A MATLAB MESH GENERATOR FOR THE TWO-DIMENSIONAL FINITE ELEMENT METHOD

JONAS KOKO*†

Abstract

A new Matlab code for the generation of unstructured (3-node or 6-node) triangular meshes in two dimensions is described. The method is based on the Matlab mesh generator `distmesh` of Persson and Strang [SIAM Rev. 46:329-345, 2004]. As input, the code takes a signed distance function for the domain geometry. A mesh size function, for the spatial node distribution, is constructed using an approximate medial axis. As outputs, the code generates a 3-node or a 6-node triangular mesh with boundary data (edges and nodes). The approach presented consists of three steps: (1) an initial nodes placement is obtained using a probabilistic node distribution, (2) an iterative smoothing is performed assuming the presence of an attractive/repulsive internode force, and (3) a fast refinement procedure is performed for 6-node triangular meshes or large scale meshes.

Keywords: Mesh generation, unstructured mesh, finite element, Matlab.

AMS subject classification: 65N50, 65M50, 65N30, 65M60

1 Introduction

Today, the finite element method (FEM) is one of the most widely used discretization method for solving Partial Differential Equations (PDEs). Finite element discretization of PDEs consists of replacing the continuous formulation by a discrete representation for the unknowns and the spatial domain. The discretization (i.e. the generation of a tessellation) of a spatial domain is a crucial step in scientific computing. The most general and versatile mesh is an unstructured triangular mesh since it can handle complex geometry and might provide better performance when the PDE solution varies rapidily (e.g. boundary layers). In PDE applications, the sufficient condition for optimal approximation results is that the minimum angle or the aspect ratio of each triangle be bounded independently of the mesh. But the generation of suitable meshes is still a major problem, see e.g. [1, 2, 3, 4, 5].

Modern mesh generators are complex codes and some of them have a specific language for the input description of the physical domain or a graphical interface to make easier the geometry definition of the domain. The mesh generator codes are therefore difficult to integrate with other codes for those who just want to mesh a domain without learning the entire meshing software. For Matlab users, there was only the mesh generator of

*Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 Clermont-Ferrand, France
†CNRS, UMR 6158, LIMOS, F-63173 Aubière, France

the (commercial) PDE Toolbox [6] until Persson and Strang proposed `distmesh` [7], a Matlab mesh generator based on signed distance functions. The signed distance functions considerably simplify the geometry description of the domain. The code is relatively short (few dozen lines) and the user can define desired mesh densities using size functions. But `distmesh` is a general purpose mesh generator (for visualization, finite element/finite volume, geometry modeling, computer graphics). For finite element applications, a post-treatment is necessary to fix some problems:

- duplicated nodes (two or mode coincident nodes should be replaced by a single node);
- triangles orientation (the triangle orientation should be the same for all mesh elements);
- boundary nodes (boundary nodes should be on the boundary);
- termination and convergence criteria (some termination criteria should be fixed).

Our goal is to propose, for finite element applications, a simple unstructured mesh generator based on `distmesh` and a fast refinement procedure. The mesh generation code consists of two Matlab functions `kmg2d` and `kmg2dref`. `kmg2d` is the mesh generator, designed to generate medium-scale meshes. `kmg2d` uses the (probabilistic) initial spatial nodes distribution of [7]. As a smoothing function, we use a modification of the repulsive/attractive force proposed by Bossen and Heckbert [8] instead of the (linear) repulsive force of `distmesh`. We propose a geometry-based mesh size function based on an approximate medial axis. Our size function is computed directly from the signed distance function. `kmg2dref` is a fast mesh refinement function. Starting from an initial coarse mesh, `kmg2dref` can produce a large-scale mesh (several hundreds of thousands of nodes) by successive refinements. In addition, the refinement procedure allows our meshing code to generate 6-node triangular meshes.

The plan of the paper is as follows. We begin by the Matlab representation of a mesh in Section 2. We recall Persson-Strang mesh generator `distmesh` in Section 3. In Section 4 we present our mesh generation code. Boundary data and triangle neighboring functions are presented in Section 5, followed by the refinement strategy in Section 6. A geometry-based mesh size function is described in Section 7. Numerical results, with various geometries, are presented in Section 8. Readers can download and edit the codes from `http://www.isima.fr/~jkoko/Codes/KMG.tar.gz`.

2 Matlab mesh representation

A triangular mesh is defined by two arrays `p(1:np,1:2)` and `t(1:nt,1:3)`, where `np` is the number of nodes and `nt` the number of triangles. The array `t` contains for each element, the node number of its vertices ordered counter-clockwise. For a 6-bode triangular mesh, the triangles array is `t(1:nt,1:6)`, with the six nodes making up each triangle listed in a particular order, in which corner nodes come first, followed by the mid-side nodes, as shown in Figure 1.

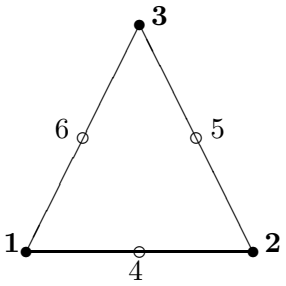


Figure 1: Local nodes numbering in a 6-node triangle

Boundary edges, for Neumann boundary conditions, are provided by an array `be(1:ne,1:2)`, for a 3-node triangular mesh, or `be(1:ne,1:3)`, for a 6-node triangular mesh. The `be(i,1:2)` contains the two nodes which bound the edge `i` and, for a 6-node case, `be(i,3)` is the midpoint.

2.1 Mesh quality

In finite element applications, the error upper bounds depend only on the smallest angle in the mesh. To quantify the mesh quality, a commonly used quality measure is the ratio between the radius of the largest inscribed circle and the smallest circumscribed circle, i.e.

$$q = 2\frac{r_{in}}{r_{out}} = \frac{(b+c-a)(c+a-b)(a+b-c)}{abc}, \tag{2.1}$$

where a, b, c are the side lengths. As a rule of thumb, a mesh is a good if all triangles have $q_{\min} > 0.5$.

3 Mesh generation with signed distance functions

The mesh generator proposed by Persson and Strang [7] is based on the physical analogy between a simplex mesh and a truss structure, and a signed distance function which quickly determines if a point is inside or outside a bounded domain $\Omega \in \mathbb{R}^2$. They implicitly assume that Ω can be defined as

$$\Omega = \left\{ \boldsymbol{x} = (x, y) \in \mathbb{R}^2 \mid d(\boldsymbol{x}) < 0 \right\}.$$

An element size function $h : \Omega \to \mathbb{R}^*_+$ is used to control the mesh density. The value $h(\boldsymbol{x})$ does not equal the actual size but gives the relative spatial node distribution over the domain. For a uniform mesh $h(\boldsymbol{x}) = 1$, for all $\boldsymbol{x} \in \Omega$.

3.1 The Persson-Strang Algorithm

The mesh generation algorithm is based on an analogy between a triangular mesh and a 2-D truss structure. Let N be the number of mesh nodes and $\boldsymbol{p}$ the N-by-2 array of node coordinates. By analogy, the edges of the triangles correspond to bars and the vertices

correspond to joints of the truss. Each bar $\boldsymbol{e}_{ij} = [\boldsymbol{p}_i, \boldsymbol{p}_j]$ has a variable spring constant $f(\boldsymbol{p}_i, \boldsymbol{p}_j)$ depending of its current length $\| \boldsymbol{e}_{ij} \|$ and a desired length r_{ij}. The equilibrium problem corresponds to the nonlinear system of equations

$$\boldsymbol{F}(\boldsymbol{p}) = 0. \tag{3.1}$$

$\boldsymbol{F}(\boldsymbol{p})$ is a N-by-2 array in which the first column contains x-components of the forces, and the second column contains the y-components of the forces.

The force vector $\boldsymbol{F}(\boldsymbol{p})$ is not a continuous function of $\boldsymbol{p}$, since the topology can be changed by Delaunay triangulation as the points move. Eq. (3.1) can be considered as a stationary solution of the following (non-physical) system of ODEs

$$\boldsymbol{p}(0) = \boldsymbol{p}_0, \tag{3.2}$$

$$\frac{d\boldsymbol{p}}{dt} = \boldsymbol{F}(\boldsymbol{p}(t)), \quad t \geq 0. \tag{3.3}$$

The time discretization ($t_k = k\Delta t$, $\boldsymbol{p}^k = \boldsymbol{p}(t_k)$) of (3.2)-(3.3) yields the iterative scheme $\boldsymbol{p}^{k+1} = \boldsymbol{p}^k + \Delta t \boldsymbol{F}(\boldsymbol{p}^k)$, that is

$$\boldsymbol{p}_i^{k+1} = \boldsymbol{p}_i^k + \Delta t \sum_{j \in \mathcal{N}_i} f(\boldsymbol{p}_i^k, \boldsymbol{p}_j^k) \frac{\boldsymbol{e}_{ij}^k}{||\boldsymbol{e}_{ij}^k||}, \quad i = 1, \ldots, N, \tag{3.4}$$

where $\mathcal{N}_i$ is the neighborhood of node i. The bar force $f(\boldsymbol{p}_i^k, \boldsymbol{p}_j^k)$ has the repulsive property only, i.e.

$$f(\boldsymbol{p}_i^k, \boldsymbol{p}_j^k) = \begin{cases} r_{ij}^k - \| e_{ij}^k \| & \text{if } \| \boldsymbol{e}_{ij}^k \| < r_{ij}^k, \\ 0 & \text{if } \| \boldsymbol{e}_{ij}^k \| \geq r_{ij}^k. \end{cases} \tag{3.5}$$

The desired bar length r_{ij}^k is computed using the size function h as follows

$$r_{ij}^k = \omega s^k h_{ij}^k \tag{3.6}$$

where $h_{ij}^k = h((\boldsymbol{p}_i^k + \boldsymbol{p}_j^k)/2)$ is the size function value at the midpoint of the edge $[\boldsymbol{p}_i^k, \boldsymbol{p}_j^k]$, and the scaling factor s^k is given by

$$s^k = \left(\frac{\sum_{i,j} \| \boldsymbol{e}_{ij}^k \|^2}{\sum_{i,j} (h_{ij}^k)^2} \right)^{1/2}.$$

The fixed factor ω is chosen such that most bars give repulsive forces ($f > 0$) in $\boldsymbol{F}$ ($\omega = 1.2$ in Persson and Strang code). It is important that most of the bars give repulsive force to help the spreading of nodes across Ω.

We can adopt the normalized length concept, introduced by Bossens and Heckbert [8], and defined on an edge by

$$\ell_{ij}^k = \frac{\| \boldsymbol{e}_{ij}^k \|}{r_{ij}^k}, \tag{3.7}$$

where r_{ij}^k is the desired edge length (3.6). The goal is therefore to create a mesh with normalized edge length of 1. With the normalized distance (3.7), the (Persson-Strang) bar force function (3.5) becomes

$$f(\boldsymbol{p}_i^k, \boldsymbol{p}_j^k) = \begin{cases} 1 - \ell_{ij}^k & \text{if } \ell_{ij}^k < 1, \\ 0 & \text{if } \ell_{ij}^k \geq 1. \end{cases} \tag{3.8}$$

Function 3.8 describes a repulsive only behavior, i.e. two vertices repulse each other if they are too close ($\ell_{ij}^k < 1$), Figure 2. Note that there exists several smoothing functions:

- the Laplacian function (each node is moved to the centroid of its neighbors) [9], the most commonly used;
- the Lennard-Jones function [10] from Chemistry, $f(d) = d^{-13} - d^{-7}$;
- the Bossens-Heckbert smoothing function [8], Figure 2,

$$f(d) = (1 - d^4)\exp(-d^4). \tag{3.9}$$

The Laplacian smoothing function describes an attraction only behavior while the Lennard-Jones and Bossen-Heckbert functions describe attraction/repulsion behavior. However, the Lennard-Jones function suffers from numerical instabilities.

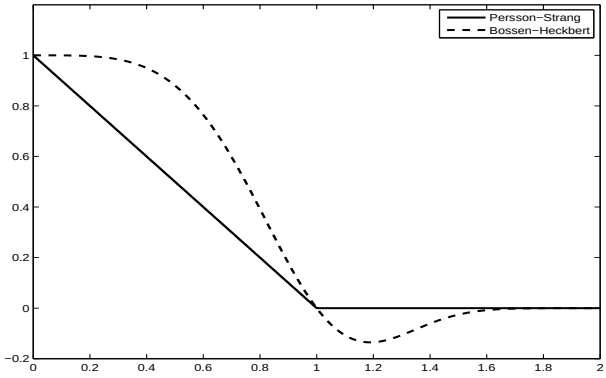


Figure 2: Smoothing functions

In contrast with some mesh adaptation procedures, see e.g. [11, 12], the update formula (3.4) is performed on all nodes simultaneously. If a node $\boldsymbol{p}_i^k$ ends up outside Ω, after the update, it is moved back on the boundary using the formula

$$\boldsymbol{p}_i^k = \boldsymbol{p}_i^k - d(\boldsymbol{p}_i^k)\nabla d(\boldsymbol{p}_i^k), \tag{3.10}$$

where the gradient $\nabla d(\boldsymbol{p}_i^k)$ is computed numerically by a finite difference scheme. Formula (3.10) is the projection of $\boldsymbol{p}_i^k$ onto the boundary of Ω, and corresponds to a reaction force normal to the boundary.

For some geometries (i.e. disc, ellipsoid) it is easy to create a signed distance function. For others, even simple like a convex polygon, the definition of a distance function can be a serious problem. Persson-Strang mesh generator allows using of simplified distance functions by introducing *fixed points*, i.e., points that must appear in the mesh and, then, are not updated according to (3.4). For example, for a rectangle, they define a distance function as the minimum distance to the four boundary lines. This is not the correct distance to the four external regions whose nearest points are corners of the rectangle. Fortunately, by fixing corner points, no mesh point ends up in these four regions.

In addition to being much shorter and simpler than other mesh generators, the Persson-Strang mesh generator produces high quality meshes.

3.2 Some flaws

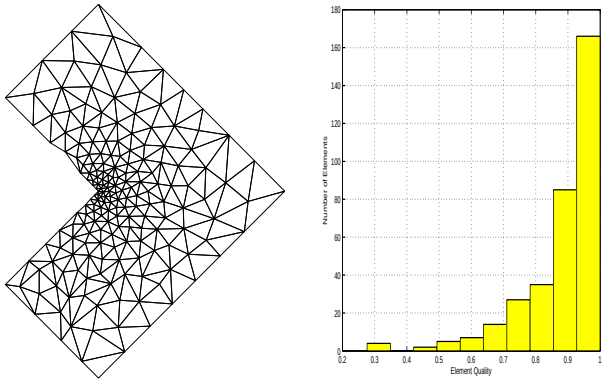


Figure 3: Mesh of an L-shaped domain and histogram of element qualities with $q_{\min} = 0.2764$. The mesh size function is $h(x, y) = 1 + 5\sqrt{x^2 + y^2}$ and $h_0 = 0.05$

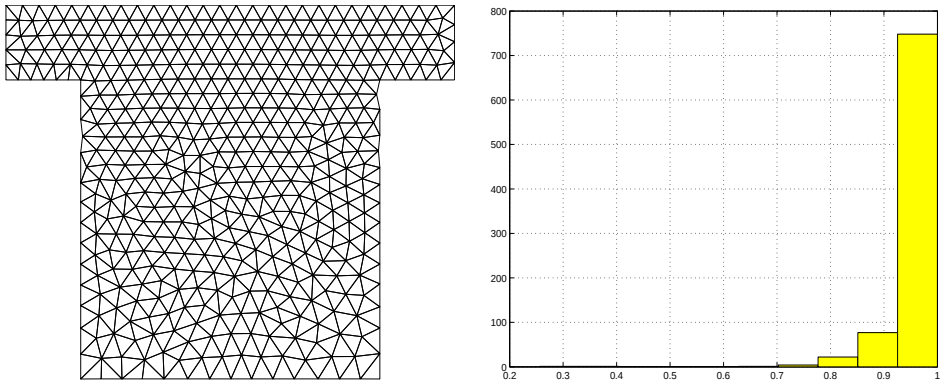


Figure 4: Mesh of a polygon and histogram of element qualities with $q_{\min} = 0.2561$. The mesh size function is $h(x, y) = 1 + \frac{1}{2}|\min(0, y)|^2$ and $h_0 = 0.055$

Run	1	2	3	4	5	6	7	8	9	10
N	366	366	366	366	366	366	366	366	366	366
CPU (in Sec.)	0.22	0.23	0.23	0.22	0.23	0.23	0.23	0.23	0.23	0.22
$q_{\min}$	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72

Table 1: Robustness of the Persson-Strang mesh generator for the unit disc with $h(x, y) = 1$ and $h_0 = 0.1$.

Run	1	2	3	4	5	6	7	8	9	10
N	589	611	563	597	602	565	609	617	581	604
CPU (in Sec.)	0.47	0.72	0.65	0.55	0.69	0.65	0.54	0.42	0.59	0.65
$q_{\min}$	0.38	0.45	0.57	0.59	0.51	0.68	0.34	0.47	0.32	0.67

Table 2: Robustness of the Persson-Strang mesh generator for the unit disc with $h(x, y) = 1 + \sqrt{x^2 + y^2}$ and $h_0 = 0.05$.

`distmesh` is a quick and flexible tool for the generation of unstructured triangular mesh (in 2D and 3D) but there are some flaws for finite element applications.

1. The non-robustness of the method for nonuniform mesh size functions. The element qualities are usually high, but occasionally drop significantly, see Figure 3-4. Running the code several times with the same parameters produces variable quality meshes, see Table 2. In contrast, for the unit disc, the method is robust and produces high quality mesh for the uniform mesh size function, see Table 1

2. It is not guaranteed that the boundary nodes are located at the boundary of the domain, see Figure 3-4.

3. The triangles created by the Matlab Delaunay triangulation function (`delaunay` and `delaunayn`) are not ordered (locally) in the standard counter clockwise form used in finite element codes. A post-treatment is necessary to avoid negative triangle area in finite element assembling functions.

4. The choice of a size function is a non trivial problem, for complex geometries.

5. The treatment of fixed-points that are also points of the bounding box is not satisfactory. If such a point survived the probabilistic rejection method, it will be duplicated since the fixed points are added after the rejection method. This can result in a catastrophe for finite element applications.

6. The mesh generator does not provide the boundary information (nodes and edges) which are crucial for the finite element applications.

7. The mesh generator is designed for small or medium scale applications. For large scale meshes, the CPU time can become prohibitive with the possibility of non-termination, see Table 3.

h_0	0.05	0.05/2	0.05/4	0.05/8	0.05/16	0.05/32
N	153	464	1950	7454	29644	115565
CPU (Sec.)	1.15	1.26	8.27	39.71	315.68	1000.98
$q_{\min}$	0.49	0.55	0.43	0.34	0.54	0.41

Table 3: Performances of the Persson-Strang mesh generator for an L-shaped domain with $h(x, y) = 1 + 5\sqrt{x^2 + y^2}$.

4 The new algorithm

We now present the new algorithm derived from the mesh generator proposed by Persson-Strang [7]. The algorithm is outlined in Algorithm 1. The initialization step (i.e. Step 0) is the same as in [7] except that the duplicated fixed points are removed from the initial distribution. The other steps of the algorithm are detailed in the next subsections.

Algorithm 1 The new algorithm

Step 0. Initialization: Generate a grid of equally spaced points in the bounding box using Matlab function `meshgrid`. Remove all points outside Ω and apply the probabilistic rejection method. Add fixed points without duplication to form the initial node positions $\boldsymbol{p}_0$.

Step 1. Triangulation: Perform a Delaunay triangulation and remove the triangles with centroid outside Ω. Reorder triangle vertices counter-clockwise. Form all edges (without duplication) and extract boundary edges.

Step 2. Mesh smoothing: Compute and assemble edge forces and move nodes. After N iterations, edges with normalized length greater than 1.5 are split by adding midpoints (and got to Step 1).

Step 3. Boundary nodes: Move (mesh) boundary nodes on the boundary of Ω. Boundary nodes of the mesh are determined as end-points of boundary edges.

Step 4. Termination criteria: Compute relative displacements of the interior points. If a large displacement is detected in the current triangulation, go to Step 1. If a large displacement is detected, in the current iteration, go to Step 2.

Step 5. Triangle quality: Check for triangle quality by computing (2.1) and triangle areas. If a triangle with negative area is found, go to Step 1. if a triangle with the quality measure $q < q_{\min}$ is found, add a new point at its center of mass and go to Step 1.

4.1 Delaunay triangulation (Step 1.)

A Delaunay triangulation is then performed using Matlab function `delaunay`. To reorder the triangle vertices, we compute triangles area using standard (FEM) counter clockwise ordering. We swap the second and the third vertices of triangles with negative area. The boundary edges are extracted.

4.2 Mesh smoothing (Step 2)

We adopt the first order motion equation (3.2)-(3.3) for our smoothing algorithm. Let $\boldsymbol{p}_i^k$ be the position of node i at the k-th time step. In order to reach a stationary state, we update node positions by the iterative procedure

$$\boldsymbol{p}_i^{k+1} = \boldsymbol{p}_i^k + \Delta t \sum_{j \in \mathcal{N}_i} f(\boldsymbol{p}_i^k, \boldsymbol{p}_j^k) \frac{\boldsymbol{e}_{ij}^k}{\ell_{ij}^k}, \quad i = 1, \ldots, N. \tag{4.1}$$

Note that in the Persson-Strang formula (3.4), the bar force $f(\boldsymbol{p}_i^k, \boldsymbol{p}_j^k)$ applies to the unitary vector $\boldsymbol{e}_{ij}^k / \parallel \boldsymbol{e}_{ij}^k \parallel$. In (4.1), the bar force applies to a vector of length r_{ij}, r_{ij} being the desired edge length.

We have tested the Persson-Strang function (3.8) and the Bossen-Heckbert function (3.9). If follows that using the Bossen-Heckbert smoothing function (3.9) it takes far less iterations to reach the stationary state. Table 4 shows the number of iterations required to reach the stationary state using the smoothing functions (3.8) and (3.9). We can notice that, with the Persson-Strang smoothing function, the algorithm requires about twice much iterations.

h_0	0.05	0.05/2	0.05/3	0.05/4	0.05/5
Persson-Strang: N/Iter	130/541	486/750	1136/1291	1971/1529	3009/1946
Bossen-Heckbert: N/Iter	110/253	497/464	1076/562	1977/725	2889/820

Table 4: Performances of the new algorithm on an L-shaped domain, Figure 3, using Persson-Strang and Bossen-Heckbert smoothing functions.

After N itererations, N being the (current) number of nodes, the edges with normalized length greater that 1.5 are split by adding midpoints as new nodes. During the smoothing process, a node i moves with the virtual "speed" $\boldsymbol{v}_i^k = (\boldsymbol{p}_i^k - \boldsymbol{p}_i^{k-1})/\Delta t$. If after "$mN$" iterations ($m \geq 1$) the smoothing process does not stop, the nodes with $\boldsymbol{v}_i^k < v_{\min}$ are fixed to accelerate the termination of the algorithm. In the code we fix $m = 5$ and $v_{\min} = \Delta t \times 0.005$.

4.3 Boundary nodes (Step 3)

In our algorithm, the boundary nodes are endpoints of boundary edges. To project boundary nodes onto the boundary, we use, instead of (3.10), the following Newton-like procedure

$$\boldsymbol{p}_i^k \leftarrow \boldsymbol{p}_i^k - d(\boldsymbol{p}_i^k) \nabla d(\boldsymbol{p}_i^k) \tag{4.2}$$

until $\| d(\boldsymbol{p}_i^k) \|$ becomes sufficiently small. In practice, the procedure converges after few iterations. The vectorized Matlab function 4.1 computes the projection.

Function 4.1 Vectorized Matlab function for computing the projection of external nodes onto the boundary

```
function [pp, iter]=pnodes(p,fd,h,tol,varargin)
%PROJNODES Project external nodes onto the boundary
%
d=feval(fd,p,varargin{:});
iter=0; pp=p;
while (max(abs(d))>tol & iter<10)
    ddx=(feval(fd,[pp(:,1)+h,pp(:,2)],varargin{:})-d)/h;
    ddy=(feval(fd,[pp(:,1),pp(:,2)+h],varargin{:})-d)/h;
    pp=pp-[d.*ddx,d.*ddy];
    d=feval(fd,pp,varargin{:});
    iter=iter+1;
end
```

4.4 Termination criteria (Step 4)

The termination criterion is based on the relative node displacement on the current iteration. The smoothing process is stopped if

$$\max_i \| \boldsymbol{p}_i^k - \boldsymbol{p}_i^{k-1} \| / h_0 < 5 \times 10^{-3},$$

h_0 is the reference edge length, provided by the user. In our code, at the end of the smoothing process, we check for triangles quality. If a triangle with negative area is found, a Delaunay triangulation is performed and the smoothing process restarts. If a triangle with quality measure $q < q_{min}$ is found, this triangle is removed and its vertices replaced by its center of mass. A Delaunay triangulation is performed and the smoothing process restarts.

5 Boundary edges and triangle neighbors

In finite element applications, boundary edges and nodes are used for assembling the boundary conditions. Boundary nodes are end points of boundary edges. We first form all edges without duplication.

```
eh0=[t(:,[2 3]); t(:,[3 1]); t(:,[1 2])]; [eh,ih]=sort(eh0,2);
ee=int64(eh(:,1)*(np+1)+eh(:,2));
[~,ie,je]=unique(ee);
e=eh(ie,:);                                          % List of edges
```

In case of a 6-node triangulation, we add the edges midpoint.

```
eq=[t(:,4); t(:,5); t(:,6)];
e=[e  eq(ie)];
```

Boundary edges appear once and boundary nodes are endpoints of boundary edges.

```
ne=size(e,1);
he=accumarray(je,1,[ne 1]); ib=find(he==1);
be=e(ib,:);                                    % List of boundary edges
bn=unique(be);                                 % List of boundary nodes
```

The Matlab function `kmg2dedg` form all edges without duplicates and extract boundary edges. This function is used in the mesh refinement (`kmg2dref`) and in computing triangle neighbor information (`kmg2dtng`).

To make our mesh generator suitable for discontinuous Galerkin methods or finite volume schemes, we have to provide for each triangle, its edges and its neighboring triangles. The Matlab function `kmg2dtng` computes for each triangle, its three edges and the three neighboring triangles. In case of a boundary triangle, at least one side has no triangle neighbor. A value of 0 is then assigned in the corresponding array entry. For a triangle t, the arrays of its edges and its neighboring triangles are arranged as in Figure 5.

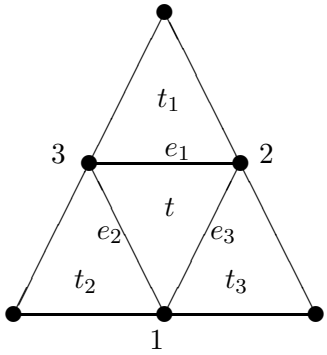


Figure 5: Triangle neighborhood

6 Mesh refinement

In our mesh generator, a coarse mesh obtained with Algorithm 1 is successively refined to produce a finer mesh. In addition, the refinement allows us to create a 6-node triangulation easily. We use only a regular refinement, i.e. all triangles are divided into four triangles of the same shape by bisection of their edges.

Let `(p0,t0)` be the Matlab representation of the coarse mesh. The refinement operations are vectorized in Matlab as follows.

(i) A list of edges is formed using `kmg2dedg`.

```
[e,ib,je]=kmg2dedg(t0);
```

`ib` is the list of boundary edges in `e`.

(ii) New nodes are created as edges midpoint. If N_e is the number of edges in the coarse mesh, the new nodes are stored in $\boldsymbol{p}_i$, $i = N+1, \ldots, N+N_e$.

```
ne=size(e,1);
p=[p0; (p0(e(:,1),:)+p0(e(:,2),:))/2];
ip1=t0(:,1); ip2=t0(:,2); ip3=t0(:,3);
lnn=[(np+1):(np+ne)]'; lnnh=lnn(je);
% edges midpoint number
mp1=lnnh(1:nt); mp2=lnnh(nt+1:2*nt); mp3=lnnh(2*nt+1:3*nt);
```

(iii) For a 3-node triangulation, the list of new triangles is formed.

```
t=zeros(4*nt,3);
t(1:nt,:)=[ip1 mp1 mp3];
t((nt+1):2*nt,:)=[mp1 ip2 mp2];
t((2*nt+1):3*nt,:)=[mp2 ip3 mp3];
t((3*nt+1):4*nt,:)=[mp1 mp2 mp3];
```

(iv) For a 6-node triangulation, we keep the coarse mesh and add the edges midpoint.

```
t=zeros(nt,6); t(:,1:3)=t0;
t(:,4:6)=[mp1 mp2 mp3];
```

(v) If during the bisection process, new boundary nodes (mid-point of boundary edges) fall outside or inside Ω, they are moved back using (4.2). Boundary edges appear once and we use Matlab function `accumarray` to count edges occurrence.

```
pb=p(np+ib,:);
[p(np+ib,:), iterb]=pnodes(pb,fd,deps,1000*deps,varargin{:});
```

Note that in case of a 6-node triangulation, the midpoints of boundary edges must not move back to boundary to keep the triangles integrity.

7 A geometry-based mesh size function

The mesh size function $h(\boldsymbol{x})$ is important for generation of high quality meshes. Our size function is designed to be adapted to the "thickness" of the computational domain. In thin regions, with small distance between boundaries, element have to be small to have high quality. The proposed function is based on the distance from the medial axis, which allows the use of any computational domain defined by a signed distance function.

The medial axis is the set of interior points that have equal distance to two or more points on the boundary. Computing a medial axis of a domain is a non-trivial problem (see,

e.g., [13, 14]). To save computational time, we only use an approximate medial axis set. For a domain $\Omega \subset \mathbb{R}^2$ with a piecewise smooth boundary, the signed distance function d is differentiable almost everywhere. In particular, d satisfies the so-called eikonal equation at every point where it is differentiable

$$\| \nabla d(\boldsymbol{x}) \| = 1.$$

The medial axis can be computed by looking for singularities of the distance function. We use the following procedure to generate the medial axis points.

1. Generate a grid of size $h_0/2$ using the Matlab function `meshgrid` and the bounding box.
2. Compute $\nabla d(\boldsymbol{x})$ using the Matlab function `gradient`.
3. Compute the medial axis $\mathcal{S} = \{\boldsymbol{x}; \quad \| \nabla d(\boldsymbol{x}) \| < .9 \text{ and } d(\boldsymbol{x}) < 0\}$.

For a point $x \in \Omega$, we compute the distance from the medial axis $d_{MA}(\boldsymbol{x}) = d(\boldsymbol{x}, \mathcal{S})$. Our mesh size function is

$$h(\boldsymbol{x}) = \alpha + |\bar{d}(\boldsymbol{x})| + \bar{d}_{MA}(\boldsymbol{x}), \tag{7.1}$$

where

- α is a strictly positive constant;
- $\bar{d}(\boldsymbol{x})$ and $\bar{d}_{MA}(\boldsymbol{x})$ are the normalized forms of d and d_{MA}, respectively, i.e.

$$\bar{d}(\boldsymbol{x}) = \frac{d(\boldsymbol{x})}{\max_{\boldsymbol{x}} |d(\boldsymbol{x})|} \quad \text{and} \quad \bar{d}_{MA}(\boldsymbol{x}) = \frac{d_{MA}(\boldsymbol{x})}{\max_{\boldsymbol{x}} d_{MA}(\boldsymbol{x})}.$$

The constant α controls the ratio between the edge lengths in "thick" regions and the edge lengths in "thin" regions. If we denote this ratio by ρ, we have that $\alpha \to 0$, implies that $\rho \to +\infty$ and $\alpha \to +\infty$, implies that $\rho \to 1$. Some typical values are:

- if $\alpha = 0.25$, then $\rho \approx 5$, that is, the edge lengths in thick regions are *about* five times the edge lengths in thin regions;
- if $\alpha = 0.5$, then $\rho \approx 3$;
- if $\alpha = 1$, then $\rho \approx 2$.

These values are obtained empirically by comparing edge lengths in "thick" and "thin" regions.

8 Numerical experiments

We present in this section some meshes generated by our algorithm. The computations have been carried out on a Dell Precision T3500 computer equipped with Intel Xeon QuadCore (2.67GHz) processor running Linux (Fedora 14). The Matlab version is 7.6. We have sometimes used the distance functions (`drectangle`, `dcircle`, etc.) introduced by [7].

The (default) quality measure (2.1) lower bound is $q_{min} = 0.5$, i.e. we accept a mesh if $q > q_{min}$ for all triangles.

8.1 Unit disc

The unit disc is defined by the distance function $d(x,y) = \sqrt{x^2+y^2} - 1$ which can be coded in Matlab as

```
function d=dcirclu(p)
d=sqrt(p(:,1).^2+p(:,2).^2)-1;
```

The bounding box for Ω is $(-1,1)^2$ and we chose $\{(-1,0),(1,0),(0,-1),(0,1),(0,0)\}$ as fixed points.

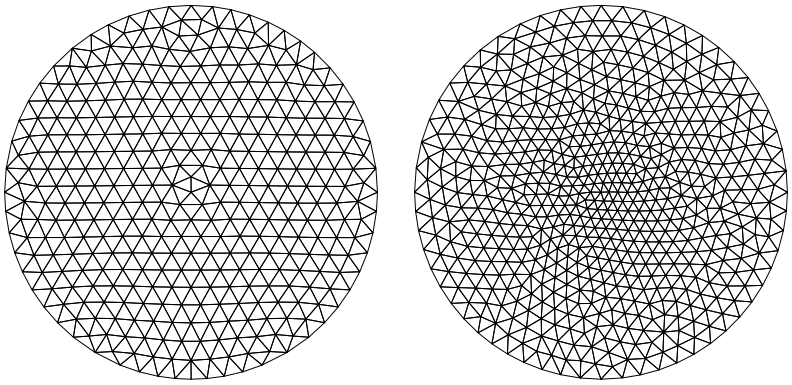
For a uniform mesh, $h(x,y) = 1$, for all $(x,y) \in (-1,1)^2$, that is, in Matlab

```
function h=hcirclu(p)
h=ones(size(p,1),1);
```

Various meshes with element size h_0 are generated using the following lines of code.

```
>>fd=@dcirclu; fh=hcirclu;
>>[p,t,be,bn]=kmg2d(fd,fh,h0,[-1,-1;1,1],1,0,[-1,0;1,0;0,-1;0,1;0,0]);
```

Figure 6.a shows the mesh obtained with $h_0 = 0.1$. Table 5 shows the performances of Algorithm 1 with a uniform size function. We can notice that the number of nodes is constant as in the Persson-Stran algorithm and the CPU time are also comparable, Table 1. However, for uniform meshes, the Persson-Strang algorithm produces better quality meshes. But in Algorithm 1, the parameter $q_{\min}$ can be modified to enhance the mesh quality.



(a) (b)

Figure 6: Mesh of a unit disc: (a) uniform size function, (b) $h(x,y) = 1 + \sqrt{x^2+y2}$

If we want to generate a mesh with a finer resolution close to the center ($(x,y) = (0,0)$) of the disc, we have to design a suitable size function $h(x,y)$. If we take $h(x,y) = 1 + \sqrt{x^2+y^2}$, the edge lengths close to the center will be about half the edge lengths close to the boundary. The corresponding Matlab function is therefore

```
function h=hcirclu(p)
h=1+sqrt(p(:,1).^2+p(:,2).^2);
```

Run	1	2	3	4	5	6	7	8	9	10
N	364	364	364	364	3064	364	364	364	364	364
Iter	203	203	203	203	203	203	203	203	203	203
CPU (Sec.)	.25	.24	.25	.25	.24	.25	.24	.25	.24	.24
$q_{\min}$	0.51	0.51	0.51	0.51	0.51	0.51	0.51	0.51	0.51	0.51

Table 5: Robustness of Algorithm 1 on a unit disc with $h(x, y) = 1$ and $h_0 = 0.05$.

Run	1	2	3	4	5	6	7	8	9	10
N	613	613	602	586	580	599	599	581	613	591
Iter	451	272	327	259	230	492	305	352	506	352
CPU (Sec.)	0.94	0.59	0.66	0.60	0.51	0.92	0.66	0.71	1.08	0.65
$q_{\min}$	0.56	0.78	0.72	0.64	0.71	0.69	0.74	0.73	0.64	0.72

Table 6: Robustness of Algorithm 1 on a unit disc with $h(x, y) = 1 + \sqrt{x^2 + y^2}$ and $h_0 = 0.05$.

Figure 6.b shows the mesh obtained with $h_0 = 0.05$. Table 6 shows the robustness of our mesh generator. All the meshes generated are such that $q_{\min} > 0.5$ as presribed. The prescribed $q_{\min}$ can be modified to increase the mesh quality, see Table 7.

Prescribed $q_{\min}$	0.5	0.55	0.6	0.65	0.7	0.75	0.8
N	613	585	578	621	614	592	493
CPU (Sec.)	0.94	0.86	1.19	0.78	1.54	1.20	5.30
$q_{\min}$	0.56	0.70	0.61	0.68	0.74	0.75	0.80

Table 7: Performances of Algorithm 1 on a unit disc with $h(x, y) = 1+\sqrt{x^2 + y^2}$, $h_0 = 0.05$ and various prescribed $q_{\min}$.

8.2 L-shaped domain

Now we consider the L-shaped domain of Figure 3. For the signed distance function `dlshape` (not shown here), we use the Persson-Strang Matlab function `dpoly` defining the distance to a given polygon. The mesh size function is $h(x, y) = 1 + 5\sqrt{x^2 + y^2}$. The following commands generate the mesh of Figure 7. We notice that the mesh has good element quality, compared to Figure 3.

```
>> bbox=[-1,-2; 2,2];
>> pfix=[-1,-1; 0,-2; 2,0; 0 2; -1,1; 0,0];
>> fd=@dlshape;  fh=@hlshape; h0=0.05;
>> [p,t,be,bn]=kmg2d(fd,fh,h0,bbox,1,0,pfix);
```

To study the numerical behavior of Algorithm 1, on the L-shaped domain, the initial reference size parameter $h_0 = 0.05$ is successively divied by 2, 4, 8, 16 and 32. Table 8

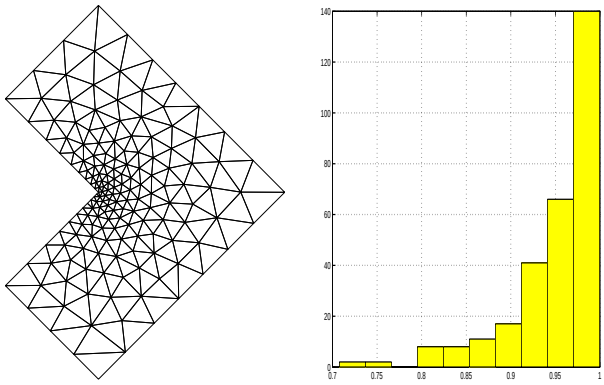


Figure 7: Mesh of an L-shaped domain and histogram of element qualities with $q_{\min} = 0.7$

h_0	0.05	0.05/2	0.05/4	0.05/8	0.05/16	0.05/32
N	129	479	1911	7453	29195	116241
Iter	270	404	1062	991	1813	4954
CPU (Sec.)	0.94	1.99	14.64	52.94	418.16	3952.44
$q_{\min}$	0.71	0.57	0.65	0.59	0.56	0.54

Table 8: Performances of Algorithm 1 on a L-shaped domain with a nonuniform mesh size function.

shows the performances of Algorithm 1. Comparing Table 8 and Table 3, we first notice that the number of nodes generated, by both algorithms, is comparable. Algorithm 1 requires much more computational time but guarantees good element quality.

8.3 Geometry-based mesh size function

We illustrate, in this subsection, the behavior of our geometry-based mesh size function, based on the "thickness" of the computational domain.

8.3.1 Hook form

We mesh the domain defined by the following signed distance function

```
function fd=dhook(p)
x=p(:,1); y=p(:,2);
d1=sqrt(x.*x+y.*y)-1; d2=sqrt((x+.4).^2+y.^2)-.55;
fd=max([d1 -d2 -y]')';
```

In the geometry-based mesh size function (7.1), we set $\alpha = 0.4$. The egde lengths ratio (ρ) will be about 4. The following Matlab lines generate the mesh of Figure 8, after 563 iterations.

```
bbox=[-1,0; 1,1]; pfix=[-1,0; 0,1; -0.95,0; 0.15 0]; h0=0.0125;
fd=@dhook; fh=@hgeom;
[p,t,be,bn]=kmg2d(fd,fh,h0,bbox,1,0,pfix);
```

The resulting mesh has 1520 nodes and $q_{\min} = 0.71$. Figure 8 also shows the approximate medial axis used in the geometry-based mesh size function (7.1). Figure 9 shows the meshes generated with $\alpha = 0.15$ and $\alpha = 1$.

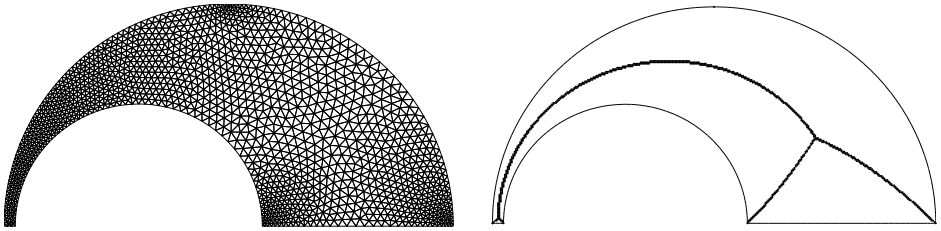
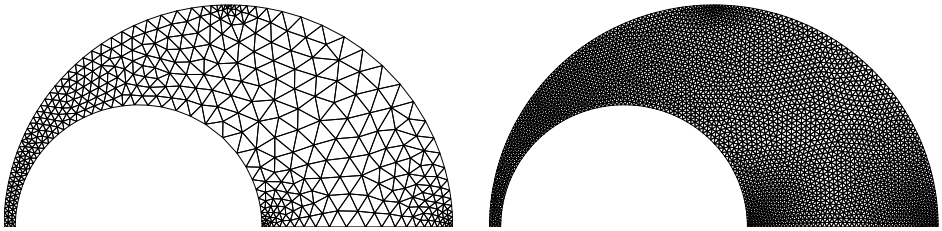


Figure 8: Mesh of a hook form using the geometry-based size function (7.1) and the approximate medial axis used in (7.1).



(a) (b)

Figure 9: Mesh of a hook form using the geometry-based size function (7.1): (a) $\alpha = 0.15$, (b) $\alpha = 1$

8.3.2 Square with holes

We now want to mesh the square $(-1, 1)^2$ with four circular holes of radius $r = 0.25$, defined by the following signed distance function

```
function fd=dsquare(p)
r=.25;
xc1=-.5; xc2=.5;  xc3=.5; xc4=-.5;
yc1=-.5; yc2=-.5; yc3=.5; yc4=.5;
dc1=dcircle(p,xc1,yc1,r); dc2=dcircle(p,xc2,yc2,r);
dc3=dcircle(p,xc3,yc3,r); dc4=dcircle(p,xc4,yc4,r);
dc=min(dc1,min(dc2,min(dc3,dc4)));
fd=max(drectangle(p,-1,1,-1,1),-dc);
```

We set $\alpha = 0.4$ in (7.1). The following Matlab commands generate the mesh of Figure 10, after 665 iterations.

```
bbox=[-1,-1; 1,1]; pfix=[-1,-1; 1,-1; 1,1; -1,1]; h0=0.025;
fd=@dquare; fh=@hgeom;
[p,t,be,bn]=kmg2d(fd,fh,h0,bbox,1,0,pfix);
```

The resulting mesh has 1184 nodes and $q_{\min} = 0.6$. Figure 10 also shows the approximate medial axis used in the computations.

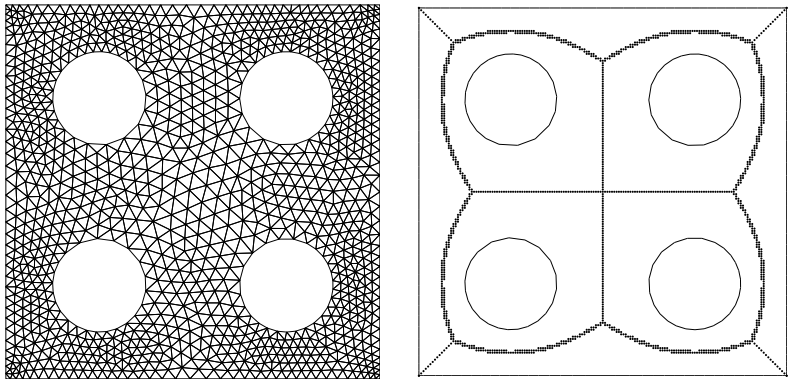


Figure 10: Mesh of a square with holes using the geometry-based size function (7.1) and the approximate medial axis used in (7.1).

8.3.3 Polygonal domain

We now consider the polygonal domain $\Omega = (-0.25,\, 1.25) \times (0,\, 0.25) \cup (0,\, 1) \times (-1,\, 0)$, defined by the following signed distance function.

```
function d=ddcavity(p)
d1=drectangle(p,0,1,-1,0);
d2=drectangle(p,-.25,1.25,0,0.25);
d=min(d1,d2);
```

In function 7.1, $\alpha = 0.4$. The following Matlab lines generate the mesh of Figure 11, after 349 iterations.

```
bbox=[-.25,-1;1.25,0.25];
pfix=[-.25,0;0,0;0,-1;1,-1;1,0;1.25,0;1.25,0.25;-0.25,0.25];
h0=0.0125; fd=@ddcavity; fh=@hgeom;
[p,t,be,bn]=kmg2d(fd,fh,h0,bbox,1,0,pfix);
```

The resulting mesh has 2629 nodes and $q_{\min} = 0.64$.

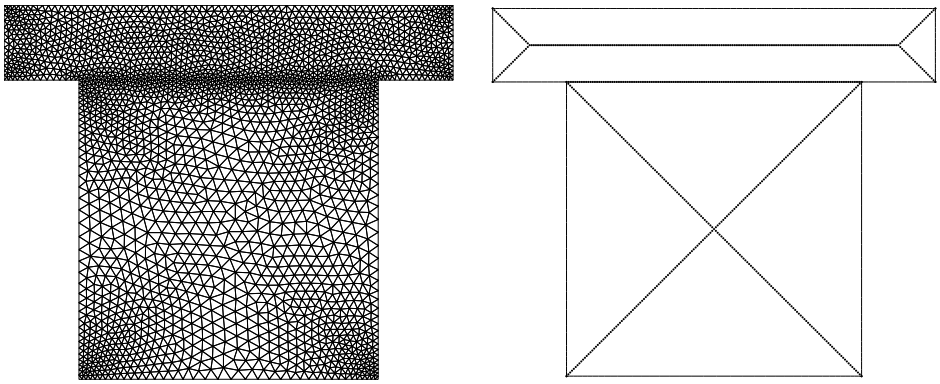


Figure 11: Mesh of a polygonal domain using the geometry-based size function (7.1) and the approximate medial axis used in (7.1)

8.4 Mesh refinement and 6-node triangulation

Large scale meshes can be generated using the refinement strategy. It suffices to specify the number of desired refinements n_r when calling the Matlab mesh generator function `kmg2d`

```
>> nr=2;
>> [p,t,be,bn]=kmg2d(fd,fh,h0,bbox,1,nr,pfix);
```

If a mesh is already available, the refinement function `kmg2dref` can be called directly, as many times as needed.

```
>> for i=1:nr, [p,t,be,bn]=kmg2dref(p,t,1,fd,sqrt(eps)*h0,tol); end
```

where `sqrt(eps)*h0` is the spacing used in the finite difference scheme to compute the gradient, `tol` is the tolerance in the projection procedure.

Consider the unit disc with the nonuniform mesh size $h(x,y) = 1 + \sqrt{x^2+y^2}$. Since the boundary is non polygonal, the projection (onto the boundary) procedure (4.2) is non trivial. With $h_0 = 0.025$, Algorithm 1 generates a mesh with 2262 nodes after 2.35 seconds. This mesh is successively refined using the function `kmg2dref`. We report in Table 9 the number of nodes and $q_{\min}$ for the resulting meshes, and the CPU times needed for the refinement. It appears that, with the refinement strategy, the saving of computational time is significant.

n_r	0	1	2	3	4	5
N	2262	8938	35475	141429	564777	2257233
CPU (Sec.)	2.35	0.02	0.15	0.48	1.88	7.94
$q_{\min}$	0.68	0.68	0.68	0.68	0.68	0.68

Table 9: Performances of Algorithm 1 on the unit disc with the refinement strategy ($h_0 = 0.025$)

For the generation of a 6-node triangular mesh, the computational cost is slightly higher than for a mesh with 3-node triangles (the cost of one refinement). Table 10 shows the performances of our algorithm for the generation of a mesh with 6-node triangles with $h_0 = 0.1$.

h_0	0.5	0.5/2	0.5/4	0.5/8	0.5/16
N	61	211	875	3604	14693
CPU (Sec.)	0.22	.26	1.31	6.93	12.92
$q_{\min}$	0.92	0.78	0.75	0.45	0.65

Table 10: Performances of Algorithm 1 for generating 6-node triangular meshes ($h_0 = 0.5$)

9 Conclusion

We have presented a Matlab mesh generator for finite element applications. Our mesh generator improves the code by Persson and Strang [7] while keeping simplicity and shortness. To generate large scale meshes, we have adopted the successive refinement strategy. We also added a geometric mesh size function based on the "thickness" of the computational domain. Numerical experiments show that various meshes, of various sizes, can be generated with our code. In the spirit of [7], we hope that readers will use this code and adapt it.

References

[1] Edelsbrunner H. *Geometry and Topology for Mesh Generation.* Cambridge University Press, 2001.

[2] Du Q. and Gunzburger M. Grid generation and optimization based on centroidal Voronoi tesselations. *Appl. Math. Comput.*, 133:591–607, 2002.

[3] George P.-L. *Mesh Generation - Application To Finite Elements.* Hermes Science Publishing, Paris, 2000.

[4] Topping B.H.V., Muylle J., Ivanyi P., Putanowicz R. and Cheng B. *Finite Element Mesh Generation.* Saxe-Coburg Publications, 2004.

[5] Zhu X., Wang Y., Zan J. and LI C. Application of fractal theory in generation and refinement of finite element mesh. *Appl. Math. Comput.*, 175:1039–1045, 2006.

[6] Langemyr L., Nordmark A., Ringh M., Ruhe A., Oppelstrup and Dorobantu M. *Partial Differential Equations Toolbox User's Guide.* The Math Works, Inc., 1995.

[7] Persson P.-O. and Strang G. A simple mesh generator in Matlab. *SIAM Rev.*, 42:329–345, 2004.

[8] Bossen F. J. and Heckbert P. S. A pliant method for anisotropic mesh generation. In *Proceeding of the 5th International meshing Roundtable*, pages 63–74, 1996.

[9] Field D. A. Laplacian smoothing and Delaunay triangulations. *Comm. Applied Numer. Meth.*, 4:709–712, 1988.

[10] Allen M. P. and Tildesley D. J. *Computer Simulation of Liquids*. Clarendon Press, Oxford, 1987.

[11] Béal P., Koko J. and Touzani R. Mesh r-adaptation for unilateral contact problems. *Int. J. Appl. Math. Comput. Sci.*, 12(1):101–108, 2002.

[12] Habashi W. G., Dompierre J., Bourgault Y., Ait-Ali-Yahia D., Fortin M. and Vallet M.-G. Anisotropic mesh adaptation: towards user-independent, mesh-independent and solver-independent CFD. Part I. *Int. J. Numer. Meth. Fluids*, 32:725–744, 2000.

[13] Persson P.-O. Mesh size functions for implicit geometries and PDE-based gradient limiting. *Engineering with Computers*, 42:95–109, 2006.

[14] Rumpf M. and Telea A. A continuous skeletonization method based on levet sets. In *EUROGRAPHICS-IEEE TCVG Symposium on Visualization*, pages 409–419, 2002.