CHALMERS
UNIVERSITY OF TECHNOLOGY

Introduction to Python and FEniCS

Eleni Gerolymatou
Geotechnical Research Group

Introduction

- A small repetition on finite elements.
- Some basics in python.
- Getting started in FEniCS.
- Some examples.

More information on fenicsproject.org
and
https://fenicsproject.org/pub/course/lectures/2017-nordic-phdcourse/

Finite Element Method: you all know this, but...

- this is a quick reminder for:
    - problem position
    - strong and weak form
    - test and trial functions
    - discretization
- and we will just be using an example

The Poisson equation

$$-\nabla^2 u(\mathbf{x}) = f(\mathbf{x})$$

- this is the Poisson equation
- ∇^2 is the Laplace operator
- f is a known function
- and u is the unknown function

The following are needed:

- the equation
- a spatial domain
- a boundary conditiom

Problem position – in the strong form

$$-\nabla^2 u(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \text{ in } \Omega,$$

$$u(\mathbf{x}) = u_b(\mathbf{x}), \quad \mathbf{x} \text{ on } \partial\Omega,$$

- this is the Poisson equation
- ∇^2 is the Laplace operator
- f is a known function
- u is the unknown function
- u_b is the value of u on the boundary
- Ω is the domain where the solution is sought
- $\partial\Omega$ is the boundary of the domain

Deriving weak form

$$-\int_{\Omega} \left(\nabla^2 u(\mathbf{x})\right) v(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{\Omega} f(\mathbf{x})v(\mathbf{x})\mathrm{d}\mathbf{x}$$

- multiply both sides by the function v and integrate over the whole domain Ω
- The function v can be any function and is often referred to as a *test* function

- If a function *u* satisfies the above equation and the boundary conditions for <u>any</u> function *v*, then *u* is the (or at least a) solution.

Deriving the weak form

- Integrating by parts and making use of the Green theorem yields

$$-\int_{\Omega}\left(\nabla^2 u(\mathbf{x})\right) v(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{\Omega} f(\mathbf{x})v(\mathbf{x})\mathrm{d}\mathbf{x} \Rightarrow$$

$$\int_{\Omega}\left(\nabla u(\mathbf{x})\right)\left(\nabla v(\mathbf{x})\right)\mathrm{d}\mathbf{x} - \int_{\partial\Omega}\mathbf{n}\cdot\nabla u(\mathbf{x})v(\mathbf{x})\mathrm{d}\mathbf{s} = \int_{\Omega} f(\mathbf{x})v(\mathbf{x})\mathrm{d}\mathbf{x}$$

- where n is the outward normal unit vector to the boundary. Demanding that

$$v(\mathbf{x}) = 0, \quad \partial\Omega$$

- means

$$\int_{\Omega}\nabla u(\mathbf{x})\nabla v(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{\Omega} f(\mathbf{x})v(\mathbf{x})\mathrm{d}\mathbf{x}$$

The weak form

$$\int_\Omega \nabla u(\mathbf{x}) \nabla v(\mathbf{x}) \mathrm{d}\mathbf{x} = \int_\Omega f(\mathbf{x}) v(\mathbf{x}) \mathrm{d}\mathbf{x}, \quad \forall v \in V$$

- This is the *weak* or *variational* or *integral* form.
- It is called weak because it is less restrictive to continuity than the strong form.
- V is called the space of test functions
- U is the called the space of trial functions

Test and Trial functions

The functions u and v should fulfill certain prerequisites in terms of continuity and integrability:

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

$$U = \left\{ v \in H^1(\Omega) : v = u_b \text{ on } \partial\Omega \right\}$$

where H^1 is the Sobolev space containing functions u such that u^2 and $|\nabla u|^2$ have finite integrals over Ω.

Discretization

The variational problem is a continuous problem.

The finite element method finds an approximate solution of the variational problem by replacing the infinite-dimensional function spaces V and U by discrete (finite-dimensional) trial and test spaces:

$$V_h \subset V = \left\{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\right\}$$
$$U_h \subset U = \left\{v \in H^1(\Omega) : v = u_b \text{ on } \partial\Omega\right\}$$

where the boundary conditions are part of the function space definitions.

$$\int_\Omega \nabla u_h(\mathbf{x}) \nabla v(\mathbf{x}) \mathrm{d}\mathbf{x} = \int_\Omega f(\mathbf{x}) v(\mathbf{x}) \mathrm{d}\mathbf{x}, \quad \forall v \in V_h$$

The variational problem

Find $u \in U$ such that:

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

where the test and trial function space definitions are

$$V_h \subset V = \left\{ v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega \right\}$$

$$U_h \subset U = \left\{ v \in H^1(\Omega) : v = u_b \text{ on } \partial\Omega \right\}$$

To a discrete system of equations

Choose a basis for the discrete function space:

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

Make an ansatz for the discrete solution:

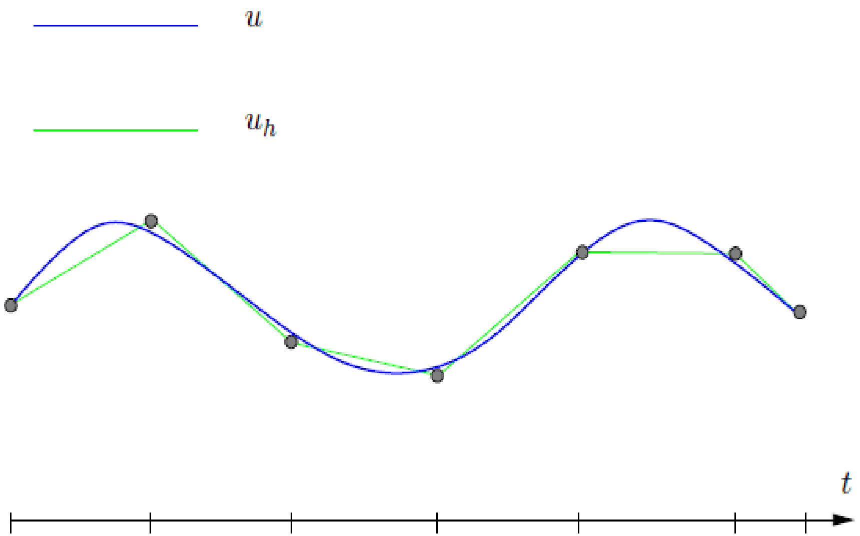
$$u_h = \sum_{j=1}^{N} U_j \phi_j$$

Test against the basis functions:

$$\int_\Omega \nabla \left(\sum_{j=1}^{N} U_j \phi_j \right) \cdot \nabla \phi_i \mathrm{d}x = \int_\Omega f \phi_i \mathrm{d}x$$

So what should be done?

- FEniCS takes care of the integration



- Define element geometry
- Define element order
- Provide the boundary conditions
- Provide the weak form

Python

- General purpose and easy
- Slow

Let‘s try this:

- Computing the sum of the integers from 1 to 100:

```
s = 0
for i in range (1,101):
      s+=i

print s
```

Python

- Summing from 1 to 100 millions:
- time python 001.py

```
5000000050000000

real    0m10.572s
user    0m7.234s
sys     0m3.109s
```

- Summing from 1 to 100 millions with c++:
- g++ -o 001c 001.cpp
- time ./001c

```
sum = 5000000050000000

real    0m0.250s
user    0m0.234s
sys     0m0.016s
```

Python program structure

```
import stuff

def some_function ( argument ):
    " Function documentation "
    return something

# This is a comment
if _name_ == " _main_ ":
    do_something
```

Python declaring variables

```
a = 5
b = 3.5
c = ”hi”
d = ’hi’
e = True
f = False
```

Python - Comparison

x == y

x != y

x > y

x < y

x >= y

x <= y

Python – Logical operators

```
not x
x and y
x or y
```

Python – If construct

```
if x > y
	x += y
elif x < y
	y += x
else:
	x += 1
```

Python – For construct

```
for variable in enumerable :
    stuff


for i in range (100 ):
    stuff
    morestuff
```

Python – While construct

```
while condition :
    stuff
i = 0
while i < 100:
    stuff
    i++
i = 0
while True :
    stuff
    if i == 99:
    break
```

Python - Functions

```
def myfunction (arg0 , arg1 , ...) :
        stuff
        ...
        return something # or not , gives None

def res(x,y):
    res = x*y+y
    return res
```

Python - Classes

```
class Foo:
    def __init__ (self , argument ):
        stuff

    def foo( self ):
        stuff
        return something

    def bar( self ):
        stuff
        return something
```

Calling it :

```
f = Foo( argument )
f.foo ()
f.bar ()
```

Python – More on classes

```
class Foo:
    def __init__ (self , argument ):
        self.x = 3          # this is a public member variable
        self.__x = 3        # this is a private member variable


    def foo( self ):        # this is a member function
        stuff
        return something
```

FEniCS - Installation

- On Ubuntu – using PPA
- Using Docker containers
- Using Anaconda – Linux and Mac only
- On Windows Subsystem – as in Ubuntu
- From source

FEniCS – What is it?

- It‘s a C++ \ Python library
- It‘s licensed under the GNU LGPL
- It‘s designed to automate the solution of PDEs
  - generation of basis functions
  - evaluation of variational forms
  - finite element assembly
  - error control
- It‘s designed for parallel execution

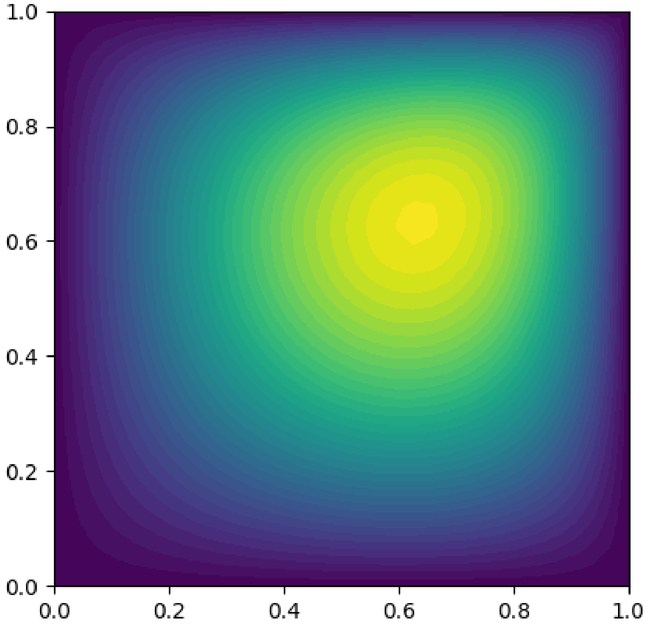
FEniCS - Documentation

- fenics.readthedocs.org
- https://fenicsproject.org/tutorial/
  - access to the book
  - several examples
- https://www.allanswered.com/community/s/fenics-project/
  - help from the community

Is it working?

- Try: python -c ’import fenics’
  - if all is well, you should get no message

- Try: python 002.py
  - if all is well, you should get the same result



FEniCS

Solving a boundary-value problem such as the Poisson equation in FEniCS consists of the following steps:

- 1. Identify the computational domain (Ω), the PDE, its boundary conditions, and source terms (f).
- 2. Reformulate the PDE as a finite element variational problem.
- 3. Write a Python program which defines the computational domain, the variational problem, the boundary conditions, and source terms, using the corresponding FEniCS abstractions.
- 4. Call FEniCS to solve the boundary-value problem and, optionally, extend the program to compute derived quantities such as fluxes and averages, and visualize the results.

A walk through 002.py

`from fenics import *`	imports the key classes from the FEniCS library
`import matplotlib.pyplot as plt`	imports plotting functionalities
`mesh = UnitSquareMesh(32 , 32)`	defines a uniform mesh over the unit square
`V = FunctionSpace(mesh , "Lagrange", 1)`	defining the finite element function space
`u = TrialFunction(V)`	defining the trial functions
`v = TestFunction(V)`	defining the test functions
`f = Expression ("x[0]*x[1]", degree =2)`	defining the expression on the right hand side

A walk through 002.py

`a = dot( grad(u), grad(v))*dx`	Defining the bilinear form
`L = f*v*dx`	Defining the linear form
`bc = DirichletBC (V, 0.0, DomainBoundary ())`	Defining the boundary condition
`u0= Function (V)`	Defining the solution function
`solve (a == L, u0, bc)`	Solving
`p=plot(u0)`	Plotting the results
`plt.show()`	

The effect of the order

Try 003.py

What is the program doing?

Which is the correct approach?

A first program

Modify 004.py to solve the problem:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 2\left(1 + 2x^2 + 2y^2\right)$$

$$x = 0: \quad u = y^2$$

$$y = 0: \quad u = 0$$

$$x = 1: \quad u = 2y^2$$

$$y = 1: \quad u = 1 + 2x^2$$

The analytical solution is $u = y^2(1 + 2x^2)$

What is the effect of order and discretization?

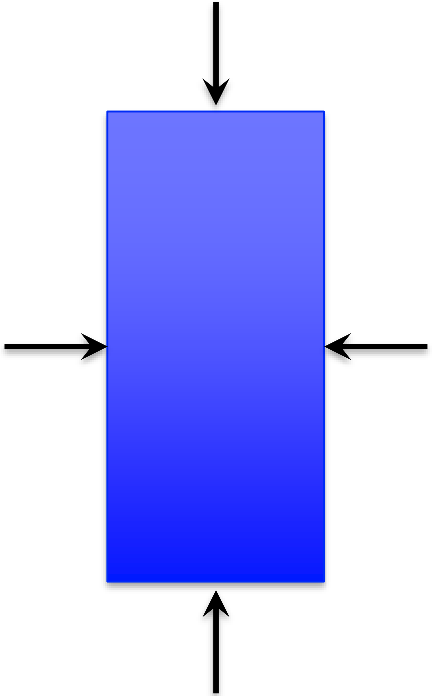
Elasticity

Consider the biaxial problem on the right.

The material is isotropically elastic with E=10 Mpa, v=0.2

The displacements at the boundary are controlled.

Evaluate the stress, strain and stored energy for this problem.



Elasticity

The balance equations in strong form read

$$\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} = f_x$$

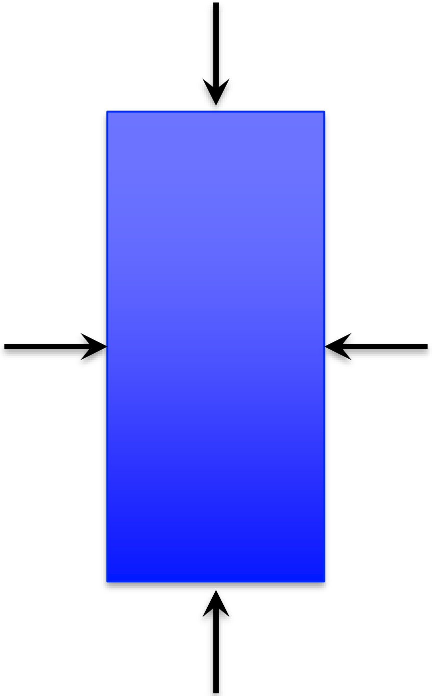
$$\frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} = f_y$$

where f expresses the body forces.

Alternatively we can write

$$\nabla . \underline{\boldsymbol{\sigma}} = \mathbf{f}$$

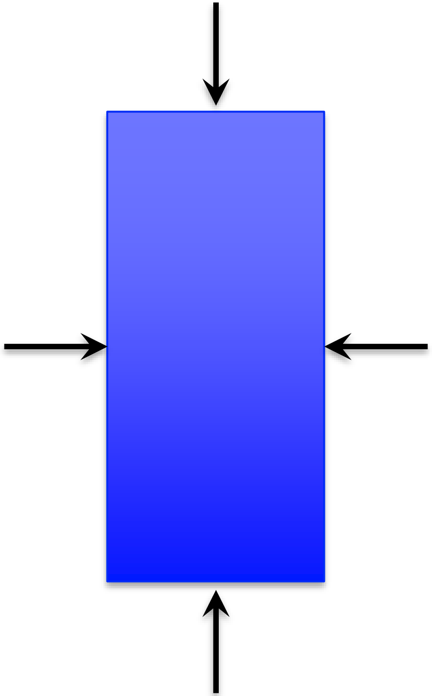
For implementation the weak form is needed.



Elasticity

$$\nabla . \underline{\boldsymbol{\sigma}} = \mathbf{f} \Rightarrow \int_{\Omega} \nabla . \underline{\boldsymbol{\sigma}} v \mathrm{d}\omega = \int_{\Omega} \mathbf{f} v \mathrm{d}\omega \Rightarrow$$

$$- \int_{\Omega} \underline{\boldsymbol{\sigma}} \cdot \nabla v \mathrm{d}\omega = \int_{\Omega} \mathbf{f} v \mathrm{d}\omega \Rightarrow \boxed{\int_{\Omega} \underline{\boldsymbol{\sigma}} \cdot \nabla v \mathrm{d}\omega = - \int_{\Omega} \mathbf{f} v \mathrm{d}\omega}$$



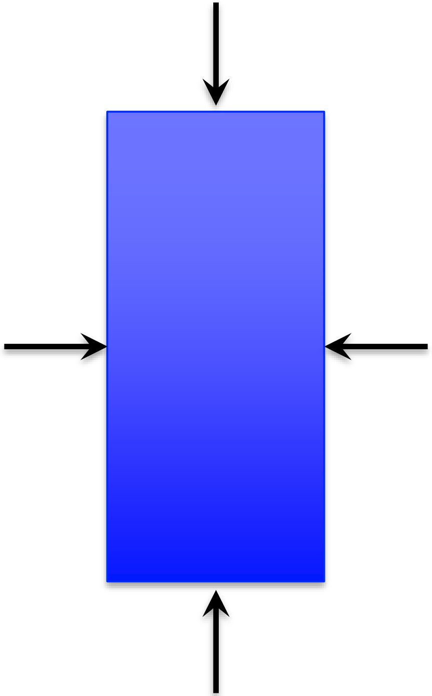
and

$$\underline{\boldsymbol{\sigma}} = \underline{\underline{\mathbf{D}}} \cdot \underline{\boldsymbol{\epsilon}}$$

Elasticity – let's walk it through

- We have a rectangular mesh

```
# Geometry
Lx = 0.1     # width
Ly = 0.2     # height
nx = 5              # number of elements in the x-direction
ny = 10             # number of elements in the y-direction
# Preparing the mesh
mesh = RectangleMesh(Point(0,0), Point(Lx,Ly), nx, ny)
```



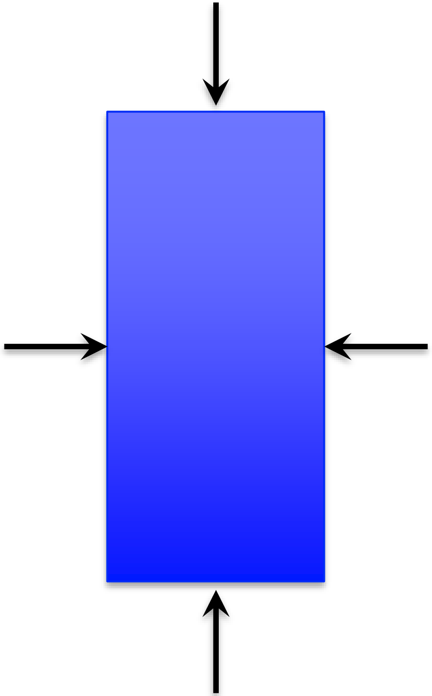
Elasticity – let’s walk it through

- We use a vector function space

```
V = VectorFunctionSpace(mesh, 'Lagrange', 2)
```

- and only constrain one component in the BC

```
# Define Dirichlet boundary conditions
# Lower boundary, zero vertical displacement
tol = 1E-14
def lower_boundary(x, on_boundary):
    return on_boundary and x[1] < tol
bcd = DirichletBC(V.sub(1), Constant( 0.001), lower_boundary)
```



Elasticity – let’s walk it through

- We also introduce the strain, the stress, the stiffness

```
# Strain
def epsilon(u):
   e =-0.5*(nabla_grad(u) + nabla_grad(u).T)
return as_tensor([[e[0, 0], e[0, 1]],
                  [e[1, 0], e[1, 1]]])
# Stress tensor
def sigma(u):
   eps=epsilon(u)
   stiffness =dsde()
   sigma = as_tensor(stiffness[i,j,k,l]*eps[k,l],(i,j))
   return as_tensor(sigma)
```

```
# Stiffness tensor
def dsde():
   stiffness = np.zeros((d,d,d,d))
   for i in range(0,d):
      for j in range(0,d):
         stiffness[i,i,j,j] +=lamda
         stiffness[i,j,i,j] +=mu
         stiffness[i,j,j,i] +=mu
   return as_tensor(stiffness)
```

Elasticity – let’s walk it through

- Change the discretization and the order of the elements. Does it make a difference?
- Change the boundary condition. What happens when the lower boundary is clamped?
- What happens when you add body forces?
- Can you assess the magnitude of the error?

Neumann conditions

- Save the same file as 006.py
- Declare the position of the boundaries

```
boundary_parts = MeshFunction("size_t", mesh, mesh.topology().dim() - 1)
left_boundary.mark(boundary_parts, 1)
right_boundary.mark(boundary_parts, 2)
ds = ds(subdomain_data = boundary_parts)
```

- Define the boundary tractions T and Ti

- Run the code

Neumann conditions

- What happens to the horizontal displacements? Find a solution.
- What is the effect of the Poisson ratio on the stored energy?
- What is the effect of the Poisson ratio on the stored energy
  - If all boundary displacements are controlled
  - If all boundary stresses are controlled

Time dependent problems

Open file 007.py

Fill in the gaps for the triaxial loading to work.

Is this the correct procedure for a triaxial test?

Iterative solutions

For nonlinear problems using an iterative method, such as the Newton-Raphson is common. For elasticity this is unnecessary but easy.

A new definition of the bilinear form and the linear form is necessary. The problem to solve is now

$$\frac{\partial A(u,v)}{\partial u}\delta u = -(A(u,v) - fv)$$

Where successive increments are evaluated, whose sum leads to the solution.

Details are not included here.

Iterative solutions

The stiffness matrix and residuals change:

```
# Newton-Raphson matrix
def NR(u):
    stiffness =dsde()
    stf=-as_tensor(stiffness[(i,j,k,l)]*nabla_grad(u)[(k,l)],(i,j))
    NR = inner(stf, nabla_grad(v))*dx
    return NR

# Newton-Raphson residual
def res(sol):
   res = -inner(sigma(sol), nabla_grad(v))*dx + dot(f, v)*dx + dot(T, v)*ds(1) + dot(Ti,v)*ds(2)
   return res
```

Iterative solutions

Open file 008.py

What is it doing?

What is wrong with the boundary conditions?
Can you fix it?

Compare the results for clamped and not clamped boundaries.
Is there an influence on the stored energy? Why?