Lab module 3: The Finite Element Method for Partial Differential Equations - assembling linear systems

Johan Jansson (jjan@kth.se), Tania Bakhos, Massimiliano Leoni

February 27, 2017

0 Jupyter-FEniCS web PDE solver environment

The address of the web Jupyter-FEniCS cloud environment, described more in detail below, is provided via email with the ip of the cloud virtual machine and Jupyter login. To run a program in Jupyter-FEniCS, open the Python 2 notebook and select the Run command under the Cell menu.

You can find example Jupyter notebooks with implementations from the lab under the `src` directory in the zip archive of the lab module, which you can upload in the Jupyter interface.

NB!: The files in the web environment are **not** stored on disk, which means they will disappear if the system is rebooted. Do not forget to save your notebook regularly to your own computer, by using Download as > IPython Notebook (.ipynb) under File in your notebook.

You can also set up the environment at your own computer by using the command:

```
sudo docker run -t -i -p 80:8000 jjan/fenics-mooc:test
```

and using the login name and passwords listed on the terminal window by accessing localhost from a web browser.

1 Introduction

In this lab session you will learn about the underlying concepts of the FEniCS [2] framework for automated solution of partial differential equations (PDE), and learn how to implement parts of the algorithms yourself.

Specifically you will investigate general assembly of linear systems, a key algorithm in FEM, and the basis for the FEniCS framework.

The goal of this session is to:

1. Learn the basic components of FEM: piecewise linear approximation and weak form of PDE.
2. Learn the basic interface of FEniCS: form language and function, mesh and solve interfaces.
3. Become familier with the Jupyter web Python notebook interface, through which FEniCS will be run in this course.
4. Understand the general algorithm and implementation for finite element assembly of linear systems

In this session we will work with the Python interface to FEniCS. We will use FEniCS version 1.6 which is installed in the Jupyter notebook (`http://jupyter.org`) Python web environment provided at the link at the top of the instructions. On the FEniCS home page [2] there is extensive documentation of the interface at both overview and detail level.

For reference material, please see the lecture notes from the DD2263 Methods in Scientific Computing course at KTH [3] and the book Computational Differential Equations [1].

2 Exercises

2.1 FEM for PDE

We start with the simplest equation: the L_2-projection, which computes the optimal projection of a function f into a finite element space V_h. To compute the L_2-projection we want to solve the equation, or "weak form":

$$r(u, v) = (u, v) - (f, v) = 0 \tag{1}$$

where u is the unknown solution function and f is a known function. v is a "test function", we will specify it below, and with the standard notation for the L_2 inner product of functions: $(v, w) = \int_\Omega vw dx$.

The finite element method (FEM) means that we require the weak form to be true for all basis functions $\phi_j(x)$ of a finite element vector space V_h, and also requiring the unkown function $U(x)$ to be a linear combination of the basis vectors of $\phi_j(x)$:

$$r(U, v) = (U, v) - (f, v) = 0, \quad U(x) = \sum_{i=0}^{N} \xi_i \phi_j(x), \quad \forall v \in \{\phi_0, ..., \phi_N\} \tag{2}$$

where ξ_i are the coefficients determining the function.

Writing the equation like this and plugging in the definitions of U and v we get:

$$(U, v) = (f, v), \quad U(x) = \sum_{i=0}^{N} \xi_i \phi_j(x), \quad \forall v \in \{\phi_0, ..., \phi_N\} \Rightarrow \tag{3}$$

$$\sum_{i=0}^{N} \int_\Omega \phi_j(x)\phi_i(x) dx \ \xi_i = \int_\Omega f \phi_i(x) dx, \quad j = 0, ..., N \tag{4}$$

Which is an $N \times N$ linear system of the form $A\xi = b$, i.e. we have N unkown coefficients for $U(x)$ representing a row and we have N rows for the N test functions.

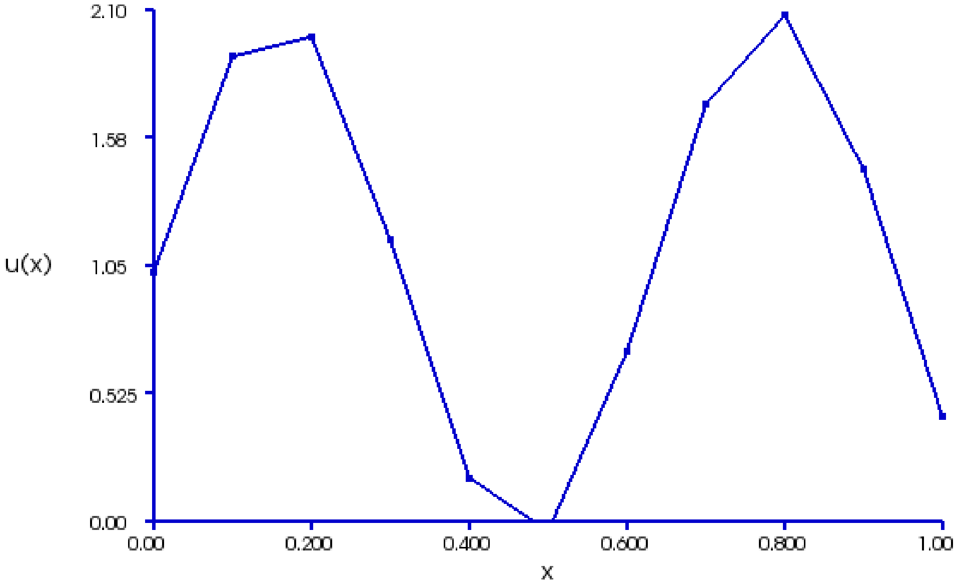
The standard choice of basis functions is the continous piecewise linear nodal basis, which thus means that $U(x)$ is a piecewise linear function. The basis functions are constructued on a mesh $\mathcal{T}$ (triangulation in 2D) with cells K (triangles in 2D), cell diameter h and nodes x_i where the coefficients ξ_i are defined. The basis functions are defined by:

$$\phi_j(x) = \begin{cases} 1, & x = x_i \\ 0, & x \neq x_i \end{cases} \tag{5}$$

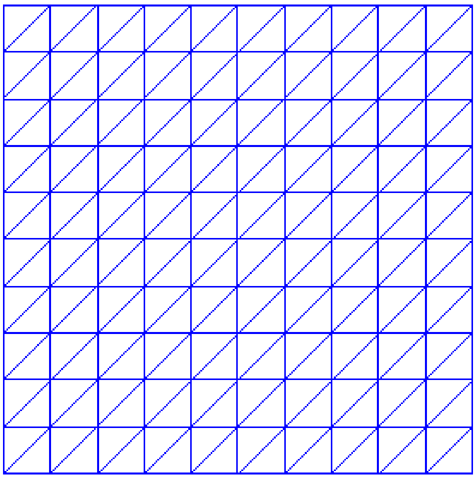
We then also have that $U(x_i) = \xi_i$, i.e. that the value of the function $U(x)$ is the same as the value of coefficient in the node x_i.

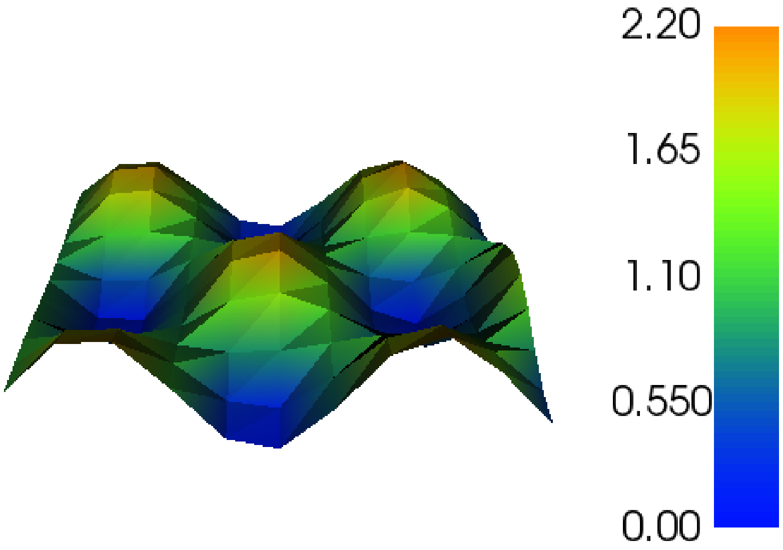
Some examples of piecewise linear functions in the function space V_h are shown below.

Here a 1D mesh of the interval $[0, 1]$ is chosen with $h = 0.1$, and a piecewise linear function approximating $1 + sin(10x) * cos(7y)$ is plotted:

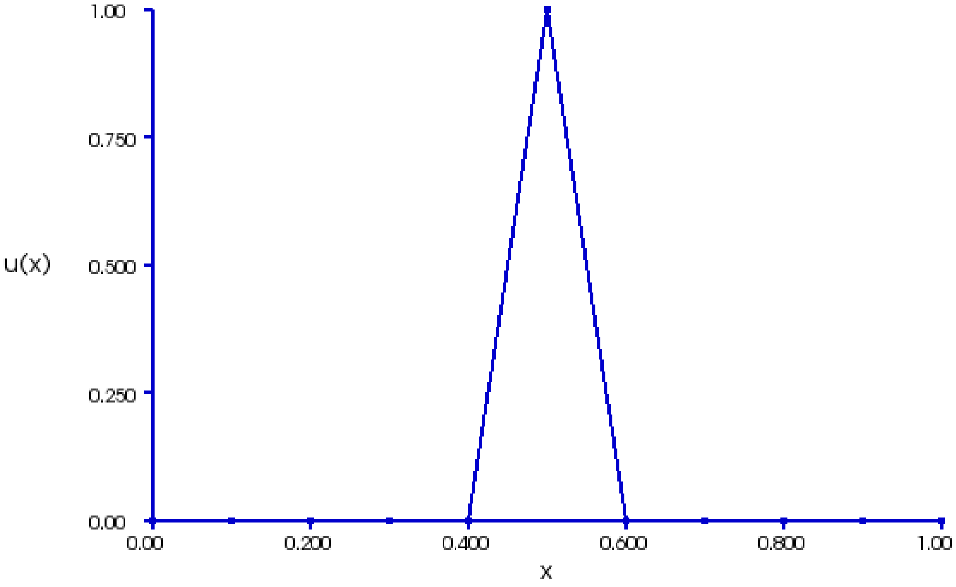


We now plot a 2D version of the above, now on the domain $[0,1] \times [0,1]$ with $h = 0.1$, and the same piecewise linear function approximating $1 + sin(10x)$, we also plot the mesh:

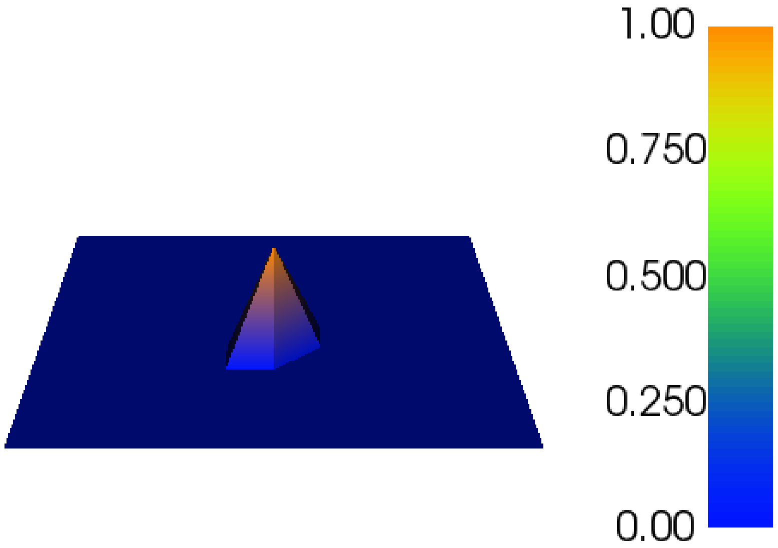




A basis function in 1D looks as follows:



And a basis function in 2D



If we replace the finite dimensional space V_h with the space V of all integrable functions (with bounded integral), the "strong" equation below is equivalent:

$$R(u) = u - f = 0 \tag{6}$$

which has the trivial solution $u = f$. In all the exercises in the lab we will use the weak form of the presented equations. The strong form can be derived as above if desired.

2.2 FEniCS interface

We first include the plotting interface for Jupyter:

```
%matplotlib inline
%run /home/fenics/fenics-matplotlib.py
```

We can then define the representation of the equation in FEniCS as:

```
from dolfin import *

mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
f = Expression("1. + sin(10*x[0])*cos(7*x[1])", degree=3)
v = TestFunction(V)
u = Function(V) # FEM solution

r = (u - f)*v*dx
```

To solve the equation we use the compact `solve()` notation:

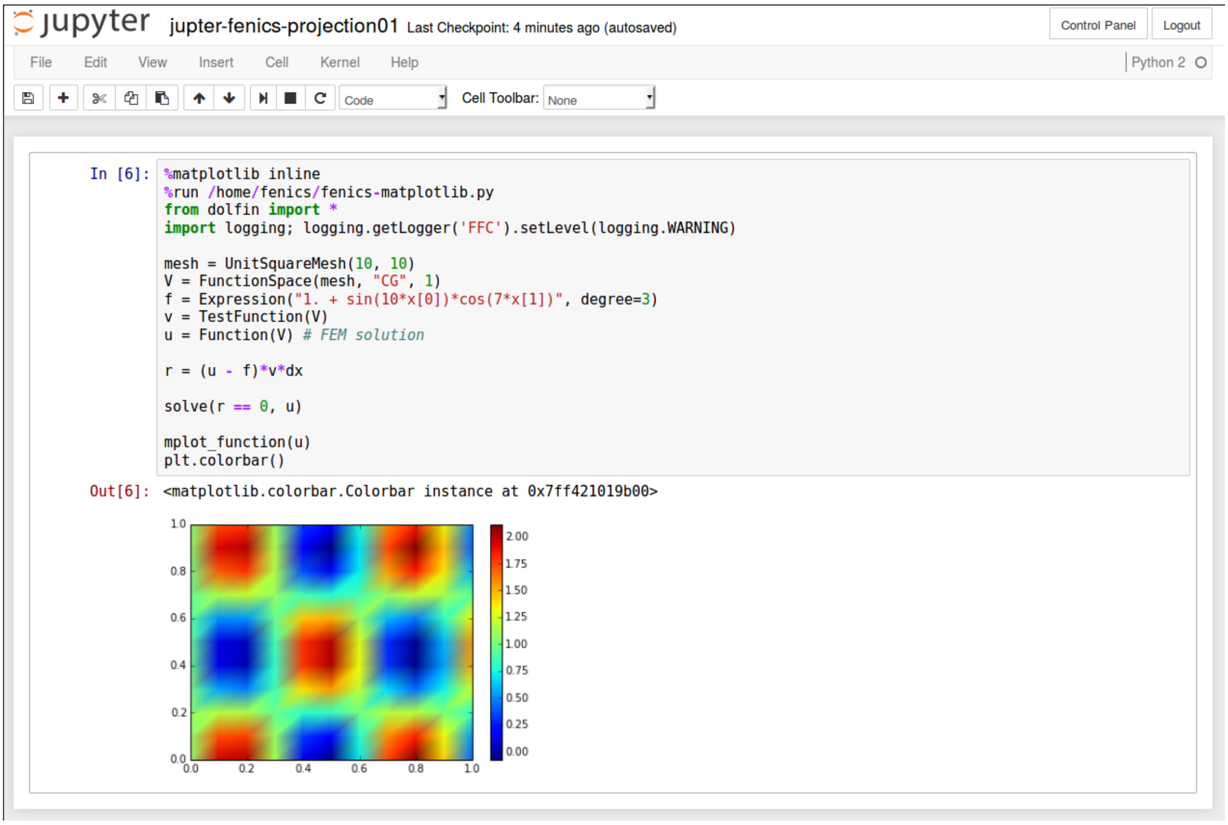
```
solve(r == 0, u)
```

and we can plot in the Jupyter environment with the provided `mplot_function()` function:

```
mplot_function(u)
```

2.2.1 Screenshot

The template program (here in compact format) and the expected output when running it should look like this:



```
%matplotlib inline
%run /home/fenics/fenics-matplotlib.py
from dolfin import *
import logging; logging.getLogger('FFC').setLevel(logging.WARNING)

mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
f = Expression("1. + sin(10*x[0])*cos(7*x[1])", degree=3)
v = TestFunction(V)
u = Function(V) # FEM solution

r = (u - f)*v*dx

solve(r == 0, u)

mplot_function(u)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar instance at 0x7ff421019b00>
```

FEniCS can also output solution functions in the VTK format, that for example ParaView can then visualize:

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

2.2.2 Questions

First try running the above given program, then try different things, for example:

a) Add a diffusion term: $\nu(\nabla u, \nabla v)$, written as `nu * inner(grad(u), grad(v))*dx` in FEniCS, to the weak form. What happens when you vary nu?

b) The coefficient vector ξ can be accessed by `u.vector()`. Compute the max-norm of the solution, this can be done by: `u.vector().norm('linf')`.

c) Try plotting a basis function yourself. Set the coefficient vector to 0 by: `u.vector()[:] = 0.0`, where : means all indices. Then choose a specific index and set its value to one, and plot the corresponding basis function of the index.

2.3 General algorithm for finite element assembly of linear systems

We saw in the introduction that applying FEM to a linear PDE generates a linear system for the unkown coefficients ξ describing the finite element solution function $U(x) = \sum_{i=0}^{N} \xi_i \phi_j(x)$, where we recall that $\phi(x)$ are the basis functions.

Standard notation is to decompose the weak residual $r(u, v)$ into a *bilinear* part $a(u, v)$ which includes all the terms with the unknown u, and a *linear* term $L(v)$, with all the known source terms. We then have $r(u, v) = a(u, v) - L(v)$.

Using this notation, we can now write out the general relations underlying the general algorithm for finite element assembly of linear systems:

In general, the matrix A_h, representing a bilinear form $a(u, v)$ is given by:

$$(A_h)_{ij} = a(\varphi_j, \hat{\varphi}_i).$$

and the vector b_h representing a linear form $L(v)$ is given by:

$$(b_h)_i = L(\hat{\varphi}_i).$$

In other words, by plugging in the basis functions with index i and j into the bilinear form $a(u, v)$, we can compute matrix element $(A_h)_{ij}$. As an example, in the L2-projection above the bilinear form is $a(u, v) = (u, v) = \int_\Omega uvdx$. Matrix element $(A_h)_{ij}$ is then simply computed by the integral: $(A_h)_{ij} = (\phi_j, \phi_i) = \int_\Omega \phi_j \phi_i dx$.

We are now ready to formulate the algorithms for assembling a matrix and vector given a weak form.

2.3.1 Matrix assembly algorithm

```
for all cells $K \in \mathcal{T}$
    for all test functions $\phi_i$ on $K$
        for all trial functions $\phi_j$ on $K$
            1. Compute $I = a(\phi_j, \phi_i)_K$
            2. Add $I$ to $(A_h)_{ij}$
        end
    end
end
```

2.3.2 Vector assembly algorithm

```
for all cells $K \in \mathcal{T}$
    for all test functions $\phi_i$ on $K$
            1. Compute $I = L(\phi_i)_K$
            2. Add $I$ to $(A_h)_{ij}$
    end
end
```

2.3.3 Python implementation of the general assembly algorithm

In FEniCS this algorithm is available as the `assemble()` function, where a typical use can be: `A = assemble(a)` with `A` a FEniCS matrix and `a` a bilinear form.

In this module we will look at a minimal, simple implementation of the `assemble()` function in 2D, we call our implementation `myassemble()`. It is available under `src/myassemble_numpy.ipynb` and can be uploaded and run in the Jupyter environment.

The implementation is based on computing the integrals on a *reference cell*, here the triangle with vertices $\{(0, 0), (1, 0), (0, 1)\}$, where the basis functions are easily defined. The integrals are then transformed through a coordinate map to the physical cell in the triangulation.

The `myassemble()` implementation is available in the file `src/myassemble_numpy`, and a verification test, comparing the assembled matrix to the one assembled by FEniCS, is available in the file `src/myassemble_numpy_test.py`.

2.3.4 Questions

Upload the given Python files to the Jupyter system to carry out the programming questions below.

a) With pen and paper, compute the integral $(\nabla\phi_0, \nabla\phi_0)$ on the reference cell defined above.

b) Why is it enough to only test against the functions $\phi_i \in V_h$ and not against all functions $v \in V_h$?

c) In `myassemble()`, the weak form $a(u, v) = (\nabla u, \nabla v)$ is assembled. Modify the implementation to instead assemble $a(u, v) = (\nabla u, \nabla v) + (u, v)$. Verify the assembled matrix against the FEniCS `assemble()` function using `src/myassemble_numpy_test.py` . Compute the max norm of the difference of the matrices.

d) Write out the quadrature rule (numerical integration method) used in `myassemble()` in mathematical notation.

References

[1] Kenneth Eriksson, Don Estep, Peter Hansbo, and Claes Johnson. *Computational Differential Equations.* Cambridge University Press New York, 1996.

[2] FEniCS. Fenics project. *http://www.fenicsproject.org*, 2003.

[3] Johan Hoffman. Lecture notes for dd2263 methods in scientific computing, 2017.