The Implementation of Finite Element Method for Poisson Equation *

Wenqiang Feng †

Abstract

This is my MATH 574 course project report. In this report, I give some details for implementing the Finite Element Method (FEM) via Matlab and Python with FEniCs. This project mainly focuses on the Poisson equation with pure homogeneous and non-homogeneous Dirichlet boundary, pure Neumann boundary condition and Mixed boundary condition on uint square and unit circle domain. Symmetric and Unsymmetric Nitsche's method will be used to deal with the non-homogeneous boundary condition. Some of the functions in this project were written for [4, 5] and some functions are from Long Chen' package [2][3]. The Python code with FEniCs are learned from [1].

1 The Model Problem

For simplicity, I consider the following three types of boundary conditions:

1. Pure Dirichlet boundary condition poisson equation:

$$\begin{cases} -\Delta u & = f \quad \text{in} \quad \Omega, \\ u & = g_D \quad \text{on} \quad \partial\Omega, \end{cases} \tag{1}$$

2. Pure Neumann boundary condition poisson equation:

$$\begin{cases} -\Delta u & = f \quad \text{in} \quad \Omega, \\ \frac{\partial u}{\partial n} & = g_N \quad \text{on} \quad \partial\Omega, \end{cases} \tag{2}$$

3. Mixed boundary condition poisson equation:

$$\begin{cases} -\Delta u & = f \quad \text{in} \quad \Omega, \\ u & = g_D \quad \text{on} \quad \Gamma_D, \\ \frac{\partial u}{\partial n} & = g_N \quad \text{on} \quad \Gamma_N. \end{cases} \tag{3}$$

where Ω is assumed to be a polygonal domain, f a given function in $L^2(\Omega)$ and g a given function in $H^{\frac{1}{2}}(\Omega)$.

Before I give the details, I would like to introduce some useful notations in this report. Let $\mathcal{T}_h$ to be the partition (Figure.1) (typically triangulation) of Ω, with piecewise constant mesh size h, i.e., $h_K = \text{diam}(K)$, similarly, to be $h_e = \text{diam}(e)$, Γ_D to be the Dirichlet boundary, Γ_N to be the

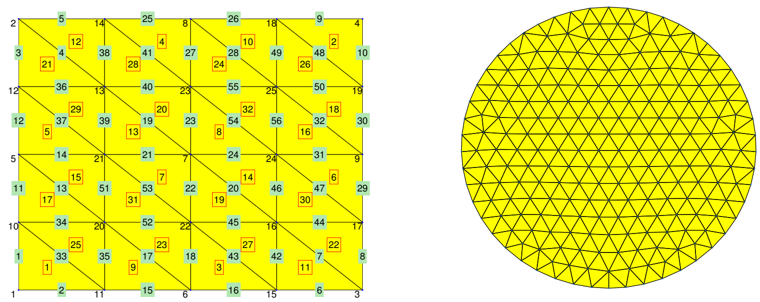

Figure 1: The triangulation on unit square and unit circle domain

Neumann boundary , and $\mathbb{P}(K)$ to be a finite dimension smooth function (typically polynomial) on the region K. This space $\mathbb{V}_h$ $(\subset H_0^1(\Omega))$ will be used to approximate the variable u.

$$\mathbb{V}_h := \{v \in L^2(\Omega) \;\; | \;\; v\,|_K \in \mathbb{P}(K) \;\; \forall K \in \mathcal{T}_h, \; v = 0 \; on \; \Gamma_D\}. \tag{4}$$

2 The weak formula and Canonical Galerkin approximation formula for model problem with

2.1 The weak formula and Galerkin approximation for the pure Dirichlet boundary problem

- Model Problem.1 : the weak formula can be written as as follows:

$$\begin{aligned} &\text{find } u \in H^1(\Omega) \text{ with } u|_{\partial\Omega} = g \text{ and} \\ &a(u,\phi) := \int_\Omega \nabla u \nabla \phi dx = \int_\Omega f \phi dx \quad \text{for} \quad \forall \phi \in H_0^1, \end{aligned} \tag{5}$$

  The Galerkin approximation formula can be read as follows:

$$\begin{aligned} &\text{find } u_h \in \mathbb{V}_h \text{ with } u_h|_{\partial\Omega} = g \text{ and} \\ &a(u_h,\phi_h) := \int_\Omega \nabla u_h \nabla \phi_h dx = \int_\Omega f_h \phi_h dx \quad \text{for} \quad \forall \phi_h \in \mathbb{V}_h, \end{aligned} \tag{6}$$

- Model Problem.2 : the weak formula can be written as as follows:

$$\begin{aligned} &\text{find } u \in H^1(\Omega) \text{ and} \\ &a(u,\phi) := \int_\Omega \nabla u \nabla \phi dx = \int_\Omega f \phi dx + \int_{\partial\Omega} g_N \phi ds \quad \text{for} \quad \forall \phi \in \mathbb{V}_h, \end{aligned} \tag{7}$$

*Key words: FEM, Pure Dirichlet boundary condition, Pure Neumann boundary condition, Mixed boundary condition, Symmetric and Unsymmetric Nitsche's method.

†Department of Mathematics,University of Tennessee, Knoxville, TN, 37909, wfeng@math.utk..edu

The Galerkin approximation formula can be read as follows:

$$\begin{aligned}&\text{find } u_h \in \mathbb{V}_h \text{ with } u_h|_{\partial\Omega} = g \text{ and}\\ &a(u_h,\phi_h) := \int_\Omega \nabla u_h \nabla \phi_h dx = \int_\Omega f_h \phi_h dx + \int_{\partial\Omega} (g_N)_k \, \phi_h ds \quad \text{for} \quad \forall \phi_h \in \mathbb{V}_h,\end{aligned} \tag{8}$$

Remark 2.1. *The necessary (and also sufficient in this setting) condition for existence of a solution is*

$$\int_\Omega f dx + \int_{\partial\Omega} g_N ds = 0.$$

The necessary condition for uniqueness of a solution is

$$\int_\Omega u dx = 0.$$

We assume $f \in L^2(\Omega)$ and $g \in L^2(\partial\Omega)$ such that $\int_\Omega f\, dx + \int_{\partial\Omega} g\, ds = 0$. Thus, there exists a unique solution in the subset of $H^2(\Omega)$ consisting of zero-average functions.

Remark 2.2. *The system arising from the pure Neumann model problem is a singular system. There are three popular approaches to deal with this singular system:*

1. *Fix one degree of freedom on the boundary (strongly impose as Dirichlet point);*
2. *Lagrange multiplier method. Changing the non-constrain problem to a constrain system;*
3. *Krylov solvers. AMG for example will do a optimal job if you make sure that the coarse solve doesn't mess with the nullspace. Using an iterative solver (like Jacobi) on the coarse solve will do the trick.*

- Model Problem.3 : the weak formula can be written as as follows:

$$\begin{aligned}&\text{find } u \in H^1(\Omega) \text{ and}\\ &a(u,\phi) := \int_\Omega \nabla u \nabla \phi dx = \int_\Omega f \phi dx + \int_{\Gamma_N} g_N \phi ds \quad \text{for} \quad \forall \phi \in \mathbb{V}_h,\end{aligned} \tag{9}$$

The Galerkin approximation formula can be read as follows:

$$\begin{aligned}&\text{find } u_h \in \mathbb{V}_h \text{ with } u_h|_{\Gamma_D} = g_D \text{ and}\\ &a(u_h,\phi_h) := \int_\Omega \nabla u_h \nabla \phi_h dx = \int_\Omega f_h \phi_h dx + \int_{\Gamma_N} (g_N)_k \, \phi_h ds \quad \text{for} \quad \forall \phi_h \in \mathbb{V}_h,\end{aligned} \tag{10}$$

2.2 The weak formula for Symmetric and Unsymmetric Nitsche's method

2.2.1 Symmetric Nitsche's method

1. The weak formula for Model Problem.1 can be written as as follows:

$$\text{find } u \in H^1(\Omega) \text{ and}$$

$$a(u,\phi) := \int_\Omega \nabla u \nabla \phi dx - \sum_{e\in\Gamma_D} \int_e \nabla u \cdot n \phi ds - \sum_{e\in\Gamma_D} \int_e \nabla \phi \cdot n u ds + \frac{\gamma}{h_e} \sum_{e\in\Gamma_D} \int_e u v ds \tag{11}$$

$$f(\phi) = \int_\Omega f \phi dx - \sum_{e\in\Gamma_D} \int_e \nabla \phi \cdot n u ds + \frac{\gamma}{h_e} \sum_{e\in\Gamma_D} \int_e u v ds \quad \text{for} \quad \forall \phi \in \mathbb{V}_h, \tag{12}$$

2. The weak formula for Model Problem.3 can be written as as follows:

$$
\begin{aligned}
&\text{find } u \in H^1(\Omega) \text{ and} \\
&a(u,\phi) := \int_\Omega \nabla u \nabla \phi dx - \sum_{e\in\Gamma_D}\int_e \nabla u \cdot n\phi ds - \sum_{e\in\Gamma_D}\int_e \nabla\phi \cdot nuds + \frac{\gamma}{h_e}\sum_{e\in\Gamma_D}\int_e uvds \qquad (13)\\
&f(\phi) = \int_\Omega f\phi dx + \sum_{e\in\Gamma_N}\int_e g_N\phi ds - \sum_{e\in\Gamma_D}\int_e \nabla\phi\cdot nuds + \frac{\gamma}{h_e}\sum_{e\in\Gamma_D}\int_e uvds \quad \text{for} \quad \forall\phi\in\mathbb{V}_h, \qquad (14)
\end{aligned}
$$

2.2.2 Unsymmetric Nitsche's method

1. The weak formula for Model Problem.1 can be written as as follows:

$$
\begin{aligned}
&\text{find } u \in H^1(\Omega) \text{ and} \\
&a(u,\phi) := \int_\Omega \nabla u \nabla \phi dx - \sum_{e\in\Gamma_D}\int_e \nabla u \cdot n\phi ds + \sum_{e\in\Gamma_D}\int_e \nabla\phi \cdot nuds + \frac{\gamma}{h_e}\sum_{e\in\Gamma_D}\int_e uvds \qquad (15)\\
&f(\phi) = \int_\Omega f\phi dx + \sum_{e\in\Gamma_D}\int_e \nabla\phi\cdot nuds + \frac{\gamma}{h_e}\sum_{e\in\Gamma_D}\int_e uvds \quad \text{for} \quad \forall\phi\in\mathbb{V}_h, \qquad (16)
\end{aligned}
$$

2. The weak formula for Model Problem.3 can be written as as follows:

$$
\begin{aligned}
&\text{find } u \in H^1(\Omega) \text{ and} \\
&a(u,\phi) := \int_\Omega \nabla u \nabla \phi dx - \sum_{e\in\Gamma_D}\int_e \nabla u \cdot n\phi ds + \sum_{e\in\Gamma_D}\int_e \nabla\phi \cdot nuds + \frac{\gamma}{h_e}\sum_{e\in\Gamma_D}\int_e uvds \qquad (17)\\
&f(\phi) = \int_\Omega f\phi dx + \sum_{e\in\Gamma_N}\int_e g_N\phi ds + \sum_{e\in\Gamma_D}\int_e \nabla\phi\cdot nuds + \frac{\gamma}{h_e}\sum_{e\in\Gamma_D}\int_e uvds \quad \text{for} \quad \forall\phi\in\mathbb{V}_h, \qquad (18)
\end{aligned}
$$

2.3 Poisson Solver

2.3.1 Data structure

Before I give the Poisson solver, I would like to introduce the data structure in Matlab. I will use the initial mesh (Figure.2) as an example to illustrate the concept of the components.

1. The basic data structure (See Table (1)) is mesh which contains
   ⋄mesh.node: The *node* vector is just the xy-value of node.
   ⋄mseh.elem: In the *elem* matrix, the first and the second column represent the start nodal indices vector and the end nodal indices vector, respectively.
   ⋄mesh.Dirichlet: The *Dirichlet* is the Dirichlet boundary condition edges.
   ⋄mesh.Neumann: The *boundary* is the Neumann boundary condition edges.

2. Another main data structure is indices (See Table (2))which will provide useful indices.
   ⋄ indices.neighbor: *indices.neighbor(1:NT,1:3)*: the indices map of neighbor information of elements, where neighbor(t,i) is the global index of the element opposite to the i-th vertex of the t-th element.
   ⋄ indices.elem2edge: *indices.elem2edge(1:NT,1:3)*: the indices map from elements to edges, elem2edge(t,i) is the edge opposite to the i-th vertex of the t-th element.

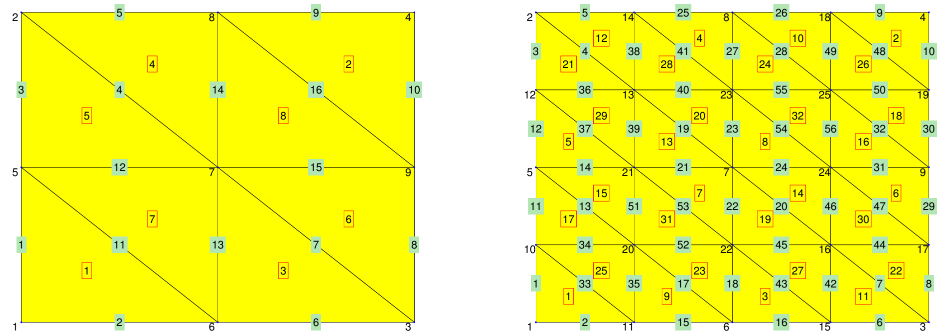

Figure 2: The initial mesh and the uniformly refinement hierarchical mesh

⋄ indices.edge: *indices.edge(1:NE,1:2)*: all edges, where edge(e,i) is the global index of the i-th vertex of the e-th edge, and edge(e,1) < edge(e,2).
⋄ indices.bdEdge: *indices.bdEdge(1:Nbd,1:2)*: boundary edges with positive oritentation, where bdEdge(e,i) is the global index of the i-th vertex of the e-th edge for i=1,2. The positive orientation means that the interior of the domain is on the left moving from bdEdge(e,1) to bdEdge(e,2). Note that this requires elem is positive ordered, i.e., the signed area of each triangle is positive. If not, use elem = fixorder(node,elem) to fix the order.
⋄ indices.edge2elem: *indices.edge2elem(1:NE,1:4)*: the indices map from edge to element, where edge2elem(e,1:2) are the global indices of two elements sharing the e-th edge, and edge2elem(e,3:4) are the local indices of e (See Figure. 3 and Table. 2).

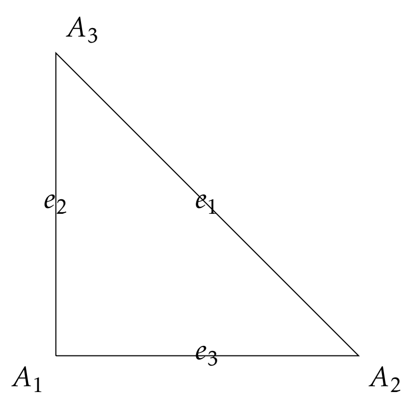

Figure 3: The local indices of vetrices and edges .

2.3.2 Poisson solver Process

The Canonical Poisson solver contains three main steps:

⋄*Step 1* Pre-process step: In this phase, we should get the information of the nodal, element and indices. In this project, I use my own function *InitialMesh* to generate for the square domain. For the complex domain, you can use Matlab's PDE tool or the package in [6] to generate the

Table 1: MESH Data structure in two dimension

Nodal NO.	node(:,1)	node(:,2)
1	-1	-1
2	-1	0
3	-1	1
4	0	-1
5	0	0
6	0	1
7	1	-1
8	1	0
9	1	1

i	Dirichlet(:,1)	Dirichlet(:,2)
	1	2
	1	4
	2	3
	3	6
	4	7
	6	9
	7	8
	8	9

	Element			neighbor			elem2edge		
NO.	1	2	3	1	2	3	1	2	3
1	1	4	2	2	1	1	4	1	2
2	5	2	4	1	5	3	4	8	5
3	2	5	3	4	3	2	6	3	5
4	6	3	5	3	7	4	6	10	7
5	4	7	5	6	2	5	11	8	9
6	8	5	7	5	6	7	11	15	12
7	5	8	6	8	4	6	13	10	12
8	9	6	8	7	8	8	13	16	14

Table 2: Indices data structure in two dimension

Edge NO.	edge edge(:,1)	edge(:,2)	EdgeNO.	elem 1	edge2elem elem 2	local	local
1	1	2	1	1	1	2	2
2	1	4	2	1	1	3	3
3	2	3	3	3	3	2	2
4	2	4	4	1	2	1	1
5	2	5	5	2	3	3	3
6	3	5	6	3	4	2	2
7	3	6	7	4	4	3	3
8	4	5	8	2	5	2	2
9	4	7	9	5	5	3	3
10	5	6	10	4	7	2	2
11	5	7	11	5	6	1	1
12	5	8	12	6	7	3	3
13	6	8	13	7	8	1	1
14	6	9	14	8	8	3	3
15	7	8	15	6	6	2	2
16	8	9	16	8	8	2	2

Boundary Element	BdEdge NO.	bdedge(:,1)	bdedge(:,2)
1	1	2	1
1	2	3	2
3	3	7	8
4	4	8	9
5	5	1	4
8	6	6	3
6	7	4	7
8	8	9	6

mesh.

⋄*Step 2* Process step: This phase contains four sub-steps as follow:

⋄*step 2.1* Compute the stiffness matrix and load vector of the elements

⋄*step 2.2* Assemble the global stiffness matrix and global load vector

⋄*step 2.3* Deal with the boundary condition (Strongly impose the boundary)

⋄*step 2.4* Solve the linear system $AU = F$

⋄*Step 3* post-process step: this phase is just to output the solution and give it visual form.

3 Templates

3.1 Basis functions template

Listing 1: Quadrature points in 1-D with barycentric coordinates

```
%Basis functions of P1
phi(:,1) = lambda(:,1);
phi(:,2) = lambda(:,2);
phi(:,3) = lambda(:,3);

% Gradient Basis functions of P1
Dphip(:,:,1) = 1;
Dphip(:,:,2) = 1;
Dphip(:,:,3) = 1;

%Basis functions of P2
phi(:,1) = lambda(:,1).*(2*lambda(:,1)-1);
phi(:,2) = lambda(:,2).*(2*lambda(:,2)-1);
phi(:,3) = lambda(:,3).*(2*lambda(:,3)-1);
phi(:,4) = 4*lambda(:,2).*lambda(:,3);
phi(:,5) = 4*lambda(:,3).*lambda(:,1);
phi(:,6) = 4*lambda(:,1).*lambda(:,2);

% Gradient Basis functions of P2
Dphip(:,:,1) = (4*lambda(p,1)-1).*Dlambda(:,:,1);
Dphip(:,:,2) = (4*lambda(p,2)-1).*Dlambda(:,:,2);
Dphip(:,:,3) = (4*lambda(p,3)-1).*Dlambda(:,:,3);
Dphip(:,:,4) = 4*(lambda(p,2)*Dlambda(:,:,3)+lambda(p,3)*Dlambda(:,:,2));
Dphip(:,:,5) = 4*(lambda(p,3)*Dlambda(:,:,1)+lambda(p,1)*Dlambda(:,:,3));
Dphip(:,:,6) = 4*(lambda(p,1)*Dlambda(:,:,2)+lambda(p,2)*Dlambda(:,:,1));


% Basis functions of P3
phi(:,1) = 0.5*(3*lambda(:,1)-1).*(3*lambda(:,1)-2).*lambda(:,1);
phi(:,2) = 0.5*(3*lambda(:,2)-1).*(3*lambda(:,2)-2).*lambda(:,2);
phi(:,3) = 0.5*(3*lambda(:,3)-1).*(3*lambda(:,3)-2).*lambda(:,3);
phi(:,4) = 9/2*lambda(:,3).*lambda(:,2).*(3*lambda(:,2)-1);
phi(:,5) = 9/2*lambda(:,3).*lambda(:,2).*(3*lambda(:,3)-1);
phi(:,6) = 9/2*lambda(:,1).*lambda(:,3).*(3*lambda(:,3)-1);
phi(:,7) = 9/2*lambda(:,1).*lambda(:,3).*(3*lambda(:,1)-1);
phi(:,8) = 9/2*lambda(:,1).*lambda(:,2).*(3*lambda(:,1)-1);
phi(:,9) = 9/2*lambda(:,1).*lambda(:,2).*(3*lambda(:,2)-1);
phi(:,10) = 27*lambda(:,1).*lambda(:,2).*lambda(:,3);
```

```
% Gradient Basis functions of P3
Dphip(:,:,1) = (27/2*lambda(p,1)*lambda(p,1)-9*lambda(p,1)+1).*Dlambda
    (:,:,1);
Dphip(:,:,2) = (27/2*lambda(p,2)*lambda(p,2)-9*lambda(p,2)+1).*Dlambda
    (:,:,2);
Dphip(:,:,3) = (27/2*lambda(p,3)*lambda(p,3)-9*lambda(p,3)+1).*Dlambda
    (:,:,3);
Dphip(:,:,4) = 9/2*((3*lambda(p,2)*lambda(p,2)-lambda(p,2)).*Dlambda(:,:,3)
    +...
                    lambda(p,3)*(6*lambda(p,2)-1).*Dlambda(:,:,2));
Dphip(:,:,5) = 9/2*((3*lambda(p,3)*lambda(p,3)-lambda(p,3)).*Dlambda(:,:,2)
    +...
                    lambda(p,2)*(6*lambda(p,3)-1).*Dlambda(:,:,3));
Dphip(:,:,6) = 9/2*((3*lambda(p,3)*lambda(p,3)-lambda(p,3)).*Dlambda(:,:,1)
    +...
                    lambda(p,1)*(6*lambda(p,3)-1).*Dlambda(:,:,3));
Dphip(:,:,7) = 9/2*((3*lambda(p,1)*lambda(p,1)-lambda(p,1)).*Dlambda(:,:,3)
    +...
                    lambda(p,3)*(6*lambda(p,1)-1).*Dlambda(:,:,1));
Dphip(:,:,8) = 9/2*((3*lambda(p,1)*lambda(p,1)-lambda(p,1)).*Dlambda(:,:,2)
    +...
                    lambda(p,2)*(6*lambda(p,1)-1).*Dlambda(:,:,1));
Dphip(:,:,9)  = 9/2*((3*lambda(p,2)*lambda(p,2)-lambda(p,2)).*Dlambda
    (:,:,1)+...
                    lambda(p,1)*(6*lambda(p,2)-1).*Dlambda(:,:,2));
Dphip(:,:,10) = 27*(lambda(p,1)*lambda(p,2)*Dlambda(:,:,3)+lambda(p,1)*
    lambda(p,3)*Dlambda(:,:,2)+...
                    lambda(p,3)*lambda(p,2)*Dlambda(:,:,1));
```

3.2 Quadrature points in 1-D with barycentric coordinates

```
function [lambda,weight] = quadpts1d(numPts)
%% QUADPTS1 quadrature points in 1-D with Bar.

if numPts > 8
   fprintf('No gauss quadrature for this case')
end

switch numPts
    case 1
        lambda =[0.500000000000000   0.500000000000000];
        weight = 1;

    case 2
        lambda =[0.788675134594813   0.211324865405187;
                 0.211324865405187   0.788675134594813];
        weight =[0.500000000000000;
                 0.500000000000000];

    case 3
        lambda =[0.500000000000000   0.500000000000000;
                 0.887298334620742   0.112701665379258;
                 0.112701665379258   0.887298334620742];
```

```
        weight =[0.444444444444444;
                 0.277777777777778;
                 0.277777777777778];

    case 4
        lambda =[0.669990521792428   0.330009478207572;
                 0.930568155797026   0.069431844202974;
                 0.330009478207572   0.669990521792428;
                 0.069431844202974   0.930568155797026];
        weight =[0.326072577431273;
                 0.173927422568727;
                 0.326072577431273;
                 0.173927422568727];

    case 5
        lambda =[0.500000000000000   0.500000000000000;
                 0.769234655052841   0.230765344947159;
                 0.953089922969332   0.046910077030668;
                 0.230765344947158   0.769234655052841;
                 0.046910077030668   0.953089922969332];

        weight =[0.284444444444444;
                 0.239314335249683;
                 0.118463442528095;
                 0.239314335249683;
                 0.118463442528095];

    case 6

        lambda =[0.619309593041598   0.380690406958402;
                 0.830604693233132   0.169395306766868;
                 0.966234757101576   0.033765242898424;
                 0.380690406958402   0.619309593041598;
                 0.169395306766868   0.830604693233132;
                 0.033765242898424   0.966234757101576];

        weight =[0.233956967286346;
                 0.180380786524069;
                 0.085662246189585;
                 0.233956967286346;
                 0.180380786524069;
                 0.085662246189585];

    case 7
        lambda =[0.500000000000000   0.500000000000000;
                 0.702922575688699   0.297077424311301;
                 0.870765592799697   0.129234407200303;
                 0.974553956171379   0.025446043828621;
                 0.297077424311301   0.702922575688699;
                 0.129234407200303   0.870765592799697;
                 0.025446043828621   0.974553956171379];

        weight =[0.208979591836735;
                 0.190915025252559;
                 0.139852695744638;
                 0.064742483084435;
```

```
                 0.190915025252559;
                 0.139852695744638;
                 0.064742483084435];

    case 8
        lambda =[0.591717321247825   0.408282678752175;
                 0.762766204958164   0.237233795041836;
                 0.898333238706813   0.101666761293187;
                 0.980144928248768   0.019855071751232;
                 0.408282678752175   0.591717321247825;
                 0.237233795041836   0.762766204958164;
                 0.101666761293187   0.898333238706813;
                 0.019855071751232   0.980144928248768];

        weight =[0.181341891689181;
                 0.156853322938944;
                 0.111190517226687;
                 0.050614268145188;
                 0.181341891689181;
                 0.156853322938944;
                 0.111190517226687;
                 0.050614268145188];
end
```

3.3 Quadrature points in 2-D with barycentric coordinates

Listing 2: Quadrature points in 2-D with barycentric coordinates

```
function [lambda,weight] = quadpts2d(order)
%% QUADPTS1 quadrature points in 2-D with barycentric coordinates.
% References:
%
% David Dunavant.
%    High degree efficient symmetrical Gaussian quadrature rules for
%    the triangle. International journal for numerical methods in
%    engineering. 21(6):1129--1148, 1985.
if order > 6
   fprintf('No gauss quadrature for this case')
end

switch order
    case 1
        lambda =[0.333333333333333   0.333333333333333
            0.333333333333333];
        weight = 1;

    case 2
        lambda =[0.666666666666667   0.166666666666667   0.166666666666667;
                 0.166666666666667   0.666666666666667   0.166666666666667;
                 0.166666666666667   0.166666666666667
                     0.666666666666667];
        weight =[0.333333333333333;
                 0.333333333333333;
                 0.333333333333333];
```

```
    case 3
        lambda =[0.333333333333333   0.333333333333333   0.333333333333333;
                 0.600000000000000   0.200000000000000   0.200000000000000;
                 0.200000000000000   0.600000000000000   0.200000000000000;
                 0.200000000000000   0.200000000000000
                     0.600000000000000];
        weight =[-0.562500000000000;
                 0.520833333333333;
                 0.520833333333333;
                 0.520833333333333];

    case 4
        lambda =[0.108103018168070   0.445948490915965   0.445948490915965;
                 0.445948490915965   0.108103018168070   0.445948490915965;
                 0.445948490915965   0.445948490915965   0.108103018168070;
                 0.816847572980459   0.091576213509771   0.091576213509771;
                 0.091576213509771   0.816847572980459   0.091576213509771;
                 0.091576213509771   0.091576213509771
                     0.816847572980459];
        weight =[0.223381589678011;
                 0.223381589678011;
                 0.223381589678011;
                 0.109951743655322;
                 0.109951743655322;
                 0.109951743655322];

    case 5
        lambda =[0.333333333333333   0.333333333333333   0.333333333333333;
                 0.059715871789770   0.470142064105115   0.470142064105115;
                 0.470142064105115   0.059715871789770   0.470142064105115;
                 0.470142064105115   0.470142064105115   0.059715871789770;
                 0.797426985353087   0.101286507323456   0.101286507323456;
                 0.101286507323456   0.797426985353087   0.101286507323456;
                 0.101286507323456   0.101286507323456
                     0.797426985353087];

        weight =[0.225000000000000
                 0.132394152788506
                 0.132394152788506
                 0.132394152788506
                 0.125939180544827
                 0.125939180544827
                 0.125939180544827];

    case 6

        lambda =[0.249286745170910   0.249286745170910   0.501426509658180;
                 0.249286745170910   0.501426509658179   0.249286745170911;
                 0.501426509658179   0.249286745170910   0.249286745170911;
                 0.063089014491502   0.063089014491502   0.873821971016996;
                 0.063089014491502   0.873821971016996   0.063089014491502;
                 0.873821971016996   0.063089014491502   0.063089014491502;
                 0.310352451033784   0.636502499121399   0.053145049844817;
                 0.636502499121399   0.053145049844817   0.310352451033784;
                 0.053145049844817   0.310352451033784   0.636502499121399;
```

```
                   0.636502499121399    0.310352451033784    0.053145049844817;
                   0.310352451033784    0.053145049844817    0.636502499121399;
                   0.053145049844817    0.636502499121399
                       0.310352451033784];

         weight =[0.116786275726379
                   0.116786275726379
                   0.116786275726379
                   0.050844906370207
                   0.050844906370207
                   0.050844906370207
                   0.082851075618374
                   0.082851075618374
                   0.082851075618374
                   0.082851075618374
                   0.082851075618374
                   0.082851075618374];

end
```

4 Numerical Experiments

4.1 Canonical Finite Element Method

4.1.1 Test. 1

1. **Square domain** In the first test, we choose the data such that the exact solution of (1) on the square domain $\Omega = [-1,1] \times [-1,1]$ is given by

$$u(x,y) = \cos(\pi x)\cos(\pi y).$$

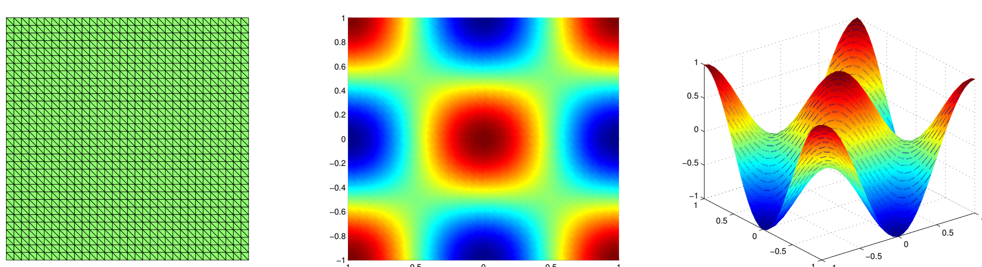

Figure 4: The canonical FEM approximation on square domain with pure Dirichlet boundary

The errors for the FEM approximation (Fig.7) using $q = 1,2,3$ on unit square domain and varying h can be found in Table (3).

2. **Unit circle domain**

The errors for the FEM approximation (Fig.5) using $q = 1,2,3$ on unit square domain and varying h can be found in Table (4).

Table 3: Errors of the computed solution on square domain in Test 1.

	#elem	$\|u-u_h\|_{L^2}$	order	$\|u-u_h\|_{H^1}$	order
$q=1$	32	5.58312×10^{-1}		4.87956×10^{0}	
	128	4.56716×10^{-1}	0.2898	2.99068×10^{0}	0.7063
	512	1.52379×10^{-1}	1.5836	1.67526×10^{0}	0.8361
	2048	4.13290×10^{-2}	1.8824	8.63415×10^{-1}	0.9563
	8192	1.05529×10^{-2}	1.9695	4.35052×10^{-1}	0.9889
	32768	2.65236×10^{-3}	1.9923	2.17948×10^{-1}	0.9972
$q=2$	32	3.19977×10^{-1}		1.90646×10^{0}	
	128	6.18139×10^{-2}	2.3720	9.35270×10^{-1}	1.0274
	512	7.45408×10^{-3}	3.0518	2.58878×10^{-1}	1.8531
	2048	9.17985×10^{-4}	3.0215	6.67841×10^{-2}	1.9547
	8192	1.14367×10^{-4}	3.0048	1.68392×10^{-2}	1.9877
	32768	1.42863×10^{-5}	3.0010	4.21912×10^{-3}	1.9968
$q=3$	32	2.36147×10^{-1}		9.85850×10^{-1}	
	128	1.45057×10^{-2}	4.0250	1.44741×10^{-1}	2.7679
	512	8.94294×10^{-4}	4.0197	1.81811×10^{-2}	2.9930
	2048	5.34187×10^{-5}	4.0653	2.23215×10^{-3}	3.0259
	8192	3.26047×10^{-6}	4.0342	2.75868×10^{-4}	3.0164
	32768	2.01648×10^{-7}	4.0152	3.42841×10^{-5}	3.0084

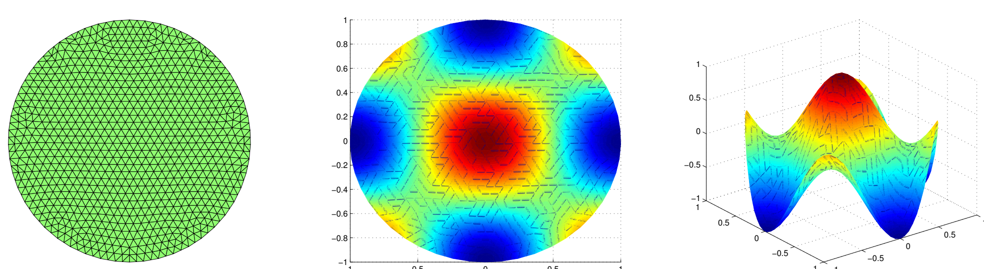

Figure 5: The canonical FEM approximation on unit circle domain with pure Dirichlet boundary

Table 4: Errors of the computed solution on unit circle domain in Test 1.

	#*elem*	$\|u-u_h\|_{L^2}$	order	$\|u-u_h\|_{H^1}$	order
$q=1$	6464	4.88656×10^{-3}		3.11195×10^{-1}	
	25856	1.22587×10^{-3}	1.9950	1.55775×10^{-1}	0.9984
	103424	3.06784×10^{-4}	1.9985	7.79145×10^{-2}	0.9995
	413696	7.67189×10^{-5}	1.9996	3.89612×10^{-2}	0.9999
	1654784	1.91814×10^{-5}	1.9999	1.94812×10^{-2}	1.0000
	6619136	4.79545×10^{-6}	2.0000	9.74067×10^{-3}	1.0000
$q=2$	6464	6.97589×10^{-5}		9.40105×10^{-3}	
	25856	8.72684×10^{-6}	2.9988	2.35466×10^{-3}	1.9973
	103424	1.09144×10^{-6}	2.9992	5.89078×10^{-4}	1.9990
	413696	1.36475×10^{-7}	2.9995	1.47313×10^{-4}	1.9996
	1654784	1.70625×10^{-8}	2.9997	3.68330×10^{-5}	1.9998
	6619136	2.13304×10^{-9}	2.9998	9.20882×10^{-6}	1.9999
$q=3$	6464	1.55711×10^{-6}		1.34110×10^{-4}	
	25856	9.65997×10^{-8}	4.0107	1.67345×10^{-5}	3.0025
	103424	6.01450×10^{-9}	4.0055	2.08997×10^{-6}	3.0013
	413696	3.75178×10^{-10}	4.0028	2.61131×10^{-7}	3.0006
	1654784	2.34006×10^{-11}	4.0030	3.26342×10^{-8}	3.0003
	6619136	4.63512×10^{-12}		4.07888×10^{-9}	3.0001

4.1.2 Test. 2

In the second test, we choose the data such that the exact solution of (3) on the square domain $\Omega = [-1,1]\times[-1,1]$ is given by

$$u(x,y) = \cos(\pi x)\cos(\pi y),$$

where the mixed boundary condition are

$$\begin{aligned} u(x,y) &= g_D, \quad x=-1 \text{ and } x=1 \\ u(x,y) &= g_N = 0, \quad y=-1 \text{ and } y=1 \end{aligned}$$

The errors for the FEM approximation using $r = 1,2,3$ and varying h can be found in Table (5).

Table 5: Errors of the computed solution on square domain in Test 2.

	$\#elem$	$\|u-u_h\|_{L^2}$	order	$\|u-u_h\|_{H^1}$	order
$q=1$	32	4.41951×10^{-1}		4.75239×10^{0}	
	128	4.38083×10^{-1}	0.0127	2.96046×10^{0}	0.6828
	512	1.49298×10^{-1}	1.5530	1.66891×10^{0}	0.8269
	2048	4.07246×10^{-2}	1.8742	8.62520×10^{-1}	0.9523
	8192	1.04122×10^{-2}	1.9676	4.34937×10^{-1}	0.9878
	32768	2.61781×10^{-3}	1.9918	2.17934×10^{-1}	0.9969
$q=2$	32	2.71445×10^{-1}		1.86894×10^{0}	
	128	6.06297×10^{-2}	2.1626	9.18462×10^{-1}	1.0249
	512	7.32915×10^{-3}	3.0483	2.56647×10^{-1}	1.8394
	2048	9.10672×10^{-4}	3.0086	6.65096×10^{-2}	1.9482
	8192	1.13981×10^{-4}	2.9981	1.68051×10^{-2}	1.9847
	32768	1.42650×10^{-5}	2.9982	4.21487×10^{-3}	1.9953
$q=3$	32	2.27969×10^{-1}		9.34541×10^{-1}	
	128	1.40805×10^{-2}	4.0171	1.39645×10^{-1}	2.7425
	512	8.78339×10^{-4}	4.0028	1.77623×10^{-2}	2.9749
	2048	5.27990×10^{-5}	4.0562	2.20197×10^{-3}	3.0120
	8192	3.23758×10^{-6}	4.0275	2.73863×10^{-4}	3.0073
	32768	2.00799×10^{-7}	4.0111	3.41554×10^{-5}	3.0033

4.2 Symmetric and Unsymmetric Nitsche's method

4.2.1 Test. 1

In the first test, we choose the data such that the exact solution of (1) on the unit domain $\Omega = [0,1]\times[0,1]$ is given by

$$u(x) = xy + \sin(\pi x) sin(\pi y).$$

1. **Symmetric Nitsche's method approximation**
   The errors for the symmetric Nitsche's method approximation (6) using $r = 1, 2, 3$ and varying h can be found in Table (6).

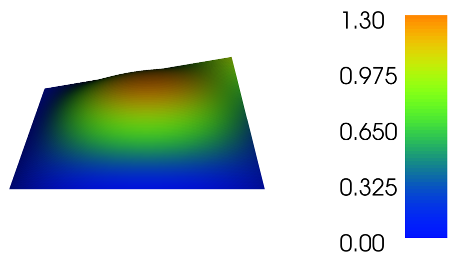

Figure 6: The symmetric Nitsche's method approximation on square domain with pure Dirichlet boundary

Table 6: Errors of the computed solution (symmetric Nitsche's method) on square domain in Test 1.

	$\#elem$	$\|\|u-u_h\|\|_{L^2}$	order	$\|\|u-u_h\|\|_{H^1}$	order
$q=1$	128	2.67237×10^{-2}	1.5056	6.03088×10^{-1}	0.8117
	512	7.53495×10^{-3}	1.6660	2.75422×10^{-1}	0.9712
	2048	1.97027×10^{-3}	1.7557	1.23602×10^{-1}	1.0328
	8192	5.02548×10^{-4}	1.8096	5.73406×10^{-2}	1.0516
	32768	1.26861×10^{-4}	1.8449	2.74357×10^{-2}	1.0540
	131072	3.18684×10^{-5}	1.8696	1.33937×10^{-2}	1.0507
	524288	7.98636×10^{-6}	1.8877	6.61383×10^{-3}	1.0461
$q=2$	128	9.12212×10^{-4}	3.3725	4.60368×10^{-2}	2.2815
	512	9.64065×10^{-5}	3.3073	1.03643×10^{-2}	2.2163
	2048	1.02791×10^{-5}	3.2813	2.32636×10^{-3}	2.1960
	8192	1.18421×10^{-6}	3.2404	5.55349×10^{-4}	2.1637
	32768	1.4137×10^{-7}	3.2056	1.35441×10^{-4}	2.1381
	131072	1.72858×10^{-8}	3.1767	3.34266×10^{-5}	2.1182
	524288	1.97191×10^{-9}	3.1703	8.30185×10^{-6}	2.1027

2. **Unsymmetric Nitsche's method approximation** The errors for the unsymmetric Nitsche's method approximation using $r = 1, 2, 3$ and varying h can be found in Table (7).

4.2.2 Test. 2

In the second test, we choose the data such that the exact solution of (3) on the unit circle domain is given by

$$u(x, y) = \sin(\pi(x^2 + y^2))cos(\pi(x - y)).$$

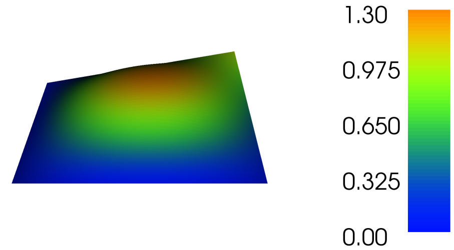

Figure 7: The unsymmetric Nitsche's method approximation on square domain with pure Dirichlet boundary

Table 7: Errors of the computed solution (unsymmetric Nitsche's method) on square domain in Test 1.

	$\#elem$	$\|\|u-u_h\|\|_{L^2}$	order	$\|\|u-u_h\|\|_{H^1}$	order
$q=1$	128	2.55734×10^{-2}	1.4155	4.40466×10^{-1}	0.9252
	512	7.92932×10^{-3}	1.5525	2.17000×10^{-1}	0.9733
	2048	2.19209×10^{-3}	1.6533	1.06718×10^{-1}	0.9901
	8192	5.74701×10^{-4}	1.7228	5.28208×10^{-2}	0.9963
	32768	1.47022×10^{-4}	1.7716	2.62667×10^{-2}	0.9986
	131072	3.71744×10^{-5}	1.8069	1.30964×10^{-2}	0.9995
	524288	9.34603×10^{-6}	1.8334	6.53886×10^{-3}	0.9999
$q=2$	128	1.79718×10^{-3}	2.8185	3.40591×10^{-2}	1.9840
	512	2.28582×10^{-4}	2.8967	8.49843×10^{-3}	1.9934
	2048	2.84656×10^{-5}	2.9329	2.11907×10^{-3}	1.9969
	8192	3.53995×10^{-6}	2.9516	5.28854×10^{-4}	1.9983
	32768	4.40984×10^{-7}	2.9622	1.32085×10^{-4}	1.9989
	131072	5.50094×10^{-8}	2.9690	3.30042×10^{-5}	1.9992
	524288	6.62729×10^{-9}	2.9811	8.24887×10^{-6}	1.9994

where

$$u(x,y) = g_D, \quad \text{on } \Gamma_D, \; x \le 0,$$
$$u(x,y) = g_N, \quad \text{on } \Gamma_N, \; x > 0,$$

with g_D = exact solution and $g_N = x*(2*pi*x*cos(pi*(x-y))*cos(pi*(x*x+y*y)) - pi*sin(pi*(x-y))*sin(pi*(x*x+y*y))) + y*(2*pi*y*cos(pi*(x-y))*cos(pi*(x*x+y*y)) + pi*sin(pi*(x-y))*sin(pi*(x*x+y*y)))$

1. **Symmetric Nitsche's method approximation** The errors for the symmetric Nitsche's method approximation (8) using $r = 1,2,3$ and varying h can be found in Table (8).
2. **Unsymmetric Nitsche's method approximation** The errors for the unsymmetric Nitsche's method approximation (Fig.9) using $r = 1,2,3$ and varying h can be found in Table (9).

Table 8: Errors of the computed solution (symmetric Nitsche's method) on unit circle domain in Test 2.

	#elem	$\|u-u_h\|_{L^2}$	order	$\|u-u_h\|_{H^1}$	order
$q=1$	70	2.5577×10^{-0}	1.4533	5.56708×10^{0}	1.3399
	263	8.03284×10^{-1}	1.5273	2.48582×10^{0}	1.2181
	998	1.85969×10^{-1}	1.7436	1.10756×10^{0}	1.2153
	4085	4.64263×10^{-2}	1.7991	5.23983×10^{-1}	1.1745
	16117	6.34069×10^{-3}	2.0046	2.74048×10^{-1}	1.1206
	64585	2.66504×10^{-3}	1.8795	1.27762×10^{-1}	1.1177
	258133	7.32207×10^{-4}	1.8764	6.77738×10^{-2}	1.0882

Table 9: Errors of the computed solution (unsymmetric Nitsche's method) on unit circle domain in Test 2.

	#elem	$\|u-u_h\|_{L^2}$	order	$\|u-u_h\|_{H^1}$	order
$q=1$	70	2.62624×10^{-0}	1.5537	5.70487×10^{0}	1.3857
	263	8.05497×10^{-1}	1.5918	2.44824×10^{0}	1.2685
	998	1.89594×10^{-1}	1.7790	1.09123×10^{0}	1.2491
	4085	4.81340×10^{-2}	1.8192	5.18076×10^{-1}	1.1983
	16117	6.98021×10^{-3}	2.0033	2.71889×10^{-1}	1.1385
	64585	2.80584×10^{-3}	1.8891	1.27267×10^{-1}	1.1316
	258133	8.82705×10^{-4}	1.8568	6.73668×10^{-2}	1.1006

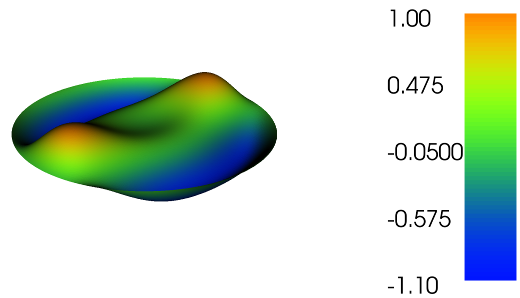

Figure 8: The symmetric Nitsche's method approximation on square domain with pure Dirichlet boundary

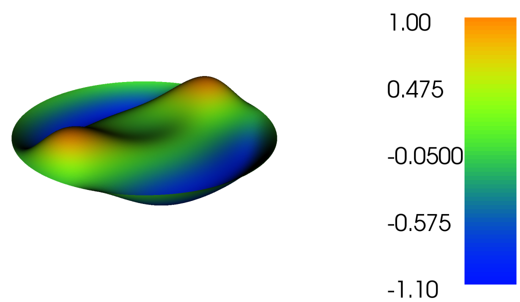

Figure 9: The unsymmetric Nitsche's method approximation on square domain with pure Dirichlet boundary

Listing 3: Python demo code

```
'''
    # This is MATH 574 course project. We try to solve the Poisson
    # Equation by using Classical FEM and FEM with Nitsche techniques
    # for pure homogeneous Dirichlet  BC.
    #          \Delta u + cu= f        in \Omega
    #                 u      = g_D      on \Gamma_D
    #                 u      = g_N      on \Gamma_N
    #
'''

from collections import namedtuple
from math import log as ln
from dolfin import *

set_log_level(ERROR)
set_log_level(WARNING)

```

```python
Result = namedtuple('Result', ['fun_name','NT','h', 'L2', 'H10', 'H1'])
f = Expression('2*pi*pi*sin(pi*x[0])*sin(pi*x[1])')
u_exact = Expression('x[0]*x[1] + sin(pi*x[0])*sin(pi*x[1])')


def classical_poisson(N,p):
    function_name= 'Classical FEM'
    'Standard formulation with strongly imposed bcs.'
    mesh = UnitSquareMesh(N, N)

    NT=mesh.num_cells()
    h = mesh.hmin()
    #print 'hmin', h
    #print  'circumradius',Circumradius(mesh)

    #plot(mesh)
    V = FunctionSpace(mesh, 'CG', p)
    u = TrialFunction(V)
    v = TestFunction(V)

    a = inner(grad(u), grad(v))*dx
    L = inner(f, v)*dx
    bc = DirichletBC(V, u_exact, DomainBoundary())

    uh = Function(V)
    solve(a == L, uh, bc)

    #plot(uh, title='numeric')
    #plot(u_exact, mesh=mesh, title='exact')
    #interactive()

    # Compute norm of error
    E = FunctionSpace(mesh, 'DG', 6)
    uh = interpolate(uh, E)
    u = interpolate(u_exact, E)
    e = uh - u

    norm_L2 = assemble(inner(e, e)*dx, mesh=mesh)
    norm_H10 = assemble(inner(grad(e), grad(e))*dx, mesh=mesh)
    norm_H1 = norm_L2 + norm_H10

    norm_L2 = sqrt(norm_L2)
    norm_H1 = sqrt(norm_H1)
    norm_H10 = sqrt(norm_H10)

    return Result(fun_name=function_name,NT=NT,h=h, L2=norm_L2, H1=norm_H1,
         H10=norm_H10)


def nitsche1_poisson(N,p):
    'Classical (symmetric) Nitsche formulation.'
    function_name= 'Symmetric Nitsche Method'
    mesh = UnitSquareMesh(N, N)
    NT=mesh.num_cells()


```

```
    V = FunctionSpace(mesh, 'CG', p)
    u = TrialFunction(V)
    v = TestFunction(V)

    beta_value = 2
    beta = Constant(beta_value)
    h_E = MinFacetEdgeLength(mesh)#mesh.ufl_cell().max_facet_edge_length
    n = FacetNormal(mesh)

    a = inner(grad(u), grad(v))*dx - inner(dot(grad(u), n), v)*ds -\
        inner(u, dot(grad(v), n))*ds + beta*h_E**-1*inner(u, v)*ds

    L = inner(f, v)*dx -\
        inner(u_exact, dot(grad(v), n))*ds + beta*h_E**-1*inner(u_exact, v)
           *ds

    uh = Function(V)
    solve(a == L, uh)

    # Save solution to file
    #file = File("poisson.pvd")
    #file << uh
    if N==64:
        viz1=plot(uh)
        viz1.write_png('snitche')
    # plot(uh, title='numeric')
    # plot(u_exact, mesh=mesh, title='exact')
    # interactive()

    # Compute norm of error
    E = FunctionSpace(mesh, 'DG', 4)
    uh = interpolate(uh, E)
    u = interpolate(u_exact, E)
    e = uh - u

    norm_L2 = assemble(inner(e, e)*dx, mesh=mesh)
    norm_H10 = assemble(inner(grad(e), grad(e))*dx, mesh=mesh)
    norm_edge = assemble(beta*h_E**-1*inner(e, e)*ds)

    norm_H1 = norm_L2 + norm_H10 + norm_edge
    norm_L2 = sqrt(norm_L2)
    norm_H1 = sqrt(norm_H1)
    norm_H10 = sqrt(norm_H10)

    return Result(fun_name=function_name,NT=NT,h=mesh.hmin(), L2=norm_L2,
       H1=norm_H1, H10=norm_H10)


def nitsche2_poisson(N,p):
    'Unsymmetric Nitsche formulation.'
    function_name= 'Unsymmetric Nitche Method'
    mesh = UnitSquareMesh(N, N)
    NT=mesh.num_cells()

    V = FunctionSpace(mesh, 'CG', p)
    u = TrialFunction(V)
```

```
    v = TestFunction(V)

    beta_value = 2
    beta = Constant(beta_value)
    h_E = MinFacetEdgeLength(mesh)#mesh.ufl_cell().max_facet_edge_length
    n = FacetNormal(mesh)

    a = inner(grad(u), grad(v))*dx - inner(dot(grad(u), n), v)*ds +\
        inner(u, dot(grad(v), n))*ds + beta*h_E**-1*inner(u, v)*ds

    L = inner(f, v)*dx +\
        inner(u_exact, dot(grad(v), n))*ds + beta*h_E**-1*inner(u_exact, v)
            *ds

    uh = Function(V)
    solve(a == L, uh)

    # plot(uh, title='numeric')
    # plot(u_exact, mesh=mesh, title='exact')
    # interactive()
    if N==64:
        viz1=plot(uh)
        viz1.write_png('nnitche')
    # Compute norm of error
    E = FunctionSpace(mesh, 'DG', 4)
    uh = interpolate(uh, E)
    u = interpolate(u_exact, E)
    e = uh - u

    norm_L2 = assemble(inner(e, e)*dx, mesh=mesh)
    norm_H10 = assemble(inner(grad(e), grad(e))*dx, mesh=mesh)
    norm_edge = assemble(beta*h_E**-1*inner(e, e)*ds)
    norm_H1 = norm_L2 + norm_H10 + norm_edge

    norm_L2 = sqrt(norm_L2)
    norm_H1 = sqrt(norm_H1)
    norm_H10 = sqrt(norm_H10)

    return Result(fun_name=function_name,NT=NT,h=mesh.hmin(), L2=norm_L2,
       H1=norm_H1, H10=norm_H10)

#
    ---------------------------------------------------------------------------


methods = [classical_poisson, nitsche1_poisson, nitsche2_poisson]

#print 'The Method:{:d}',format( method
for m in [1,2]: #[0,1,2]:
    for p in [1,2]:
        method = methods[m]

        #print "The Method:{}".format(method)

        #norm_type = 'H1'

```

```
        R = method(N=4,p=p)
        print "The method: {0} with degree of polynomial {1:}".format(R.fun_name,p)
        print "{0:>6s} {1:>6s} {2:>10s} {3:>7s} {4:>6s} {5:>7s}".format("Elem", "h", "L^2", "rate", "H^1","rate")
        h_ = R.h
        e_ = getattr(R, 'H1')
        eL2_= getattr(R,'L2')
        for N in [8, 16, 32, 64,128,256,512]:
            R = method(N,p=p)
            h = R.h
            NT = R.NT
            e = getattr(R, 'H1')
            eL2 = getattr(R,'L2')
            rate = ln(e/e_)/ln(h/h_)
            rateL2 = ln(eL2/eL2_)/ln(h/h_)
            # print 'h  error  rate'
            print '{0:6d}  {h:.3E} {eL2:.5E} {rateL2:.4f} {e:.5E} {rate:.4f}'.format(NT,h=h,eL2=eL2,rateL2=rateL2, e=e, rate=rate)
```

References

[1] M. S. Alnæs, J. Hake, R. C. Kirby, H. P. Langtangen, A. Logg, and G. N. Wells, *The FEniCS Manual*, October, 2011. 1

[2] L. Chen, *AFEM@MATLAB: a matlab package of adaptive finite element methods*, Technique Report, (2006). 1

[3] ——, *iFEM: an innovative finite element methods package in MATLAB*, Technique Report, (2009). 1

[4] W. Feng, X. He, Y. Lin, and X. Zhang, *Immersed finite element method for interface problems with algebraic multigrid solver*, Commun. Comput. Phys., 15 (2014), pp. 1045–1067. 1

[5] W. Feng, X. He, and Y. L. X. Zhang, *Immersed finite element method for interface problems with algebraic multigrid solver*, Commun.Comput.Phys., (To appear). 1

[6] P. Persson and G. Strang, *A simple mesh generator in matlab*, SIAM Review, 46 (2004), p. 2004. 5