FEniCS Course

Lecture 0: Introduction to FEM

Contributors
Anders Logg, Kent-Andre Mardal

What is FEM?

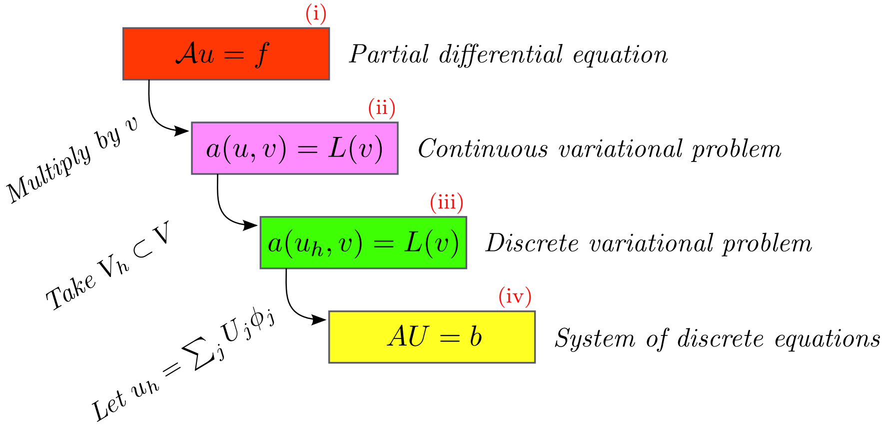
The finite element method is a framework and a recipe for discretization of mathematical problems in general

Examples:

- Ordinary differential equations
- Partial differential equations
- Integral equations

- A recipe for discretization of PDE
- That is, PDE $\rightarrow Ax = b$
- Issues that arise: choice of bases, stabilization, error control, adaptivity, computational complexity

The FEM cookbook



The method is a truely practical method that allows you to discretize *any* PDE on *any* domain and at the same time analyze (or even control) *accuracy*, *stability*, and *computational complexity* from a theoretical point of view

The PDE (i)

Consider Poisson's equation, the Hello World of partial differential equations:

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

Poisson's equation arises in numerous applications:

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

From PDE (i) to variational problem (ii)

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

$$-\int_\Omega (\Delta u) v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

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

$$-\int_\Omega (\Delta u) v \, \mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \, \mathrm{d}s}_{=0}$$

We find that:

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

The variational problem (ii)

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

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

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

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}$$

From continuous (ii) to discrete (iii) problem

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

$$\int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

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

From discrete variational problem (iii) to discrete system of equations (iv)

Choose a basis for the discrete function space:

$$V_h = \text{span}\,\{\phi_j\}_{j=1}^N$$

That is, we go from an abstract problem which applies to any bases to a concrete linear system in a given basis

Then, we make an ansatz for the discrete solution:

$$u_h = \sum_{j=1}^{N} U_j \phi_j$$

Test against the basis functions:

$$\int_\Omega \nabla(\underbrace{\sum_{j=1}^{N} U_j \phi_j}_{u_h}) \cdot \nabla \phi_i \,\mathrm{d}x = \int_\Omega f \phi_i \,\mathrm{d}x$$

From discrete variational problem (iii) to discrete system of equations (iv), cont'd.

Rearrange to get:

$$\sum_{j=1}^{N} U_j \underbrace{\int_\Omega \nabla \phi_j \cdot \nabla \phi_i \,\mathrm{d}x}_{A_{ij}} = \underbrace{\int_\Omega f \phi_i \,\mathrm{d}x}_{b_i}$$

A linear system of equations:

$$AU = b$$

where

$$A_{ij} = \int_\Omega \nabla \phi_j \cdot \nabla \phi_i \,\mathrm{d}x \qquad (1)$$

$$b_i = \int_\Omega f \phi_i \,\mathrm{d}x \qquad (2)$$

The canonical abstract problem

(i) Partial differential equation:

$$\mathcal{A}u = f \quad \text{in } \Omega$$

(ii) Continuous variational problem: find $u \in V$ such that

$$a(u, v) = L(v) \quad \text{for all } v \in \hat{V}$$

(integrate by parts and employ boundary conditions for trial or test functions)

(iii) Discrete variational problem: find $u_h \in V_h \subset V$ such that

$$a(u_h, v) = L(v) \quad \text{for all } v \in \hat{V}_h$$

(choose an appropriate subspace)

(iv) Discrete system of equations for $u_h = \sum_{j=1}^{N} U_j \phi_j$:

$$AU = b$$
$$A_{ij} = a(\phi_j, \phi_i),\ b_i = L(\phi_i)$$

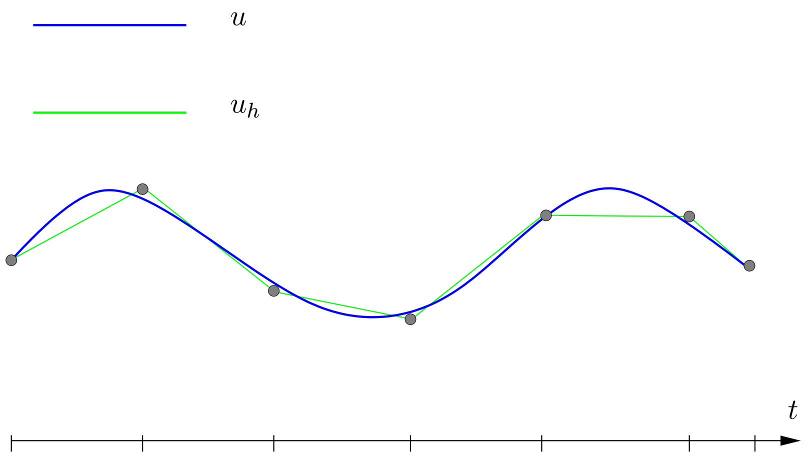
(choose a concrete basis for the appropriate subspace)

Important topics

- *How to choose* V_h*?*
- *How to compute* A *and* b
- *How to solve* $AU = b$*?*
- *Can we quantify/control How large the error* $e = u - u_h$ *is?*
- *Can we assess the cost of solving the system?*
- Extensions to nonlinear, time-dependent, complicated problems

How to choose V_h

Finite element function spaces



The finite element definition (Ciarlet 1975)

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 $d = 1, 2, 3, \ldots$) 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) $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_n\}$ is a basis for the dual space $\mathcal{V}'$; that is, the space of bounded linear functionals on $\mathcal{V}$

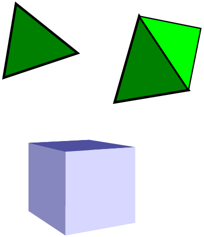
The finite element definition is kind of abstract

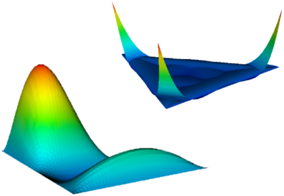
A finite element is a triple $(T, \mathcal{V}, \mathcal{L})$, is used as follows

- the domain T is used to divide the mesh into subdomains represented by T
- the space $\mathcal{V}$ is used to evaluate the variational forms locally for each subdomain T
- the set of degrees of freedom $\mathcal{L}$ is used to glue together the localized function space ($\mathcal{V}$) to a global function space using the degrees of freedom

The finite element definition (Ciarlet 1975)

T $\quad$ $\mathcal{V}$ $\quad$ $\mathcal{L}$





$v(\bar{x})$

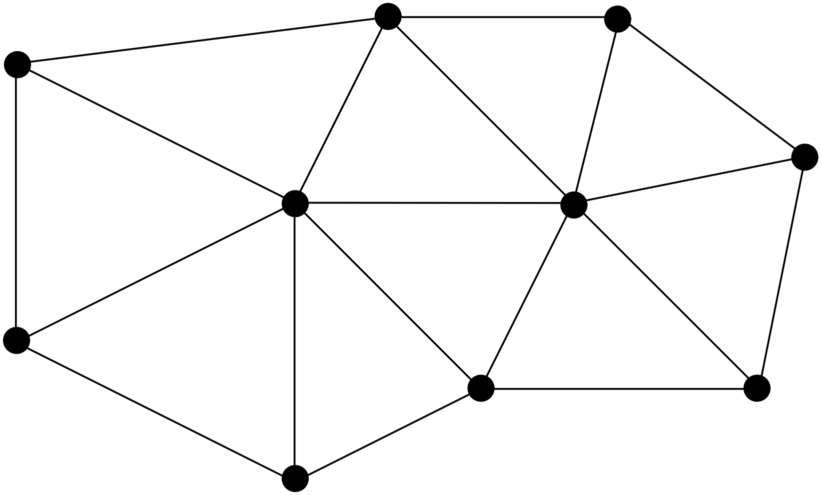
$v(\bar{x}) \cdot n$

$\int_T v(x)w(x)\,\mathrm{d}x$

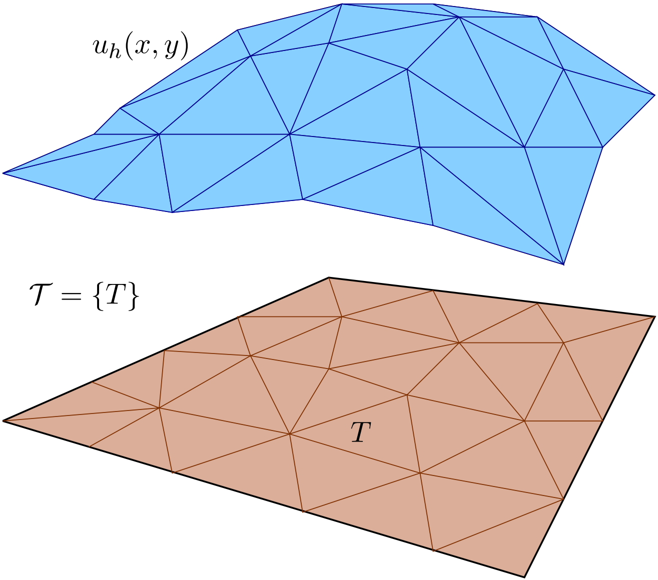
The linear Lagrange element: $(T, \mathcal{V}, \mathcal{L})$

- T is a line, triangle or tetrahedron
- $\mathcal{V}$ is the first-degree polynomials on T
- $\mathcal{L}$ is point evaluation at the vertices

The linear Lagrange element: $\mathcal{L}$



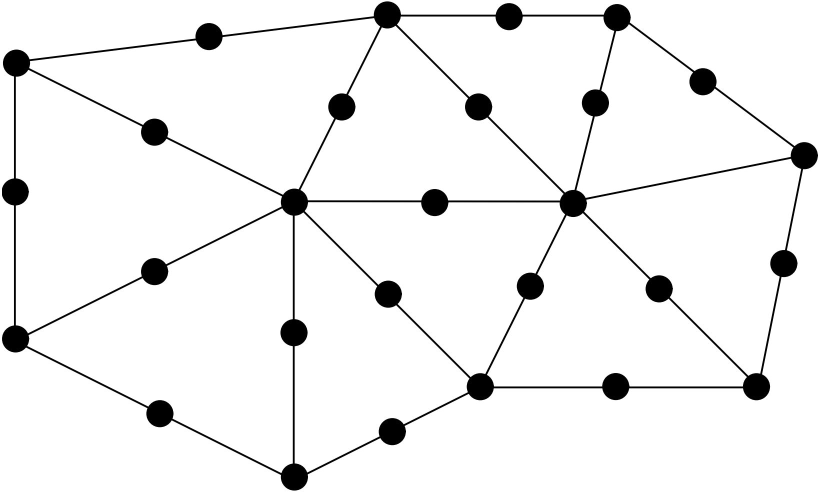
The linear Lagrange element: V_h



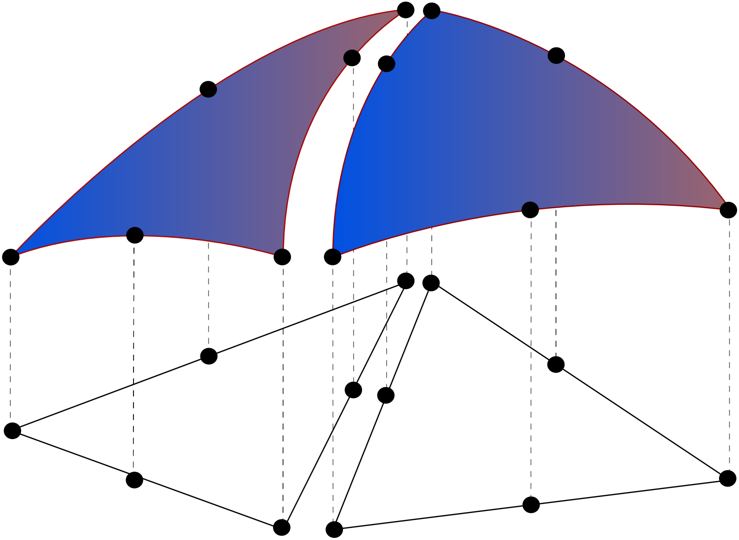
The quadratic Lagrange element: $(T, \mathcal{V}, \mathcal{L})$

- T is a line, triangle or tetrahedron
- $\mathcal{V}$ is the second-degree polynomials on T
- $\mathcal{L}$ is point evaluation at the vertices and edge midpoints

The quadratic Lagrange element: $\mathcal{L}$

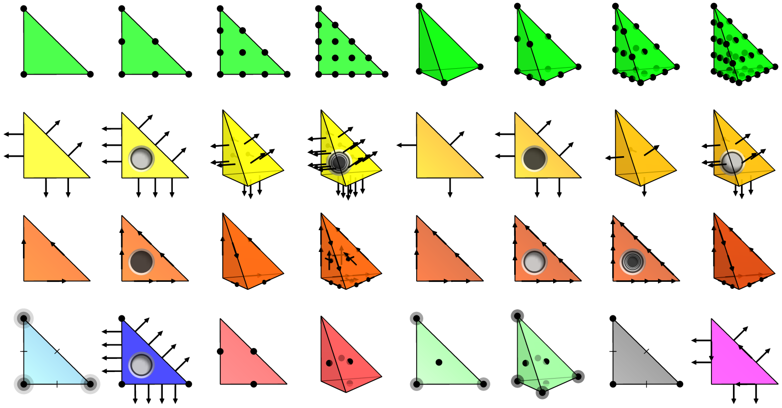


The quadratic Lagrange element: V_h



Families of elements

Families of elements



Computing the sparse matrix A

Why is the matrix sparse?

Naive assembly algorithm

$A = 0$

for $i = 1, \ldots, N$

    for $j = 1, \ldots, N$

        $A_{ij} = a(\phi_j, \phi_i)$

    end for

end for

The element matrix

The global matrix A is defined by

$$A_{ij} = a(\phi_j, \phi_i)$$

The *element matrix* A_T is defined by

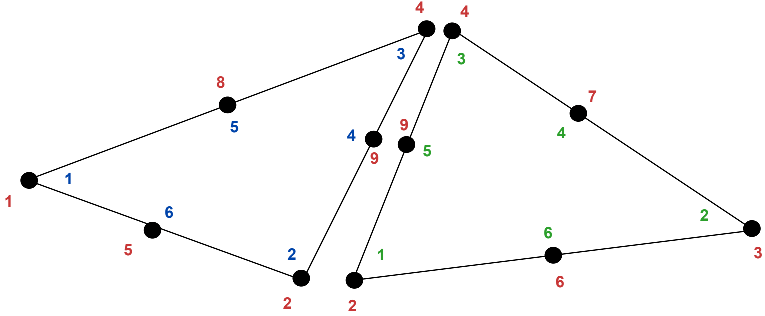
$$A_{T,ij} = a_T(\phi_j^T, \phi_i^T)$$

The local-to-global mapping

The global matrix ι_T is defined by

$$I = \iota_T(i)$$

where I is the *global index* corresponding to the *local index* i



The assembly algorithm

$A = 0$

for $T \in \mathcal{T}$

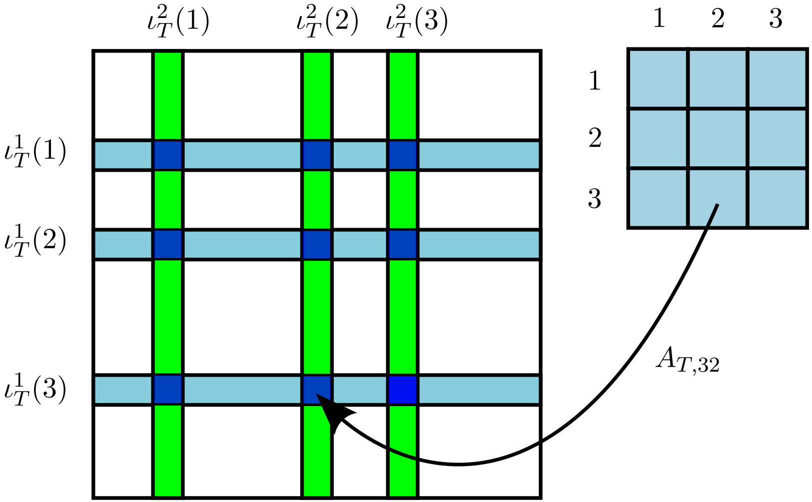
Compute the element matrix A_T

Compute the local-to-global mapping ι_T

Add A_T to A according to ι_T

end for

Adding the element matrix A_T



Solving $AU = b$

Direct methods

- Gaussian elimination
  - Requires $\sim \frac{2}{3}N^3$ operations
- LU factorization: $A = LU$
  - Solve requires $\sim \frac{2}{3}N^3$ operations
  - Reuse L and U for repeated solves
- Cholesky factorization: $A = LL^\top$
  - Works if A is symmetric and positive definite
  - Solve requires $\sim \frac{1}{3}N^3$ operations
  - Reuse L for repeated solves

Iterative methods

Krylov subspace methods

- GMRES (Generalized Minimal RESidual method)
- CG (Conjugate Gradient method)
  - Works if A is symmetric and positive definite
- BiCGSTAB, MINRES, TFQMR, …

Multigrid methods

- GMG (Geometric MultiGrid)
- AMG (Algebraic MultiGrid)

Preconditioners

- ILU, ICC, SOR, AMG, Jacobi, block-Jacobi, additive Schwarz, …

Which method should I use?

Rules of thumb

- Direct methods for small systems
- Iterative methods for large systems
- Break-even at ca 100–1000 degrees of freedom
- Use a symmetric method for a symmetric system
    - Cholesky factorization (direct)
    - CG (iterative)
- Use a multigrid preconditioner for Poisson-like systems
- GMRES with ILU preconditioning is a good default choice

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)
```

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 Ω:

```
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)
```

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 `u0` 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: