

# FEniCS Course

## Lecture 2: Static linear PDEs

---

### *Contributors*

Hans Petter Langtangen, Anders Logg  
Marie E. Rognes, André Massing



FENICS  
PROJECT

# Hello World!

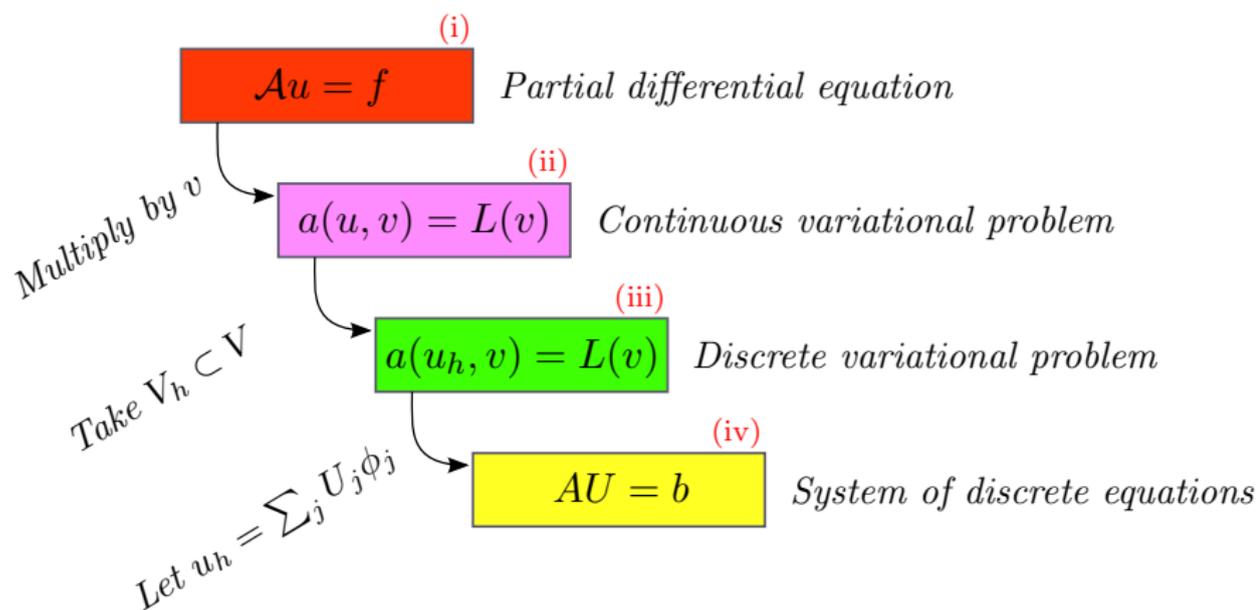
We will solve Poisson's equation, the Hello World of scientific computing:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \\ u &= u_0 && \text{on } \partial\Omega \end{aligned}$$

Poisson's equation arises in numerous contexts:

- heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, water waves, magnetostatics
- as part of numerical splitting strategies of more complicated systems of PDEs, in particular the Navier–Stokes equations

# The FEM cookbook



# Solving PDEs in FEniCS

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  $\Omega$
- 4 Add statements in the program for solving the variational problem, computing derived quantities such as  $\nabla u$ , and visualizing the results

## Deriving a variational problem for Poisson's equation

The simple recipe is: multiply the PDE by a test function  $v$  and integrate over  $\Omega$ :

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} f v \, dx$$

Then integrate by parts and set  $v = 0$  on the Dirichlet boundary:

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds}_{=0}$$

We find that:

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx$$

# Variational problem for Poisson's equation

Find  $u \in V$  such that

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx$$

for all  $v \in \hat{V}$

The trial space  $V$  and the test space  $\hat{V}$  are (here) given by

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}$$

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}$$

# Discrete variational problem for Poisson's equation

We approximate the continuous variational problem with a discrete variational problem posed on finite dimensional subspaces of  $V$  and  $\hat{V}$ :

$$V_h \subset V$$

$$\hat{V}_h \subset \hat{V}$$

Find  $u_h \in V_h \subset V$  such that

$$\int_{\Omega} \nabla u_h \cdot \nabla v \, dx = \int_{\Omega} f v \, dx$$

for all  $v \in \hat{V}_h \subset \hat{V}$

## Canonical variational problem

The following canonical notation is used in FEniCS: find  $u \in V$  such that

$$a(u, v) = L(v)$$

for all  $v \in \hat{V}$

For Poisson's equation, we have

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

$a(u, v)$  is a *bilinear form* and  $L(v)$  is a *linear form*

## A test problem

We construct a test problem for which we can easily check the answer. We first define the exact solution by

$$u(x, y) = 1 + x^2 + 2y^2$$

We insert this into Poisson's equation:

$$f = -\Delta u = -\Delta(1 + x^2 + 2y^2) = -(2 + 4) = -6$$

This technique is called the *method of manufactured solutions*

# Implementation in FEniCS

```
from fenics import *

mesh = UnitSquareMesh(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)

u0 = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]",
                degree=2)
bc = DirichletBC(V, u0, "on_boundary")

f = Constant(-6.0)
u = TrialFunction(V)
v = TestFunction(V)
a = inner(grad(u), grad(v))*dx
L = f*v*dx

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

plot(u)
interactive() # If using VTK plotting
```

## Step by step: the first line

The first line of a FEniCS program usually begins with

```
from fenics import *
```

This imports key classes like `UnitSquareMesh`, `FunctionSpace`, `Function` and so forth, from the FEniCS user interface (DOLFIN)

## Step by step: creating a mesh

Next, we create a mesh of our domain  $\Omega$ :

```
mesh = UnitSquareMesh(8, 8)
```

This defines a mesh of  $8 \times 8 \times 2 = 128$  triangles of the unit square.

Other useful classes for creating built-in meshes include `UnitIntervalMesh`, `UnitCubeMesh`, `UnitCircleMesh`, `UnitSphereMesh`, `RectangleMesh` and `BoxMesh`

More complex geometries can be built using Constructive Solid Geometry (CSG) through the FEniCS component `mshr`:

```
from mshr import *
r = Rectangle(Point(0.5, 0.5), Point(1.5, 1.5))
c = Circle(Point(1.0, 1.0), 0.2)
g = r - c
mesh = generate_mesh(g, 10)
```

## Step by step: creating a function space

The following line creates a finite element function space relative to this mesh:

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

The second argument specifies the type of element, while the third argument is the degree of the basis functions on the element

Other types of elements include "Discontinuous Lagrange", "Brezzi-Douglas-Marini", "Raviart-Thomas", "Crouzeix-Raviart", "Nedelec 1st kind H(curl)" and "Nedelec 2nd kind H(curl)"

## Step by step: defining expressions

Next, we define an expression for the boundary value:

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

The formula must be written in C++ syntax, and the polynomial degree must be specified.

The `Expression` class is very flexible and can be used to create complex user-defined expressions. For more information, try

```
from fenics import *  
help(Expression)
```

in Python or, in the shell:

```
$ pydoc fenics.Expression
```

## Step by step: defining a boundary condition

The following code defines a Dirichlet boundary condition:

```
bc = DirichletBC(V, u0, "on_boundary")
```

This boundary condition states that a function in the function space defined by  $V$  should be equal to  $u_0$  on the domain defined by "on\_boundary"

Note that the above line does not yet apply the boundary condition to all functions in the function space

## Step by step: more about defining domains

For a Dirichlet boundary condition, a simple domain can be defined by a string

```
"on_boundary" # The entire boundary
```

Alternatively, domains can be defined by subclassing `SubDomain`

```
class Boundary(SubDomain):  
    def inside(self, x, on_boundary):  
        return on_boundary
```

You may want to experiment with the definition of the boundary:

```
"near(x[0], 0.0)" #  $x_0 = 0$   
"near(x[0], 0.0) || near(x[1], 1.0)"
```

There are many more possibilities, see

```
help(SubDomain)  
help(DirichletBC)
```

## Step by step: defining the right-hand side

The right-hand side  $f = -6$  may be defined as follows:

```
f = Expression("-6.0", degree=0)
```

or (more efficiently) as

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

## Step by step: defining variational problems

Variational problems are defined in terms of *trial* and *test* functions:

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

We now have all the objects we need in order to specify the bilinear form  $a(u, v)$  and the linear form  $L(v)$ :

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

## Step by step: solving variational problems

Once a variational problem has been defined, it may be solved by calling the `solve` function:

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

Note the reuse of the variable name `u` as both a `TrialFunction` in the variational problem and a `Function` to store the solution.

## Step by step: post-processing using Notebooks

Add these incantations on top (after importing dolfin/fenics)

```
import pylab
%matplotlib inline
parameters["plotting_backend"] = "matplotlib"
```

The solution and the mesh may be plotted by simply calling:

```
plot(u)
pylab.show()
plot(mesh)
pylab.show()
```

For postprocessing in ParaView or MayaVi, store the solution in VTK format:

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

## *The FEniCS challenge!*

Solve the partial differential equation

$$-\Delta u = f$$

with homogeneous Dirichlet boundary conditions on the unit square for  $f(x, y) = 2\pi^2 \sin(\pi x) \sin(\pi y)$ . Plot the error in the  $L^2$  norm as function of the mesh size  $h$  for a sequence of refined meshes. Try to determine the convergence rate.

- *Who can obtain the smallest error?*
- *Who can compute a solution with an error smaller than  $\epsilon = 10^{-6}$  in the fastest time?*

*The best students(s) will be rewarded with a FEniCS surprise!*

Hints: `help(errornorm)`, `help(assemble)`