HOMEWORK 7

CEE 361-513: Introduction to Finite Element Methods

Due: Monday Dec. 10

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`.

```
# Define the half width of the element
w = 2.

# Create the mesh with a single element of width 2w
# and centered at (3,3)
coordinates = np.array([ (-w,-w), (w,-w), (w,w), (-w,w) ])

# Connectivity
connectivity = np.array([[0,1,2,3]])

# Construct an element
element_index = 0
my_element = quadrilateral_element.element(  )  # <- fill here
```

2. One of the methods (the functions of the element object) is the function that takes a base index $i \in \{0, \dots, (\text{poly\_order} + 1)^2 - 1\}$ 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. 1 for all the basis functions . A starter code is provided in the function `part_2` in the file `problem_1.py`.

3. Read and summarize §3.3 Isoparametric Elements of 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

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

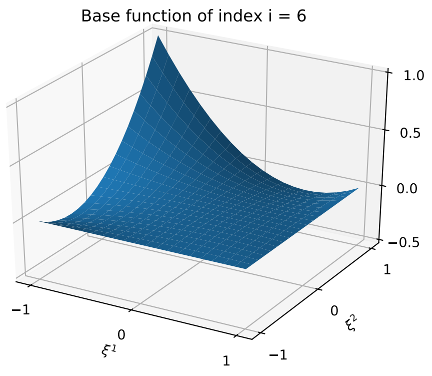
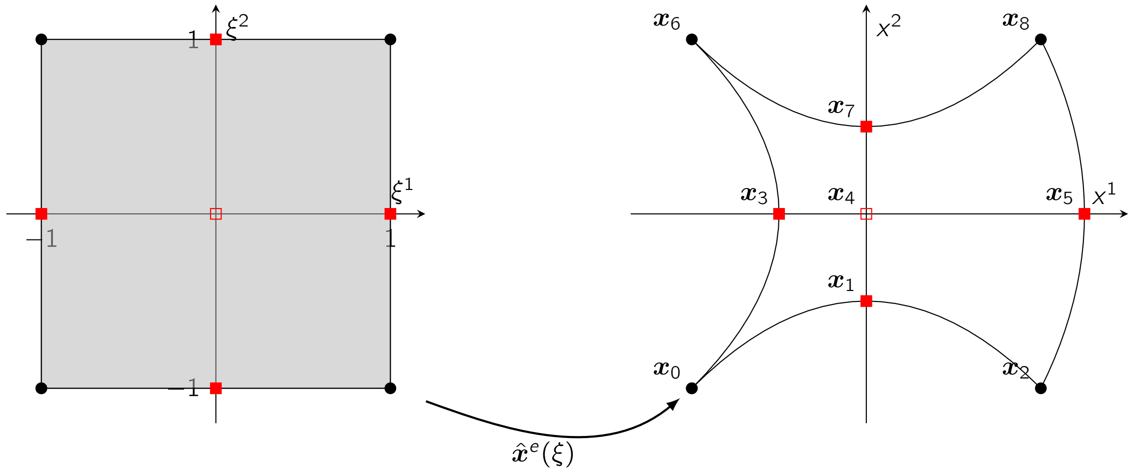


Figure 1: Sample plot of a base function

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 $\boldsymbol{x}_0, \boldsymbol{x}_2, \boldsymbol{x}_6, \boldsymbol{x}_8$ 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 68 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.

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

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

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?

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 184 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.

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.

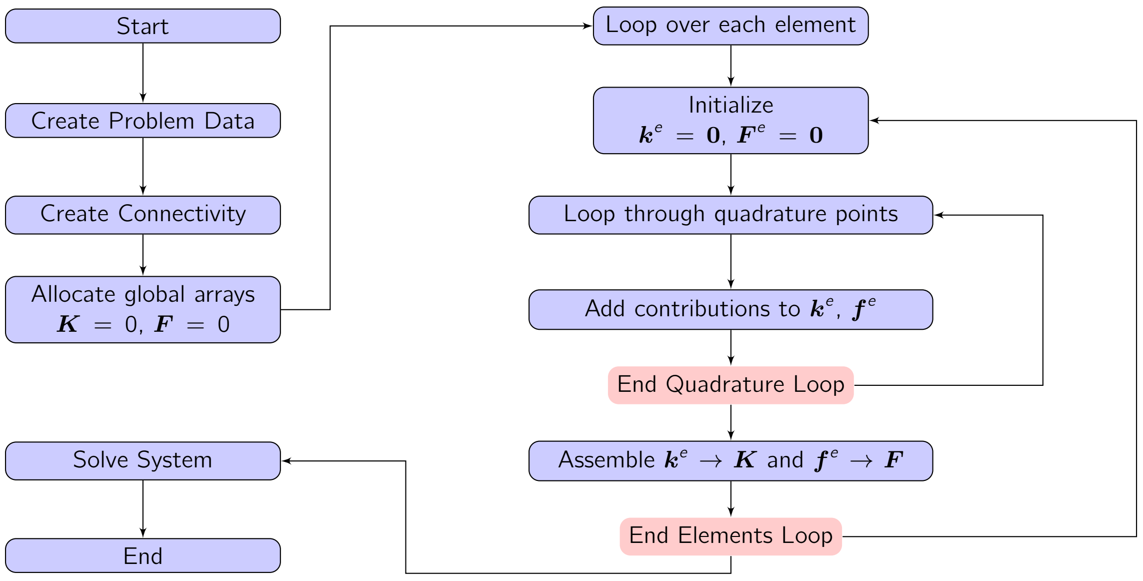

Start
Create Problem Data
Create Connectivity
Allocate global arrays $\boldsymbol{K} = 0$, $\boldsymbol{F} = 0$
Solve System
End
Loop over each element
Initialize $\boldsymbol{k}^e = \mathbf{0}$, $\boldsymbol{F}^e = \mathbf{0}$
Loop through quadrature points
Add contributions to $\boldsymbol{k}^e$, $\boldsymbol{f}^e$
End Quadrature Loop
Assemble $\boldsymbol{k}^e \to \boldsymbol{K}$ and $\boldsymbol{f}^e \to \boldsymbol{F}$
End Elements Loop

Figure 2: The flow of a finite element code

2. ⋆ 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

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

(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.