HOMEWORK 8

CEE 361-513: Introduction to Finite Element Methods

Due: Friday Dec. 8 @ 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 and summarize App. 3.1 Triangular and Tetrahedral Elements

```
Solution :
Refer to the book.
```

PROBLEM 2:

In `triangular_element/` and `utils/` there are several files containing important functions to assist the construction of a triangular element. The `triangular_element` object is implemented in the file named `triangular_element/triangular_element.py`.

1. The `triangular_element` object possesses a map from the parametric domain to the physical domain $\hat{\boldsymbol{x}}^e(\boldsymbol{\xi})$ which is implemented in the method `get_map` on line 111. The map is based upon isoparametric mapping, namely

$$\hat{\boldsymbol{x}}^e(\boldsymbol{\xi}) = \hat{\phi}_i(\boldsymbol{\xi})\boldsymbol{x}_i^e$$

where $\boldsymbol{x}_i^e$ denotes the positions in real space of the i^{th} degree of freedom (see figure below). When the element object is first created the class assumes by default that only the vertex nodes are passed to the object (for example, for polynomials degree $p = 2$ only the vertex $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2$ are passed to the object) and the remainder of the nodes should be interpolated linearly. Recalling from class the meaning of barycentric coordinates, using barycentric coordinates only, implement the linear interpolation of edge and interior nodes in the method `interpolate_interior_nodes` on line 94 of `triangular_element.py`. (hint: to do so you you should use `self.basis.dof_nodes` as well as `crds_vertex`.)

```
Solution :

        self.nodes_crds_phys[ a ] += self.basis.dof_nodes[a,i]*crds_vertex[i]# <-
```

2. A triangular element in `triangular_element/triangular_element.py` is constructed by passing the element index, the coordinates of the entire mesh, the connectivity of the entire mesh, and the polynomial order of the interpolating functions. Look at line 39 of `triangular_element.py` for more details. If we are interested in studying only one element we can then construct a mesh with one element, and construct the element object as illustrated in the starter code `problem_1.py` and also shown below. Fill in the arguments of `my_element`.

```
Solution :

  my_element = triangular_element.element( element_index, coordinates,\
   connectivity, poly_order)  # <-- fill me here
```

3. One of the methods (the functions of the element object), as you can imagine, is the function that takes a base index $i \in \{0, \ldots, \text{num of dof}\}$ and a coordinate in the parametric domain and returns the base function at that point. This particular method is implemented at line 154 of `triangular_element.py` and is named `get_base_function_val`. For this part of the problem we would like you to plot all the basis

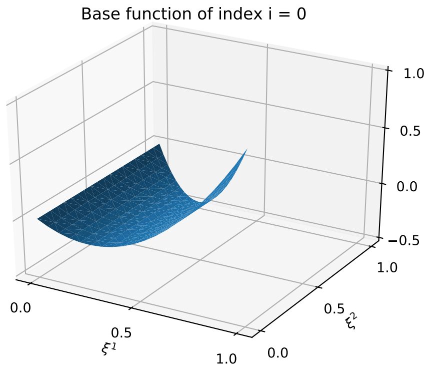

Figure 1: Sample plot of a base function

functions for degree 2 polynomial basis function. You should get a figure similar to the one shown in Fig. 1 for all the basis functions. A starter code is provided in the function `part_2` in the file `problem_1.py`.

Solution :

```
        z[idx] = element.get_base_function_val(base_index, crd) #<-- fill here1
```

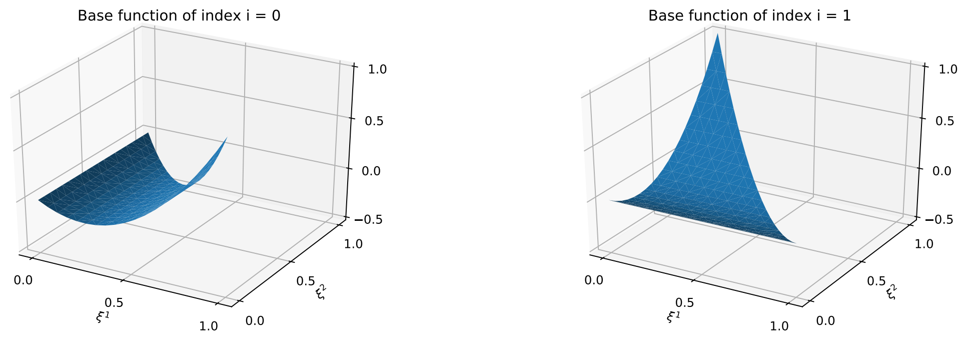

Figure 2: Base function of index 0 and 1

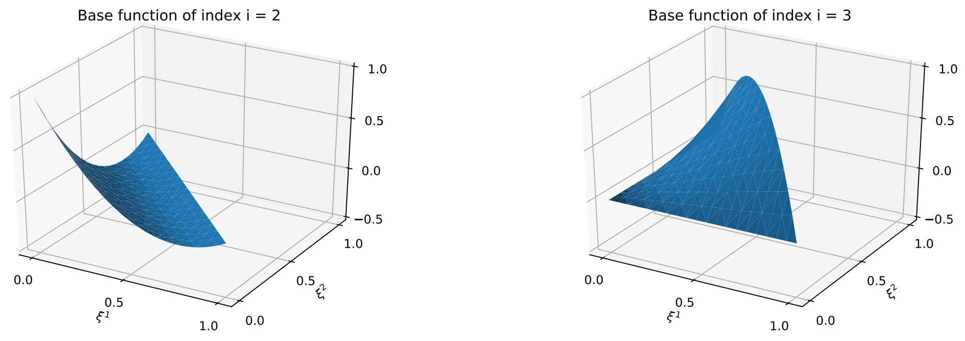

Figure 3: Base function of index 2 and 3

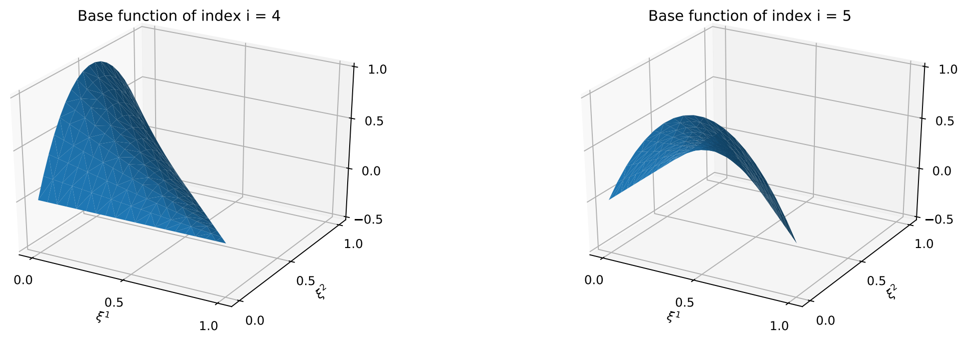

Figure 4: Base function of index 4 and 5

4. If we would like to use curved boundaries by interpolating all the nodes with the basis functions we must pass the coordinates $\boldsymbol{x}_i^e$ to the element object. Luckily this is implemented in the method `set_interior_nodes_coordinates` on line 100 of `triangular_element.py`.

   Your job is to create an array of coordinates of dimension $(p+1)(p+2)/2 \times d$, where $p = 2$ is the polynomial order and $d = 2$ is the space dimension, containing the following coordinates. Then you must pass this array to `set_interior_nodes_coordinates`. Refer to the starter function `part_4.py` in `problem_1.py` for guidance.

Node	x^1	x^2
$\boldsymbol{x}_0$	2.0	0.0
$\boldsymbol{x}_1$	0.0	2.0
$\boldsymbol{x}_1$	-2.0	0.0
$\boldsymbol{x}_3$	0.5	1.0
$\boldsymbol{x}_2$	-1.0	1.5
$\boldsymbol{x}_5$	0.0	0.4

   Again, using the same starter function as reference, plot the shape of the element in the physical space before and after you interpolate quadratically the element edges.

   Solution :

```
Xe = np.array([(2.0,0.),(0.,2.0),(-2.0,0.0)\
,(0.5,1.0),(-1.0,1.5),(0.0, 0.4)]) # <-- fill here

element.set_interior_nodes_coordinates( Xe ) # <- fill here
```

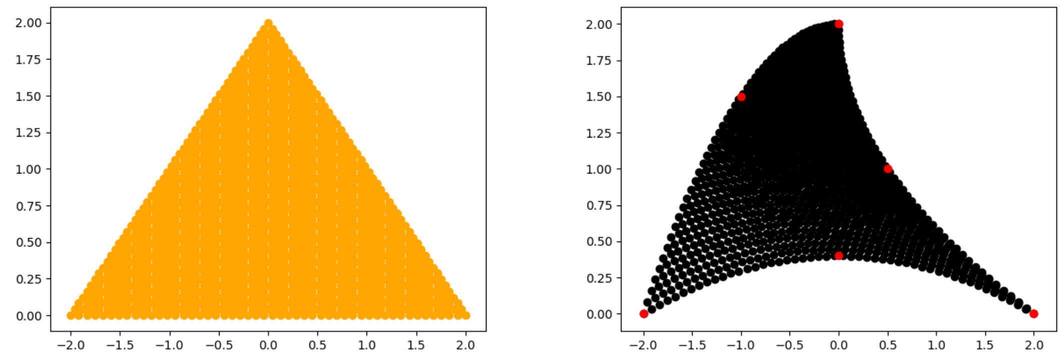

Figure 5: Curved element before (orange) and after interpolation (black)

5. You recall from class that we mentioned that the Jacobian $\hat{j}^e(\boldsymbol{\xi}) = \det(\nabla_{\boldsymbol{\xi}}\hat{\boldsymbol{x}}^e(\boldsymbol{\xi}))$ is a measure of change of a differential area element in the parametric domain to the physical domain. Namely

$$\hat{j}^e = \frac{d\Omega^e}{d\hat{\Omega}}.$$

   As the Jacobian is a fundamental quantity in performing a lot of calculations in finite elements, the computation of the Jacobian is implemented in the method `get_dmap` on line 114 of `triangular_element.py`. The value of the Jacobian can vary spatially. Plot the value of the Jacobian for the curved edge element from the previous exercise. Can you interpret what you are seeing?

Solution :

```
    Y[i] = element.get_map(x)# <-- fill me here

    J[i] = element.get_dmap(x, jacobian=True)# <-- fill me here
```

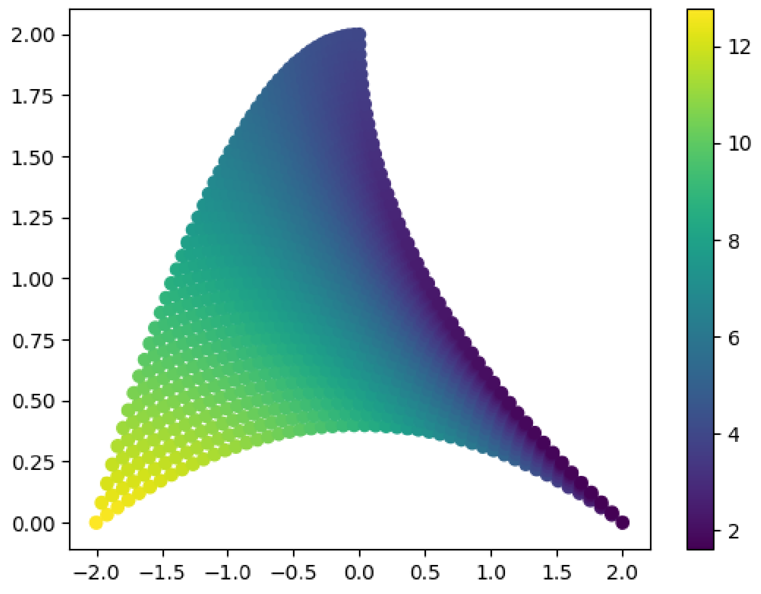

Figure 6: The jacobian for the curved element

6. As we saw in class the area of an element is given by

$$A^e = \int_{\omega^e} d\Omega = \int_{\hat{\Omega}} \hat{j}(\boldsymbol{\xi}) d\hat{\Omega} = \int_0^1 \int_0^{1-\xi^2} \hat{j}^e(\boldsymbol{\xi}) d\xi^1 d\xi^2$$

and the above integral can be approximated using numerical quadrature, namely

$$A^e = \int_0^1 \int_0^{1-\xi^2} \hat{j}^e(\boldsymbol{\xi}) d\xi^1 d\xi^2 \approx \sum_{(\tilde{\xi}_Q, \omega_q) \in \mathcal{Q}} j^e(\tilde{\boldsymbol{\xi}}_Q) \, \omega_Q.$$

Luckily for us the quadrature rule $\mathcal{Q}$ (the set of tuples of quadrature points $\tilde{\boldsymbol{\xi}}_Q$ and quadrature weights ω_Q) has been implemented in the method called `get_quadrature` on line 195 of `triangular_element.py`. With the above and the starter code in function `part_6.py` compute the area for the element with curved edges.

Solution :

```
    jacobian = element.get_dmap(q_points[i], jacobian=True)#<-fill me here
```

```
The area of the curved element is 2.93.
```

7. Suppose now that you want to integrate the function $g(\boldsymbol{x}) = \sin(x_1)\exp(x_2)$ over the curved element from the previous parts of the problem. Unfortunately in `triangular_element/simplex_quadrature.py` only

quadrature rules of degree precision up to order 3 were hard coded. Your job is to implement the 6-point quadrature formula of degree precision 4 as found in App. 3.1 of the textbook and use it to compute the integral of $g(\boldsymbol{x})$ over the curved domain (note that the weights in Table 3.I.1 should be multiplied by 1/2). Namely, if we let Ω^e denote the curved domain you want to compute

$$\int_{\Omega^e} g(\boldsymbol{x})d\Omega = \int_{\hat{\Omega}} g(\boldsymbol{x}^e(\boldsymbol{\xi}))j^e(\boldsymbol{\xi})d\hat{\Omega} \approx \sum_{i=1}^{n_Q} g(\boldsymbol{x}^e(\tilde{\boldsymbol{\xi}}_i))\omega_i$$

where $\tilde{\boldsymbol{\xi}}_i$ and ω_i are the the quadrature points and weights you just implemented.

```
Solution :
```

```
def part_7(element):
  # Get the quadrature rule
  q_points, q_weights = element.get_quadrature(4)

  f = lambda x : np.sin(x[0])*np.exp(x[1])

  # Initialize the value of the integral
  val = 0
  # Loop over all quadrature points
  for i in range(len(q_weights)):

    # Get the jacobian at the quadrature point
    jacobian = element.get_dmap(q_points[i], jacobian=True)#<-fill me here

    # Add the contribution
    val += f(q_points[i])*q_weights[i] *jacobian

  print 'The value of the integral is %.2f'%val
```

```
The value of the integral is 1.03.
```

PROBLEM 3: Elasticity with FEniCS

This problem solves a 2-dimensional elasticity problem in FEniCS

1. What do the Lamè parameters λ and μ represent

   Solution :
   The Lamè parameters are material properties that arise in stress-strain relationships. λ roughly denotes the resistance to volumetric change while μ denotes the resistance to shear.

2. Derive the weak form of the problem.

Solution :
The trial and test functions are:

$$\mathcal{S} = \{\boldsymbol{u} | \boldsymbol{u} \in [H^1(\Omega)]^2, \boldsymbol{u} = \boldsymbol{g} \ \forall \ \boldsymbol{x} \in \Gamma_D\}$$

$$\mathcal{V} = \{\boldsymbol{v} | \boldsymbol{v} \in [H^1(\Omega)]^2, \boldsymbol{v} = \boldsymbol{0} \ \forall \ \boldsymbol{x} \in \Gamma_D\}$$

The weak form as derived in class is:

$$\int_{\Gamma_N} \boldsymbol{v} \cdot \boldsymbol{\sigma} \boldsymbol{n} d\Gamma - \int_{\Omega} \boldsymbol{\sigma} : \nabla \boldsymbol{v} \ d\Omega - \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{v} \ d\Omega = 0$$
$$\int_{\Omega} \boldsymbol{\sigma} : \nabla \boldsymbol{v} \ d\Omega = \int_{\Gamma_N} \boldsymbol{v} \cdot \boldsymbol{t} d\Gamma - \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{v} \ d\Omega$$

3. We would like to manufacture the boundary conditions $\boldsymbol{g}$, source term $\boldsymbol{f}$ and tractions $\boldsymbol{t}$ such that the solutions at the above boundary value problem is exactly:

$$\boldsymbol{u}^e(\boldsymbol{x}) = sin(x_1)x_2\mathbf{e}_1 + (-x_1^2 + x_1)\mathbf{e}_2 \tag{1}$$

Derive the forcing function $\boldsymbol{f}$ in terms of Lamè parameters λ and mu

Solution :

$$\boldsymbol{f} = -(\lambda + 2.0\mu) sin(x_1)x_2\mathbf{e}_1 + ((\mu + \lambda)cos(x_1) - 2.0\mu)\mathbf{e}_2$$

4. Consider the boundary $\Gamma_N^{top} = [\mathbf{0}, \ell] \times \{d\}$ (namely the top part of the domain). We are interested in applying boundary tractions on this top part of the domain such that we recover the solution of Eq. (1). What is the outward unit normal $\boldsymbol{n}$ to this boundary? Obtain an expression for the boundary tractions $\boldsymbol{t}^{top}$ that should be applied on Γ_N^{top} in order to recover the solution of (1).

Solution :

$$\boldsymbol{t}^{top} = \mu * (1 - 2x_1 + sin(x_1)\mathbf{e}_1 + \lambda * cos(x_1) * x_2\mathbf{e}_2$$

5. The code above is implemented in FEniCS in `elasticity_fenics.py`. Fill in the values for $\boldsymbol{f}$ and $\boldsymbol{t}$ in the code (`<-fill here`, lines 105, 110) and complete the forcing functional F($\boldsymbol{v}$) on line 117.

Solution :

```
f = Expression(('-sin(x[0])*x[1]*(lmbda+2.0*mu)',\
    '((mu+lmbda)*cos(x[0])-2.0*mu)'),mu = mu, lmbda = lmbda, degree = 7 )
```

```
traction_top = Expression(('mu*(1.0-2.0*x[0]+sin(x[0]))',\
    'lmbda*(cos(x[0]))*x[1]'), lmbda=lmbda, mu = mu, degree = 7)
```

```
F = -dot(f, v)*dx + dot(traction_top, v)*ds(2)+ \
    dot(traction_right, v)*ds(3) + dot(traction_bottom, v)*ds(4)
```

6. Run the code and report the L^2-norm of error for default mesh and plot the solution using ParaView.

Solution :
The L^2-norm of the error was 0.000386

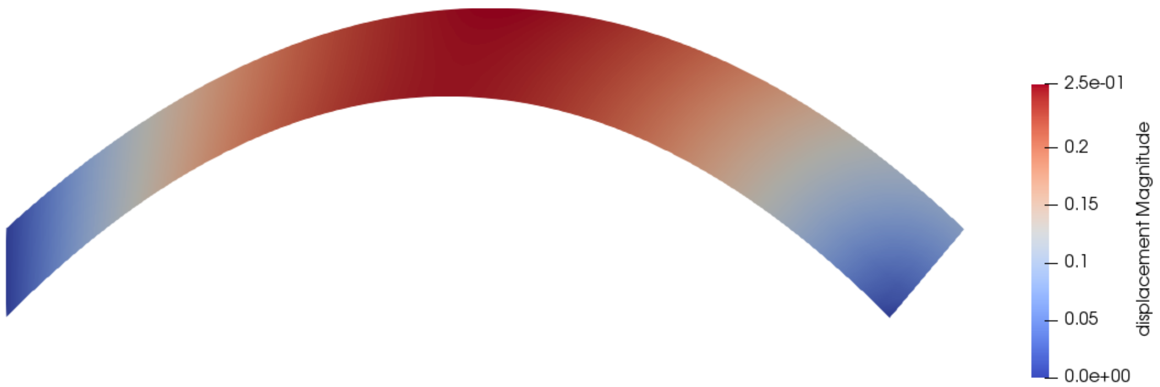

Figure 7: Deformed shape of the beam

7. Run the code on refined meshes iteratively (modify `ndiv_x, ndiv_y`) to obtain the rate of convergence using L^2-norm and H^1-norm of the error for a linear interpolation.

Solution :
The rate of convergence using L^2-norm of the error ≈ 2 The rate of convergence using H^1-norm of the error ≈ 1