

4 多边形网格加密

4.1 加密规则

在对单元进行加密时, 一个必要的规则是单元的边必须进行细分, 从而对多边形网格, 一个自然的加密方式是图 11 (a) 所示的四点划分. 这里, 目标单元通过连接重心和各边的中点划分为若干个四边形. 注意, 对有悬挂点 Q 的单元, 最好补充从重心到点 Q 的一个新边, 而不是对边 PQ 和 QR 进行二分, 否则可能出现退化三角形. 我们称这种处理为容许二分. 如图 11 (b) 所示, 三角形 $\Delta 123$ 由二分 PQ 和 QR 而产生, 新的三角形 $\Delta 567$ 由三角形 $\Delta 123$ 经类似方式获得. 根据三角形的性质, 重心 z_7 位于中线 e_{34} 上且满足 $|e_{47}| : |e_{73}| = 1 : 2$, 这意味着 $\angle 7 > \angle 3$. 这样, 如果继续“双边二分”, 那么退化三角形就会出现. 另一方面, 容许二分也导致更简单的多边形单元.

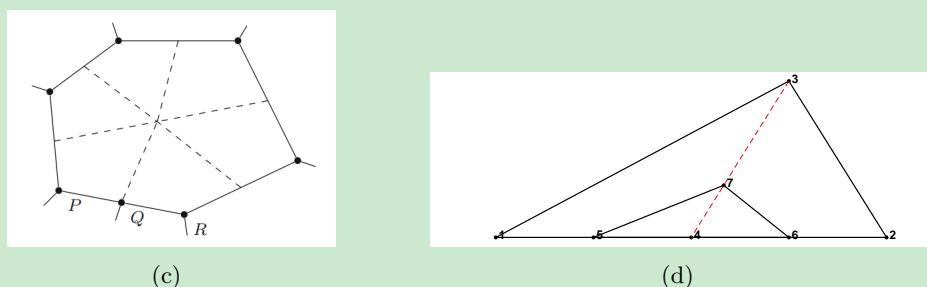


图 11. 边二分. (a) 容许二分: 添加从悬挂点到重心的新边; (b) 双边二分: 将悬挂点连接的两条边均进行二分

一个网格加密函数应具有如下形式

```
[node, elem] = PolyMeshRefine(node, elem, idElemMarked).
```

这里, `node` 和 `elem` 是基本数据结构, 存储网格的节点坐标和顶点连通性, 而数组 `idElemMarked` 给出标记单元的索引. 一般而言, 待加密单元要多于标记单元. 若不对单条边上的悬挂点个数做出限制, 则很可能违反无短边假设. 尽管这种要求对 VEMs 不是必要的, 但我们仍期望获得没有短边的高质量网格. 为此, 一些额外的单元需要加入到待加密单元集合中.

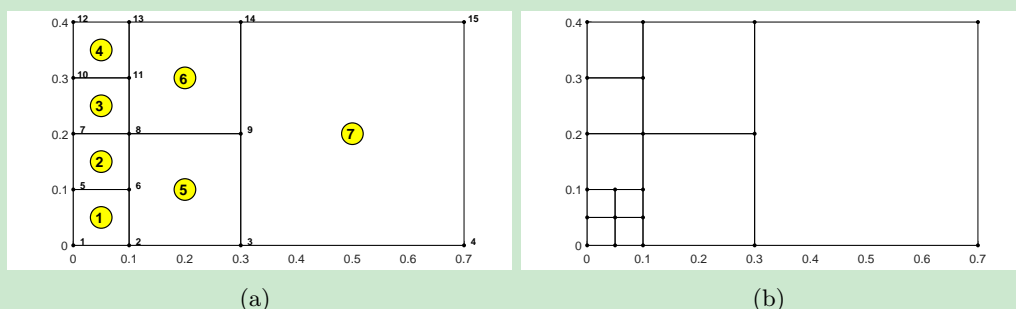


图 12. (a) Initial mesh; (b) Refinement of only the marked elements.

给定图 12 (a) 中的初始网格, 我们发现短边问题归结为两个相邻单元 ① 和 ⑤ 的加密这一典型情形.

- 假设 ① 是标记单元: `idElemMarked = 1`. 如果我们只按图 12 (b) 那样加密标记单元, 并且继续划分右下角的小单元, 那么单元 ⑤ 中就会出现短边 (由于不断地添加悬挂点). 这种现象也出现在图 11 (b) 中的 two-edge bisection 中, 后者通过使用容许二分得以解决.

- 为了避免短边的出现, 我们进一步加密相邻单元 ⑤, 见图 13. 也就是说, 对以 e 为公共边的两个单元 K_1 和 K_2 , 若 K_1 在待加密集中且 e 的一个端点是 K_2 的悬挂点, 则同时加密 K_1 和 K_2 .

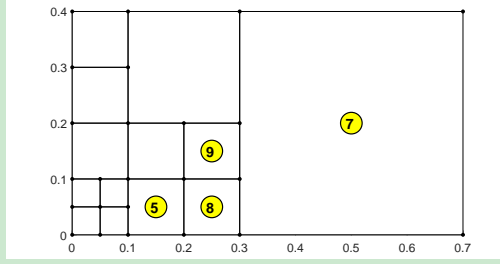


图 13. Refine both ① and ⑤

- 对图 12 (a) 中的单元, ⑤ 和 ⑦ 可分别视为 ① 和 ⑤, 基于同样的原因, 我们也需要划分单元 ⑦.
- 重复上面的过程, 我们就可以获得所有新的待加密单元.

注意, 标记单元本身可能出现上面指出的情况. 例如, 图 14 中红色的单元 9, 19 和 20 就是如此, 此时含悬挂点的单元 9 的地位和额外需要加密的单元的地位类似. 把这些地位等同的标记单元也加入额外单元中, 就可避免产生短边, 因为上面的处理确保每条边仅有一个悬挂点 (a midpoint of the collinear edge).

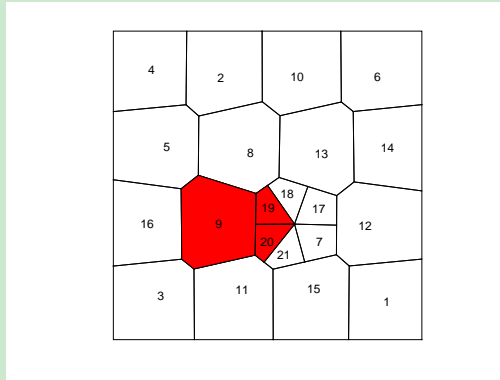


图 14. 悬挂点出现在标记单元加密后的子单元中

在下文中, 称一条边为非平凡的, 如果它的一个端点是某个单元的悬挂点; 称含悬挂点的单元为非平凡单元; 称新产生的待加密单元为额外待加密单元. 根据该规定, 待加密单元分为

$$\text{待加密单元} \begin{cases} \text{平凡待加密单元: 恰为平凡标记单元} \\ \text{非平凡待加密单元} \begin{cases} \text{非平凡标记单元} \\ \text{额外待加密单元} \end{cases} \end{cases} \quad (4.5)$$

4.2 确定待加密单元

根据 (4.5) 的分类, 我们先确定标记单元中的平凡和非平凡单元. 根据“单悬挂点规则”, 悬挂点一定是单元共线边的中点, 为此可通过计算如下误差找到悬挂点

$$\text{err}(i) = \left| z_i - \frac{1}{2}(z_{i-1} + z_{i+1}) \right|, \quad i = 1, \dots, N_v,$$

式中, z_i 是单元顶点, N_v 是顶点数. 非平凡标记单元按如下代码确定

```
1 ismElem = cell(NT,1); % store the elementwise mid-point locations
2 %% Determine the trivial and nontrivial marked elements
3 % nontrivial marked elements: marked elements with hanging nodes
4 nMarked = length(idElemMarked);
5 isT = false(nMarked,1);
6 for s = 1:nMarked
7     % current element
8     iel = idElemMarked(s);
9     % local logical index of elements with hanging nodes
10    p0 = node(elem{iel},:); p1 = circshift(p0,1,1); p2 = circshift(p0,-1,1);
11    err = vecnorm(p0-0.5*(p1+p2),2,2);
12    ism = (err<tol); ismElem{iel} = ism; % is midpoint
13    if sum(ism)<1, isT(s) = true; end % trivial
14 end
15 idElemMarkedT = idElemMarked(isT);
16 idElemMarkedNT = idElemMarked(~isT);
```

这里, `idElemMarkedT` 是平凡标记单元, `idElemMarkedNT` 是非平凡标记单元. 为了避免后面重复确定单元悬挂点, 用 `ismElem` 存储每个单元悬挂点的局部逻辑数组.

现在收集额外的待加密单元. 用 `idElemMarked` 记录标记单元, `idElemNew` 记录每一步新增加的单元. 向量 `idElemMarkedNew` 收集直到当前步的所有待加密单元 (包括给定的标记单元和所有的额外的待加密单元). 在循环找额外单元的过程中, `idElemMarkedNew` 的地位与初始标记单元的地位等同. 这是找额外单元的核心思想, 即把每一循环步归结到一开始的局面.

4.1 节的描述可总结为算法 2.

Algorithm 2 找到额外的待加密单元

- 初始化 `idElemMarkedNew` 和 `idElemNew` 为标记单元集合 `idElemMarked`.
- 找到 `idElemNew` 的相邻单元 (去除已找到的待加密单元), 编号存储为 `idElemNewNeighbor`, 并获得已找到的待加密单元集 `idElemMarkedNew` 中所有单元的边的序号.
- 对 `idElemNewNeighbor` 中单元循环, 并更新 `idElemNew`: 确定当前单元非平凡边的索引, 记为 `idEdgeDg`. 若 `idEdgeDg` 与 `idElemMarkedNew` 确定的边有交集 (实际上只要与新产生的单元 `idElemNew` 的边有交集即可), 则当前单元需要加密.
- 更新 `idElemMarkedNew`, 即添加 `idElemNew` 中单元.
- 若 `idElemNew` 是空集, 停止循环; 否则, 可视当前所有待加密单元 `idElemMarkedNew` 为初始的标记单元, 回到第一步.
- 所有额外待加密单元为
`idElemRefineAddL = setdiff(idElemMarkedNew,idElemMarked);`

上面的算法对应的 MATLAB 代码如下.

```
1 %% Find the additional elements to be refined
2 % initialized as marked elements
3 idElemMarkedNew = idElemMarked; % marked and all new elements
4 idElemNew = idElemMarked; % new elements generated in current step
5 isEdgeMarked = false(NE,1);
6 while ~isempty(idElemNew)
```

```

7      % adjacent polygons of new elements
8      for iel = idElemNew(:)'
9          if ~isempty(neighbor{iel}); continue; end
10         index = elem2edge{iel};
11         ia = edge2elem(index,1); ib = edge2elem(index,2);
12         ia(ia==iel) = ib(ia==iel);
13         neighbor{iel} = ia';
14     end
15     idElemNewNeighbor = unique(horzcata(neighbor{idElemNew}));
16     idElemNewNeighbor = setdiff(idElemNewNeighbor,idElemMarkedNew); % delete the ...
17     % ones in the new marked set
18     % edge set of new marked elements
19     isEdgeMarked(horzcata(elem2edge{idElemNew})) = true;
20     % find the adjacent elements to be refined
21     nElemNewAdj = length(idElemNewNeighbor);
22     isRefine = false(nElemNewAdj,1);
23     for s = 1:nElemNewAdj
24         % current element
25         iel = idElemNewNeighbor(s);
26         index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
27         % local logical index of elements with hanging nodes
28         v1 = [Nv,1:Nv-1]; v0 = 1:Nv; % left,current
29         p0 = node(elem{iel},:); p1 = circshift(p0,1,1); p2 = circshift(p0,-1,1);
30         err = vecnorm(p0-0.5*(p1+p2),2,2);
31         ism = (err<tol); ismElem{iel} = ism; % is midpoint
32         if sum(ism)<1, continue; end % start the next loop if no hanging nodes exist
33         % index numbers of edges connecting hanging nodes in the adjacent elements ...
34         % to be refined
35         idEdgeDg = indexEdge([v1(ism),v0(ism)]);
36         % whether or not the above edges are in the edge set of new marked elements
37         if sum(isEdgeMarked(idEdgeDg)), isRefine(s) = true; end
38     end
39     idElemNew = idElemNewNeighbor(isRefine);
40     idElemMarkedNew = unique([idElemMarkedNew(:); idElemNew(:)]);
41     idElemRefineAddL = setdiff(idElemMarkedNew,idElemMarked);

```

4.3 单元的加密与扩展

单元加密与扩展的预说明

先说明本文的单元扩展思路, 因为它影响单元的加密. 需要扩展的单元由三部分组成:

- 一是所有待加密单元的相邻单元;
- 二是额外待加密单元加密后的子单元;
- 三是某些标记单元加密后的子单元. 如图 (14) 所示, 单元 9, 19, 20 都是标记单元. 当单元 9 进行划分后, 因 19 和 20 要划分, 从而单元 9 的子单元会出现悬挂点.

简单分析一下扩展单元的特点.

1. 若某个单元需要扩展, 即该单元某条边需要二分, 则该条边必是某个待加密单元的平凡边.
2. 但要指出的是, 这种平凡边可能是另一个待加密单元的非平凡边, 如图 14 中单元 19 的左侧边是单元 9 的非平凡边.

3. 需要二分的边仍是原始剖分的边, 即悬挂点仅添加在原始剖分的边上.

为此, 我们先找到所有待加密单元中的平凡边集合, 记为 idEdgeCut . 注意, 这些平凡边只要是某个待加密单元的即可. 需扩展单元的特点是: 至少有一条边对应 idEdgeCut 中的索引.

本文的扩展方法是: 查看单元的边序号 elem2edge , 确定其中是否有位于 idEdgeCut 中的元素. 正因为如此, 非平凡待加密单元要先加密, 以获得子单元的数据结构 elem2edge .

由于标记单元中有一些单元不需要扩展 (它们直接加密), 但它们也有 idEdgeCut 中的边, 如单元 19 和 20 的左侧边. 为此, 单元加密时, 这部分单元要最后考虑.

非平凡待加密单元的加密

悬挂点可能出现在非平凡待加密单元加密后的子单元中, 且本文的单元扩展方法需要数据结构 elem2edge . 为此, 先对非平凡待加密单元进行加密.

某些边的中点和单元重心需要加入到节点矩阵 node 中. 我们将顶点、边和单元用单指标 $i = 1, 2, \dots, N + NE + NT$ 按如下顺序重新编号

$$z_1, \dots, z_N; e_1, \dots, e_{NE}; K_1, \dots, K_{NT},$$

称其为连接序号. 然而, 在大多数情形下, 使用 edge 的索引更加方便.

为了构造四点子单元, 首先考虑一个例子, 其顶点和边的连接序号为

$$z_1, e_1, z_2, e_2, z_3, e_3, z_4, e_4, z_5, e_5, \quad N_v = 5,$$

其中 z_2 和 z_5 是悬挂点, 下标表示局部索引. 接下来, 将非平凡边的连接序号用悬挂点序号如下代替

$$\begin{aligned} & z_1, e_1, z_2, e_2, z_3, e_3, z_4, e_4, z_5, e_5 \\ \hookrightarrow & z_1, z_2, z_2, z_2, z_3, e_3, z_4, z_5, z_5, z_5, \quad N_v = 5. \end{aligned}$$

这样可以用统一的方式构造数据结构 elem . 该方法也适用于标记单元, 无论它是否有悬挂点.

对数据结构 elem2edge , 子单元在内部的非平凡边仅用 0 进行标记, 而边界边用边的连接序号标记.

程序如下

```

1 %% Partition the nontrivial elements to be refined
2 % these elements are composed of:
3 % - nontrivial marked elements
4 % - additional elements to be refined
5 idElemRefineNT = [idElemMarkedNT; idElemRefineAddL];
6 nRefineNT = length(idElemRefineNT);
7 elemRefineNT = cell(nRefineNT,1);
8 elem2edgeRefineNT = cell(nRefineNT,1);
9 for s = 1:nRefineNT
10     % current element
11     iel = idElemRefineNT(s);
12     index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
13     % find midpoint
14     v1 = [Nv,1:Nv-1]; v0 = 1:Nv;
15     ism = ismElem{iel};

```

```

16 % modify the edge number
17 ide = indexEdge+N; % the connection number
18 ide(v1(ism)) = index(ism); ide(ism) = index(ism);
19 % elem (with or without hanging nodes)
20 nsub = Nv-sum(ism);
21 z1 = ide(v1(¬ism)); z0 = index(¬ism);
22 z2 = ide(¬ism); zc = iel*ones(nsub,1)+N+NE;
23 elemRefineNT{s} = [z1(:), z0(:), z2(:), zc(:)];
24 % elem2edge
25 ise = false(Nv,1); ise([v1(ism),v0(ism)]) = true;
26 idg = zeros(Nv,1); idg(ise) = indexEdge(ise);
27 e1 = idg(v1(¬ism)); e0 = idg(¬ism);
28 elem2edgeRefineNT{s} = [e1(:), e0(:), zeros(nsub,2)]; % e2 = ec = 0
29 end
30 addElemRefineNT = num2cell(vertpcat(elemRefineNT{:}), 2);
31 addElemRefineNT2edge = num2cell(vertpcat(elem2edgeRefineNT{:}), 2);

```

注意, num2cell.m 对空元胞也可操作. Line 23 的 1, 0, 2, c 标记分别对应当前顶点的上一个顶点、当前顶点、当前顶点的下一个顶点和重心.

单元扩展: 增加悬挂点

需要扩展的单元由两部分组成: 一些是待加密单元的相邻单元, 另一些是非平凡单元加密后的子单元.

```

1 %% Determine the elements to be expanded
2 % these elements are composed of
3 % - adjacent polygonals of elements to be refined
4 % - subcells of nontrivial elements
5 % adjacent polygons of elements to be refined
6 idElemRefine = unique([idElemRefineAddL(:); idElemMarked(:)]); % ascending order ...
7 for update
8   for iel = idElemRefine(:)
9     if isempty(neighbor{iel}); continue; end
10    index = elem2edge{iel};
11    ia = edge2elem(index,1); ib = edge2elem(index,2);
12    ia(ia==iel) = ib(ia==iel);
13    neighbor{iel} = ia';
14  end
15 idElemRefineNeighbor = unique(horzcat(neighbor{idElemRefine}));
16 idElemRefineNeighbor = setdiff(idElemRefineNeighbor,idElemRefine);
17 % basic data structure of elements to be extended
18 elemExtend = [elem(idElemRefineNeighbor); addElemRefineNT];
19 elem2edgeExtend = [elem2edge(idElemRefineNeighbor); addElemRefineNT2edge];

```

注意, 悬挂点仅出现在原始剖分的边上, 且这些边是某个待加密单元的平凡边. 但要指出的是, 这种平凡边可能是另一个待加密单元的非平凡边, 如图 14 中单元 19 的左侧边是单元 9 的非平凡边. 为此, 我们先找到所有待加密单元中的平凡边, 这些平凡边只要是某个待加密单元的即可. 为了方便, 我们用逻辑数组 isEdgeCut 对所有边的平凡和非平凡性质进行标记, true 的位置表示该条边要进行二分, 即为平凡边.

```

1 %% Extend elements by adding hanging nodes
2 % natural numbers of trivial edges w.r.t some element to be refined
3 isEdgeCut = false(NE,1);

```

```

4 for s = 1:length(idElemRefine)
5     iel = idElemRefine(s);
6     index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
7     v1 = [Nv,1:Nv-1];
8     ism = ismElem{iel};
9     idx = true(Nv,1); idx(v1(ism)) = false; idx(ism) = false;
10    isEdgeCut(indexEdge(idx)) = true;
11 end

```

需扩展单元的特点是: 至少有一条边对应 `isEdgeCut` 中逻辑真的索引. 为了扩展单元, 先初始化一个长度为 $2N_v$ 的零向量, 在奇数位插入当前单元的顶点序号, 但仅在偶数位插入 `isEdgeCut` 中逻辑真的边索引. 这样, 删除零元素后即得连通性向量.

```

1 % extend the elements
2 for s = 1:length(elemExtend)
3     index = elemExtend{s}; indexEdge = elem2edgeExtend{s};
4     id = find(indexEdge>0);
5     id = id(isEdgeCut(indexEdge(id)));
6     idvec = zeros(1,2*length(index));
7     idvec(1:2:end) = index; idvec(2*id) = indexEdge(id)+N;
8     elemExtend{s} = idvec(idvec>0);
9 end

```

划分平凡标记单元

剩下的未加密单元是平凡标记单元, 它们类似之前的方法进行划分.

```

1 %% Partition the trivial marked elements
2 nMarkedT = length(idElemMarkedT);
3 addElemMarkedT = cell(nMarkedT,1);
4 for s = 1:nMarkedT
5     iel = idElemMarkedT(s);
6     index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
7     ide = indexEdge+N; % connection number
8     z1 = ide([Nv,1:Nv-1]); z0 = index;
9     z2 = ide; zc = iel*ones(Nv,1)+N+NE;
10    addElemMarkedT{s} = [z1(:), z0(:), z2(:), zc(:)];
11 end
12 % addElem
13 addElemMarkedT = num2cell(vvertcat(addElemMarkedT{:}), 2);
14 addElem = [elemExtend; addElemMarkedT];

```

更新基本数据结构

如下更新基本数据结构 `node` 和 `elem`:

```

1 %% Update node and elem
2 % node
3 nodeEdgeCut = (node(edge(isEdgeCut,1),:) + node(edge(isEdgeCut,2),:))/2;
4 nodeCenter = zeros(length(idElemRefine),2);
5 for s = 1:length(idElemRefine)
6     iel = idElemRefine(s); index = elem{iel};
7     verts = node(index(~ismElem{iel}),:); verts1 = verts([2:end,1],:);
8     area_components = verts(:,1).*verts1(:,2)-verts1(:,1).*verts(:,2);

```

```

9     ar = 0.5*abs(sum(area_components));
10    nodeCenter(s,:) = sum((verts+verts1).*repmat(area_components,1,2))/(6*ar);
11 end
12 node = [node; nodeEdgeCut; nodeCenter];
13 % elem
14 elem([idElemRefine(:); idElemRefineNeighbor(:)]) = []; % delete old
15 elem = [elem; addElem];
16
17 %% Reorder the vertices
18 [~,~,totalid] = unique(horzcat(elem{:})');
19 elemLen = cellfun('length',elem);
20 elem = mat2cell(totalid', 1, elemLen)';

```

4.4 程序整理

完整的加密程序总结如下.

CODE 7. PolyMeshRefine.m (多三角形网格加密)

```

1 function [node,elem] = PolyMeshRefine(node,elem,elemMarked)
2 %PolyMeshRefine refines a 2-D polygonal mesh satisfying one-hanging node rule
3 %
4 % We divide elements by connecting the midpoint of each edge to its
5 % barycenter.
6 % We remove small edges by further partitioning some adjacent elements.
7 %
8 % Copyright (C) Terence Yu.
9
10 idElemMarked = unique(elemMarked(:)); % in ascending order
11 tol = 1e-10; % accuracy for finding midpoint
12
13 %% Get auxiliary data
14 NT = size(elem,1);
15 if ~iscell(elem), elem = num2cell(elem,2); end
16 % diameter
17 diameter = cellfun(@(index) max(pdist(node(index,:))), elem(idElemMarked));
18 if max(diameter)<tol
19     disp('The mesh is too dense'); return;
20 end
21 % totalEdge
22 shiftfun = @(verts) [verts(2:end),verts(1)];
23 T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
24 v0 = horzcat(elem{:})'; v1 = horzcat(T1{:})';
25 totalEdge = sort([v0,v1],2);
26 % edge, elem2edge
27 [edge, i1, totalJ] = unique(totalEdge,'rows');
28 elemLen = cellfun('length',elem);
29 elem2edge = mat2cell(totalJ',1,elemLen)';
30 % edge2elem
31 Num = num2cell((1:NT)'); Len = num2cell(elemLen);
32 totalJelem = cellfun(@(n1,n2) n1*ones(n2,1), Num, Len, 'UniformOutput', false);
33 totalJelem = vertcat(totalJelem{:});
34 i2(totalJ) = 1:length(totalJ); i2 = i2(:);
35 edge2elem = totalJelem([i1,i2]);
36 % neighbor
37 neighbor = cell(NT,1);
38 % number

```



```

39 N = size(node,1); NE = size(edge,1);
40
41 ismElem = cell(N,1); % store the elementwise mid-point locations
42 %% Determine the trivial and nontrivial marked elements
43 % nontrivial marked elements: marked elements with hanging nodes
44 nMarked = length(idElemMarked);
45 isT = false(nMarked,1);
46 for s = 1:nMarked
47     % current element
48     iel = idElemMarked(s);
49     % local logical index of elements with hanging nodes
50     p0 = node(elem{iel},:); p1 = circshift(p0,1,1); p2 = circshift(p0,-1,1);
51     err = vecnorm(p0-0.5*(p1+p2),2,2);
52     ism = (err<tol); ismElem{iel} = ism; % is midpoint
53     if sum(ism)<1, isT(s) = true; end % trivial
54 end
55 idElemMarkedT = idElemMarked(isT);
56 idElemMarkedNT = idElemMarked(~isT);
57
58 %% Find the additional elements to be refined
59 % initialized as marked elements
60 idElemMarkedNew = idElemMarked; % marked and all new elements
61 idElemNew = idElemMarked; % new elements generated in current step
62 isEdgeMarked = false(NE,1);
63 while ~isempty(idElemNew)
64     % adjacent polygons of new elements
65     for iel = idElemNew(:)'
66         if ~isempty(neighbor{iel}); continue; end
67         index = elem2edge{iel};
68         ia = edge2elem(index,1); ib = edge2elem(index,2);
69         ia(ia==iel) = ib(ia==iel);
70         neighbor{iel} = ia';
71     end
72     idElemNewNeighbor = unique(horzcat(neighbor{idElemNew}));
73     idElemNewNeighbor = setdiff(idElemNewNeighbor,idElemMarkedNew); % delete the ...
74     % edge set of new marked elements
75     isEdgeMarked(horzcat(elem2edge{idElemNew})) = true;
76     % find the adjacent elements to be refined
77     nElemNewAdj = length(idElemNewNeighbor);
78     isRefine = false(nElemNewAdj,1);
79     for s = 1:nElemNewAdj
80         % current element
81         iel = idElemNewNeighbor(s);
82         index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
83         % local logical index of elements with hanging nodes
84         v1 = [Nv,1:Nv-1]; v0 = 1:Nv; % left,current
85         p0 = node(elem{iel},:); p1 = circshift(p0,1,1); p2 = circshift(p0,-1,1);
86         err = vecnorm(p0-0.5*(p1+p2),2,2);
87         ism = (err<tol); ismElem{iel} = ism; % is midpoint
88         if sum(ism)<1, continue; end % start the next loop if no hanging nodes exist
89         % index numbers of edges connecting hanging nodes in the adjacent elements ...
90         % to be refined
91         idEdgeDg = indexEdge([v1(ism),v0(ism)]);
92         % whether or not the above edges are in the edge set of new marked elements
93         if sum(isEdgeMarked(idEdgeDg)), isRefine(s) = true; end
94     end
95     idElemNew = idElemNewNeighbor(isRefine);

```

```

95     idElemMarkedNew = unique([idElemMarkedNew(:); idElemNew(:)]);
96 end
97 idElemRefineAddL = setdiff(idElemMarkedNew,idElemMarked);
98
99 %% Partition the nontrivial elements to be refined
100 % these elements are composed of:
101 % - nontrivial marked elements
102 % - additional elements to be refined
103 idElemRefineNT = [idElemMarkedNT; idElemRefineAddL];
104 nRefineNT = length(idElemRefineNT);
105 elemRefineNT = cell(nRefineNT,1);
106 elem2edgeRefineNT = cell(nRefineNT,1);
107 for s = 1:nRefineNT
108     % current element
109     iel = idElemRefineNT(s);
110     index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
111     % find midpoint
112     v1 = [Nv,1:Nv-1]; v0 = 1:Nv;
113     ism = ismElem{iel};
114     % modify the edge number
115     ide = indexEdge+N; % the connection number
116     ide(v1(ism)) = index(ism); ide(ism) = index(ism);
117     % elem (with or without hanging nodes)
118     nsub = Nv-sum(ism);
119     z1 = ide(v1(~ism)); z0 = index(~ism);
120     z2 = ide(~ism); zc = iel*ones(nsub,1)+N+NE;
121     elemRefineNT{s} = [z1(:), z0(:), z2(:), zc(:)];
122     % elem2edge
123     ise = false(Nv,1); ise([v1(ism),v0(ism)]) = true;
124     idg = zeros(Nv,1); idg(ise) = indexEdge(ise);
125     e1 = idg(v1(~ism)); e0 = idg(~ism);
126     elem2edgeRefineNT{s} = [e1(:), e0(:), zeros(nsub,2)]; % e2 = ec = 0
127 end
128 addElemRefineNT = num2cell(vertcats(elemRefineNT{:}), 2);
129 addElemRefineNT2edge = num2cell(vertcats(elem2edgeRefineNT{:}), 2);
130
131 %% Determine the elements to be expanded
132 % these elements are composed of
133 % - adjacent polygonals of elements to be refined
134 % - subcells of nontrivial elements
135 % adjacent polygons of elements to be refined
136 idElemRefine = unique([idElemRefineAddL(:); idElemMarked(:)]); % ascending order ...
    for update
137 for iel = idElemRefine(:) '
138     if ~isempty(neighbor{iel}); continue; end
139     index = elem2edge{iel};
140     ia = edge2elem(index,1); ib = edge2elem(index,2);
141     ia(ia==iel) = ib(ib==iel);
142     neighbor{iel} = ia';
143 end
144 idElemRefineNeighbor = unique(horzcats(neighbor{idElemRefine}));
145 idElemRefineNeighbor = setdiff(idElemRefineNeighbor,idElemRefine);
146 % basic data structure of elements to be extended
147 elemExtend = [elem(idElemRefineNeighbor); addElemRefineNT];
148 elem2edgeExtend = [elem2edge(idElemRefineNeighbor); addElemRefineNT2edge];
149
150 %% Extend elements by adding hanging nodes
151 % natural numbers of trivial edges w.r.t some element to be refined

```

```

152 isEdgeCut = false(NE,1);
153 for s = 1:length(idElemRefine)
154     iel = idElemRefine(s);
155     index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
156     v1 = [Nv,1:Nv-1];
157     ism = ismElem{iel};
158     idx = true(Nv,1); idx(v1(ism)) = false; idx(ism) = false;
159     isEdgeCut(indexEdge(idx)) = true;
160 end
161 % extend the elements
162 for s = 1:length(elemExtend)
163     index = elemExtend{s}; indexEdge = elem2edgeExtend{s};
164     id = find(indexEdge>0);
165     id = id(isEdgeCut(indexEdge(id)));
166     idvec = zeros(1,2*length(index));
167     idvec(1:2:end) = index; idvec(2*id) = indexEdge(id)+N;
168     elemExtend{s} = idvec(idvec>0);
169 end
170
171 %% Partition the trivial marked elements
172 nMarkedT = length(idElemMarkedT);
173 addElemMarkedT = cell(nMarkedT,1);
174 for s = 1:nMarkedT
175     iel = idElemMarkedT(s);
176     index = elem{iel}; indexEdge = elem2edge{iel}; Nv = length(index);
177     ide = indexEdge+N; % connection number
178     z1 = ide([Nv,1:Nv-1]); z0 = index;
179     z2 = ide; zc = iel*ones(Nv,1)+N+NE;
180     addElemMarkedT{s} = [z1(:), z0(:), z2(:), zc(:)];
181 end
182 % addElem
183 addElemMarkedT = num2cell(vertpcat(addElemMarkedT{:}), 2);
184 addElem = [elemExtend; addElemMarkedT];
185
186 %% Update node and elem
187 % node
188 nodeEdgeCut = (node(edge(isEdgeCut,1),:) + node(edge(isEdgeCut,2),:))/2;
189 nodeCenter = zeros(length(idElemRefine),2);
190 for s = 1:length(idElemRefine)
191     iel = idElemRefine(s); index = elem{iel};
192     verts = node(index(~ismElem{iel}),:); verts1 = verts([2:end,1],:);
193     area_components = verts(:,1).*verts1(:,2)-verts1(:,1).*verts(:,2);
194     ar = 0.5*abs(sum(area_components));
195     nodeCenter(s,:) = sum((verts+verts1).*repmat(area_components,1,2))/(6*ar);
196 end
197 node = [node; nodeEdgeCut; nodeCenter];
198 % elem
199 elem([idElemRefine(:); idElemRefineNeighbor(:)]) = []; % delete old
200 elem = [elem; addElem];
201
202 %% Reorder the vertices
203 [~,~,totalid] = unique(horzcat(elem{:})');
204 elemLen = cellfun('length',elem);
205 elem = mat2cell(totalid', 1, elemLen)';

```
