Unified Implementation of Adaptive Finite Element Methods in Matlab

Yue Yu*

School of Mathematical Sciences, Institute of Natural Sciences, MOE-LSC, Shanghai Jiao Tong University, Shanghai, 200240, P. R. China.

Abstract

We provide a unified implementation of the adaptive finite element methods for the Poisson equation in Matlab by combing the use of the software package varFEM. The design idea can be extended to other Galerkin-based methods, for example, the discontinues Galerkin methods and the virtual element methods.

1 Finite element methods for the Poisson equation

In this section we briefly review the finite element method and introduce the numerical implementation in varFEM, a Matlab software package for the finite element method. For simplicity, we consider the Poisson equation with the homogeneous Dirichlet boundary conditions.

1.1 The finite element method for the Poisson equation

Let Ω be a bounded Lipschitz domain in $\mathbb{R}^2$ with polygonal boundary $\partial\Omega$. Consider the Poisson equation

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases} \tag{1.1}$$

where $f \in L^2(\Omega)$ is a given function. The continuous variational problem is to find $u \in V := H_0^1(\Omega)$ such that

$$a(u,v) = \ell(v), \quad v \in V,$$

where

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \mathrm{d}\sigma, \qquad \ell(v) = \int_\Omega f v \mathrm{d}\sigma.$$

For the finite element discretization, we discuss the conforming Lagrange elements. Let $\mathcal{T}_h$ be a shape regular triangulation. The generic element will be denoted as K in the sequel. Define

$$V_h = \{v \in V : v|_K \in \mathbb{P}_k(K), \quad K \in \mathcal{T}_h\},$$

---

*terenceyuyue@sjtu.edu.cn

where $k \leq 3$. The discrete problem is to seek $u_h \in V_h$ satisfying

$$a(u_h, v) = \ell(v), \qquad v \in V_h. \tag{1.2}$$

1.2 Data structures for triangular meshes

We adopt the data structures given in iFEM [1]. All related data are stored in the Matlab structure `Th`, which is computed by using the subroutine FeMesh2d.m as

```
1 Th = FeMesh2d(node,elem,bdStr);
```

where the basic data structures `node` and `elem` are generated by

```
1 [node,elem] = squaremesh([x1 x2 y1 y2], h1, h2);
```

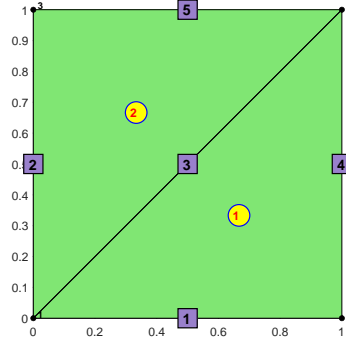For clarity, we take a simple mesh shown in Fig. 1 as an example.

Fig. 1: Illustration of the data structures

The triangular meshes are represented by two basic data structures `node` and `elem`, where `node` is an $\text{N} \times 2$ matrix with the first and second columns contain x- and y-coordinates of the nodes in the mesh, and `elem` is an $\text{NT} \times 3$ matrix recording the vertex indices of each element in a counterclockwise order, where `N` and `NT` are the numbers of the vertices and triangular elements. For the mesh given in Fig. 1,

$$\texttt{elem} = \begin{bmatrix} 2 & 4 & 1 \\ 3 & 1 & 4 \end{bmatrix}$$

In the current version, we only consider the $\mathbb{P}_k$-Lagrange finite element spaces with k up to 3. In this case, there are two important data structures `edge` and `elem2edge`. In the matrix `edge(1:NE,1:2)`, the first and second rows contain indices of the starting and ending points. The column is sorted in the way that for the k-th edge, `edge(k,1)<edge(k,2)` for $k = 1, 2, \cdots, \text{NE}$. For the given triangulation,

$$\texttt{edge} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 2 & 4 \\ 3 & 4 \end{bmatrix}$$

2

The matrix `elem2edge` establishes the map of local index of edges in each triangle to its global index in matrix `edge`. For the given mesh,

$$\texttt{elem2edge} = \begin{bmatrix} 3 & 1 & 4 \\ 3 & 5 & 2 \end{bmatrix}.$$

By convention, we label three edges of a triangle such that the i-th edge is opposite to the i-th vertex. We refer the reader to https://www.math.uci.edu/~chenlong/ifemdoc/mesh/auxstructuredoc.html for some detailed information.

To deal with boundary integrals, we first extract the boundary edges from `edge` and store them in matrix `bdEdge`. In the input of `FeMesh2d`, the string `bdStr` is used to indicate the interested boundary part in `bdEdge`. For example, for the unit square $\Omega = (0,1)^2$,

- `bdStr = 'x==1'` divides `bdEdge` into two parts: `bdEdgeType{1}` gives the boundary edges on $x = 1$, and `bdEdgeType{2}` stores the remaining part.

- `bdStr = {'x==1','y==0'}` separates the boundary data structure `bdEdge` into three parts: `bdEdgeType{1}` and `bdEdgeType{2}` give the boundary edges on $x = 1$ and $y = 0$, respectively, and `bdEdgeType{3}` stores the remaining part.

- `bdStr = []` implies that `bdEdgeType{1} = bdEdge`.

We also use `bdEdgeIdxType` to record the index in matrix `edge`, and `bdNodeIdxType` to store the nodes for respective boundary parts. Note that we determine the boundary of interest by the coordinates of the midpoint of the edge, so `'x==1'` can also be replaced by a statement like `'x>0.99'`.

1.3 Implementation of the FEM in varFEM

FreeFEM is a popular 2D and 3D partial differential equations (PDE) solver based on finite element methods [2], which has been used by thousands of researchers across the world. The highlight is that the programming language is consistent with the variational formulation of the underlying PDEs, referred to as the variational formulation based programming in [3], where we have developed an FEM package in a similar way of FreeFEM using the language of Matlab, named varFEM. The similarity here only refers to the programming style of the main or test script, not to the internal architecture of the software.

Consider the Poisson equation with Dirichlet boundary condition on the unit square. The exact solution is given by

$$u(x,y) = xy(1-x)(1-y)\exp\left(-1000((x-0.5)^2 + (y-0.117)^2)\right).$$

The PDE data is generated by `pde = Poissondata_afem()`. The function file is simply given as follows.

```
1  function uh = varPoisson(Th,pde,Vh,quadOrder)
2
3  %% Assemble stiffness matrix
```

```
 4 Coef  = 1;
 5 Test  = 'v.grad';
 6 Trial = 'u.grad';
 7 kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
 8
 9 %% Assemble right hand side
10 % Omega
11 Coef = pde.f;   Test = 'v.val';
12 ff = assem2d(Th,Coef,Test,[],Vh,quadOrder);
13
14 %% Apply Dirichlet boundary conditions
15 g_D = pde.g_D;
16 on = 1;
17 uh = apply2d(on,Th,kk,ff,Vh,g_D);
```

In the above code, the structure `pde` stores the information of the PDE, including the exact solution `pde.uexact`, the gradient `pde.Du`, etc. We set up the triple (`Coef`, `Test`, `Trial`) for the coefficients, test functions and trial functions in variational form, respectively. It is obvious that `v.grad` is for ∇v and `v.val` is for v itself. The routine `assem2d.m` computes the stiffness matrix corresponding to the bilinear form on the two-dimensional region, i.e.

$$A = (a_{ij}), \quad a_{ij} = a(\Phi_i, \Phi_j),$$

where Φ_i are the global shape functions of the finite element space `Vh`. The integral of the bilinear form, as $(\nabla\Phi_i, \nabla\Phi_j)_\Omega$, is approximated by using the Gaussian quadrature formula with `quadOrder` being the order of accuracy.

We remark that the `Coef` has three forms:

1. A function handle or a constant.

2. The numerical degrees of freedom of a finite element function.

3. A coefficient matrix `CoefMat` resulting from the numerical integration.

In the computation, the first two forms in fact will be transformed to the third one. Given a function $c(p)$, where $p = (x, y)$, the coefficient matrix is in the following form

$$\texttt{CoefMat} = \begin{bmatrix} c(p_1^1) & c(p_2^1) & \cdots & c(p_{n_g}^1) \\ c(p_1^2) & c(p_2^2) & \cdots & c(p_{n_g}^2) \\ \vdots & \vdots & \vdots & \vdots \\ c(p_1^{\text{NT}}) & c(p_2^{\text{NT}}) & \cdots & c(p_{n_g}^{\text{NT}}) \end{bmatrix}. \tag{1.3}$$

Here, $p_1^i, p_2^i, \cdots, p_{n_g}^i$ are the quadrature points on the element K_i.

We display the numerical result in Fig. 2 for the uniform triangular mesh with $h_1 = h_2 = 1/50$ generated by

```
 1 [node,elem] = squaremesh([0 1 0 1], 1/50, 1/50);
```

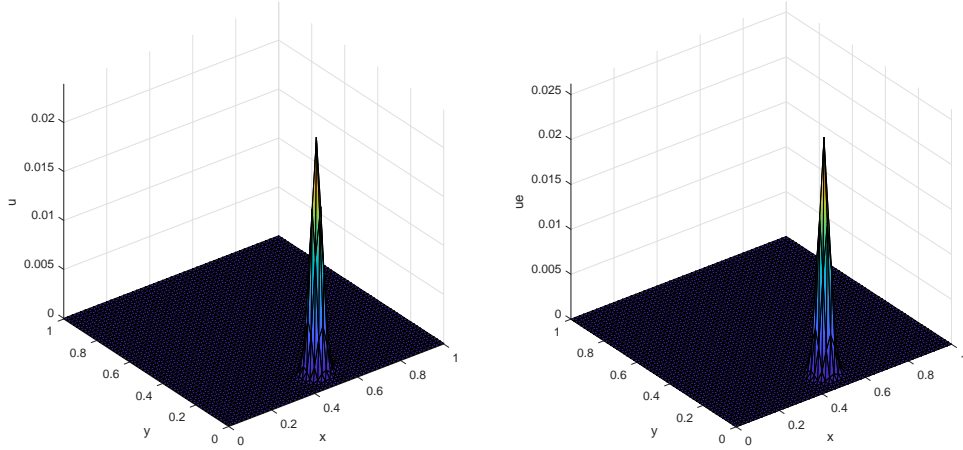Fig. 2: Numerical and exact solutions with $h_1 = h_2 = 1/50$

2 Adaptive finite element methods for the Poisson equation

In this section we briefly introduce the ingredients of the adaptive finite element method, and provide the overall structure of the implementation.

2.1 The ingredients of the adaptive FEM

For the conforming FEM, one can establish the following residual based a-posteriori error estimate

$$\|u - u_h\|_1 \lesssim \eta(u_h),$$

where $\eta = \left(\sum_{K \in \mathcal{T}_h} \eta_K^2 \right)^{1/2}$ and

$$\eta_K^2 = h_K^2 \|f + \Delta u_h\|_{0,K}^2 + \sum_{e \subset \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2 \tag{2.1}$$

are referred to as the global and local error indicators, respectively. Here, $f + \Delta u_h$ is the interior residual, and

$$[\partial_{n_e} u_h] = \partial_{n_e} u_h|_{K_1} - \partial_{n_e} u_h|_{K_2}$$

is the jump term, where n_e is the unit outer normal vector of the target element $K = K_1$.

Standard adaptive algorithms based on the local mesh refinement can be written as loops of the form

$$\textbf{SOLVE} \rightarrow \textbf{ESTIMATE} \rightarrow \textbf{MARK} \rightarrow \textbf{REFINE}.$$

Given an initial subdivision $\mathcal{T}_0$, to get $\mathcal{T}_{k+1}$ from $\mathcal{T}_k$ we first solve the FEM problem under consideration to get the numerical solution u_k on $\mathcal{T}_k$. The error is then estimated by using u_k, $\mathcal{T}_k$ and the a posteriori error bound $\eta(u_h)$. The local error bound η_K is used to mark a subset $\mathcal{M}$ of elements in $\mathcal{T}_k$ for refinement. The marked triangles and possible more neighboring elements are refined in such a way that the subdivision meets certain conditions, for example, the resulting triangular

5

mesh is still shape regular. The above procedures are included in the test script with an overview given as follows

```matlab
for k = 1:maxIt
    % Step 1: SOLVE
    Th = FeMesh2d(node,elem,bdStr);
    uh = varPoisson(Th,pde,Vh,quadOrder);
    % Step 2: ESTIMATE
    eta = Poisson_indicator(Th,uh,pde,Vh,quadOrder);
    % Step 3: MARK
    elemMarked = mark(elem,eta,theta);
    % Step 4: REFINE
    [node,elem] = bisect(node,elem,elemMarked);
end
```

Three important modules are involved: the local error indicator in Step 2, the marking algorithm in Step 3 and the local refinement algorithm in Step 4. We employ the Dörfler marking strategy to select the subset of elements and then use the newest vertex method to refine the mesh. Note that the subroutines `mark.m` and `bisect.m` are extracted from *i*FEM [1] with some minor modifications. In this article, we are only concerned with the computation of the error indicator. For the later two steps, we refer the reader to [1] for details on the implementation.

2.2 The unified implementation of the error indicator

2.2.1 The computation of the elementwise residuals

With the help of varFEM, the first term $h_K^2 \|f + \Delta u_h\|_{0,K}^2$ in (2.1) can be simply computed as

```matlab
%% elementwise residuals
fc = interp2dMat(pde.f,'.val',Th,Vh,quadOrder);
uxxc = interp2dMat(uh,'.dxx',Th,Vh,quadOrder);
uyyc = interp2dMat(uh,'.dyy',Th,Vh,quadOrder);
Coef = (fc + uxxc + uyyc).^2;
[¬,elemIh] = integral2d(Th,Coef,Vh,quadOrder);
elemRes = diameter.^2.*elemIh;
```

In the above code, `elemIh` stores all the local error indicators $[\eta_{K_1}, \cdots, \eta_{K_{\mathrm{NT}}}]^T$; The function `interp2dMat` is used to generate the coefficient matrix (see (1.3)). It is obvious that the coefficient matrix of $(f + \Delta u_h)^2$ is `(fc+uxxc+uyyc).^2`, where `fc`, `uxxc` and `uyyc` are the coefficient matrices of f, $\partial_{xx} u_h$ and $\partial_{yy} u_h$, respectively.

In what follows, we focus on the unified implementation of the jump term or jump integral $\sum_{e \subset \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2$. We remark that any other type of jump terms can be easily adapted or designed accordingly as will be seen.

2.2.2 The elementwise interior and exterior indices of the quadrature points

The integral over e is calculated by using the one-dimensional Gaussian numerical integration formula

$$\int_e f \mathrm{d}s = |e|(w_1 f(p_1) + w_2 f(p_2) + \cdots + w_{n_g} f(p_{n_g})),$$

where w_i and p_i are the quadrature weights and points on e, and n_g is the number of the quadrature points. Note that the endpoints of e are not included.
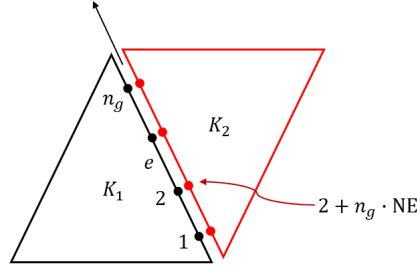

Fig. 3: Illustration of the quadrature points

1. Indexing rule for the quadrature points. In Fig. 3 the direction of the edge e is specified by the arrow. In the implementation, the direction is determined by the data structure `edge` which satisfies `edge(k,1)<edge(k,2)`. For the first edge $e = e_1$ in `edge`, the quadrature points will be numbered by $1, 2, \cdots, n_g$ or $1 : n_g$ for short when restricted to the left triangle. Similarly, the quadrature points on the i-th edge e_i will be numbered by

$$p_i^L = (1 : n_g) + (i-1)n_g, \qquad i = 1, 2, \cdots, \text{NE}.$$

The right-hand restrictions are accordingly numbered as

$$p_i^R = p_i^L + n_g \cdot \text{NE}, \qquad i = 1, 2, \cdots, \text{NE}.$$

2. Elementwise sign matrix. To characterize the direction of an edge on an element K, one can introduce the elementwise sign matrix

```
1 % sign of elementwise edges
2 sgnelem = sign([elem(:,3)-elem(:,2), elem(:,1)-elem(:,3), elem(:,2)-elem(:,1)]);
```

In some cases, it is better to restore the positive sign for edges on the boundary of the domain:

```
1 E = false(NE,1); E(bdEdgeIdx) = 1; sgnbd = E(elem2edge);
2 sgnelem(sgnbd) = 1;
```

Here, the data structures `elem2edge` and `bdEdgeIdx` have been introduced in Subsect. 1.2.

3. Elementwise interior indices. According to the indexing rule, one easily obtains the interior indices of the first sides of all triangles:

```
1 % first side
2 e1 = elem2edge(:,1); sgn1 = sgnelem(:,1);
3 id = repmat(1:ng, NT,1);
4 id(sgn1<0,:) = repmat((ng:-1:1)+NE*ng, sum(sgn1<0), 1);
5 elem2dofe1 = id + (e1-1)*ng;
```

Note that for the edges with positive signs the natural indices are $1 : ng$, while for those with negative signs the natural indices are reversed. One can similarly introduce the interior indices of other sides, and hence give the elementwise interior indices

```
1 elem2dofM = [elem2dofe1,elem2dofe2,elem2dofe3];
```

where the letter `M` stands for "Minus".

4. Elementwise exterior indices. To compute the jump, we also need to introduce the elementwise exterior indices `elem2dofP`, where `P` is for "Plus". Given an edge, let the interior indices of the quadrature points are $i_1, \cdots, i_{n_g}$, and the exterior indices are $i'_1, \cdots, i'_{n_g}$. Since $|i_k - i'_k| = n_g \cdot$ `NE`, one just needs to subtract $n_g \cdot$ `NE` for those indices in `elem2dofM` greater than $n_g \cdot$ `NE`, and add $n_g \cdot$ `NE` to those indices less than $n_g \cdot$ `NE`.

```
1 index = (elem2dofM>ng*NE);
2 elem2dofP = elem2dofM + (-ng*NE)*index + ng*NE*(¬index);
```

2.2.3 The elementwise interior and exterior evaluations

Given u_h, we now determine the interior evaluations `elemuhM` and the exterior evaluations `elemuhP` of u_h corresponding to `elem2dofM` and `elem2dofP`. To this end, one can compute the evaluations of the basis functions at the quadrature points along the boundary ∂K. Given a triangle K, let λ_1, λ_2 and λ_3 be the barycentric coordinate functions. Since λ_i are usually used to construct the basis functions for the FEMs, one can first specify the evaluations of λ_i and the derivatives $\partial_x \lambda_i$ and $\partial_y \lambda_i$ at the quadrature points.

Given some points $p_1, p_2, \cdots, p_{n_G}$ on the triangle K, for example, the quadrature points for the integration of the bilinear forms. In varFEM, the evaluations of λ_i are stored in the following form

$$
\begin{bmatrix}
\lambda_1(p_1) & \lambda_2(p_1) & \lambda_3(p_1) \\
\lambda_1(p_2) & \lambda_2(p_2) & \lambda_3(p_2) \\
\vdots & \vdots & \vdots \\
\lambda_1(p_{n_G}) & \lambda_2(p_{n_G}) & \lambda_3(p_{n_G})
\end{bmatrix}.
$$

At this time, one can choose p_j as the quadrature points along the boundary ∂K. Let $\partial K = e_1 \cup e_2 \cup e_3$. The quadrature points on e_i are denoted by $p_{1,e_i}, \cdots, p_{n_g,e_i}$. In this case, $n_G = 3n_g$, and one can specify the values $\lambda_i(p_j)$ by using the 1-D quadrature points $r_1, r_2, \cdots, r_{n_g}$. The Gaussian quadrature points and weights $r = [r_1, r_2, \cdots, r_{n_g}]$ and $w = [w_1, w_2, \cdots, w_{n_g}]$ on $[0, 1]$ are given by `quadpts1.m`:

```
1 [lambda1d,weight1d] = quadpts1(4);  ng = length(weight1d);
2 [¬,id] = sort(lambda1d(:,1));
3 lambda1d = lambda1d(id,:); weight1d = weight1d(id);
```

Here we use `sort` to guarantee $r_1 < r_2 < \cdots < r_{n_g}$. Note that

$$
\text{lambda1d} = \begin{bmatrix}
r_1 & r_{n_g} \\
r_2 & r_{n_g-1} \\
\vdots & \vdots \\
r_{n_g} & r_1
\end{bmatrix},
$$

and all the row sums are 1, i.e. $r_1 + r_{n_g} = r_2 + r_{n_g - 1} = \cdots = 1$. By the definition of the barycentric coordinate functions, the coordinates $(\lambda_1, \lambda_2, \lambda_3 = 1 - \lambda_1 - \lambda_2)$ on the three sides of K are:

- 1-th side: $(0, r_{n_g}, r_1), (0, r_{n_g - 1}, r_2), \cdots, (0, r_1, r_{n_g})$;

- 2nd side: $(r_1, 0, r_{n_g}), (r_2, 0, r_{n_g - 1}), \cdots, (r_{n_g}, 0, r_1)$;

- 3rd side: $(r_{n_g}, r_1, 0), (r_{n_g - 1}, r_2, 0), \cdots, (r_1, r_{n_g}, 0)$.

Therefore the $n_G = 3n_g$ points can be given as

```
 1 function [lambdaBd,weightBd] = quadptsBd(order)
 2 %%Gauss quadrature points along the boundary of triangles
 3
 4 % quadrature on [0,1]
 5 [lambda1d,weight1d] = quadpts1(order);  ng = length(weight1d);
 6 [¬,id] = sort(lambda1d(:,1));
 7 lambda1d = lambda1d(id,:); weight1d = weight1d(id);
 8 % quadrature on each side
 9 lambdae1 = [zeros(ng,1), lambda1d(:,2), lambda1d(:,1)];
10 lambdae2 = [lambda1d(:,1), zeros(ng,1), lambda1d(:,2)];
11 lambdae3 = 1 - lambdae1 - lambdae2;
12 % quadrature along the boundary of each element
13 lambdaBd = [lambdae1; lambdae2; lambdae3];
14 weightBd = repmat(weight1d,1,3);
```

With these quadrature points one can construct the evaluations of the basis functions. The details are omitted. Please refer to Base2dBd.m for implementation details. For the $\mathbb{P}_1$ element, the basis functions are $\phi_i = \lambda_i$, and the elementwise interior evaluations of u_h are given as follows

```
 1 elemuhM = zeros(NT,3*ng);
 2 for p = 1:3*ng
 3     % interpolation at the p-th quadrture point
 4     for i = 1:length(phi)
 5         base = phi{i};
 6         elemuhM(:,p) = elemuhM(:,p) + uh(elem2dof(:,i)).*base(:,p);
 7     end
 8 end
```

For other types of finite elements, one just needs to modify the basis functions base. For evaluations of the derivatives, for example, $\partial_x u_h$, one needs to change base to the partial derivative ∂_x of the basis functions. The elementwise exterior evaluations of u_h can be computed as

```
 1 uhI = zeros(2*NE*ng,1);
 2 uhI(elem2dofM) = elemuhM;
 3 elemuhP = uhI(elem2dofP);
```

The above discussions are summarized in a subroutine:

```
 1 function [elemuhM,elemuhP] = elem2edgeInterp(wStr,Th,uh,Vh,quadOrder)
```

One can obtain the evaluations of u_h, $\partial_x u_h$ and $\partial_y u_h$ by setting wStr as .val, .dx and .dy, respectively.

2.2.4 The computation of the jump integral

With the above preparations, we are able to compute the jump integral

$$h_e\|[\partial_{n_e}u_h]\|_{0,e}^2 = h_e \int_e ([\partial_x u_h]n_x + [\partial_y u_h]n_y)^2 \mathrm{d}s = h_e \cdot |e| \sum_{i=1}^{n_g} w_i([\partial_x u_h]n_x + [\partial_y u_h]n_y)^2(p_{i,e}).$$

- The first step is to compute the elementwise interior and exterior evaluations of $\partial_x u_h$ and $\partial_y u_h$:

```
%% elementwise interior and exterior evaluations at quadrature points
[elemuhxM,elemuhxP] = elem2edgeInterp('.dx',Th,uh,Vh,quadOrder);
[elemuhyM,elemuhyP] = elem2edgeInterp('.dy',Th,uh,Vh,quadOrder);
```

- The elementwise jumps are

```
elem2Jumpx = elemuhxM - elemuhxP;
elem2Jumpy = elemuhyM - elemuhyP;
```

- The jump integral can be computed by looping of triangle sides:

```
elemJump = zeros(NT,1);
[¬,weight1d] = quadpts1(quadOrder);
ng = length(weight1d);
for i = 1:3 % loop of triangle sides
    hei = he(elem2edge(:,i));
    cei = hei;
    neix = ne(elem2edge(:,i),1); neiy = ne(elem2edge(:,i),2);
    Jumpnx = elem2Jumpx(:,(1:ng)+(i-1)*ng).*repmat(neix,1,ng);
    Jumpny = elem2Jumpy(:,(1:ng)+(i-1)*ng).*repmat(neiy,1,ng);
    Jumpn = (Jumpnx+Jumpny).^2;
    elemJump = elemJump + cei.*hei.*(Jumpn*weight1d(:));
end
```

- We finally get the local error indicators

```
%% Local error indicator
eta = (abs(elemRes) + elemJump).^(1/2);
```

3 Numerical example

Consider the example given in Subsect. 1.3. We employ the Dörfler marking strategy with parameter $\theta = 0.4$ to select the subset of elements for refinement.

The initial mesh and the final adapted meshes after 20 and 30 refinement steps are presented in Fig. 4 (a-c), respectively, where the $\mathbb{P}_1$ element is used. We also plot the adaptive approximation in Fig. 5, which almost coincides with the exact solution. The convergence rates of the error estimators and the errors in H^1 norm are shown in Fig. 6, from which we observe the optimal rates of convergence as predicted by the theory. The full code is available from varFEM package (https://github.com/Terenceyuyue/mVEM). The subroutine PoissonVEM_indicator.m is used to compute the local indicator and the test script is main_Poisson_afem.m.
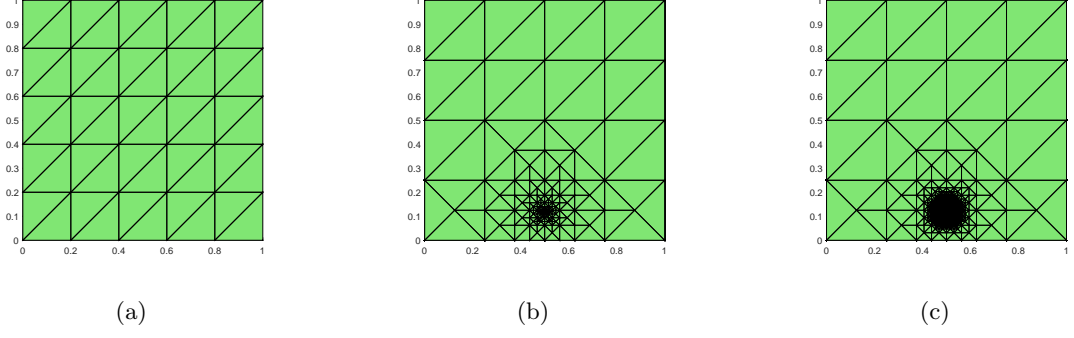
Fig. 4: The initial and the final adapted meshes. (a) The initial mesh; (b) After 20 refinement steps; (c) After 30 refinement steps
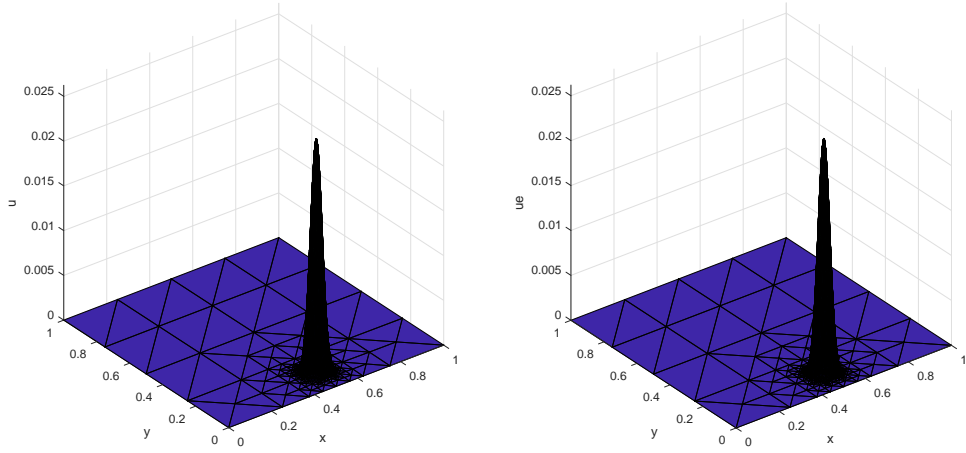

Fig. 5: The exact and numerical solutions

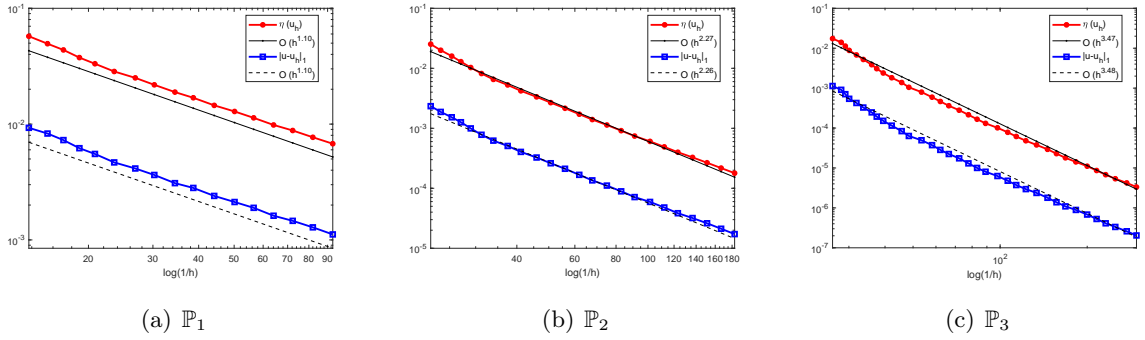(a) $\mathbb{P}_1$ (b) $\mathbb{P}_2$ (c) $\mathbb{P}_3$

Fig. 6: The convergence rates of the error estimators and the errors in H^1 norm

4 Concluding remarks

In this paper, a unified implementation of the adaptive finite element methods was presented for the model problem, which combines the use of the Matlab software package varFEM. The design idea can be extended to other types of Galerkin methods.

11

References

[1] L. Chen. iFEM: an integrated finite element method package in MATLAB. Technical report, University of California at Irvine, 2009.

[2] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[3] Y. Yu. varFEM: variational formulation based programming for finite element methods in Matlab. arXiv:2206.06918, 2022.