Our task is to numerically solve the second order elliptic problem in two dimensional space using finite element method.

We consider the following model problem: let \( \Omega = (-1,1)^2 \setminus [0,1)^2 \) be the L-shaped domain, we seek \( u : \Omega \to \mathbb{R} \), such that

\[
- \Delta u + qu = f \quad \text{in } \Omega, \quad u = 0 \quad \text{on } \partial \Omega.
\]

Here the functions \( q(x) \geq 0 \) and \( f \) are both given. We can write a weak formulation:

\[
\text{find } u \in H^1(\Omega) \text{ satisfying } \\
\int_{\Omega} (\nabla u \cdot \nabla v + quv) \, dx = \int_{\Omega} fv \, dx, \quad \text{for all } v \in H^1_0(\Omega).
\]

The approximation scheme consists of developing finite element method to find approximated solutions to the weak problem (0.2).

1. Triangulation

To start with, we need to discretize the open bounded domain \( \Omega \subset \mathbb{R}^2 \) into triangles. The discussion in this section applies to generic domains.

**Notations.** To simplify our problem, let us assume that our bounded domain \( \Omega \) is polygonal. Non-polygonal domains such as unit disc requires additional treatment for the geometry. We then subdivide \( \overline{\Omega} \) (including \( \Omega \) and the boundary \( \partial \Omega \)) with many small closed triangles (quadrilaterals are also allowed, but we will not consider here). Let us denote \( \mathcal{T} \) to be the collection of all the triangles. Since \( \Omega \) is open (not including boundary), \( \Omega \) is equal to interior of the union of all the triangles, i.e. \( \Omega = (\bigcup_{\tau \in \mathcal{T}} \tau)^{\circ} \). Figure 1 shows a triangular mesh in the square domain \((-1,1)^2\).

There are two basic rules when we are talking about a mesh for our finite element implementation:

a) One triangle cannot overlap any other triangles. This means that if \( \tau_i \) and \( \tau_j \) are two distinct triangles in \( \mathcal{T} \), the intersection of their interiors should be empty, i.e. \( \tau_i^{\circ} \cap \tau_j^{\circ} = \emptyset \).

b) Given two distinct triangles \( \tau_i \) and \( \tau_j \), their intersection \( \tau_i \cap \tau_j \) should be only one of the following cases: an empty set \( \emptyset \), a common vertex or a common edge. This indicates that the case in Figure 2 is not allowed. In Figure 2, we call the vertex at the interior of an edge as a hanging node. We say a mesh is admissible or **conforming** if the mesh has no hanging nodes (see Figure 1). Here we remark that we can deal with the hanging nodes using some special techniques but it is a bit more complicated (this is in fact quite useful when the...
element is quadrilateral). For simplicity, we only consider admissible triangular meshes in our problem.

![Figure 1. A uniform structured mesh in $[-1,1]^2$.](image1)

**Figure 1.** A uniform structured mesh in $[-1,1]^2$.

**Figure 2.** A non-conforming mesh.

*Shape regularity.* To make sure that we can generate a sequence of meshes so that our approximated solutions on these meshes will convergence to the exact solution, we need two more requirements on the meshes. The first one is the *shape regularity.* For a triangle $\tau$ in the mesh $\mathcal{T}$, there exists a smallest circle so that the triangle $\tau$ is inside this circle (circumcircle), we denote $h_{\tau}$ the diameter of such circle. Also, there exists a largest circle inscribed in $\tau$ (inscribed circle) and we denote $r_{\tau}$ the diameter of the inscribed circle. We say a mesh $\mathcal{T}$ is shape-regular
provided that there exists a constant \( c \) so that
\[
\frac{h_\tau}{r_\tau} \leq c
\]
holds for any triangle \( \tau \in \mathcal{T} \). The constant \( c \) is called the shape regularity constant and \( h = \max_\tau h_\tau \) denotes the size of the mesh.

For a given triangle \( \tau \) in a mesh \( \mathcal{T} \), let \( \theta_\tau \) be the smallest angle in \( \tau \). We can prove that
\[
\frac{h_\tau}{r_\tau} \leq \frac{2}{\sin \theta_\tau}.
\]
This inequality implies that if \( \theta_\tau \), the smallest angle in \( \tau \), is greater than a certain angle \( \theta_0 \) for any triangle \( \tau \in \mathcal{T} \), then the mesh is shape-regular. So roughly speaking, shape-regularity requires all the triangles not to be too “flat”.

**Quasi-uniformity.** We say a mesh \( \mathcal{T} \) is *quasi-uniform* provided that \( \mathcal{T} \) is shape-regular and there exists a constant \( \rho \) so that
\[
\rho h \leq h_\tau \quad \text{for all } \tau \in \mathcal{T}.
\]
This shows that the sizes of any two triangles are similar up to a constant ratio \( \rho \).

In a uniform mesh such as the one in Figure 1, we shall observe that \( \rho = 1 \). So here we extend the uniform mesh to a more general setting.

**Constructing a mesh.** We will solve our 2D problem on shape-regular and quasi-uniform meshes. The first task is to construct a quasi-uniform mesh so that we can use it in MATLAB. Unfortunately, to our best knowledge, in MATLAB we do not have a direct method to build a mesh that can be controlled by the shape regularity and quasi-uniform constants (e.g. Figure 3).

An alternative software we are going to use is TRIANGLE. The source code is available at https://www.cs.cmu.edu/˜quake/triangle.html. For a Linux OS or macOS computer, we can compile the source code in terminal by typing
```
make all
```

**Remark 1.** For mac users, the installation of Xcode, command tools and X11 are required. A modification of the `makefile` from the source code is also necessary. In line 76, we need to change the `SWITCHES` flag to
```
CSWITCHES = -O -DNO_TIMER -I/usr/X11/include -L/usr/X11/lib
```

After successfully compiling the code, we obtain the executable `triangle`. The poly-file `Lshaped.poly` provides the definition of the domain (see here for an instruction). In terminal, we can generate a L-shaped mesh by using command
```
./triangle -a.02 -e -q30 Lshaped.poly
```

Here the parameters
Figure 3. A quasi-uniform mesh for a L-shaped domain.

- \( a_0.02 \) means that the area of all the triangles is no larger than 0.02. We can control the mesh size \( h \) by changing the area.
- \( -e \) outputs (to an .edge file) a complete list of edges in the triangulation. In some cases (not in our problem) we need to integrate over edges.
- \( -q30 \) ensures that the smallest angle of each triangle is no less than 30 degrees. This guarantees the quasi-uniformity of the mesh we generate.

The output contains several files. In our problem we will need three of them: \texttt{Lshaped.1.node}, \texttt{Lshaped.1.ele} and \texttt{Lshaped.1.edge}. We also need the MATLAB function \texttt{readmesh.m}. It reads the files and store the mesh data into MATLAB. The function can be called as follows:

\[
[n\_node,n\_ele,node,ele,n\_edge,edge,is\_node\_at\_boundary] = \texttt{readmesh('Lshaped.1.node','Lshaped.1.ele','Lshaped.1.edge');}
\]

Here the output values \( n\_node, n\_ele, \) and \( n\_edge \) correspond to the number of vertices, elements, and edges respectively. \( \texttt{node} \) is an \( n\_node \) by 2 matrix storing the coordinates of all the vertices. \( \texttt{ele} \) and \( \texttt{edge} \) are lists of triangles and edges. The vector \( \texttt{is\_node\_at\_boundary} \) is a boolean vector to check whether each node in the \( \texttt{node} \) list is on the boundary.

For instance, in the first row of \( \texttt{ele} \), \( \{8, 21, 77\} \) gives the triangle with vertices to be the 8th, 21st and 77th nodes in the \( \texttt{node} \) list.

The function \texttt{meshplot.m} can plot the continuous piecewise linear function on the corresponding mesh. For example, suppose that we have read and stored the
Figure 4. Interpolation of \( f(x, y) = \sin(\pi x) + \sin(\pi y) \) on the L-shaped domain.

Mesh described as above into MATLAB, we can plot the interpolation of the function \( f(x, y) = \sin(\pi x) + \sin(\pi y) \) on the mesh by

```matlab
y = @(x1,x2) sin(pi*x1)+sin(pi*x2);
meshplot(ele,node,y(node(:,1),node(:,2)));
```

See Figure 4 for an illustration. We can use this function to visualize the approximated solution.

2. Finite element approximation

In this section we first discuss the continuous piecewise linear finite element space subordinated in a quasi-uniform mesh.

The finite element space. Given a quasi-uniform and shape-regular mesh \( \mathcal{T}_h \) with the mesh size \( h > 0 \), we denote by \( \mathcal{V}_h \) the space of the continuous piecewise linear functions on \( \mathcal{T}_h \). Namely, for any \( v_h \in \mathcal{V}_h \), the function \( v_h \) is continuous and its restriction to any triangle \( \tau \in \mathcal{T}_h \) is a linear function (see e.g. Figure 4). Similar to the 1D case, the finite element basis functions have the following property: given \( x_i \) in the node list, the corresponding basis function \( \phi_i \) satisfies

\[
\phi_i(x_i) = 1, \quad \text{and} \quad \phi_i(x_j) = 0 \quad \text{for all} \quad x_j \neq x_i.
\]

Figure 5 shows two example basis functions.
Figure 5. (Left) A finite element basis function at the boundary and (Right) a finite element basis function in the interior of the domain.

The set \( \{ \phi_i \} \) forms a complete basis for the finite element space \( \mathbb{V}_h \), therefore any \( v_h \in \mathbb{V}_h \) is a linear combination of \( \{ \phi_i \} \), that is

\[
v_h(x) = \sum_{i=1}^{M} c_i \phi_i(x),
\]

where \( M \) is the dimension of \( \mathbb{V}_h \).

**Reference triangle, affine mapping and local basis functions.** Recall that we build the stiffness matrix in 1D by assembling local stiffness matrices for each subinterval. In 2D, we follow the same element-by-element approach, thus we need to build the local stiffness matrix for every cell (triangle). Since a triangle has three vertices, we shall have three local basis functions. Therefore, the local stiffness matrix should be a \( 3 \times 3 \) matrix.

In 1D it was convenient to write the local basis functions and their derivatives directly. However, in 2D a direct representation is not readily available, hence we define the local basis functions in a different approach.

We define the reference triangle \( \hat{\tau} \) with three vertices (0,0), (1,0) and (0,1), see Figure 6. Given any arbitrary triangle \( \tau \in \mathcal{T}_h \) with three vertices \((x_1, y_1)\), \((x_2, y_2)\) and \((x_3, y_3)\), we can map \( \hat{\tau} \) to \( \tau \) using the mapping

\[
\begin{pmatrix} x \\ y \end{pmatrix} = T(\hat{x}, \hat{y}) := \begin{pmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{pmatrix} \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \quad \text{for} \ (\hat{x}, \hat{y}) \in \hat{\tau}.
\]
Clearly, a sanity check shows that $T$ maps the vertices of $\hat{\tau}$ to the vertices of $\tau$ (that is, $(0,0) \mapsto (x_1,y_1)$, $(1,0) \mapsto (x_2,y_2)$, and $(0,1) \mapsto (x_3,y_3)$). We can also prove that, if $f(x,y)$ is a linear function in $\tau$, $f(T(\hat{x},\hat{y}))$ is a linear function in $\hat{\tau}$. The mapping $T$ is also one-to-one, therefore the inverse mapping $T^{-1}: \tau \to \hat{\tau}$ exists.

![Figure 6](image)

**Figure 6.** (a) The reference triangle $\hat{\tau}$ and (b) the real arbitrary triangle $\tau$.

It is not difficult to derive that the local basis functions on $\hat{\tau}$ are

$$\hat{\phi}_1 = 1 - \hat{x} - \hat{y}, \quad \hat{\phi}_2 = \hat{x}, \quad \text{and} \quad \hat{\phi}_3 = \hat{y}.$$  

Their first order derivatives are

$$\hat{\nabla} \hat{\phi}_1 = (-1, -1)^t, \quad \hat{\nabla} \hat{\phi}_2 = (1, 0)^t, \quad \text{and} \quad \hat{\nabla} \hat{\phi}_3 = (0, 1)^t.$$  

Here $\hat{\nabla} \hat{\phi}_i$ denotes the gradient of $\hat{\phi}_i$ with respect to $(\hat{x}, \hat{y})$.

With the assistance of mapping $T^{-1}$, the local basis functions on $\tau$ can be defined as

$$\phi_i(x,y) = \hat{\phi}_i(T^{-1}(x,y)), \quad \text{for} \quad i = 1, 2, 3.$$  

Also, invoking the chain rule, we can compute the gradient of $\phi_i$ by

$$\nabla \phi_i = (J(T^{-1}))^t \hat{\nabla} \hat{\phi}_i.$$  

Here $J(T^{-1})$ is the Jacobian of the inverse of $T$. Since $T$ is a linear mapping,

$$J(T^{-1}) := B^{-1} = \begin{pmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{pmatrix}^{-1}.$$
**Numerical integration.** To compute the entries of the local stiffness matrix on an arbitrary triangle $\tau \in \mathcal{T}_h$,

$$ a^\tau_{i,j} := \int_{\tau} a(x, y) \nabla \phi_i \cdot \nabla \phi_j \, dx dy, \quad \text{for } i, j = 1, 2, 3, $$

we will map the triangle $\tau$ to the reference element $\hat{\tau}$, and apply the numerical integration on $\hat{\tau}$. In the first step, by change of the variables using the mapping $T$, we have

$$ a^\tau_{i,j} := \int_{\tau} a(T(\hat{x}, \hat{y})) \nabla \phi_i \cdot \nabla \phi_j \, dx dy = \int_{\hat{\tau}} a(T(\hat{x}, \hat{y})) [(J(T^{-1}))^t \hat{\nabla} \hat{\phi}_i] \cdot [(J(T^{-1}))^t \hat{\nabla} \hat{\phi}_j] |\text{det}B| \, d\hat{x} d\hat{y}. $$

In the second step, given the quadrature points $q_1, q_2, \ldots, q_k$ and the quadrature weights $w_1, w_2, \ldots, w_k$ for the integral on $\hat{\tau}$, we define the approximation to the above integral by

$$ \int_{\tau} a(x, y) \nabla \phi_i \cdot \nabla \phi_j \, dx dy \approx \sum_{l=1}^k w_l a(T(q_l)) [(J(T^{-1}))^t \hat{\nabla} \hat{\phi}_i(q_l)] \cdot [(J(T^{-1}))^t \hat{\nabla} \hat{\phi}_j(q_l)] |\text{det}B|, $$

for $i, j = 1, 2, 3$. An example quadrature formula is

$$ \int_{\tau} f(x) \, dx \approx \frac{1}{24} (f(0,0) + f(1,0) + f(0,1) + 9f(\frac{1}{3}, \frac{1}{3})). $$

The above formula is exact for polynomial $f \in \mathbb{P}^2$.

**Remark 2.** Another type of commonly used quadrature rule is the Gaussian quadrature:

$$ \int_{\tau} f(x, y) \, dx \, dy \approx \sum_{j=1}^{n_q} \hat{w}_j f(\hat{x}_j, \hat{y}_j). $$

Table 1 lists the first five quadrature rules. Notice that these rules have the property (can be verified by taking $f = 1$)

$$ \sum_{j=1}^{n_q} \hat{w}_j = 0.5. $$

Higher orders quadrature rules can be found here.
Table 1. Quadrature rules exact for $p \in P^k(\hat{\tau})$ on the standard 2D reference triangle $\hat{\tau}$.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$n_q$</th>
<th>$\hat{x}_j$</th>
<th>$\hat{y}_j$</th>
<th>$\hat{w}_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.333333333333330</td>
<td>0.333333333333330</td>
<td>1.000000000000000</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.1666666666666670</td>
<td>0.1666666666666670</td>
<td>0.333333333333330</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1666666666666670</td>
<td>0.6666666666666670</td>
<td>0.333333333333330</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.6666666666666670</td>
<td>0.1666666666666670</td>
<td>0.333333333333330</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.333333333333330</td>
<td>0.333333333333330</td>
<td>-0.562500000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.20000000000000000</td>
<td>0.20000000000000000</td>
<td>0.520833333333330</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.20000000000000000</td>
<td>0.60000000000000000</td>
<td>0.520833333333330</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.60000000000000000</td>
<td>0.20000000000000000</td>
<td>0.520833333333330</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>0.4459484909159700</td>
<td>0.4459484909159700</td>
<td>0.223815896780100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.4459484909159700</td>
<td>0.1081030181680700</td>
<td>0.223815896780100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1081030181680700</td>
<td>0.4459484909159700</td>
<td>0.223815896780100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0915762135097700</td>
<td>0.0915762135097700</td>
<td>0.109951743655320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0915762135097700</td>
<td>0.8168475729804600</td>
<td>0.109951743655320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8168475729804600</td>
<td>0.0915762135097700</td>
<td>0.109951743655320</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>0.333333333333330</td>
<td>0.333333333333330</td>
<td>0.225000000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.4701420641051100</td>
<td>0.4701420641051100</td>
<td>0.132391527885100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.4701420641051100</td>
<td>0.0597158717897700</td>
<td>0.132391527885100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0597158717897700</td>
<td>0.4701420641051100</td>
<td>0.132391527885100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1012865073234600</td>
<td>0.1012865073234600</td>
<td>0.125939180544830</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1012865073234600</td>
<td>0.7974269853530900</td>
<td>0.125939180544830</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.7974269853530900</td>
<td>0.1012865073234600</td>
<td>0.125939180544830</td>
</tr>
</tbody>
</table>

**Approximation of local mass matrix.** Similarly, we can approximate the local mass matrix and the local right hand side vector by

\[
m_{i,j}^\tau = \int_\tau b(x,y)\phi_i(x)\phi_j(x) \, dx \, dy \\
\approx \sum_{l=1}^k w_{l}b(T(q_{l}))\phi_i(T(q_{l}))\phi_j(T(q_{l}))|\det B| \\
= \sum_{l=1}^k w_{l}b(T(q_{l}))\hat{\phi}_i(q_{l})\hat{\phi}_j(q_{l})|\det B| \quad \text{for } i, j = 1, 2, 3.
\]
and
\[ f_i^T = \int_{\tau} f(x, y) \phi_i(x, y) \, dx \, dy \]
\[ \approx \sum_{l=1}^{k} w_l f(T(q_l)) \phi_i(T(q_l)) |\det B|, \]
\[ = \sum_{l=1}^{k} w_l f(T(q_l)) \hat{\phi}_i(q_l) |\det B|, \quad \text{for} \ i = 1, 2, 3. \]

Note that the values of \( \hat{\phi}_i(q_l) \) and \( \hat{\nabla} \hat{\phi}_j(q_l) \) can be precomputed and hard-coded into the code. Here is an example code for the computation of the local mass matrix:

```matlab
function [ML] = LocalMass(v1,v2,v3,w,q,hat_phi_at_q, bcoef)
% Output:
% [AL] : [3X3 matrix] local mass matrix.
% Input:
% v1, v2, v3: [2X1 matrices] x, y coordinates for a triangle.
% w: [1Xk matrix] list of quadrature weights. k is the number of quadrature points.
% q: [2Xk matrix] list of quadrature points. Each column gives the x, y coordinates of quadrature points.
% hat phi at q: [3Xk matrix] list of evaluations at quadrature points for each local basis functions at the reference triangle. For example, the first row should store all the values at k quadrature points for the first local basis function 1-x-y.
% bcoef: function b(p). Here p is matrix with two rows. The first row is the list of x-coordinate and the second row is the list of y-coordinate. So you have to build the function operating corresponding elements in these two rows.

ML = zeros (3,3); % initialize the local mass matrix
n_q_point = length(w); % get the number of quadrature points
B = [v2(1)-v1(1), v3(1)-v1(1);
    v2(2)-v1(2), v3(2)-v1(2)]; % define B
detB = abs(det(B)); % compute the determinant of B
% map the quadrature points from the reference triangle to the one we have
q_point = B*q+repmat(v1,1,k); % compute b(x) at each quadrature point
b_at_q = bcoef(q_point);
% apply the quadrature rule to compute the local mass matrix
for l = 1:n_q_point
    for i = 1:3
        for j = 1:3
            ML(i,j) = w(l)*b_at_q(l)*hat_phi_at_q(i,l)*hat_phi_at_q(j,l)*detB;
        end
    end
end
```
The computation of local stiffness matrix and local right hand side vector is similar.

**Assembly of global matrices and vectors.** The “element-by-element” technique is used to assemble the global matrices and rhs vector.

We can either create a single `assembly_matrices` function that assembles matrices and rhs at the same time, or we can write separate routines for each. Here we choose the first approach and write them into the same function.

For notational convenience, we denote $N_i$ to be the number of interior nodes. Same as in the 1D case, the solution at the boundary nodes are already determined - they are fixed to be zero, therefore the global mass matrix $M$ and global stiffness matrix $A$ should only include the $N_i$ interior nodes. In the following routine the `I_full` array serves as a boundary/interior indicator.

```matlab
% set boundary indicator
k = 0;
j = 0;
I_full = zeros(n_nodes, 1);
for m = 1:n_nodes
    if (is_node_at_boundary(m))
        k = k+1;
        boundary(k) = m;
        I_full(m) = -k;
    else
        j = j+1;
        interior(j) = m;
        I_full(m) = j;
    end
end
```

We now assembly the matrices:

```matlab
function [A,M] = assembly_matrices(n_ele,eles,nodes,I_full,N_i)
l=0;
for j=1:n_ele
    v1 = nodes(eles(j,1),:);
v2 = nodes(eles(j,2),:);
v3 = nodes(eles(j,3),:);
[B, absdB, InvB] = mapping(v1,v2,v3);
ls = localStiff(InvB, absdB);
lm = localMass(absdB);
    for m = 1:3
```
if I_full(eles(j,m))>0
    for k = 1:3
        if I_full(eles(j,k))>0
            l = l+1;
            ii(l) = I_full(eles(j,m));
            jj(l) = I_full(eles(j,k));
            v_mass(l) = ls(m,k);
            v_stiff(l) = ls(m,k);
        end
    end
end
end

% create the sparse matrices
A = sparse(ii,jj,v_mass,N_i,N_i);
M = sparse(ii,jj,v_stiff,N_i,N_i);
end

Solve the linear system. Recall that the system matrix \( S \) (in this example, \( S = A + M \)) is assembled as a sparse matrix. We shall solve the linear equation system using the backslash solver to obtain the solution vector

\[
sol_{\text{interior}} = S\backslash F.
\]

Note that the solution vector has values on the \( N_i \) interior nodes only. We need to map back to the full vector. This is done by using the \textit{interior} array:

\[
sol = \text{zeros}(n\_nodes,1); \% \text{all boundary nodes have value 0}
\]

for \( j = 1:n\_\text{interior} \)
    sol(interior(j)) = sol_{\text{interior}}(j);
end

Solution visualization. MATLAB has a convenient function \texttt{trimesh} that readily plots the solution associated to the triangulation. The inputs are the element array, the node array, and the solution array:

\[
\text{trimesh (eles, nodes(:,1), nodes(:,2), sol);}
\]

Errors. The \( L^2 \) and \( H^1 \) error computations follow the same procedure as in the 1D case, i.e. collecting the local error at each element using the quadrature formula.

\textbf{Remark 3.} For non-homogeneous Dirichlet boundary conditions, extra procedures are required for the right hand side vector; refer to Homework 8.
3. Appendix: A python code for finite elements

The reason for writing this Python program is two fold. On one hand, Python is nowadays a popular programming language, and it is easy to learn, fully featured for scientific computation, and includes strong science libraries such as NumPy. On the other hand, the university has stopped providing free MATLAB student license, which is making the open-source Python programming more favorable. When compared with MATLAB, the generic logic in python is the same, but the syntax is different. Here is a simple comparison between Python and MATLAB.

The Python installation package is available here. You also need the NumPy package.

After downloading my Python code, you may unzip it, navigate to the directory in terminal, and run the code by typing

code: python FEMdriver.py

You shall see the error table printed in terminal, as well as the plots for solutions and error convergence rates. Figures 7 and 8 are the solutions to Problem 1 of homework 8.

Have fun!

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
Mesh Size & L2 Error & H1 Error \\
\hline
1.280369E-01 & 5.967612E-04 & 2.751635E-03 \\
6.78842E-02 & 1.611432E-04 & 9.578743E-04 \\
3.498557E-02 & 4.123644E-05 & 2.848826E-04 \\
1.776392E-02 & 1.037576E-05 & 8.008386E-05 \\
\hline
\end{tabular}
\caption{Error table for Problem 1 of homework 8.}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{The errors to Problem 1 of homework 8. (a) the printed errors in terminal; (b) the log-log plot.}
\end{figure}
Figure 8. The solution to Problem 1 of homework 8. (a) the coarsest mesh, (d) the finest mesh.