PatchTest

March 25, 2020

[1]:
```
%config InlineBackend.figure_format='retina'
```

0.0.1 Verification and Validation (Fish and Belytschko, Section 8.2)

Two of the most important things from any numerical simulation are:

1. Verification: Are we solving the equations correctly.
2. Validation: Are we solving the right equations for the problem that we are trying to model.

Validation is a significantly more involved issue, and cuts right to the core of how we try to model a physical phenomena. For example, if we are modeling a material then is the material elastic, viscous or plastic? Is the constitutive relation that we are writing correct? Are the material properties correct? Do the boundary conditions correctly represent the real problem we are modeling? This requires significant experience, connection with the actual experiments and finding parallels with other similar systems. We are not going to discuss this matter further.

Verification is a more well defined problem. We believe that certain equations that we have are **correct**. The more relatively modest goal is to ensure that numerical solution to the differential equation is the *correct* solution, i.e., the **strong form** of the differential equation is solved correctly. However, since the finite element solution almost never gets the exact solution in the **strong** sense, the problem of verification becomes a bit more involved. The most basic test that any correct FEA code should satisfy is called as the **patch test**.

The simple idea behind this test is as follows. If an analytical solution θ could be exactly represent by the finite element solution θ^h, then even with a few number of arbitrary shaped and sized elements, one should **exactly** get $\theta = \theta^h$. Since any *good* finite element interpolation should at least be linear complete (linear elements, e.g., 3 node triangle), one can perform this test with respect to a **linear** field. In the case of heat conduction problem that we are trying to solve, if the temperature field is of the form

$$T(x, y) = \alpha_0 + \alpha_1 x + \alpha_2 y$$

, where the coefficients α_i are some arbitrary constants. Now this field $T(x, y)$ satisfies the heat conduction equation (which is a Poisson equation)

$$\nabla^2 T + s = 0$$

, where s, the source term is zero. Hence, if we provide **essential** boundary conditions $T(x, y) = \bar{T}$ on Γ_T, corresponding to this function, we have a boundary value problem whose solution is the same as that above. Since, the solution to a well-posed, linear, boundary value problem (BVP) is

unique, this solution is **THE** solution to the problem. Also, since, as mentioned earlier, any FEA formulation is linear complete, if we create a mesh like: On this particular mesh with arbitrary size elements, if we now now solve the BVP using FEA we should get an answer that should be extremely close to the exact solution (error $< 10^{-8}$).

In FEniCS, since the quadrilateral element support is not so prominent as of now, we use triangular elements for the patch test. Using the following steps.

1. We create a mesh of our own as below. Please note the steps.

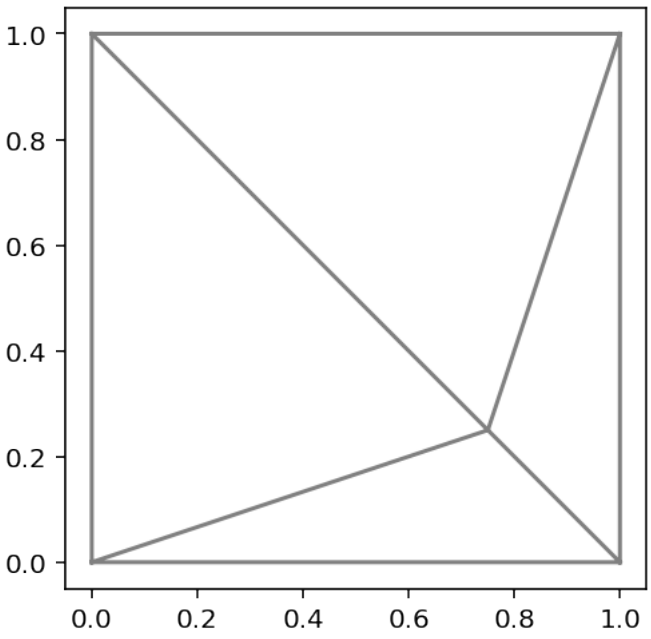
[7]:
```
# import the libraries

import dolfin as df
import matplotlib.pyplot as plt
import mshr
import numpy as np # numpy library for arrays etc.

# define an editor to write the mesh
%matplotlib inline
mesh = df.Mesh()
editor = df.MeshEditor()
editor.open(mesh, 'triangle', 2, 2)
editor.init_vertices(5)
editor.init_cells(4)
```

[8]:
```
# add vertices
editor.add_vertex(0, np.array([0.0, 0.0]))
editor.add_vertex(1, np.array([1.0, 0.0]))
editor.add_vertex(2, np.array([0.75, 0.25]))
editor.add_vertex(3, np.array([1.0, 1.0]))
editor.add_vertex(4, np.array([0.0, 1.0]))
```

[9]:
```
# add cells
editor.add_cell(0, np.array([0, 1, 2], dtype=np.uintp))
editor.add_cell(1, np.array([2, 1, 3], dtype=np.uintp))
editor.add_cell(2, np.array([2, 3, 4], dtype=np.uintp))
editor.add_cell(3, np.array([2, 4, 0], dtype=np.uintp))
```

[10]:
```
# final formalities
editor.close() # close the editor
mesh.order # order the mesh
df.plot(mesh) # plot
```

[10]:
```
[<matplotlib.lines.Line2D at 0x113a5c9e8>,
 <matplotlib.lines.Line2D at 0x113a5ce10>]
```



Define Laplace equation. The exact solution is:

$$\theta = \alpha_0 + \alpha_1 x + \alpha_2 y$$

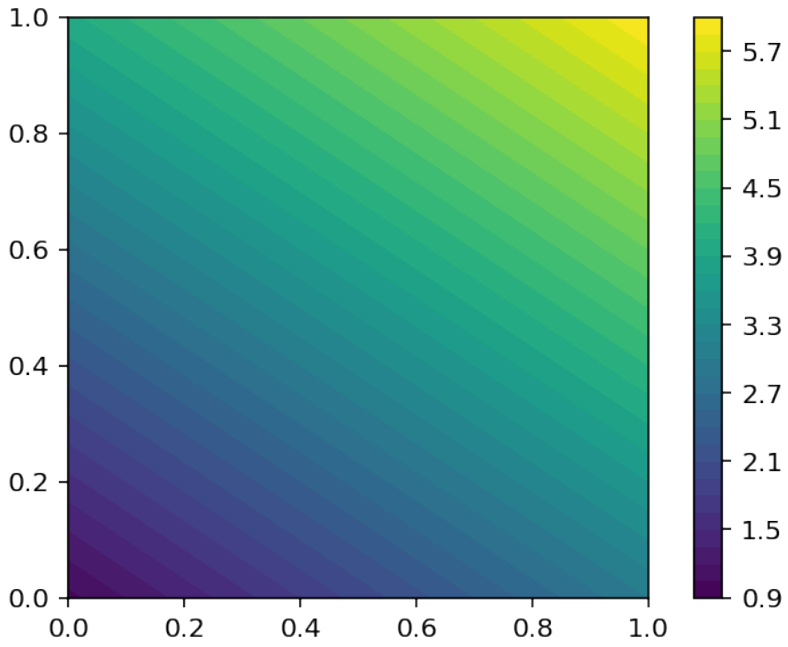
,

we define $\alpha_0 = 1, \alpha_1 = 2, \alpha_2 = 3$

[15]:
```
V = df.FunctionSpace(mesh, 'P', 1)

# The exact solution
u_D = df.Expression('a0 + a1*x[0] + a2*x[1]', a0 = 1, \
                    a1 = 2, a2 = 3, degree=1)

# Define boundary condition
def boundary(x, on_boundary):
    return on_boundary

bc = df.DirichletBC(V, u_D, boundary)

# Define variational problem
u = df.TrialFunction(V)
v = df.TestFunction(V)
f = df.Constant(0.0)
a = df.dot(df.grad(u), df.grad(v))*df.dx
L = f*v*df.dx

# Compute solution
u = df.Function(V)
```

```
df.solve(a == L, u, bc)

# Plot solution and mesh
c = df.plot(u)
plt.colorbar(c)
#plot(mesh)

print("L2_error is = ", df.errornorm(u_D, u, 'L2'))
```

```
L2_error is =  3.624438086253896e-16
```



The L_2 error is really small as expected. Also, since FEniCS is a well tested program, of course, it passes the patch test.

This concludes our discussion on Poisson equation which is of **elliptic**. The next topic is solving the dynamic heat (or diffusion) equation using FEniCS which is extension of the Poisson equation to the PDEs of **parabolic** type.

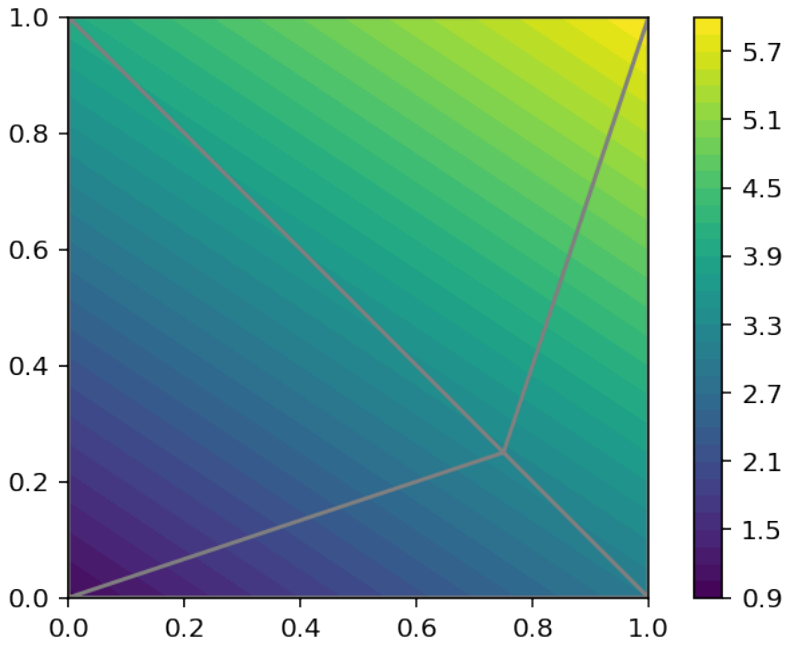
Below, I am attaching the entire patch code, which could be save as a .py (python) file.

[16]:
```
# import the libraries

import dolfin as df
import matplotlib.pyplot as plt
import mshr
import numpy as np # numpy library for arrays etc.
```

```
# define an editor to write the mesh
%matplotlib inline
mesh = df.Mesh()
editor = df.MeshEditor()
editor.open(mesh, 'triangle', 2, 2)
editor.init_vertices(5)
editor.init_cells(4)

# add vertices
editor.add_vertex(0, np.array([0.0, 0.0]))
editor.add_vertex(1, np.array([1.0, 0.0]))
editor.add_vertex(2, np.array([0.75, 0.25]))
editor.add_vertex(3, np.array([1.0, 1.0]))
editor.add_vertex(4, np.array([0.0, 1.0]))

# add cells
editor.add_cell(0, np.array([0, 1, 2], dtype=np.uintp))
editor.add_cell(1, np.array([2, 1, 3], dtype=np.uintp))
editor.add_cell(2, np.array([2, 3, 4], dtype=np.uintp))
editor.add_cell(3, np.array([2, 4, 0], dtype=np.uintp))

# final formalities
editor.close() # close the editor
mesh.order # order the mesh
df.plot(mesh) # plot

# create a BVP on FEniCS and solve

V = df.FunctionSpace(mesh, 'P', 1)

# The exact solution
u_D = df.Expression('a0 + a1*x[0] + a2*x[1]', a0 = 1, \
                 a1 = 2, a2 = 3, degree=1)

# Define boundary condition
def boundary(x, on_boundary):
    return on_boundary

bc = df.DirichletBC(V, u_D, boundary)

# Define variational problem
u = df.TrialFunction(V)
v = df.TestFunction(V)
f = df.Constant(0.0)
a = df.dot(df.grad(u), df.grad(v))*df.dx
L = f*v*df.dx
```

```
# Compute solution
u = df.Function(V)
df.solve(a == L, u, bc)

# Plot solution and mesh
c = df.plot(u)
plt.colorbar(c)
#plot(mesh)

print("L2_error is = ", df.errornorm(u_D, u, 'L2'))
```

```
L2_error is =  3.624438086253896e-16
```



Satisfying patch test is a sufficient condition that the code is correct. However, it is also shown that satisfaction of patch test also implies that the element is convergent. The other approach to check convergence is comparing with known solutions. However, for many geometries there is no knowledge of closed form solution. In that case, one uses the **method of manufactured solutions** that was discussed in one of the previous classes and also uploaded on the course website.

[ ]: