# 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.

A. Logg et al., *Automated Solution of Differential Equations by the Finite Element Method*, Lecture Notes in Computational Science and Engineering 84, DOI 10.1007/978-3-642-23099-8_1,

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

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

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

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.