PROGRAMMING OF LINEAR VIRTUAL ELEMENT METHODS

LONG CHEN, AND MIN WEN

In this notes, we present an efficient MATLAB implementation of the linear virtual element method for the two dimensional Poisson equation. In [6], it aims to give short implementation of algorithms for the education purpose. The code in [6] is not vectorized which is crucial to achieve comparable performance in MATLAB with respect to compiled languages.

1. Introduction to Virtual Element Methods

Virtual Element Method (VEM) [1] can be viewed as an extension of Finite Element Methods (FEM) to general polygonal and polyhedral meshes. In [2], a hitchhiker guide to programming of VEM is also given. Here we present a simple example: linear VEM for Poisson equation in 2-D, and wish to make the method and its implementation more accessible for beginners. For theoretical results and implementation for more general cases, we refer to [1, 2, 8].

1.1. Formulation.

Consider the model problem of the two dimensional Poisson equation:

$$-\Delta u = f \text{ in } \Omega, \quad u = g \text{ on } \partial\Omega. \tag{1}$$

Here Ω is a polygonal domain in $\mathbb{R}^2$, $f \in L^2(\Omega)$ and $g \in H^{1/2}(\partial\Omega)$. Define $H^1_0(\Omega) := \{v \in H^1(\Omega) : \operatorname{tr} v = 0 \text{ on } \partial\Omega\}$ and $H^1_g(\Omega) := \{v \in H^1(\Omega) : \operatorname{tr} v = g \text{ on } \partial\Omega\}$. The weak formulation of (1) is: find $u \in H^1_g(\Omega)$ such that

$$a(u,v) = (f,v), \quad \forall v \in H^1_0(\Omega), \tag{2}$$

where the bilinear form $a(u,v) = (\nabla u, \nabla v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x$ and $(\cdot,\cdot)$ is the L^2 inner product. By Lax-Milgram Lemma, it has a unique solution.

Let $\mathcal{T}_h$ be a collection of partitions of domain Ω into polygonal elements without self-intersecting boundary. The linear VEM space $V_h(E)$ for a polygon $E \in \mathcal{T}_h$ is defined as:

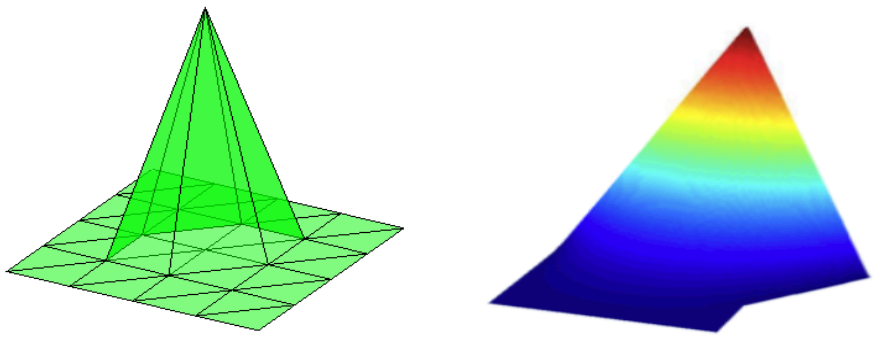
$$V_h(E) := \{v \in H^1(E) : \Delta v|_E = 0, v|_{\partial E} \text{ is continuous and piecewise linear}\}.$$

Namely restricted to boundary edges of a polygon, the function is the standard linear Lagrange element. The interior is defined by the harmonic extension. Since a piecewise linear function will be uniquely determined by its value on vertices, $\dim V_h(E) = n_E$, where n_E is the number of vertices of E. The global virtual element space is defined as

$$V_h = \{v_h \in H^1(\Omega) : v_h|_E \in V_h(E) \text{ for all } E \in \mathcal{T}_h\}.$$

Let $\mathcal{N}(\mathcal{T}_h)$ be the set of vertices of mesh $\mathcal{T}_h$ and $N = |\mathcal{N}(\mathcal{T}_h)|$ be the number of vertices. We define the operator dof (degree of freedom) from V_h to $\mathbb{R}^N$ as $\mathrm{dof}_i(v_h) = v_h(\boldsymbol{x}_i)$, for a vertex $\boldsymbol{x}_i \in \mathcal{N}(\mathcal{T}_h), i = 1, \ldots, N$. The canonical basis $\{\phi_1, \cdots, \phi_N\}$ of V_h is chosen satisfying $\mathrm{dof}_i(\phi_j) = \delta_{ij}, i, j = 1, \cdots, N$. The nodal interpolation $I_h : C(\bar{\Omega}) \to V_h$ is defined as $I_h u = \sum_{i=1}^N u(\boldsymbol{x}_i)\phi_i$ and denoted by $u_I = I_h u$.

A basis function ϕ_i is like the classical hat function of linear FE defined on triangular meshes. The difference is: for FEM functions, inside the element it is a linear polynomial while the VEM function is defined by the harmonic extension and function values inside the element is not explicitly known. As we shall show later, the basis does not need to be known explicitly and this is the reason for the term "virtual" in VEM.



(a) A linear FEM basis (b) A linear VEM basis

FIGURE 1. Nodal basis functions of Linear FEM and VEM.

The classic Galerkin approximation of (2) is: find $u_h \in V_h \cap H^1_g(\Omega)$ such that:

$$(\nabla u_h, \nabla v_h) = (f, v_h), \quad \forall v_h \in V_h \cap H^1_0(\Omega).$$

Suppose $u_h = \sum_{j=1}^N u_j \phi_j$, by linearity, we have for $i \in 1, \cdots, N$,

$$\sum_{j=1}^{N} (\nabla \phi_j, \nabla \phi_i) u_j = (f, \phi_i). \tag{3}$$

If we denote the matrix $\boldsymbol{A}_{ij} = (\nabla \phi_j, \nabla \phi_i)$, $\boldsymbol{u}_h = (u_h^1, \cdots, u_h^N)^\intercal$, and the vector $\boldsymbol{b} = (b_1, \cdots, b_N)^\intercal$ by $b_i = (f, \phi_i)$. Equation (3) is written in the matrix form as

$$\boldsymbol{A}\boldsymbol{u}_h = \boldsymbol{b}. \tag{4}$$

It is easy to show the matrix $\boldsymbol{A}$ is symmetric and positive definite which guarantees the solvability of (4).

For finite element methods, it suffices to compute the local stiffness matrix in each element, and then assemble the matrix $\boldsymbol{A}$ by summing the contribution from each element. In VEM, the central question is to compute an accurate approximation of $(\nabla \phi_j, \nabla \phi_i)_E$ without knowing the basis functions explicitly.

1.2. **Equivalent bilinear form.** We do not know a VEM function inside each element. What we do know is its dof and the function values on edges. The novelty of VEM is to use an equivalent and accurate bilinear form which can be computed using the information we have.

To do so, we first introduce a projection operator. In each polygonal element E, the operator $\Pi^\nabla : V_h(E) \to \mathbb{P}_1(E)$ is an H^1 projection to $\mathbb{P}_1(E)$ space:

$$(\nabla \Pi^\nabla v_h, \nabla p)_E = (\nabla v_h, \nabla p)_E \quad \text{for all } p \in \mathbb{P}_1(E). \tag{5}$$

Here $\mathbb{P}_1(E)$ is the space of linear polynomials. The right hand side can be computed by the integration by parts

$$(\nabla v_h, \nabla p)_E = -(v_h, \Delta p)_E + \langle v_h, \partial_n p\rangle_{\partial E} = \langle v_h, \nabla p \cdot n\rangle_{\partial E}.$$

On the boundary v_h is piecewise linear while $p \in \mathbb{P}_1(E)$ is a linear polynomial whose normal derivative is of course computable.

It is obvious that (5) only defines $\Pi^\nabla v_h$ up to a constant. The constant can be determined by a projection operator onto constants $P_0 : V_h(E) \to \mathbb{P}_0(E)$ and requiring

$$P_0(\Pi^\nabla v_h - v_h) = 0.$$

One simple choice is

$$P_0 v_h = \frac{1}{n_E}\sum_{i=1}^{n_E} \mathrm{dof}_i(v_h) = \frac{1}{n_E}\sum_{i=1}^{n_E} v_h(\boldsymbol{x}_i).$$

It is easy to show $\nabla\Pi^\nabla v$ is indeed the L^2 projection of ∇v to the constant vector space on E. We thus have an explicit formula of Π^∇

$$\Pi^\nabla v = \frac{1}{|E|}\int_E \nabla v \,\mathrm{d}\boldsymbol{x}\cdot(\boldsymbol{x}-\boldsymbol{x}_E) + \frac{1}{n_E}\sum_{i=1}^{n_E} v(\boldsymbol{x}_i), \tag{6}$$

where

$$\boldsymbol{x}_E = \frac{1}{n_E}\sum_{i=1}^{n_E}\boldsymbol{x}_i$$

is the barycentric center of E and satisfying $P_0(\boldsymbol{x}-\boldsymbol{x}_E) = 0$.

Now we can rewrite the basis function $\phi_i \in V_h(E)$ as $\Pi^\nabla\phi_i + (I-\Pi^\nabla)\phi_i$, and as Π^∇ is an H^1-projection

$$(\nabla\phi_i, \nabla\phi_j)_E = (\nabla\Pi^\nabla\phi_i, \nabla\Pi^\nabla\phi_j)_E + (\nabla(I-\Pi^\nabla)\phi_i, \nabla(I-\Pi^\nabla)\phi_j)_E.$$

The first term is computable as $\Pi^\nabla\phi_i$ is a polynomial. The second term is still not computable since again the basis ϕ_i is unknown. Fortunately this term is of high frequency which can be replaced by a so-called stabilization term $S_E(\cdot,\cdot)$

$$(\nabla\Pi^\nabla\phi_i, \nabla\Pi^\nabla\phi_j)_E + S_E((I-\Pi^\nabla)\phi_i, (I-\Pi^\nabla)\phi_j).$$

Define $\|v\|^2_{S_E} = S_E(v,v)$. The stabilization S_E defines a norm equivalent to the H^1-seminorm restricted to the kernel space of Π^∇, i.e.,

$$c_1\|\nabla v\|^2_E \le \|v\|^2_{S_E} \le c_2\|\nabla v\|^2_E, \quad \forall v \in V_h(E) \text{ and } \Pi^\nabla v = 0,$$

where some positive constants c_1 and c_2 are independent of E [1]. Consequently we have the norm equivalence

$$\|\nabla v\|^2_E \eqsim \|\nabla\Pi^\nabla v\|^2_E + \|(I-\Pi^\nabla)v\|^2_{S_E} \quad \forall v \in V_h(E).$$

There are many types of stabilization. The most popular one is the correctly scaled l^2 inner product of dofs and for linear VEM in 2D, it reads as

$$S_E(u,v) = \sum_{r=1}^{n_E}\mathrm{dof}_r(u)\,\mathrm{dof}_r(v).$$

We summarize the explicit expression of the local stiffness matrix of the virtual element method as follows:

$$(\boldsymbol{A}_h^E)_{ij} := (\nabla\Pi^\nabla\phi_i, \nabla\Pi^\nabla\phi_j)_E + \sum_{r=1}^{n_E} \mathrm{dof}_r((I-\Pi^\nabla)\phi_i)\,\mathrm{dof}_r((I-\Pi^\nabla)\phi_j), \tag{7}$$

and will discuss its matrix representation in the next subsection.

1.3. **Matrix representation.** We shall use boldface and capital letters for matrices and give explicit formulation of the projection Π^∇ and stabilization S_E.

Recall that

$$\boldsymbol{x}_E = (x_E, y_E) = \frac{1}{n_E}\sum_{i=1}^{n_E} \boldsymbol{x}_i$$

is the center of E, and let $h_E = |E|^{1/2}$. Usually h_E is chosen as the diameter of E. We chose h_E as the square root of the area of each element which will simplify the projection matrix. A scaled monomial basis of $\mathbb{P}_1(E)$ is chosen as

$$m_1 = 1, \quad m_2 = (x - x_E)/h_E, \quad m_3 = (y - y_E)/h_E.$$

Given a function $v_h \in V_h(E)$, its vector representation is $\boldsymbol{v} = \mathrm{dof}(v_h)$. By definition,

$$\Pi^\nabla v_h = \sum_{\alpha=1}^{3} s^\alpha m_\alpha \tag{8}$$

and the coefficient vector $\boldsymbol{s} = (s^\alpha)$ is determined by the following linear systems

$$(\nabla m_\alpha, \nabla\Pi^\nabla v_h)_E = (\nabla m_\alpha, \nabla v_h)_E \quad \alpha = 1,2,3. \tag{9}$$

Denote the matrix representation of the operator Π^∇ relative to the basis (m_α) by $\boldsymbol{\Pi}^\nabla$ which is of size $3 \times n_E$. In matrix-vector terminology,

$$\boldsymbol{s} = \boldsymbol{\Pi}^\nabla \boldsymbol{v} = \boldsymbol{G}^{-1}\boldsymbol{B}\boldsymbol{v},$$

which is the matrix realization of the linear algebraic system (8).

Let $\boldsymbol{G}$ be a modified stiffness matrix of $\mathbb{P}_1(E)$ of size 3×3. By the choice of bases and h_E, the matrix $\boldsymbol{G}_{3\times 3}$ becomes

$$\boldsymbol{G} := \begin{pmatrix} P_0 m_1 & P_0 m_2 & P_0 m_3 \\ 0 & (\nabla m_2, \nabla m_2)_E & (\nabla m_3, \nabla m_2)_E \\ 0 & (\nabla m_2, \nabla m_3)_E & (\nabla m_3, \nabla m_3)_E \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

and thus $\boldsymbol{G}^{-1}$ comes at free. The matrix $\boldsymbol{B}_{3\times n_E}$ is:

$$\boldsymbol{B} := \begin{pmatrix} 1/n_E & 1/n_E & \cdots & 1/n_E \\ (\nabla m_2, \nabla\phi_1)_E & (\nabla m_2, \nabla\phi_2)_E & \cdots & (\nabla m_2, \nabla\phi_{n_E})_E \\ (\nabla m_3, \nabla\phi_1)_E & (\nabla m_3, \nabla\phi_2)_E & \cdots & (\nabla m_3, \nabla\phi_{n_E})_E \end{pmatrix}.$$

As the derivative of constant is zero, the first row of $\boldsymbol{B}$ is replaced by the constraint, i.e. to impose $P_0\Pi^\nabla\phi_i = P_0\phi_i$. Therefore the first row of $\boldsymbol{B}$ is $P_0\phi_1 = \cdots = P_0\phi_{n_E} = 1/n_E$. For the other components $(\nabla m_j, \nabla\phi_i)_E, j = 2, 3$, we have

$$(\nabla m_j, \nabla\phi_i)_E = \int_{\partial E} \nabla m_j \cdot \boldsymbol{n}\,\phi_i\,\mathrm{d}s = \frac{1}{2h_E}(\boldsymbol{n}_e^{j-1} + \boldsymbol{n}_{e'}^{j-1}), \tag{10}$$

where e and e' are two edges containing the vertex i, the first $\boldsymbol{n}$ is the unit outwards normal vector of ∂E and $\boldsymbol{n}_e = (n_e^x, n_e^y) = (n_e^1, n_e^2)$ is the scaled normal vector by multiplying the edge length.

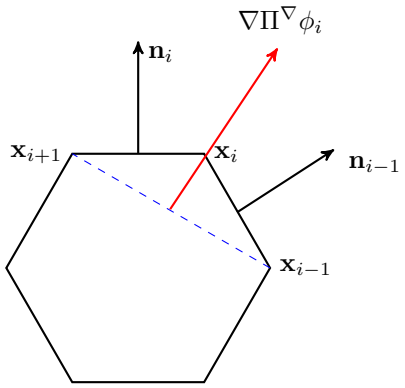


FIGURE 2. $\nabla\Pi^\nabla\phi_i$ is the average of two adjacent normal vectors which is also the normal vector of the line of adjacent vertices.

We thus conclude that

$$\boldsymbol{\Pi}^\nabla = \boldsymbol{G}^{-1}\boldsymbol{B} = \boldsymbol{B}. \tag{11}$$

The matrix-vector representation of

$$\mathrm{dof}(\Pi^\nabla v) = \mathrm{dof}\left(\sum_{\alpha=1}^{3} s^\alpha m_\alpha\right) = \sum_{\alpha=1}^{3} s^\alpha \,\mathrm{dof}\,(m_\alpha)$$

is

$$\boldsymbol{D}\boldsymbol{s} = \boldsymbol{D}\boldsymbol{\Pi}^\nabla\boldsymbol{v} = \boldsymbol{D}\boldsymbol{B}\boldsymbol{v}.$$

In order to compute the stabilization term, one more matrix $\boldsymbol{D}_{n_E\times 3}$ is needed.

$$\boldsymbol{D} := \big(\mathrm{dof}_i(m_j)\big) = \begin{pmatrix} 1 & (x_1-x_E)/h_E & (y_1-y_E)/h_E \\ 1 & (x_2-x_E)/h_E & (y_2-y_E)/h_E \\ 1 & (x_3-x_E)/h_E & (y_3-y_E)/h_E \\ 1 & (x_4-x_E)/h_E & (y_4-y_E)/h_E \\ 1 & (x_5-x_E)/h_E & (y_5-y_E)/h_E \\ \cdots & \cdots & \cdots \\ 1 & (x_{n_E}-x_E)/h_E & (y_{n_E}-y_E)/h_E \end{pmatrix},$$

where $(x_i, y_i), i = 1, \cdots, n_E$ is the coordinate of vertex $\boldsymbol{x}_i$ of E.

Finally the matrix formulation of $\boldsymbol{A}_h^E$ is given by

$$\boldsymbol{A}^E = \boldsymbol{B}^\intercal(:, 2:3)\boldsymbol{B}(2:3,:) + (\boldsymbol{I}-\boldsymbol{D}\boldsymbol{B})^\intercal(\boldsymbol{I}-\boldsymbol{D}\boldsymbol{B}).$$

Here in order to compute the inner product of Π^∇, we only need the second and third rows of $\boldsymbol{B}$. Similarly in the matrix product $\boldsymbol{D}\boldsymbol{B}$ the product the first column of $\boldsymbol{D}$ and the first row of $\boldsymbol{B}$ is simply $1/n_E$.

For the right hand side vector b_i in (4), we also use the inner product of d.o.f f and ϕ_i and approximate

$$(f, \phi_i) \approx \sum_{E\in\mathcal{T}_h, \boldsymbol{x}_i\in E} |E| f(\boldsymbol{x}_i)/n_E.$$

The area of a polygon can be computed by the Green's formulae

$$|E| = \frac{1}{2}\left(\int_{\partial E} x\,\mathrm{d}y - y\,\mathrm{d}x\right) = \frac{1}{2}\sum_{i=1}^{n_E}(x_i y_{i+1} - y_i x_{i+1})$$

assuming the boundary ∂E is oriented counter-clockwise.

2. Implementation in MATLAB

In this section, we discuss how to implement the above formulation efficiently in MATLAB. The code in [6] is not efficient for two reasons. One is that there are large `for` loops. Vectorization should be taken into account to avoid `for` loop as much as possible. Another one is that it does not take advantage of sparsity. The stiffness matrix obtained is very sparse and sparse matrix algorithms require significantly less computational time and computer memory [5]. Note that it is highly inefficient to update a sparse matrix inside a large loop since a lot of operation and memory relocation are involved when the sparse pattern keeps changing. We need to compute all non-zeros first and call `sparse` only once to generate the sparse matrix.

The polygonal mesh we used is generated by `PolyMesher` in MATLAB [7]. The output contains a matrix named `node` which represents the coordinates of vertices and a cell array named `elem` of which each cell records the vertices of each element with a counter-clockwise order.

```
%% Assemble the matrix equation
N = size(node,1); % number of nodes
elemVertexNumber = cellfun('length',elem);% number of vertices per element
nnz = sum(elemVertexNumber.^2); % a upper bound on non-zeros
ii = zeros(nnz,1); % initialization
jj = zeros(nnz,1);
ss = zeros(nnz,1);
b = zeros(N,1);
edge = zeros(sum(elemVertexNumber),2);
index = 0;
edgeIdx = 1;
tic;
fv = pde.f(node); % right hand side evaluated at vertices
for nv = min(elemVertexNumber):max(elemVertexNumber)
    % find polygons with nv vertices
    idx = (elemVertexNumber == nv); % index of elements having nv vertices
    NT = sum(idx); % number of elements having nv vertices
    if NT == 0     % no element has nv vertices
        continue;
    end
    % vertex index and coordinates
    nvElem = cell2mat(elem(idx));
    x1 = reshape(node(nvElem,1),NT,nv);
    y1 = reshape(node(nvElem,2),NT,nv);
    x2 = circshift(x1,[0,-1]);
    y2 = circshift(y1,[0,-1]);
    % record edges
    nextIdx = edgeIdx + NT*nv;
    newEdgeIdx = edgeIdx:nextIdx-1;
    edge(newEdgeIdx,1) = nvElem(:); % get edge per element
```

```
    vertexShift = circshift(nvElem,[0,-1]);
    edge(newEdgeIdx,2) = vertexShift(:);
    edgeIdx = nextIdx;
    % compute geometry quantity: edge, normal, area, center
    bdIntegral = x1.*y2 - y1.*x2;
    area = sum(bdIntegral,2)/2; % the area per element
    h = repmat(sign(area).*sqrt(abs(area)),1,nv); % h is not the diameter
    cx = sum(reshape(node(nvElem(:),1),NT,nv),2)/nv;
    cy = sum(reshape(node(nvElem(:),2),NT,nv),2)/nv;
    normVecx = y2 - y1; % normal vector is a rotation of edge vector
    normVecy = x1 - x2;
    % matrix B, D, I - P
    Bx = (normVecx + circshift(normVecx,[0,1]))./(2*h); % average of normal vectors
    By = (normVecy + circshift(normVecy,[0,1]))./(2*h); % in adjaency edges
    Dx = (x1 - repmat(cx,1,nv))./h; %  m = (x - cx)/h
    Dy = (y1 - repmat(cy,1,nv))./h;
    IminusP = zeros(NT,nv,nv);
    for i = 1:nv
        for j = 1:nv
            IminusP(:,i,j) = - 1/nv - Dx(:,i).*Bx(:,j) - Dy(:,i).*By(:,j);
        end
        IminusP(:,i,i) = ones(NT,1) + IminusP(:,i,i);
    end
    % assemble the matrix
    for i = 1:nv
        for j = 1:nv
            ii(index+1:index+NT) = nvElem(:,i);
            jj(index+1:index+NT) = nvElem(:,j);
            ss(index+1:index+NT) = Bx(:,i).*Bx(:,j) + By(:,i).*By(:,j) ...
                                 + dot(IminusP(:,:,i),IminusP(:,:,j),2);
            index = index + NT;
        end
    end
    % compute the right hand side
    patchArea = accumarray(nvElem(:),repmat(area/nv,nv,1),[N 1]);
    b = b + fv.*patchArea;
end
A = sparse(ii,jj,ss,N,N);
```

We still have a `for` loop but of small size. We loop over all possible number of vertices: $3, 4, 5, \ldots$ and assume the maximum is uniformly bounded. For all elements of the same number of vertices, we use vectorization to compute geometric quantities for all elements. For example, `x1` is a tall matrix of size `NT x nv` representing the x-coordinate of vertices. `Bx` is also of size `NT x nv` which is the second row of matrix $\boldsymbol{B}$ but for all elements. `circshift` is used for getting neighbors for each node and then areas can be computed using matrix operations which drops the need to loop over all elements.

Besides the `sparse` command for generating sparse matrices, we also use the build in command `accumarray` in MATLAB to generate dense matrices with small number of columns, again avoiding the `for` loop over all elements.

```
%% Find boundary edges and nodes
totalEdge = sort(edge(:,1:2),2);
[i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
```

```
bdEdge   = [j(s==1), i(s==1)]; % find the boundary edge
isBdNode = false(N,1);
isBdNode(bdEdge) = true;
%% Impose Dirichlet boundary conditions
u = zeros(N,1);
u(isBdNode) = pde.g_D(node(isBdNode,:));
b = b - A*u;
%% Solve Au = b
isFreeNode = ~isBdNode; % all interior nodes are free
u(isFreeNode) = A(isFreeNode,isFreeNode)\b(isFreeNode);
```

The interior edges are repeated twice in `totalEdge`. We use the summation property of `sparse` command to merge the duplicated indices. The nonzero vector `s` takes value 1 (for boundary edges) or 2 (for interior edges). We can find the boundary edges using the subset of indices pair corresponding to the nonzero value 1. More explanation can be found in [3].

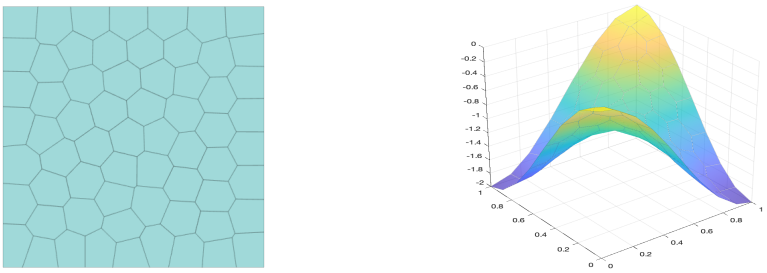
We then impose the Dirichlet boundary conditions and solve the matrix equation by the build-in direct solver. Other boundary conditions can be implemented following the standard FEM procedure; see [5].

3. Numerical Results

The subroutine `PoissonVEM.m` has been added into *i*FEM [4]. The test script is `squarePoissonVEM.m` stored in `ifem/example/Poisson/` folder. We choose the exact solution is $u = \cos(\pi x)\cos(\pi y) - 1$ and compute the right hand side $f = 2\pi^2 \cos(\pi x)\cos(\pi y)$. We shall compute the following errors:

$$\|u_I - u_h\|_A = \left[(u_I - u_h)^\intercal \boldsymbol{A}(u_I - u_h)\right]^{1/2},$$
$$\|u_I - u_h\|_\infty = \max|u_I - u_h|,$$

where u_h is the numerical solution obtained by the linear virtual element methods; u_I is the nodal interpolation of the exact solution u in $\mathcal{T}_h$ and $\boldsymbol{A}$ is the stiffness matrix of linear VEM.



(a) A polygonal mesh. (b) Linear VEM solution

FIGURE 3. Use `showmeshpoly(node,elem)` to plot a polygonal mesh and `showsolutionpoly(node,elem,u)` to plot the linear VEM function.

The errors of $\|u_I - u_h\|_A$ and $\|u_I - u_h\|_\infty$ are presented in Table 4 and the decay rate is shown in Fig. 5. We use number of nodes instead traditional maximal diameter h to measure the decay rate. In the uniform mesh case, $N^{-1} = \mathcal{O}(h^2)$ and $N^{-0/5} = \mathcal{O}(h)$.

We conclude that the order of accuracy in energy norm is more than 1 and in maximum norm is almost 2. It verifies that virtual element method is feasible for solving Poisson equations with polygonal meshes in two dimensions.

```
  Dof      NT    ||DuI-Du_h||  ||uI-u_h||_{max}

  130      64   7.32287e-02   2.80832e-02
  514     256   2.63733e-02   7.54620e-03
 2044    1024   1.06655e-02   1.66407e-03
 8168    4096   4.93804e-03   5.48253e-04
32652   20014   2.28291e-03   1.16791e-04
```

FIGURE 4. Error of Linear VEM to Poisson equation.

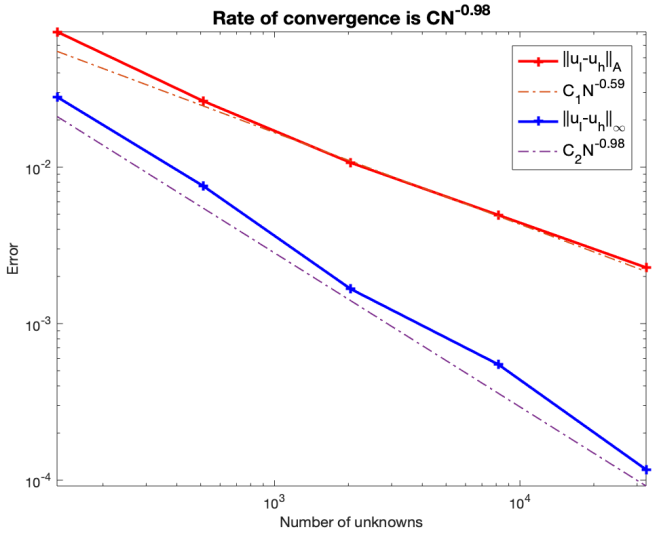


FIGURE 5. Decay rate of error of Linear VEM to Poisson equation.

Finally, we also compare the corresponding assembling CPU time along with that in [6] in Table 1, which reveals that our assembling time scales linearly. While it is more expensive to assemble the stiffness matrix due to the large `for` loop as the number of nodes is large enough in [6].

ACKNOWLEDGEMENT

The authors would like to thank Dr. Shuhao Cao for adding subroutines on polygon meshes into *i*FEM.

REFERENCES

[1] L. Beirão da Veiga, F. Brezzi, A. Cangiani, G. Manzini, L. Marini, and A. Russo. Basic principles of virtual element methods. *Mathematical Models and Methods in Applied Sciences*, 23(01):199–214, 2013. 1, 3

TABLE 1. The comparison of assembling CPU time (in seconds)

#Dofs	Assemble algorithm in this notes	Assemble algorithm in [6]
130	0.019	0.042
514	0.012	0.194
2,044	0.016	0.521
8,168	0.049	3.441
32,652	0.196	40.236

[2] L. Beirao da Veiga, F. Brezzi, L. Marini, and A. Russo. The hitchhiker's guide to the virtual element method. *Mathematical Models and Methods in Applied Sciences*, 24(08):1541–1573, 2014. 1

[3] L. Chen. Data structure for triangulations, 2008. 8

[4] L. Chen. ifem: an integrated finite element methods package in matlab. *University of California at Irvine*, 2009. 8

[5] L. Chen. Programming of finite element methods in MATLAB. *arXiv preprint arXiv:1804.05156*, 2018. 6, 8

[6] O. J. Sutton. The virtual element method in 50 lines of matlab. *Numerical Algorithms*, 75(4):1141–1159, 2017. 1, 6, 9, 10

[7] C. Talischi, G. H. Paulino, A. Pereira, and I. F. Menezes. Polymesher: a general-purpose mesh generator for polygonal elements written in matlab. *Structural and Multidisciplinary Optimization*, 45(3):309–328, 2012. 6

[8] H. Wei. FEALPy: Finite element analysis library in python. https://github.com/weihuayi/fealpy, 2017-2020. 1