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method — Its Basis and Fundamentals*. Elsevier, 6th edition, 2005.

# *Index*