PRECEPT 09

CEE 361-513: Introduction to Finite Element Methods

Monday Nov. 27

Poisson's Equation with FEniCS

1. We are going to solve the simplest 2-D problem, the Poisson Equation. The problem reads: find $u : [0,1] \times [0,1] \to \mathbb{R}$ such that

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \Gamma_D \end{aligned}$$

We solve the problem on the domain $\Omega = [0,1] \times [0,1]$. Γ_D represents the Dirichlet boundary of the domain. For this problem we employ the method of manufactured solution to study the convergence. The manufactured solution is:

$$u = exp(-((x-0.5)^2 + (y-0.5)^2)/0.125^2))$$

First and foremost we use sympy to create the analytical solution ue and obtain the expressions for g and f.

```
# Create manufactured solution                                          1
x = sp.symbols('x[0]')                                                  2
y = sp.symbols('x[1]')                                                  3
solution = sp.exp(-((x-0.5)**2+(y-0.5)**2)/0.125**2)                    4
ue_code = sp.ccode(solution).replace('M_PI','pi')                       5
f_code = sp.ccode( -sp.diff(solution,x,2)-sp.diff(solution,y,2))\        6
      .replace('M_PI','pi')                                             7
```

Next we create the mesh/subdivision of the domain:

```
                                                                        1
V = FunctionSpace(mesh, 'Lagrange', poly_order)                         2
```

The mesh looks like:

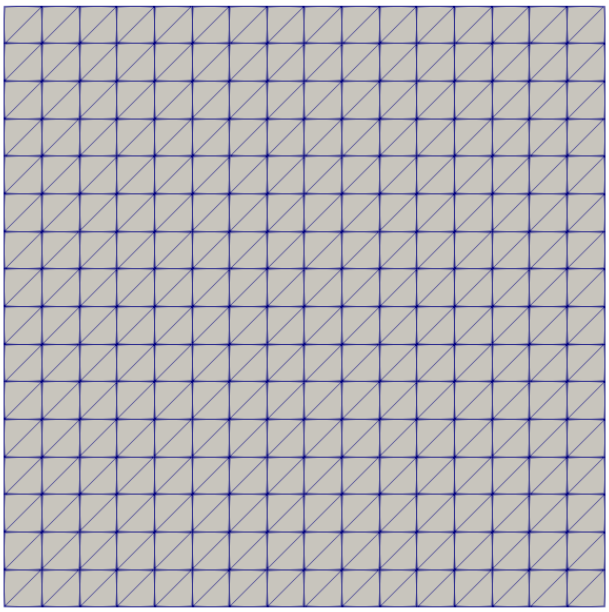

Figure 1: Mesh

We now create the subspace of piecewise linear polynomials as we saw in class.

```
# Create the function space                                    1
poly_order = 1                                                 2
V = FunctionSpace(mesh, 'Lagrange', poly_order)                3
```

Next we identify the parts of the boundary of our domain that are Dirichlet Γ_D. We do not have a Neumann Boundary in this problem

```
# Define the dirichlet boundary                                1
class dirichlet_boundary(SubDomain):                           2
  def inside(self, x, on_boundary):                            3
    return on_boundary                                         4
```

We define the exact solution as FEniCS expression

```
# Define the analaytical solution                              1
ue = Expression(ue_code, pi=np.pi, degree=5)                   2
```

and we define the Dirichlet boundary conditions

```
# Define the dirichlet boundary conditions                     1
bc = DirichletBC(V, ue, dirichlet_boundary())                  2
```

Now we are ready to define the trial function and the test function, the expression for the source term

```
# Define the trial and test function                           1
u = TrialFunction(V)                                           2
v = TestFunction(V)                                            3
```

```
# Define the forcing function                                  1
f = Expression(f_code,pi=np.pi, degree=5)                      2
```

We now define the bilinear form and the forcing functional

```
# Define the bilinear form
a = dot(grad(u),grad(v))*dx

# Define the forcing
F = f*v*dx
```

and we finally solve the problem

```
# Compute solution
uh = Function(V, name='displacement')
solve(a == F, uh, bc)
```

Since we know the exact the solution we can compute the $L2 - norm$, $H1 - norm$ of error , and the maximum error.

```
# Compute maximum error at vertices
vertex_values_ue = ue.compute_vertex_values(mesh)
vertex_values_uh = uh.compute_vertex_values(mesh)

error_L2 = errornorm(ue, uh, 'L2')
error_H1 = errornorm(ue, uh, 'H1')

error_max = np.max(np.abs(vertex_values_ue - vertex_values_uh))
```

The maximum error was 0.0204
We can solve the problem on successively refined meshes and obtain the rate of convergence.

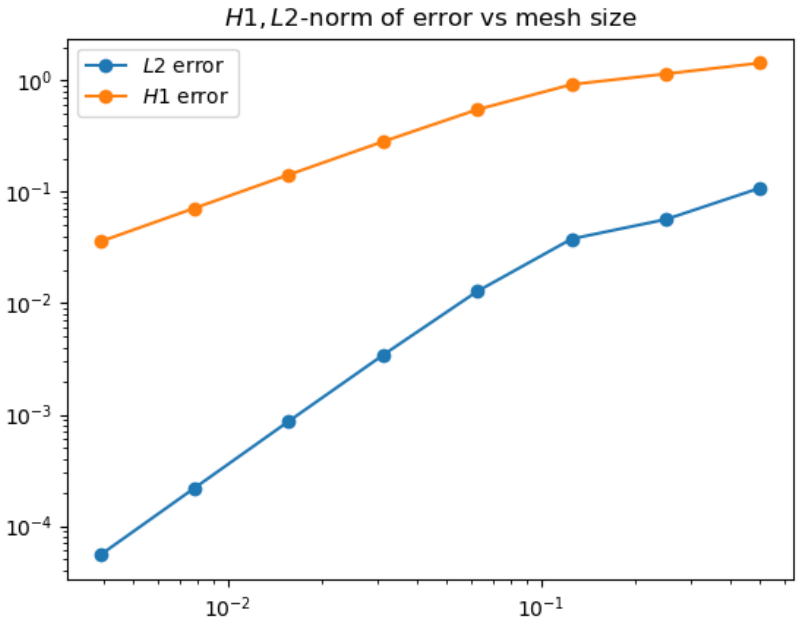

Figure 2: Convergence plot

The rate of convergence using $L2 - norm$ was 2
The rate of convergence using $H1 - norm$ was 1

2. Saving the solution in XDMF format

```
# Save as xdmf format                                                    1
# The file in which the solution is stored                               2
file_xdmf = XDMFFile('poisson.xdmf')                                     3
file_xdmf.write(uh,0)                                                    4
```

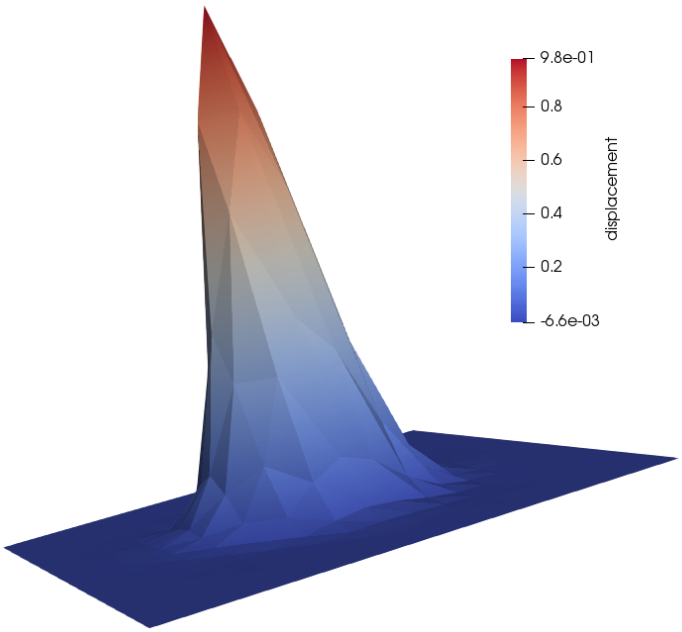

Figure 3: Solution with initial mesh

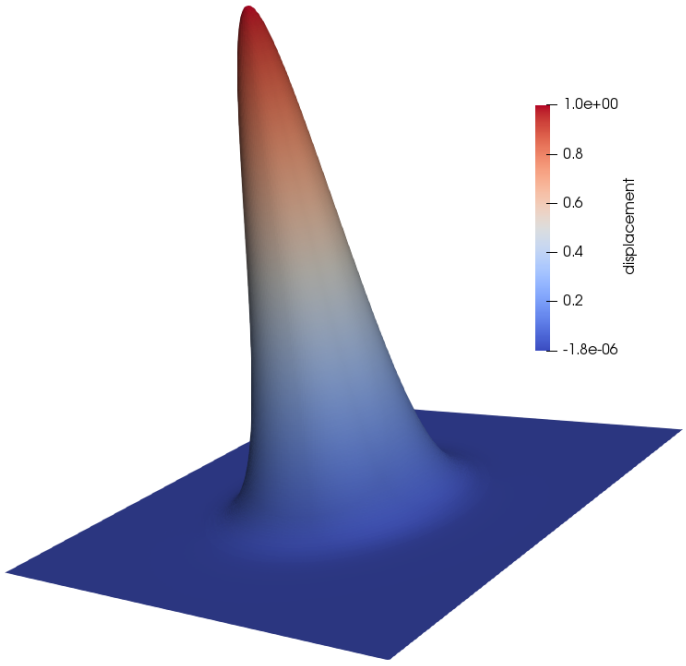

Figure 4: Converged Solution

Poisson's Equation with Quad Elements

Here we modify the poisson's problem file provided along with the homework code to add the function for computing $L1-norm$ and $H1-norm$ of the errors and determine the rate of convergence.

1. Compute $L2-norm$ of error

```
def compute_l2_norm_error(local_to_global_map, elements, uh, ue ):

  l2_error = 0.

  for element in elements:
    # Get the element local degress of freedom
    num_dofs = element.get_num_dofs( )

    # Get the quadrature rule
    gauss_points, gauss_weights = element.get_quadrature()

    # Fill in the stiffness matrix
    for q in range(len(gauss_points)):

      # Get the jacobian at the quadrature point
      jacobian = element.get_dmap( gauss_points[q], jacobian=True )

      # Get the image of the gauss point on the physical domain
      x_g = element.get_map( gauss_points[q] )

      uh_g = 0.

      # Loop over all degree of freedoms
      for i in range(num_dofs):
        # Get the global index of the local i dof
        i_global = local_to_global_map.get_global_dof( \
          element.element_index , i )

        # If we specified a source term
        uh_g += element.get_base_function_val(i,\
          gauss_points[q])*uh[i_global]

      # Add contribution of intergral at gauss point
      l2_error += pow( ue( x_g ) - uh_g , 2 )*jacobian*gauss_weights[q]

  return np.sqrt(l2_error)
```

2. Compute $H1-norm$ of error

```
import numpy as np
def compute_h1_norm_error(local_to_global_map, elements, uh, ue, due):

  h1_error = 0.

  for element in elements:
    # Get the element local degress of freedom
```

```
    num_dofs = element.get_num_dofs( )

    # Get the quadrature rule
    gauss_points, gauss_weights = element.get_quadrature()

    # Fill in the stiffness matrix
    for q in range(len(gauss_points)):

      # Get the jacobian at the quadrature point
      jacobian = element.get_dmap( gauss_points[q], jacobian=True )

      # Get the image of the gauss point on the physical domain
      x_g = element.get_map( gauss_points[q] )

      # Get the value of gradient of all basis at quadrature point

      uh_g = 0.
      duh_g = 0.

      # Loop over all degree of freedoms
      for i in range(num_dofs):
        # Get the global index of the local i dof
        i_global = local_to_global_map.get_global_dof( \
          element.element_index , i )

        # If we specified a source term
        uh_g += element.get_base_function_val(i,gauss_points[q])*\
            uh[i_global]
        duh_g +=element.get_base_function_grad(i,gauss_points[q])*\
            uh[i_global]

      # Add contribution of intergral at gauss point
      h1_error += pow( ue( x_g ) - uh_g , 2 )*jacobian*gauss_weights[q]\
            +  np.dot((due( x_g ) - duh_g),  (due( x_g ) - duh_g))\
            *jacobian*gauss_weights[q]

  return np.sqrt(h1_error)
```

3. Compute rate of convergence.

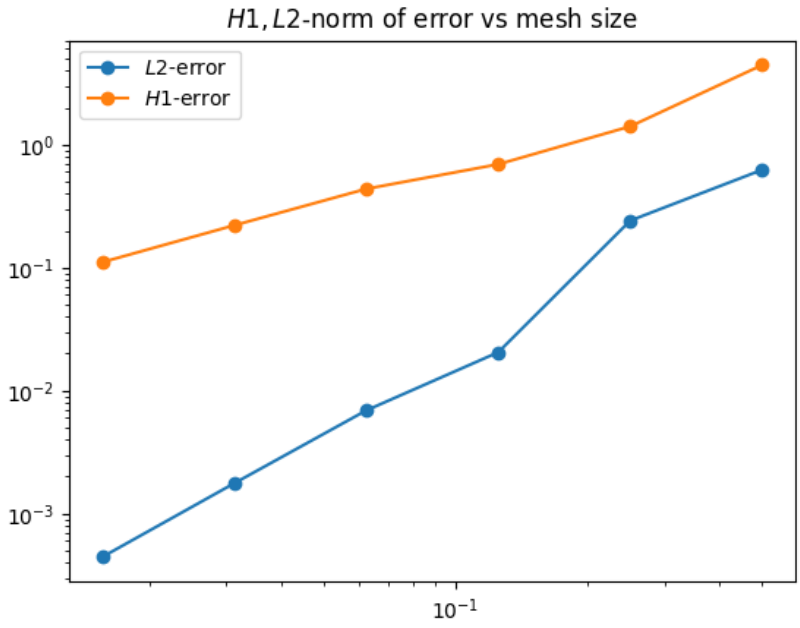

Figure 5: Convergence plot for quad elements

The rate of convergence for $L2 - norm$ of error is 2 and the rate of convergence of $H1 - norm$ of error is 1