HOMEWORK 7

CEE 361-513: Introduction to Finite Element Methods

Due: Friday Dec. 1 @ 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.

Introduction

The objective of this homework is to reconnect theoretical results derived in class with an actual implementation of Finite Element Methods. Particularly in this homework assignment we will be looking at quadrilateral elements in two dimension.

PROBLEM 1:

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

1. A quadrilateral element in `quadrilateral_element/quadrilateral_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 `quadrilateral_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`.

```
    ax.set_zlim([-0.5,1])                                              1
    plt.title(r'Base function of index i = %i'%base_index)             2
    plt.savefig('../../figures/base_function_%i.pdf'%base_index)       3
    plt.close('all')                                                   4
                                                                       5
def part_4(element):                                                   6
                                                                       7
  # Create a list of sample points in the parametric domain            8
  xs = np.linspace(-1,1,100)                                           9
  X = np.meshgrid( xs, xs )                                            10
  X = np.c_[ X[0].flatten(), X[1].flatten() ]                          11
                                                                       12
  # Map the points in the physical domain before                       13
  # interpolating the edge nodes quadratically                         14
  Y = X*0                                                              15
  for i,x in enumerate(X):                                             16
```

Solution :

```
  element_index = 0                                                    1
  my_element = quadrilateral_element.element( element_index,\          2
```

2. One of the methods (the functions of the element object), as you can imagine, is the function that takes a base index $i \in \{0, \ldots, \text{poly_order}\}$ and a coordinate in the parametric domain and returns the base function at that point. This particular method is implemented at line 144 of `quadrilateral_element.py` and is named `get_base_function_val`. For this part of the problem we would like you to plot all the

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

Solution :

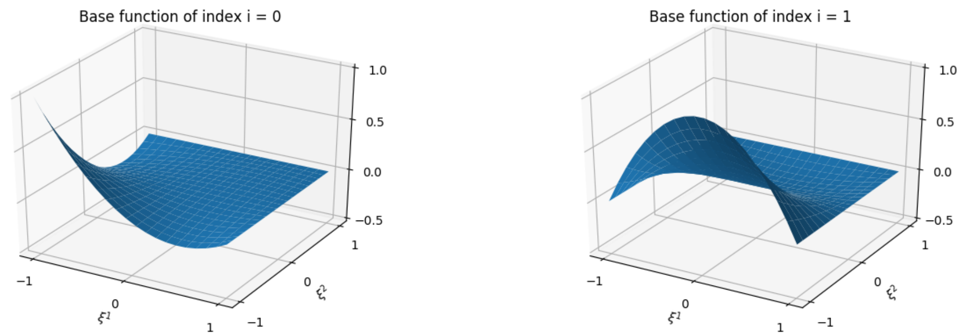


Figure 1: Plot of a base function with index 0 and 1

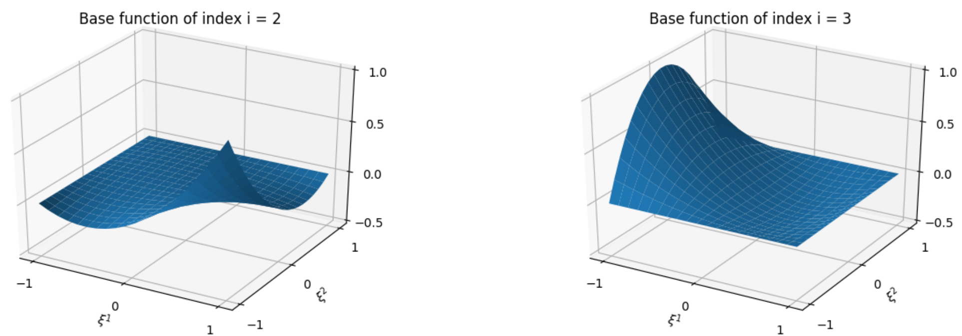


Figure 2: Plot of a base function with index 2 and 3

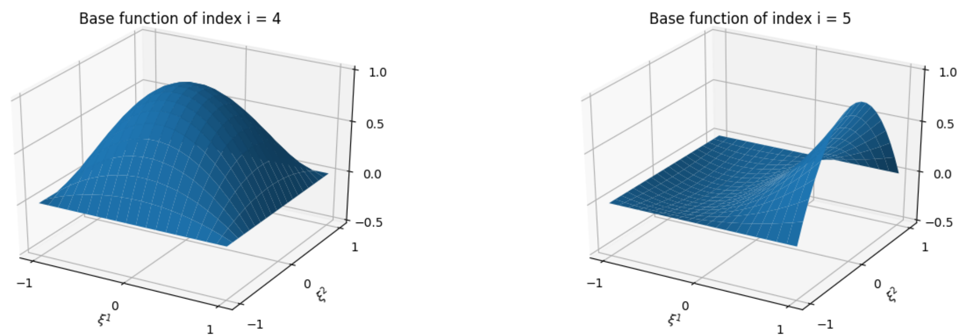


Figure 3: Plot of a base function with index 4 and 5

```
Solution :
```

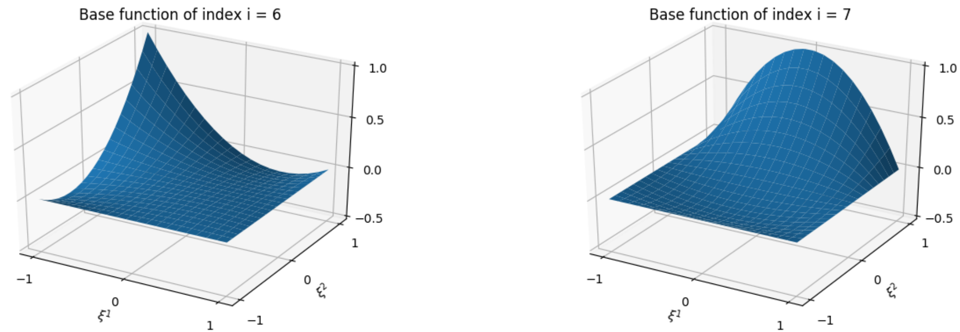


Figure 4: Plot of a base function with index 6 and 7

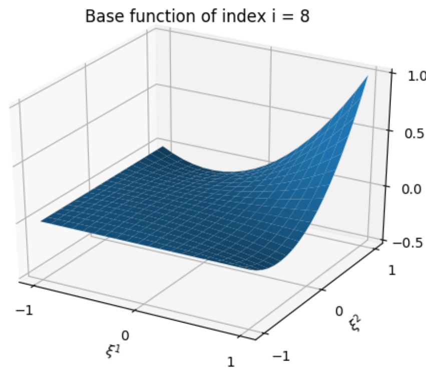


Figure 5: Plot of a base function with index 8

3. Read and summarize §3.3 Isoparametric Elements of the textbook.

```
Solution :
Refer to the textbook.
```

4. The `quadrilateral_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 99. The map is based upon isoparametric mapping, namely

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

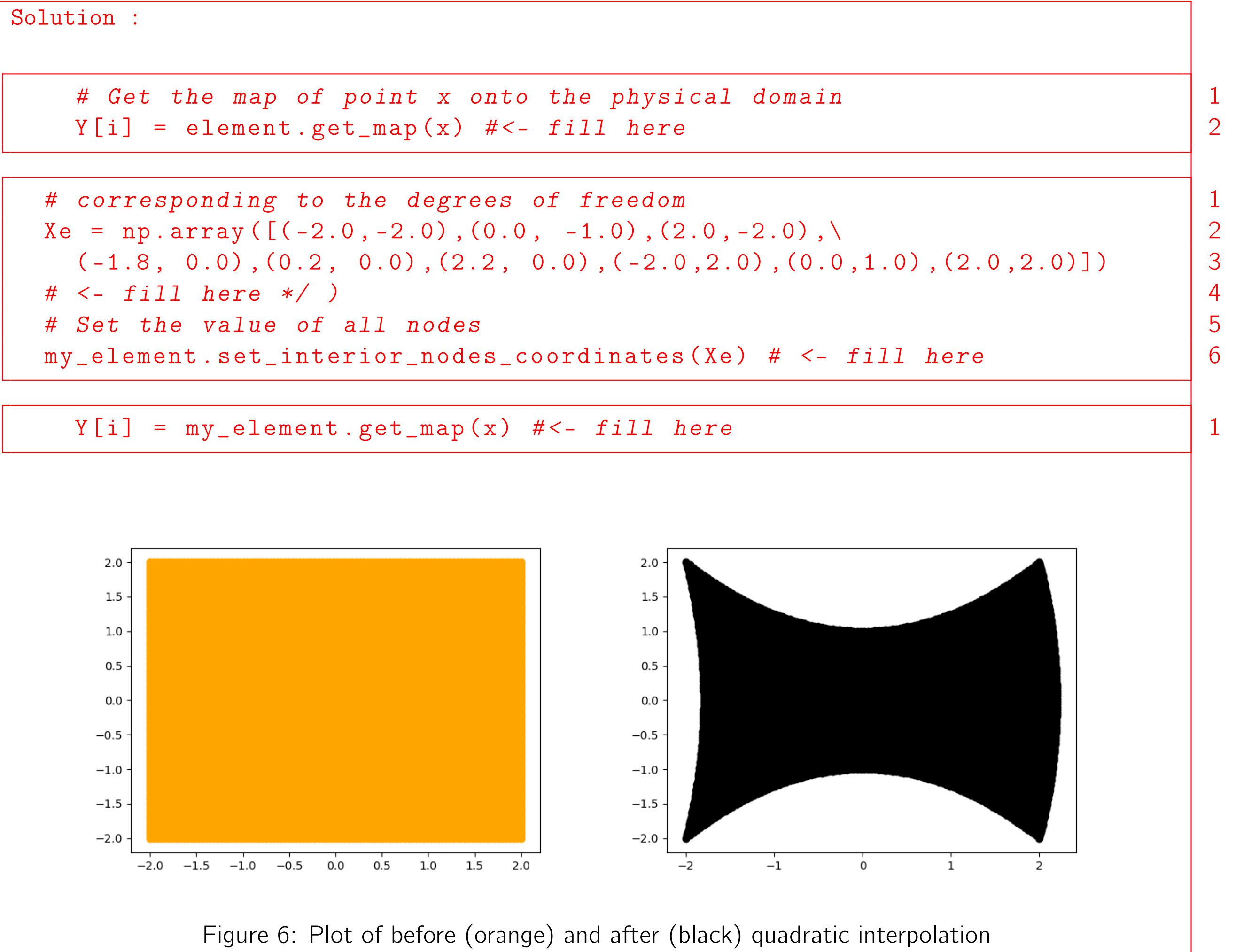
where $\boldsymbol{x}_i^e$ denotes the positions in real space of the ith degree of freedom (see figure below). When the element object is first created the class assumes by default that only the vertex nodes are passed to the object (for example, for polynomials degree $p = 2$ only the vertex $\boldsymbol{x}_0, \boldsymbol{x}_2, \boldsymbol{x}_6, \boldsymbol{x}_8$ are passed to the object) and the remainder of the nodes are interpolated linearly. To see how this is done visit `interpolate_interior_nodes` on line 68 of `quadrilateral_element.py`.

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 89 of `quadrilateral_element.py`.

Your job is to create an array of coordinates of dimension $(p+1)^d \times d$, 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
$\boldsymbol{x}_0$	-2.00	-2.00
$\boldsymbol{x}_1$	0.00	-1.00
$\boldsymbol{x}_2$	2.00	-2.00
$\boldsymbol{x}_3$	-1.80	0.00
$\boldsymbol{x}_4$	0.20	0.00
$\boldsymbol{x}_5$	2.20	0.00
$\boldsymbol{x}_6$	-2.00	2.00
$\boldsymbol{x}_7$	0.00	1.00
$\boldsymbol{x}_8$	2.00	2.00

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 :

```
    # Get the map of point x onto the physical domain
    Y[i] = element.get_map(x) #<- fill here
```

```
  # corresponding to the degrees of freedom
  Xe = np.array([(-2.0,-2.0),(0.0, -1.0),(2.0,-2.0),\
    (-1.8, 0.0),(0.2, 0.0),(2.2, 0.0),(-2.0,2.0),(0.0,1.0),(2.0,2.0)])
  # <- fill here */ )
  # Set the value of all nodes
  my_element.set_interior_nodes_coordinates(Xe) # <- fill here
```

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

Figure 6: Plot of before (orange) and after (black) quadratic interpolation

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 `quadrilateral_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 :

```
    # Get the map of point x onto the physical domain
    Y[i] = my_element.get_map(x) #<- fill here

    # Get the jacobian
    J[i] = my_element.get_dmap(x, jacobian=True) #<- fill here
```

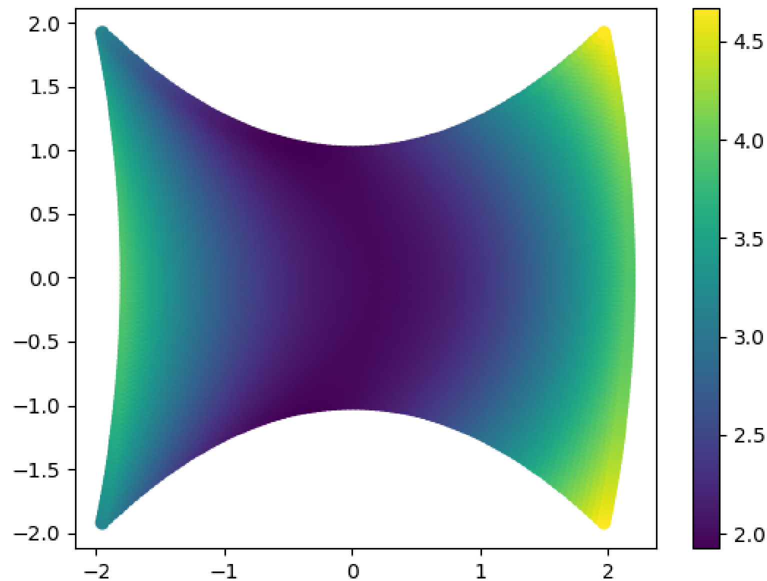


Figure 7: Plot of the jacobian

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

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

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

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

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 185 of `quadrilateral_element.py`.

With the above and the starter code in function `part_6.py` compute the area for the element with curved edges.

Solution :

```
q_points, q_weights = element.get_quadrature()   # <- fill here
```

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

    # Add the contribution
    A += jacobian*q_weights[i]#<- fill here
```

The value of area of the curved element is 10.67.

PROBLEM 2:

Recall from class the structure of a finite element code as shown below. In the file `problem_2.py` we effectively implemented the flow chart below for the differential problem of : find $u \in \Omega = [-w, w] \times [w, w],\ w = 2$ such that, for $f(\boldsymbol{x}) = \sin(x^1)$, we have

$$\Delta u = f \quad \forall \boldsymbol{x} \in \Omega$$

and

$$u = 0 \quad \forall \boldsymbol{x} \in \Gamma$$

(namely all of our boundaries are Dirichlet boundaries).

1. For each cloud of the cloud in the flow chart identify which lines of code and in which file we are performing the operation. You will likely have to go through all the code provided.

   Solution :

   (a) Create Problem Data : problem_2.py: lines 30-37

   (b) Create Connectivity : problem_2.py: lines 39-44

   (c) Allocate global arrays : problem_2.py: lines 60-62

   (d) Loop over each element: assembly.py: lines 56-73

   (e) Initialize $\boldsymbol{k}^{=}\mathbf{0}$, $\boldsymbol{f}^e = \mathbf{0}$: poisson.py: lines 35-39

   (f) Loop through quadrature points : poisson.py; lines 45-71

   (g) Add contributions to $\boldsymbol{k}^e$ and $\boldsymbol{f}^e$: poisson.py: lines 60-71

   (h) Assemble $\boldsymbol{k}^e \to \boldsymbol{K}$ and $\boldsymbol{f}^e \to \boldsymbol{F}$: problem_2.py: line 71; assembly.py: lines 26-45

   (i) Solve System: problem_2.py: lines 75-85; apply_bc.py: lines 6-19

2. $\star$ The differential problem specific part of the code is found in `element_operations`. Now suppose we are interested in solving a similar problem but in addition to the diffusion term (Δu) we are adding a reaction term. Namely we are interested in solving: find $u \in \Omega = [-w, w] \times [w, w],\ w = 2$ such that, for $f(\boldsymbol{x}) = \sin(x^1)$, $k = 0.1$, we have

   $$\Delta u + ku = f \quad \forall \boldsymbol{x} \in \Omega$$

   and

   $$u = 0 \quad \forall \boldsymbol{x} \in \Gamma$$

(namely all of our boundaries are Dirichlet boundaries).

Perform the following steps:

(a) Derive the weak form for the above problem

Solution :
The set of trial and test functions are:

$$\mathcal{S} = \{u | u \in H^1(\Omega), u = 0 \ \ \forall \boldsymbol{x} \in \Gamma\}$$

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

The residual is:

$$R = \Delta u + ku - f = 0$$

Multiplying by the test function and integrating over the domain gives:

$$\int_\Omega \Delta uv d\Omega + \int_\Omega kuv d\Omega - \int_\Omega fv d\Omega = 0$$
$$int_\Omega \nabla \cdot (\nabla uv) d\Omega - \int_\Omega \nabla u \cdot \nabla v d\Omega + \int_\Omega kuv d\Omega - \int_\Omega fv d\Omega = 0$$
$$\int_\Gamma v(\nabla u) \cdot \boldsymbol{n} d\Gamma - \int_\Omega \nabla u \cdot \nabla v d\Omega + \int_\Omega kuv d\Omega - \int_\Omega fv d\Omega = 0$$
$$\int_\Omega \nabla u \cdot \nabla v d\Omega - \int_\Omega kuv d\Omega = - \int_\Omega fv d\Omega$$

Therefore:

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v d\Omega - \int_\Omega kuv d\Omega$$

$$F(v) = - \int_\Omega fv d\Omega$$

(b) Create a new file in `element_operations` named `diffusion_reaction.py`. Here you can effectively copy `poisson.py`.

Solution :
Create the file as instructed.

(c) Create a class inside the above file named `diffusion_reaction` that implements your derived weak form. Here the changes will be minimal and very similar to what you did for homework 6.

Solution :

We mainly modify the stiffness term:

```
# Get the value of the gradient of all basis at the quadrature point
grad_phi = []
phi = []
k_val = 0.1  # Value of the coefficient k in the problem
for k in range(num_dofs):
  # Get the base function gradients
  grad_phi.append( element.get_base_function_grad( k, gauss_points[q] ) )
  phi.append(element.get_base_function_val( k, gauss_points[q] ) )
  # Loop over all degree of freedoms
  for i in range(num_dofs):
    for j in range(num_dofs):

    # Add the contribution to the stiffness matrix
    ke[i,j] +=  np.dot( grad_phi[i] , grad_phi[j] )*jacobian*\
           gauss_weights[q]-\
           k_val*phi[i]*phi[j]*jacobian*gauss_weights[q]
```