HOMEWORK 6

CEE 361-513: Introduction to Finite Element Methods

Due: Friday Nov. 17 @ Midnight

NB: Students taking CEE 513 must complete all problems. All other students will not be graded for problems marked with $\star$, but are encourage to attempt them anyhow.

PROBLEM 1:

Read the following sections from the textbook and summarize the content of each section

1. 1.1-1.9
2. $\star$ 1.10
3. 1.12-1.15
4. 3.6-3.9 (stopping atop of page 148 before the discussion of shape function subroutines)

```
Solution :
Refer the relevant sections from the book.
```

PROBLEM 2: Learning FEniCS

This problem is to familiarize yourself with FEniCS, a very useful, high-level, and open-source finite element library.

1. Visit the FEniCS webpage (`https://fenicsproject.org`) and go to the Download tab at the top. If you are using Mac or Linux we suggest you use the Anaconda prebuilt distribution. If you are using Windows we suggest you use Docker distribution. If you have any issues along the installation please contact us for help.

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

$$\frac{d^2u}{dx^2} = f \quad \forall x \in (-1, 1)$$

and

$$u(-1) = g, \quad \frac{du}{dx}(1) = t.$$

We would like to manufacture boundary conditions (t and g) and source terms (f) such that the solution of the above boundary problem is exactly

$$u^e(x) = \frac{1}{2}\cos(2\pi x) + \exp(x) + x^3.$$

To do so we simply set

$$g = u^e(-1), \quad t = \frac{du^e}{dx}(1), \quad f(x) = \frac{d^2u^e}{dx^2}.$$

With the above we can then derive the weak form of the problem statement to be: find $u \in \mathcal{S}$, with $\mathcal{S}$ being the set of trial functions, such that

$$a(u, v) = F(v) \quad \forall v \in \mathcal{V}$$

where $\mathcal{V}$ is the set of test functions and

$$a(u, v) = \int_{-1}^{1} \frac{du}{dx}\frac{dv}{dx}dx, \quad F(v) = -\int_{-1}^{1} f\, v dx + t\, v(1).$$

All of the above is implemented for you in the attached code named `fenics_truss.py`. Your job for this part of the problem is to go through the code and make sure it runs and it makes sense. Once it does plot the analytical solution u^e and the finite element approximation u^h.

Here all I am going to do is to break down what we are doing.

First and foremost we use `sympy` to create the analytical solution u^e and obtain the expressions for $g, t, f(x)$. We also cast the variable `solution`, which is a `sympy` function, into a `python` lambda function for later plotting.

```
x = sp.symbols('x[0]')                                                    1
solution = 1./2*sp.cos( 2.*sp.pi*x ) + sp.exp(x) + x**3                   2
ue_code = sp.ccode(solution).replace('M_PI','pi')                         3
t_code = sp.ccode( sp.diff(solution,x,1)).replace('M_PI','pi')            4
f_code = sp.ccode( sp.diff(solution,x,2)).replace('M_PI','pi')            5
solution = sp.lambdify(x,solution)                                        6
```

Next we create the mesh (aka the subdivision) of our domain by first creating a mesh of a domain $[0, 1]$ then scaling the domain by 2 and shifting it to the left by 1.

```
# Start by creating a unit interval mesh
# subdivided in ndiv elements
ndiv = 10
mesh = UnitIntervalMesh(ndiv)

# We are going to shift the above mesh
# such that it dicretizes the interval [-1,1]
mesh.coordinates()[:] *= 2
mesh.coordinates()[:] -= 1
```

We now create the subspace of piecewise linear polynomials as we saw in class (the triangle-looking functions also shown in the second plot of the problem on Lagrange polynomials).

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

Next we identify the parts of the boundary of our domain (in $1-D$ these are two points, $\Gamma = \{x = -1, x = 1\}$) that are Dirichlet Γ_D and the ones that are Neumann Γ_N.

```
# Define the dirichlet boundary
class dirichlet_boundary(SubDomain):
  def inside(self, x, on_boundary):
    return on_boundary and abs( x + 1. ) < DOLFIN_EPS

# Define the neumann boundary
class neumann_boundary(SubDomain):
  def inside(self, x, on_boundary):
    return on_boundary and abs( x - 1. ) < DOLFIN_EPS
```

We next mark the boundaries of our domain

```
boundaries = FacetFunction("size_t", mesh)
dirichlet_boundary().mark(boundaries,1)
neumann_boundary().mark(boundaries,2)
ds = ds(subdomain_data=boundaries)
```

and we define the Dirichlet boundary conditions

```
bc = DirichletBC(V, ue, dirichlet_boundary())
```

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

```
u = TrialFunction(V)
v = TestFunction(V)
```

```
f = Expression(f_code,pi=np.pi, degree=5)
t = Expression(t_code,pi=np.pi, degree=5)
```

We now define the bilinear form and the forcing functional

```
a = dot(u.dx(0), v.dx(0))*dx
```

```
F = -f*v*dx + t*v*ds(2)
```

and we finally solve the proble

```
uh = Function(V)
solve(a == F, uh, bc)
```

Solution :
The plot obatined:

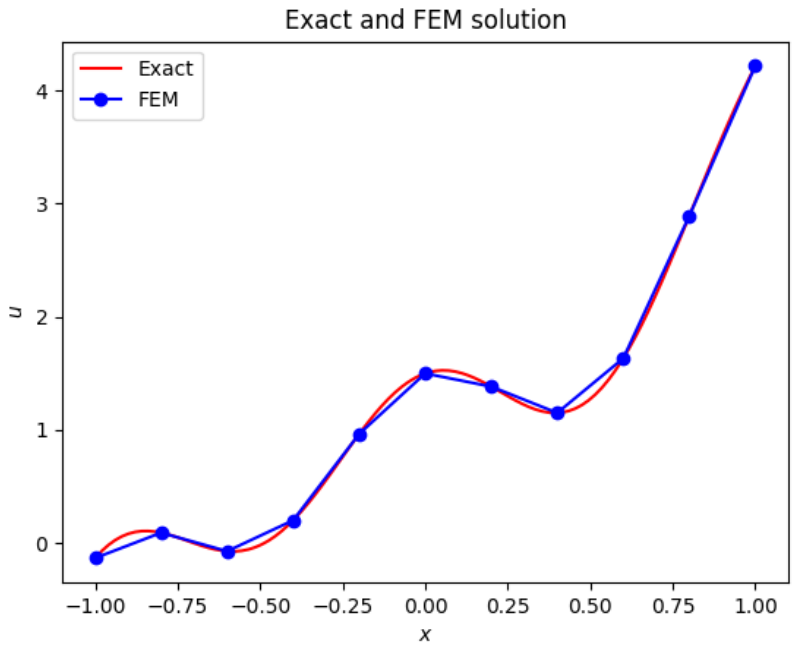

Figure 1: Exact and FEM solution

3. With the above code construct boundary conditions and source terms such that the analytical solution is of the form

$$u^e(x) = \frac{1}{2}\sin(3\pi x)\, x + \exp(x) + x^2.$$

Then compute the L^2 norm of the error for a mesh with $N = 2^1, 2^2, 2^3, 2^4, \ldots, 2^8$ elements (here you will have to write a `for` loop) and plot the the L^2 norm of the error against the average element size $h = 2/2^1, 2/2^2, \ldots, 2/2^8$ on a log-log plot (use `matplotlib loglog` plot).

Solution :

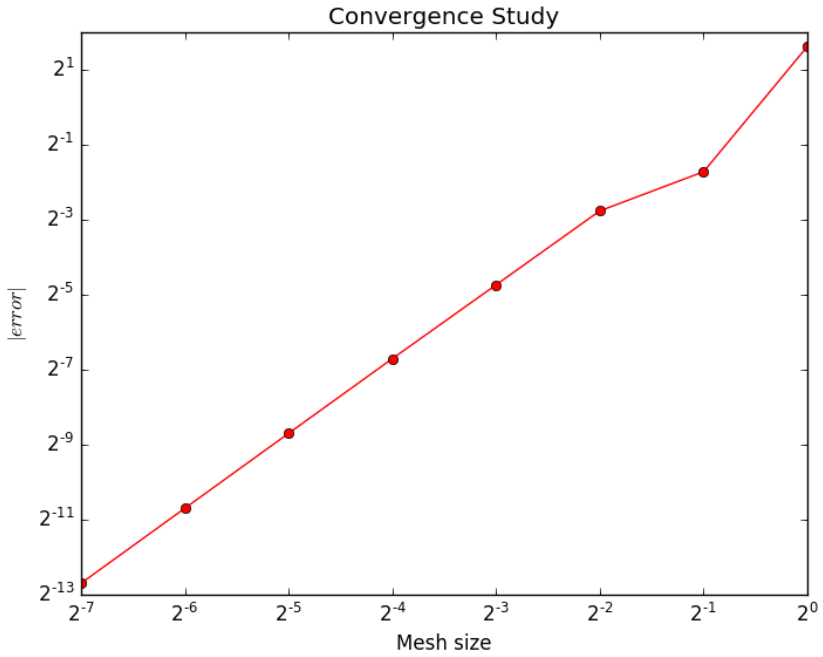

Figure 2: Convergence plot

4. What is the rate of convergence of the solution? Namely, we see that for a mesh size h the L^2 norm of the error is

$$\|e^h\|_{L^2} = \|u^h - u^e\| \approx \alpha h^k$$

where α is a constant independent of h. Our goal is to determine k.

To do so choose a fixed h (for example $h = 2/2^7$) then look at

$$\frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}} \approx \frac{\alpha\, h^k}{\alpha\,(h/2)^k} = 2^k \Rightarrow k = \log_2 \frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}}.$$

Solution :
The rate of convergence for piece-wise linear polynomials is 2.

5. What is the rate of convergence if instead of linear polynomials we use quadratic or cubic polynomials? Namely, repeat the above steps by changing the `poly_order` variable in

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

Note, that when we change the polynomial order we are simply replacing the "triangle" functions with the higher order Lagrange polynomials of the previous problem constructed over the interior of each element.

Solution :

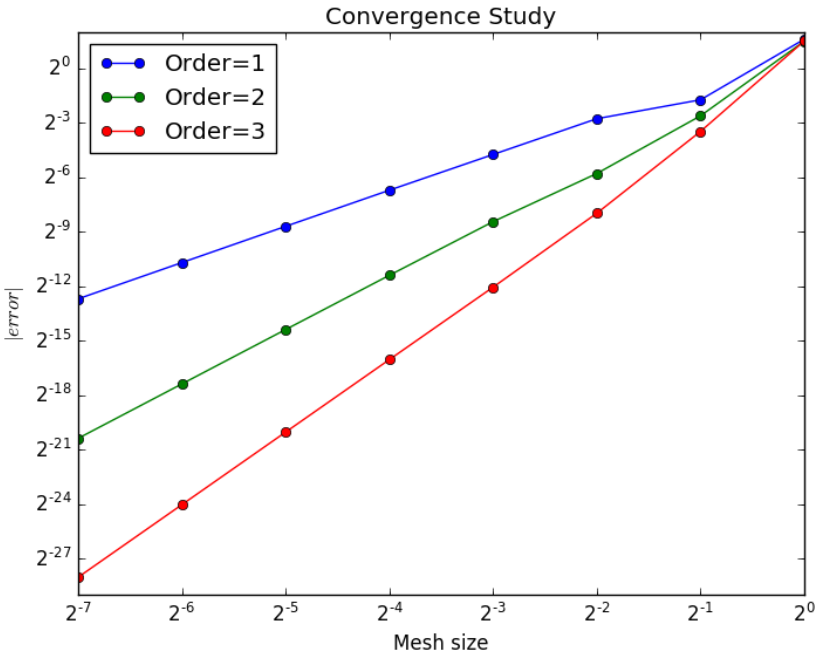

Figure 3: Convergence plot for different polynomial orders

Polynomial Order	Rate of Convergence
1	2
2	3
3	4

Table 1: Convergence rates for different polynomial orders

PROBLEM 3: Solving A Differential Problem

Consider the problem of solving for u : [-1,1] $\rightarrow \mathbb{R}$ such that

$$\frac{d^2u}{dx^2} + ku = f \quad \forall x \in (-1, 1)$$

where k = 0.1 and

$$f(x) = 0.5k\ cos(2.0\pi x) - 2.0\pi^2 cos(2.0\pi x)$$

and

$$u(-1) = \frac{1}{2},\ u(1) = \frac{1}{2}$$

For this problem refer to the attached code `finite_element_example.py`

1. Derive the weak form of the problem

   Solution :

   $$R = \frac{d^2u}{dx^2} + ku - f$$

   The set of trial and test functions are:
   Trail Functions:

   $$\mathcal{S} = \{u | u \in \ \text{Smooth}, u(-1) = 0.5, u(1) = 0.5\}$$

   Test Functions:

   $$\mathcal{V} = \{w | w \in \ \text{Smooth}, w(-1) = 0., w(1) = 0.\}$$

   Multiplying the residual with the test function and integrating over the domain:

   $$\int_{-1}^{1} \frac{d^2u}{dx^2} w\ dx\ + k\ u\ w\ -\ f\ w = 0$$
   $$\frac{du}{dx} w \Big|_{-1}^{1} - \int_{-1}^{1} \frac{du}{dx}\frac{dw}{dx}\ dx + \int_{-1}^{1} kuw\ dx\ - \int_{-1}^{1} fw\ dx = 0$$
   $$\int_{-1}^{1} \frac{du}{dx}\frac{dw}{dx}\ dx - \int_{-1}^{1} kuw\ dx\ = - \int_{-1}^{1} fw\ dx$$

2. Solve the problem using linear (Lagrange polynomials) finite elements. (fill in the missing parts in the starter code `finite_element_example.py` labeled with `# <- fill here`)

Solution :
The modified code:

Finite Element Example Starter Code Solution

```
# terms for a given solution                                                    1
k = 0.1 # <- fill here                                                          2
# Convert ue and f to lambda functions                                          3
x = sp.symbols('x')                                                             4
f = 0.5* k *sp.cos(2.0*sp.pi*x) - 2.0*sp.pi*sp.pi*sp.cos(2.0*sp.pi*x)           5
f = sp.lambdify(x, f)# <- fill here                                             6
```

Modifying the stiffness matrix

```
for j in range(num_dofs):                                                       1
  ke[i,j] +=  gauss_weights[q]*\                                                2
          d_lagrange_basis(nodes_crds_param,i,poly_order,gauss_points[q])*\      3
          d_lagrange_basis(nodes_crds_param,j,poly_order,gauss_points[q])*\      4
          jacobian**(-1.)\                                                      5
        - 0.1*gauss_weights[q]*\                                                6
          lagrange_basis(nodes_crds_param,i,poly_order,gauss_points[q])*\        7
          lagrange_basis(nodes_crds_param,i,poly_order,gauss_points[q])*\        8
          jacobian                                                              9
```

Modifying the boundary conditions

```
# Apply boundary dirichlet bc at the left and right                             1
bc_vals = [0.5, 0.5]#<- fill here                                               2
bc_dofs = [local_to_global_map(poly_order, 0, connectivity_array, 0),\          3
                local_to_global_map(poly_order, num_elements - 1,\              4
                 connectivity_array, poly_order ) ]#<- fill here                5
```

3. Modify the code from Problem 2 to solve the same differential problem as above.

Solution :
The modified code:

Modified FEniCS code

```
# Compute the boundary terms and source
# terms for a given solution
ndiv = 10
x = sp.symbols('x[0]')
k0 = 0.1
force = 1./2*k0*sp.cos( 2.*sp.pi*x ) -2.0*sp.pi**2*sp.cos(2.0*sp.pi*x)
f_code = sp.ccode(force).replace('M_PI','pi')
force = sp.lambdify(x,force)

# Define the dirichlet boundary
class dirichlet_boundary(SubDomain):
  def inside(self, x, on_boundary):
    return on_boundary and abs( x + 1. ) < DOLFIN_EPS \
    or abs( x - 1. ) < DOLFIN_EPS

# Define the dirichlet boundary conditions
bc = DirichletBC(V, 0.5, dirichlet_boundary())


# Define the bilinear form
a = dot(u.dx(0), v.dx(0))*dx - k0*dot(u,v)*dx

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

4. Plot the solution obtained with your finite element code and the one obtained with FEniCS.

Solution :

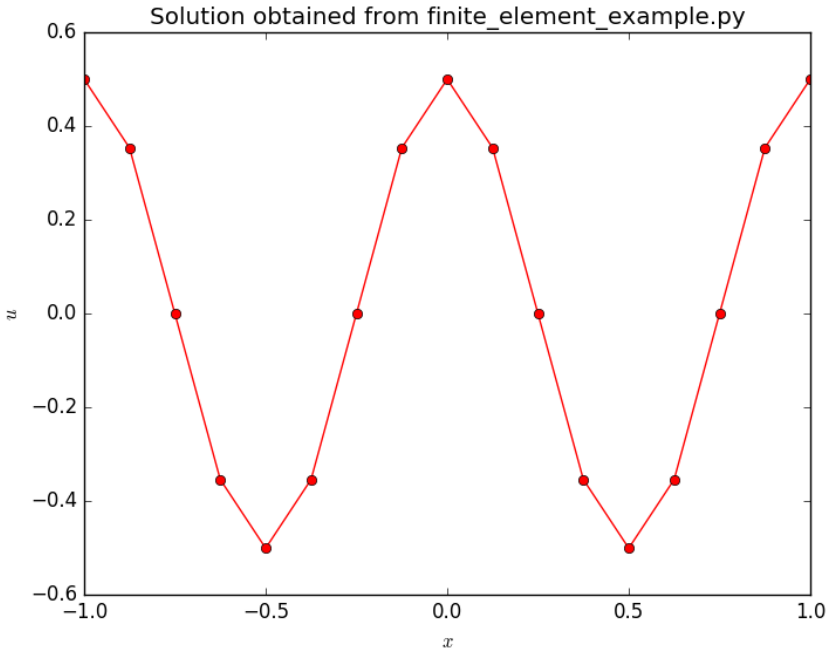

Figure 4: Solution obtained using finite element code

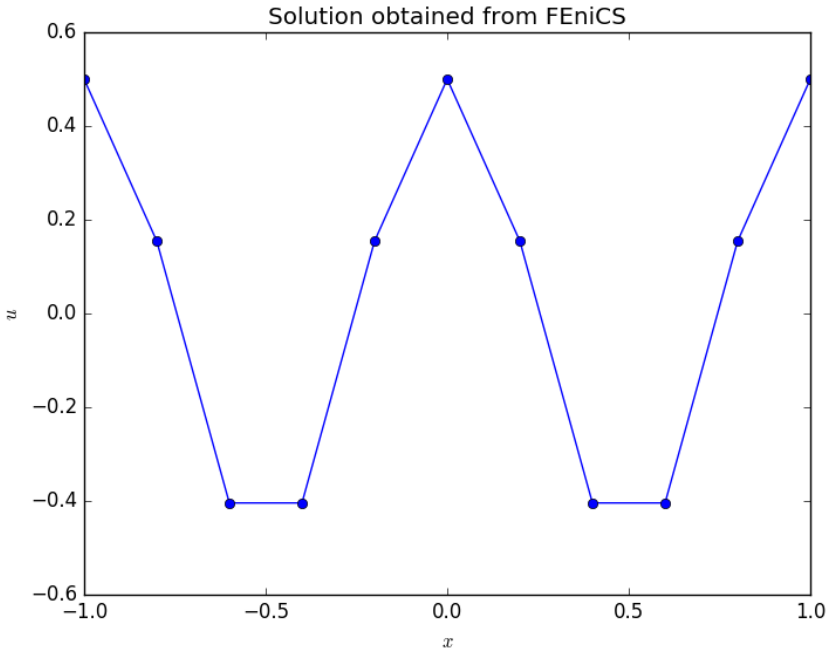

Figure 5: Solution obtained using FEniCS